Technical University of Crete, Greece

School of Electrical and Computer Engineering

Architecture and Implementation of a Distributed Complex Event Processing System

Vasiliki Manikaki

Thesis Committee:

Professor Antonios Deligiannakis (Supervisor)

Professor Minos Garofalakis

Professor Vasilis Samoladas

Chania, April 2017

Abstract

Distributed event detection is the process of identifying specific occurrences of interest in incoming data available at a number of distributed nodes. The traditional approach for detecting events implies central collection and processing of data, which is impractical for a number of reasons. Firstly, since the number of nodes might be large, collecting information centrally is not always possible or efficient. This happens because the amount of information to be transmitted may be huge and the available bandwidth insufficient to accommodate the transmission. Secondly, central processing of distributed data is not balancing the cost for answering more complex queries and for big data applications the processing speed may introduce additional latency in complex event detection. Additionally, processing all data in a distributed network in a single node generates a single point of failure. In-situ processing for complex event detection systems is an architectural scheme that can alleviate the aforementioned limitations. It provides a mechanism for balancing the work load of both event processing and network traffic by distributing coordinating duties to multiple nodes. The additional division of monitoring complex queries in multiple steps, based on event frequencies, enables each node to relay just the absolutely required events to the coordinating nodes for evaluation. Additionally, the geometric method allows a network to monitor in a distributed way if the value of a complex function, even nonlinear, calculated using incoming data is over or under a specific threshold value. Thus, composite events can be distributely detected if they are expressed as a threshold monitoring function. The geometric method imposes a set of local constraints on each node and manages to reduce the need for communication between the nodes as long as the constraints are satisfied.

In this work, a unified architectural integration of in-situ complex event processing and the geometric method is implemented, using the real-time distributed computation framework named Storm, for distributed event detection. A topology is implemented to handle both the monitoring of complex functions as well as complex event queries using Storm components. All necessary mechanisms for intra and inter node communication are also addressed to facilitate the optimization objectives. Finally, the system is designed to recover after a node transient failure and special care is taken to allow real-time system adaptivity in case event frequencies drift significantly over time.

Acknowledgements

This thesis is the end of my long journey in finishing my studies. This degree is the result of the effort and support of a lot of people. Here, I am going to thank those people without whom I might not have been able to finish my studies successfully. I am so lucky that I got many people around me who were always ready to help me.

The first person that deserves my gratitude is my supervisor, Professor Antonios Deligiannakis for his continuous guidance, support, and encouragement during our co-operation. I would like to express my deepest thanks for the opportunity he gave me to deal with such an interesting topic. I am grateful for his generous help and professionalism throughout the elaboration of this study. It is not often that one finds an advisor that always finds the time to listen to the little problems that unavoidably crop up in the course of this work. I am obliged to him more than he knows. It has been a great experience for me to work under his supervision.

I would also like to thank the rest of the members of my examination committee, Prof. Minos Garofalakis and Prof. Vasileios Samoladas for the time they spent on reading and evaluating this master's thesis.

I would like to cease this opportunity to thank my fellow labmates in Softnet Lab: Giannis Flouris and Nikos Giatrakos, for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in our business trips.

Apart from work, these acknowledgements would not be complete without thanking my family for their constant support, care and love. They have always encouraged me to explore my potential and pursue my dreams. I thank my mother Giota, my father

Stratos and my sister Konstantina for supporting me spiritually throughout my life.

As for my dear Michalis, I find it difficult to express my appreciation because it is so boundless. He is my rock. Without his love and support, I would be lost. I am grateful to my precious Michalis, not just because he has given up so much to make my career a priority in our lives, but because always reminded me that "it's OK to stress just not to stress out".

To all of you, thanks for always being there for me.

Vasiliki
Chania 2017

To Michalis Foukarakis,
my wonderful husband, my best friend and my true Love

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis Motivation and Contribution

Complex event processing is used in distributed systems for extracting relevant information. In this scope, many methodologies exist that achieve this purpose. However, the most common approach, central collection and processing of information, encounters a number of issues. First of all, it is not efficient in cases where the number of nodes is large, since there is a huge amount of information that needs to be transmitted. Another reason is that complex queries introduce latency during processing. Finally, the central collecting node is prone to failure, which could prove catastrophic for the system.

An efficient way to combat these common issues that occur during complex event processing is to perform in-situ processing and distribute the workload for processing among nodes. This work has been conducted in the context of the FERARI[1] (Flexible Event pRocessing for big dAta aRchItectures) project, which proposes a highly scalable distributed streaming architecture that can support efficient complex event processing with this method [2] [3].

In this work, a system that enables realtime Complex Event Processing (CEP) for large volume event data streams over distributed topologies has been developed. The components of the system architecture have been described and implemented using entities of the Storm real-time distributed computation framework. The resulting topologies

[1]FERARI is a collaborative project within the European Commission's FP7 ICT Work Programme: http://www.ferari-project.eu/

describe the nodes of the system and the intra- and inter-site communication between the nodes has been developed in an efficient way.

The first way to achieve in-situ processing is the use of the geometric method for function monitoring. This approach decomposes the monitoring problem into local constraints that can be imposed on the geographically distributed data streams, thus achieving reduction in communication. Each node checks these constraints locally for each data received from the stream. Collecting data centrally is only required when a local constraint on a stream has been violated [1].

The architecture has additionally given the nodes the capability to send and receive only the events that are required at each point in time by following a push/pull paradigm instead of blindly sending each possibly relevant event to all potential receiver nodes. In addition to that, in cases of temporary node failure, the system is able to continue the processing and possibly recover the delayed events. Extending the distributed network is also supported by the system, by adding new nodes.

Finally, in cases where the latency for the inter-site communication has changed significantly or there is a drift in the event occurrence frequencies, the system is able to adapt and handle changes in the event processing plans.

1.2 Thesis Outline

The thesis is structured and organized in the following chapters:

Chapter 2 describes the relevant theoretical scientific background. It covers complex event processing concepts and the geometric approach for function monitoring.

Chapter 3 includes information on the systems and technologies that have been utilized during the implementation of this work, which are the Apache Storm system, IBM Proton and Redis.

Chapter 4 presents the components of the system architecture on both inter- and intra- site level. Their composition, general functionalities setup and interaction are described. The capability of the system to handle the addition of nodes and delayed messages during potential communication failures at runtime is also presented.

Chapter 5 covers the methodology for implementing geometric monitoring including a thorough example using the geometric approach.

Chapter 6 describes the ordinary complex event processing work flow and the query plan adaptivity capability of the system.

Chapter 7 concludes the thesis.

Chapter 2

Theoretical Scientific Background

2.1 Complex Event Processing

Event processing is a method of tracking and analyzing (processing) of streams of information (data) about things that happen. Complex Event Processing (CEP) combines data from multiple sources to infer events or patterns for complicated situations. The objective of CEP systems is to process Machine to Machine (M2M) data in an efficient manner and immediately recognize the occurence of interesting situations (events).
Some examples of CEP applications:

- Mobile and sensor networks
- Computer clusters and smart energy grids
- Network health monitoring applications
- Security attacks detection

Performance in CEP systems is very important. There are many performance metrics such as Throughput [4], [5], [6], [7], [8] which refers to the number of events processed per time unit, or CPU cost of the language operators [4], [8]. The aforementioned metrics are mostly used in CEP systems that receive and process data from only one source. On the other hand, there are CEP systems that receive data from a collection of streams and where there is a central site that decides in which site each event will be processed. In this case, the most common performance metrics are the Transmission Cost and the

Detection Latency, which is the time between the occurrence of an event and its detection from the central site.

2.1.1 Events

An event is an occurrence of interest in time within a particular system or domain [7] [8]; it is something that is contemplated as having happened in that domain or something that has already happened.
The authors in [6] define the event tuples as e = $< s ; t >$, where:

- **e**: represents the event of interest.
- **s**: refers to a list of attributes.
- **t**: is a list of timestamps, the first timestamp represents the start of the event and the last timestamp represents the end of it.

Typically in literature [5], [6], [7], [9] events are categorized as simple (primitive) and complex (also called composite or deferred). Simple events are atomic occurrences of interest, while complex events are events that summarize, represent, or denote a set of other events (primitive and/or complex). Complex events are detected by the CEP system based on defined patterns (rules) that involve other events.

2.1.2 Queries

CEP applications are rule-driven. Usually rules are expressed as queries that are submitted in the CEP system in order to detect complex events using a given pattern. A complex event query is commonly expressed using the following form [4], [7], [8], [10]:

```
PATTERN OPERATOR (list of TypeOfEvents)
WHERE (event value constraint)
AND (another value constraint)
WITHIN (time)
RETURN (ComplexEvent or Events to return)
```

PATTERN OPERATOR: Includes operators such as SEQ, AND, OR and others.

More specifically the SEQ pattern operator demands the occurrence of all the involved events to be sequential. The AND pattern operator requires all the involved events to occur, while the OR pattern operator requires any of them to occur.
WHERE: This clause is used to define equality or inequality constraints among the attributes of the events. The constraints can be separated by logical operators.
WITHIN: In order to fully match the pattern and successfully detect a complex event, the events must occur within the specified time window that this pattern operator defines.
RETURN: The RETURN operator specifies the complex event or list of simple events to output.

While the events arrive on the stream that the CEP receives, the query pattern can be partially matched. Two methods for tracking the state of a partial match are i) **Automata-based** and ii) **Graph-based**.

2.2 The Geometric Approach

Considering a network of n nodes, each node S_i $(i = 1...n)$ maintains a local d-dimensional vector, termed as the *local statistics vector*, with the j-th $(j = 1...d)$ 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 among 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 monitoring framework in [11], [12], [13], [14], [15], [16], [17] to be applicable, any supported monitoring function $f : R^d \rightarrow R$ 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 monitoring function can in general be non-linear. Given a threshold T, the framework 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

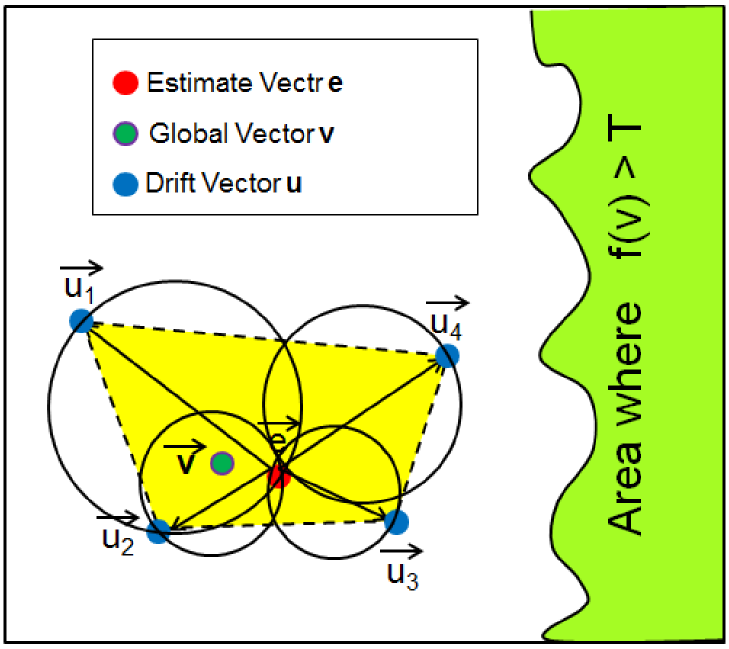

Figure 2.1: Local constraints using the Geometric Approach. Each node constructs a sphere with diameter the drift vector $\vec{u}$, of the node and the estimate vector $\vec{e}$,. The global statistics vector $\vec{v}$, is guaranteed to lie in the convex hull of $\vec{e}$, $\vec{u}_1$, $\vec{u}_2$, $\vec{u}_3$, $\vec{u}_4$. The union of the local spheres covers the convex hull.

times, and (ii) a delta vector $\Delta\vec{vi}$, 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 $\vec{u}_i = \vec{e} + \vec{v}_i$. Additional optimizations 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 included in the calculation of the drift vector.

The domain space R^d represents the potential locations of the global statistics vector at any time. Let all points in R^d where $f(\vec{v}) \leq T$ be colored by the same color (i.e., white in Figure 2.1), while the remaining points be colored by a different color (i.e., green in Figure 2.1). 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 $Conv(\vec{u}_1, ..., \vec{u}_n)$ of the drift vectors and, thus, within the convex hull $Conv(\vec{e}, \vec{u}_1, ..., \vec{u}_n)$ of the drift vectors and the estimate vector. Thus, if $Conv(\vec{e}, \vec{u}_1, ..., \vec{u}_n)$ 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 $Conv(\vec{e}, \vec{u}_1, ..., \vec{u}_n)$, since it is not aware of the current drift vectors of other sites. However, an important observation [15] 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.

Using Safe-Zones. The more recent work of [11], [12], [13] simplifies the local tests performed by nodes by having each node test whether its drift vector [12], [13] or its local statistics vector [11] lies within a convex region, also known as a safe zone. This test is very efficient and only depends on the complexity of the bounding convex region. Nodes do not transmit information as long as their local vectors lie within their safe zone. This condition also makes sure that global threshold violations cannot occur unless at least one local vector of a node lies outside the safe zone. This method is more effective than using the simple spherical constraints, as the bounding regions that can be calculated using a better reference vector than the estimate vector lead to fewer communications. The most important challenge in this method is appropriately selecting large safe zones. In the simplest case, each node has the same safe zone regardless of the data distribution, however there have been optimizations where the shapes of the safe zones are different for each node [18].

The work in [13] demonstrates how a safe zone 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.

Chapter 3

Supporting Systems

3.1 Apache Storm

Apache Storm is a distributed real-time computational framework. Storm was created at Backtype, a social analyticscompany acquired by Twitter in 2011. It became an open-source project on September 19, 2011 and it is used today by over 60 companies including Twitter[1], Groupon[2], Alibaba[3], and Spotify[4]. It is written in both Java and Clojure and can be used with any programming language. The main properties of Storm are:

- **Scalable**: Storm can handle an enormous number of messages per second. A topology can be scaled by adding machines and increasing its parallelism settings. The topology's tasks can be assigned to the new machines as soon as they are added.

- **Fault-tolerant**: Storm is responsible for reassigning tasks as necessary if faults (e.g. a worker is down) are discovered during execution of computations and for making sure that computation can run forever unless killed.

- **Guarantees no data loss**: Storm never leaves any messages unprocessed. If errors are detected the messages might be processed more than once, so it is guaranteed that no message will be lost.

[1] https://twitter.com/
[2] http://groupon.co.uk/
[3] https://alibaba.com/
[4] https://spotify.com/

- **Extremely robust**: Storm clusters are easy to manage and Storm has been built with novice users in mind.
- **Programming language agnostic**: Storm is implemented in Java, but it is possible to use other languages such as Python or Ruby to implement a topology.
- **Simple to program**: Compared to other real-time processing systems and methods, writing programs using Storm is quite simple.
- **Fast**: Storm has been designed with speed in mind and manages to perform real-time processing quickly (e.g. over a million tuples processed per second per node) and reliably.

3.1.1 Main Concepts

3.1.1.1 Streams

Streams are unbounded sequences of tuples. Tuples are named list of values of any data type. The data types supported by Storm are all the primitive types, strings, and byte arrays, as well as serializable custom objects. Storm allows streams to be transformed into other streams between its components reliably.

3.1.1.2 Spouts

The role of a storm project is to process streams. Stream emission in a topology is achieved using spouts. A spout instance can be implemented for reading from many different external sources such as distributed file systems, databases, messaging or queueing frameworks, and for reading directly from a file as well. They transmit their data to other components to process them further. Spouts can be configured to emit multiple streams.

3.1.1.3 Bolts

A bolt is a component that receives tuples as input (from spouts or bolts) and emits tuples as output (to other bolts). Bolts process input streams by performing a number of tasks such as running functions, filtering, streaming aggregations, streaming joins and others. Similarly to spouts, bolts can also be configured to emit multiple streams.

3.1.1.4 Topologies

A specific arrangement of spouts, bolts and their connections is called a topology and contains the whole application logic. Figure 3.1 shows how the components of a topology are connected. Each edge in the figure represents a stream subscription. For example, the first spout sends data to all the first level bolts, while the second spout only sends data to one of the first level bolts. The components of a topology are also called nodes and run in parallel. The level of parallelism for each node is defined in the topology's configuration and represents the number of threads spawned to perform execution. It is possible to adjust (decrease or increase) the worker processes without requiring restarting the topology or cluster. A topology runs forever until it is explicitly terminated.

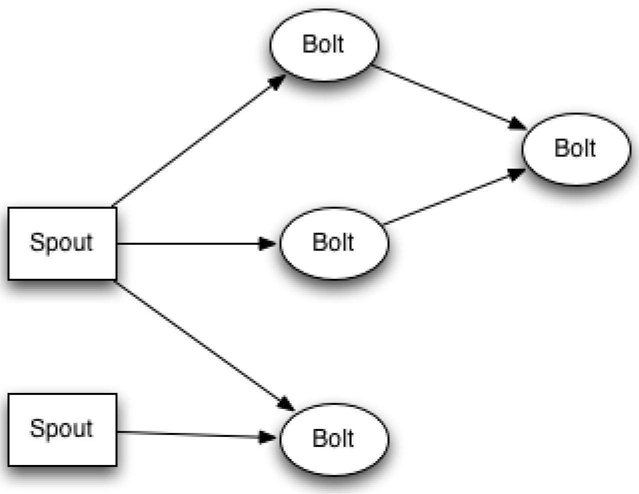

Figure 3.1: Storm Topology (figure from: http://storm.apache.org/releases/current/Tutorial.html).

3.1.2 Stream Grouping

A topology needs to define which streams each bolt should receive as input. This is achieved using stream groupings. Storm offers some built-in groupings, while custom groupings are also supported. The built-in groupings are the following:

- **Shuffle Grouping**: In this grouping tuples evenly and randomly (round robin) distributed across the tasks. This means that each bolt is guaranteed to get an equal number of tuples.
- **Fields Grouping**: A fields grouping is able to group a stream by a subset of its fields. This means that equal values for that subset of fields go to the same task.
- **All Grouping**: The tuple is sent to all tasks. All grouping is used to send signals to bolts.
- **Global Grouping**: In global grouping all tasks send tuples to a single task, the one with the lowest ID.
- **None Grouping**: At the time of this writing, this grouping works the same as a shuffe grouping. In the future, bolts with this grouping will execute in the same thread as the component (bolt or spout) they subscribe from, if possible.
- **Direct Grouping**: In direct grouping, tuples are emitted directly to a specific consumer task.
- **Local / Shuffle Grouping**: This grouping is used to emit tuples to task in the same worker process. If this is not possible, it works similar to shuffe grouping.

3.1.3 Storm Architecture

A Storm cluster has two kinds of nodes: the master node and the worker nodes. The master node runs Nimbus, a daemon which is responsible for monitoring the cluster for failures, assigning tasks to machines, and distributing code around the cluster. Worker nodes run two types of processes: one or more Workers and a single instance of the Supervisor daemon. The Supervisor starts and stops worker processes when necessary by listening for work assigned by nimbus to the node it runs on. Each worker process executes a subset of a topology. A running topology consists of many worker processes spread across many machines. Storm requires a Zookeeper cluster1 which coordinates Nimbus and the Supervisors. Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

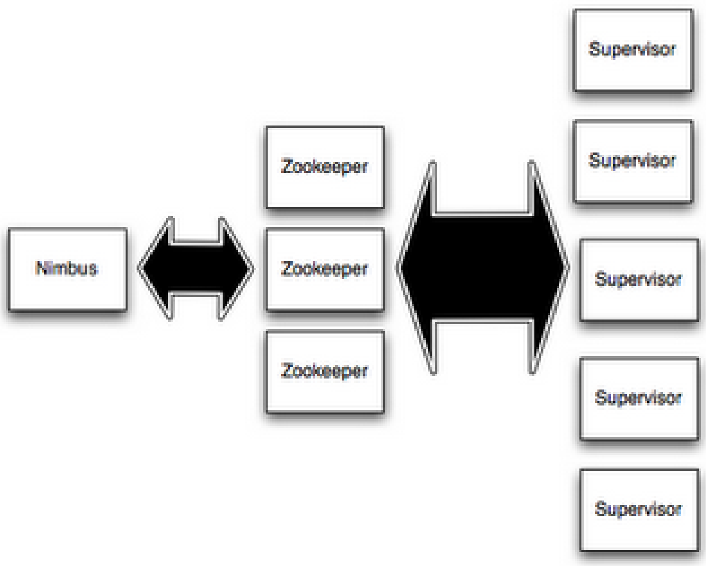

Figure 3.2: Storm cluster architecture (figure from: http://storm.apache.org/releases/current/Tutorial.html).

Zookeeper can also store Nimbus's and Supervisors' state. The cluster architecture is shown in Figure 3.2.

3.2 IBM Proactive Technology Online (Proton)

3.2.1 Standalone Proton

At IBM Research - Haifa, a complex event processing platform named Proton - IBM Proactive Technology Online has been developed. The platform supports both event-driven and complex event processing applications and one of its strengths is that it responds to situations instead of just single events. Its engine component utilizes pattern matching for series of events that occur within a dynamic time window, also known as a context.

The project is open source, platform-independent, written in Java and is available as both a J2EE (Java to Enterprise Edition) application and a J2SE (Java to Standared Edition) application. A Proton development project contains the following definitions:

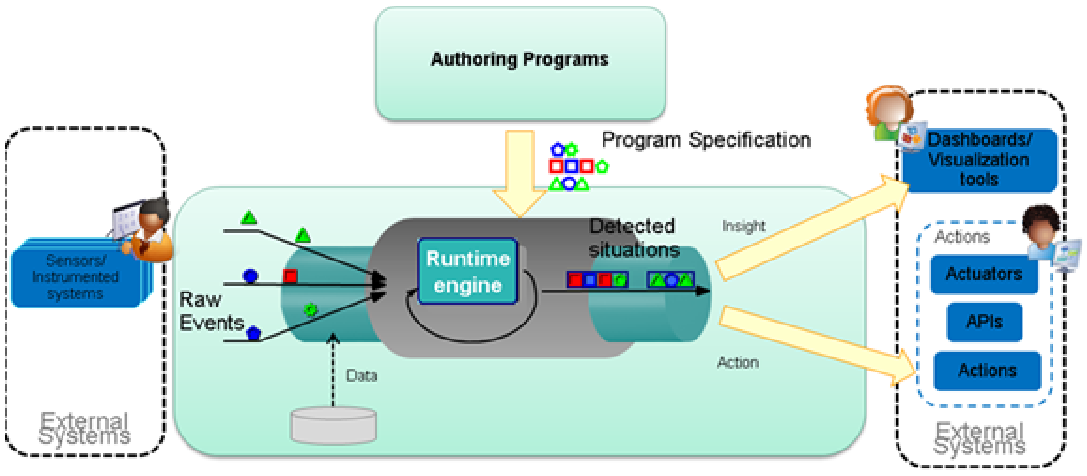

Figure 3.3: Proton Authoring Tool and Runtime Engine (figure from: https://github.com/ishkin/Proton).

events, producers, consumers, temporal contexts, segmentation contexts, composite contexts, and event processing agents (EPAs). The combination of these definitions constitutes an Event Processing Network (EPN), which is a directed graph with EPAs and event classes as nodes.) The Proton engine processes these definitions and takes action whenever needed, reporting anything that is required to the engine consumers. The above definitions are described in more detail below.

- **Events**: The Proton engine continuously receives a number of events while running. These events contain data related to the application domain (system or user's business) and are represented by classes where the event data is contained within the class attributes. Event instances include both built-in attributes and user-defined attributes. The most significant built-in attributes are the following:
  - **Name**: The name of the relevant event class. This is the only attribute that is mandatory. For the rest, if they have no value, the engine constructs them and assigns a default value.
  - **DetectionTime**: The Proton engine assigns the time of event detection to this attribute. The value represents milliseconds that have passed since the

  Unix Epoch (00:00:00 Thursday, 1 January 1970), a standard way of measuring time instants.

  - **OccurrenceTime**: This attribute represents a date, which can possibly be assigned to the event by the source that created it and has the value of the event occurence time. The default value is equal to the DetectionTime attribute in cases where the engine finds the value empty.
  - **Duration**: If the event occurs within a time interval, this attribute measures the duration of the event in milliseconds. The default value is 0.0.
  - **Certainty**: This attribute can have a value between 0.0 and 1.0 and represents the certainty of the event. The default value is 1.0.
  - **EventId**: The event source may provide a string identification of the event. The default value that the engine uses if the attribute has no value is an auto-generated identifier.
  - **EventSource**: This attribute represents the the name of the source of the event. It has a default empty string value.

  In addition to the above, as well as some other attributes such as **Annotation**, the event instance can also have user-defined attributes. The only limitation is that the user-defined attributes must not have the same name as a built-in attribute.

- **Producers**: Producers are entities that bring events related to the outside world into the event-processing network. The producers utilize adapters for pushing or pulling events into the EPN. There are three main adapter types supported:

  1. **File**: With this adapter, the events are read from a given file. The events are injected either at a constant rate (default file adapter) or in a more sophisticated way (timed adapter) that depends on the relative difference between the OccurenceTime attribute values of consecutive events in the file.
  2. **REST**: This adapter is a REST client. The events are periodically retrieved from an external REST service using GET methods. The adapter also provides a number of parameters that include information on the URL of the REST service, the content type and the polling mode (single or batch event instances).

  3. **Custom**: The producer reads events using a custom, user-defined mechanism.

  Apart from adapters, the producers can have a number of other parameters such as polling interval which dictates how often the producer will retrieve events from the source, input formatters (e.g., json, csv), delimiters to use for distinguishing event attributes, separators, date formatters and others. Custom adapters may include additional parameters.

- **Consumers**: A consumer consumes events generated by the EPN and sends them to the outside world. Similar to the producers, consumers support a number of adapter types. Their parameters have names and values and there are built-in parameters and optional additional parameters. There are three types of adapters:

  1. **File**: With this adapter, the events are written to a given file.
  2. **REST**: This adapter is a REST client. The events are periodically sent to an external REST service using POST methods. The adapter also provides a number of parameters that include information on the URL of the REST service, the content type and an optional authorization token which may be used for authorization purposes in the POST HTTP request header.
  3. **Custom**: The consumer writes events using a custom, user-defined mechanism.

  The consumers also include parameters that define the formatting of the events (event/date formatters, delimiters, separators, etc.).

- **Contexts**: Contexts determine when a particular event-processing agent is relevant. The event processing agents (described later) can open multiple context instances at the same time and evaluations for each open context are made in parallel. The contexts are divided into three types:

  – **Temporal contexts**: A temporal context defines a time window in which the event-processing agent is relevant. The context is started by an **Initiator** and its termination is done by a **Terminator**. The temporal context may have several different kinds of initiators and terminators.

    Initiators belong to one of the following types:

1. **Startup**: In this case, the temporal context is open at the beginning of the run or when the event processing agent is defined.
2. **Event**: The initiator for the temporal context is a specific event.
3. **Absolute time**: This defines the exact time that the temporal context is initiated.

Finally, the initiators employ a correlation policy, which dictates whether to open a new temporal context if another temporal context instance of this event processing agent is already open. The default is to not to initiate a new temporal context if another appropriate temporal context is already active.

Terminators belong to one of the following types:

1. **Event**: The terminator for the temporal context is a specific event.
2. **Relative time**: In this case, for the temporal context to terminate, a predefined time interval from the initiation of the temporal context has to pass.
3. **Absolute time**: This defines the exact time that the temporal context will terminate.
4. **Never ends**: The temporal context will remain open for the duration of the run. This is the default option when no other terminators have been specified for the temporal context.

The terminators specify a quantifier parameter which defines if the first, last or every temporal context is terminated and in case of a terminating event, there may be conditions attached to the termination. Their activation is defined according to their order in the temporal context definition. When a possible terminator event instance is detected by the Proton engine, one or more temporal context instances of the same temporal context are terminated or discarded. In case of normal termination, an event-processing agent can still derive events, while in case of discard the event instances that have accumulated during this temporal context are discarded, and no detection can occur for this temporal context instance.

– **Segmentation contexts**: Events that refer to the same entity are grouped by a semantic equivalent defined by a segmentation context, according to a set of attributes. A segmentation context value can be either an attribute or an expression based on some attribute values of a certain event.

  - **Composite complex**: A composite context groups one or more contexts.

- **Event Processing Agents**: The event-processing agents are nodes in the directed graph of the event-processing network and have the following properties:

  1. **Name**: A unique string that identifies the agent.
  2. **EPA(Operator) Type**: This property defines the event detection pattern. There are several operator types, each with its own set of properties and operands.

     (a) The **Basic (Filter)** operator is stateless and does not correlate between its participant events and its function is to detect patterns if the incoming events pass a threshold condition.

     (b) The **Join** operator where the pattern is detected if all its listed participant events arrive either in any order (All operator) or in the exact order of the operands (Sequence operator).

     (c) The **Absence** operator where the pattern is detected if none of the listed events have arrived during the context.

     (d) The **Aggregation** operator. An Aggregate EPA is a transformation EPA that takes as input a collection of events and computes values by applying functions over the input events. These computed values can be used in the EPA condition and in its derived events.

     (e) The **Trend** operator. Patterns that track the value of a specific attribute over time are called Trend patterns. This operator is used to detect increment, decrement, or stable patterns among a series of input events and operates only on a single event type. It can be used to detect trends among a minimum specified number of event instances.

     The timing that patterns are detected and reported is defined by the **evaluation policy** of the EPA. In the **immediate mode**, these happen immediately as long as the conditions of the pattern composition are satisfied. The alternative is called **deferred mode**, in which the patterns are detected at the end of the context. The policy is defined in cases where the all, sequence, aggregation and trend operators are involved. The number of times a pattern is allowed to be detected in a context is defined in the **cardinality policy**. The value of the policy attribute can be either **single**, where the pattern should be detected only once or **unrestricted**, where there is no limit in the pattern calculation as long as its conditions are satisfied. The policy is defined in cases where the all, sequence, aggregation and trend operators are involved.

3. **Participant Events (Operands)**: These are the input events to the EPA and contain several properties, such as name (and alias), condition and consumption. The **Condition** property is used for event filtering, ignoring the events that participate in the EPA if they do not satisfy the condition. When the operator type is join, aggregation or selection, the **Consumption** property defines the condition for events to be reused later in the same pattern.

4. **Segmentation Contexts**: Semantically related events are involved in segmentation contexts employed by the EPA. These events are partitioned according to the values of the attributes (or expression) defined by the segmentation context and for each partition the detection process is performed separately. Each segmentation context in the EPA must have a segmentation context segment for every EPA operand and it defines matching only between the EPA's operands.

5. **Composite Contexts**: Several context instances can be open in parallel by an EPA with a composite context. A composite context instance is open if all the contexts listed in the composite context are matched. In the case where the composite context contains a segmentation context, this segmentation context should be defined over all the event initiators and event terminators of the temporal contexts of this composite context.

6. **Derived Events**: These are events that are a composition of events or other derived events, or content filtering on events, or both. They can be generated by an EPA after pattern detection and must be specified in the EPA definition, along with their properties (name, condition and expressions). The name property corresponds to an already defined event, while the condition property dictates the circumstances for event derivation, if present (the default is to

derive the event). Expressions are defined for each derived event's attribute and provide information on how to calculate the attribute value. The derived event instances have the same characteristics as an input event, with user-defined and built-in attributes which are defined in a similar way. The derived events are re-entered as input events to the system.

EPNs are completely defined in JSON files. These files are created by the Proton Authoring tool and include definitions for event types, action types, EPAs, contexts, producers and consumers. The basic format of the input JSON file is as follows:

```
“epn”: {
        “events”: [ ],
        “epas”: [ ],
        “contexts”: {
             “temporal”: [ ],
             “segmentation”: [ ],
             “composite”: [ ]
         }
}
```

The goal of this system is to provide support for a programmer to develop, deploy and maintain proactive event-driven applications by responding to raw events and identifying meaningful events within contexts. This means that Proton is able to compute and emit derived events by applying patterns defined within a context on received raw events. Proton receives information on the occurence of events from event producers, detects situations (conditions based on series of events that have occured within a dynamic time window), and outputs the detected situations to external consumers. The Proton architecture includes a run-time engine, producer and consumer adapters and an authoring tool (see Figure 3.3).

The main functional components of the Proton architecture and the interaction among them are presented below:

- **Adapters**: User-defined producers and concumers for event data are translated into input and output adapters in Proton execution time and represent the communication of Proton with external systems.
- **Parallelizing agent-contexts queues**: Used for parallelization of processing of single or multiple event instances.
- **Context service**: For managing of context's lifecycle -initiation of new context partitions, termination of partitions based on events/timers, segmenting incoming events into context groups which should be processed together.
- **EPA manager**: For managing Event Processing Agent (EPA) instances per context partition, managing its state, pattern matching and complex event derivation based on that state.

3.2.2 Proton on Storm

The Proton on Storm architecture is similar to the Proton standalone architecture, including the queues, the context service and the EPA manager. These logical components are implemented using Storm primitives such as spouts and bolts. The Proton on Storm topology is shown in Figure 3.4. The Routing bolt has multiple independent parallel instances running and determines the routing metadata of an incoming event. Afterwards, it adds the agent name and the context name to the event tuple and sends the tuple the next logical component which is the Context bolt. The Context bolt processes the tuple, adds more metadata and sends the tuple to the EPA manager bolt. The EPA manager bolt performs pattern matching and if it detects a derived event it sends it back into the Routing bolt.

3.3 Redis

Redis[1] (REmote DIctionary Server) is a NoSQL fast in-memory key-value store which is used when performance and scaling are important. As the keys can contain strings, lists, sets, hashes and other data structures, Redis is also known as a data structure server.

[1] https://redis.io/

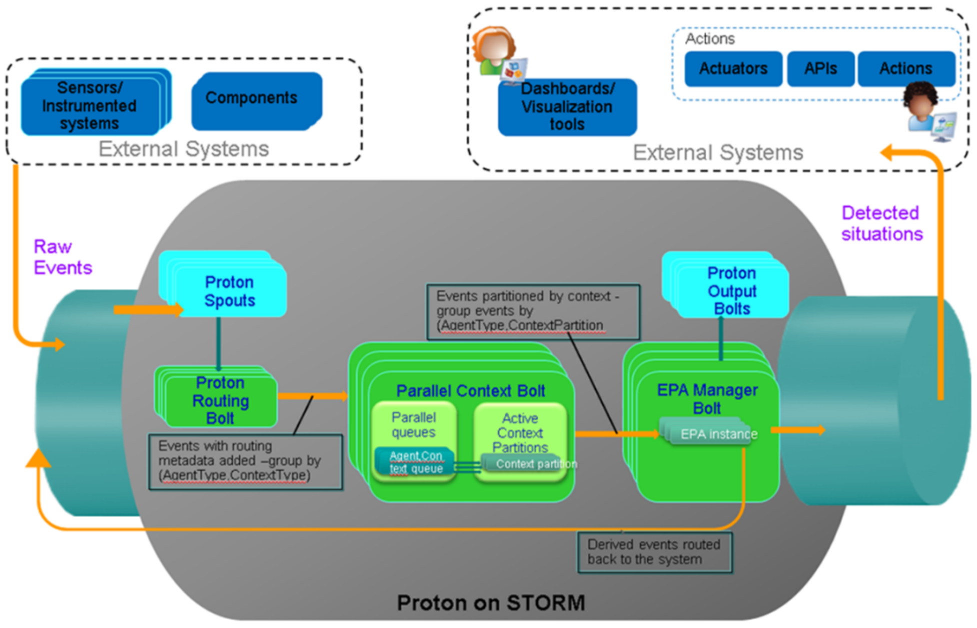
Sensors/ Instrumented systems
Components
External Systems
Dashboards/ Visualization tools
Actions
Actuators
APIs
Actions
External Systems
Raw Events
Detected situations
Proton Spouts
Proton Routing Bolt
Events with routing metadata added –group by (AgentType,ContextType)
Events partitioned by context - group events by (AgentType,ContextPartition
Parallel Context Bolt
Parallel queues
Agent,Context queue
Active Context Partitions
Context partition
Proton Output Bolts
EPA Manager Bolt
EPA instance
Derived events routed back to the system
Proton on STORM

Figure 3.4: Architecture of Proton on Storm.

Redis is used in this work for its implementation of the publish/subscribe messaging paradigm. In it, the components that are registered as publishers send messages to assigned channels without knowing who their receiver will be. On the other hand, the subscribers express interest in a number of channels and receive messages through them without knowing who sent them. This paradigm is suitable for inter-node communication, since the publishers do not need to be aware of the nodes they are sending their mesages to and the number of nodes throughout the run of the system may change. In this work, Redis subscriber queues are used in the entry points of each node to receive broadcasted information from the node network and process it accordingly.

Chapter 4

System Architecture and Setup

4.1 Network Architecture

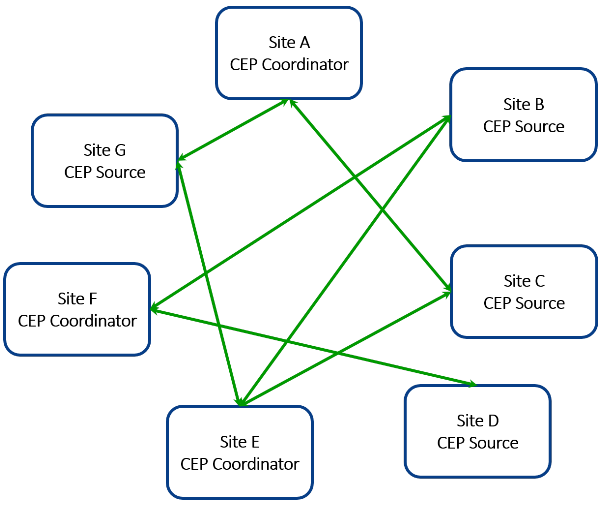

Figure 4.1: Architecture example with multi-star micro-coordinators.

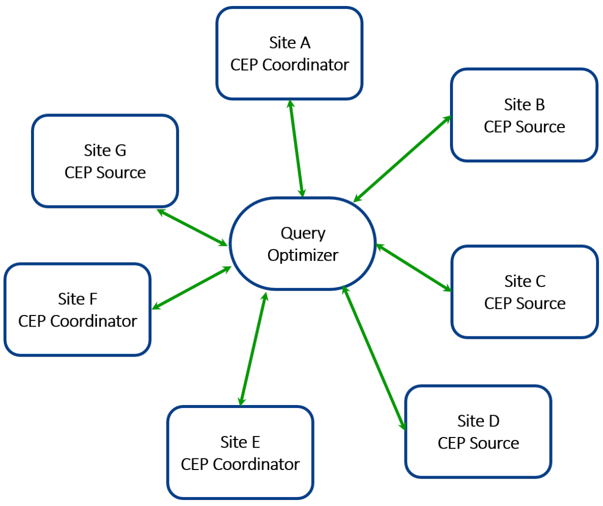

Figure 4.2: Query Optimizer's Place in the Architecture.

This work attempts to develop an architecture that is able to support in-situ processing of distributed, continuous data streams in a flexible scalable and communication efficient way. In this section, the basic components of this architecture will be presented.

The distributed environment of the architecture comprises a number of potentially geographically dispersed data collection entities, termed sites (also known as network nodes or just nodes) that participate in complex event processing. All the sites constitute a streaming cloud platform with an installed topology as described in section 4.2. One of the sites, however, has an additional functionality enabling it to act as a query Optimizer. This query Optimizer is an essential part of the architecture, as it enables the generation of optimal plans that efficiently balance communication cost and complex event detection latency. Each site of the network is responsible for monitoring its own stream of events and the Optimizer is responsible for providing plans that manage inter-site communication for the timely detection of complex events. In essence, it breaks down a global Complex Event Processing Expression to sub-queries that are distributed to local sites and define the Complex Event Processing Expression to be monitored at each local site.

The query Optimizer is considered as a distinct, daemon-like entity in the architecture. Despite the fact that the Optimizer resides in a specific site of the distributed network, its inclusion in this site does not affect the remaining fuctionality of that site. Therefore, this allows, for conceptual reasons, to consider the Optimizer cut off the underlying distributed architecture and be treated as a distinct object (not as part of a specific site) that can interact with all the sites. This distinction can be seen in Figure 4.2.

Additionally, in order to facilitate the in-situ processing, the Optimizer selects a number of sites that act as CEP coordinator sites (namely micro-coordinators or mini-coordinators). These sites are responsible for answering specific parts of a query and for supporting all necessary communication for detecting complex events regardless of the source that generated them. Any of the existing sites can take the role of micro-coordinator and there may be many micro-coordinators inside the network. The CEP coordinators are able to communicate (link) with every source site that is needed for plan execution. Each micro-coordinator essentially is the central node of a star network topology, therefore the whole architecture constitutes a multi-star topology. In Figure 4.1, such an architectural scheme can be seen, where there are three micro-coordinators (marked as “CEP Coordinator”) that are linked with every site (marked as “CEP Source”) that is needed for plan execution. It is important to mention that each site may simultaneously be a source site and a CEP micro-coordinator, depending on the Optimizer’s planning.

A micro-coordinator is, in essence, a site where operators upon event types are installed. This means that it expects the other sites to send it event types that are relevant i.e., events involved in the installed operator. The best communication paradigm to use for this purpose is called push/pull [9] and the Optimizer produces plans that take full advantage of it. In this paradigm, the Optimizer instructs the micro-coordinators, which in turn instruct the source sites, which instances of event types to send immediately (push mode events) and for which event types communication should be postponed (pull request). The push/pull paradigm is efficient because if the micro-coordinator instructed the rest of the sites to communicate all events involved in the installed operator when they occurred, it would greatly increase the communication cost. Furthermore, if the micro-coordinator postponed the transmission of an event tuple to the micro-coordinator, it would delay the evaluation of the installed operator(s), increasing a potential event's detection latency.

Another key element of the architecture is the use of communication efficient distributed methods for monitoring global functions by a partitioning of the global functions to distributed local functions that communicate only if needed. This is achieved using geometric monitoring techniques (described in section 2.2) generally used in detecting events, which are emitted when a function, computed over the data of different distributed nodes, has crossed a specific global threshold. For this purpose, all the sites have been designed to support generic geometric monitoring tasks, i.e. each monitoring task is defined using a vector of statistics derived at the sites, a function to be applied on the average of these vectors and a threshold value.

4.2 Intra-Site Architecture

The architecture's intra-site processing is built on top of a streaming cloud platform, namely Apache Storm. Each site runs its own Storm instance, including its own Zookeeper and Nimbus nodes. A Storm site instance runs both the Proton CEP and a block of components that are involved in the complex event processing and implement the push/pull paradigm and the geometric monitoring operators. Since it was important to keep the architecture as generic as possible, whether a site is also a micro-coordinator or not does not affect the composition of its Storm components. However, the Geometric Monitoring (GM) operator is more complex than the other supported operators, therefore it requires a few additional components for its implementation.

The intra-site architecture is described in this section. Each site's Storm topology contains spouts, which read data from files or receive events from other sites and several different bolts, as illustrated in figure 4.3. A site's Storm topology comprises the following components:

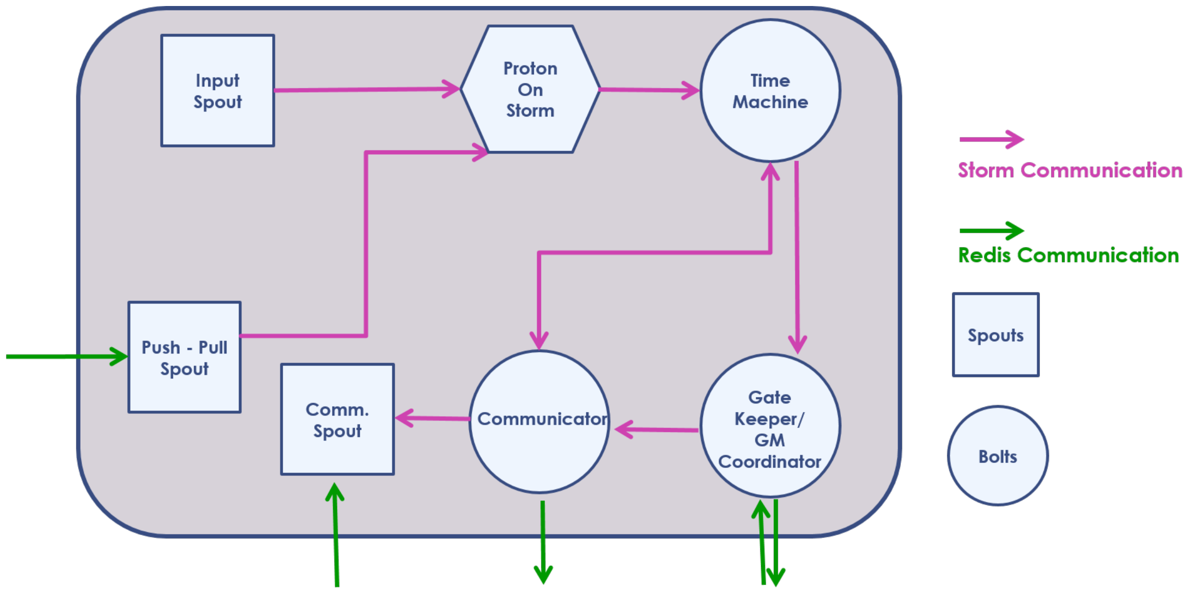

Figure 4.3: Intra-site Architecture.

4.2.0.1 Input Spout

The Input spout is the intra-site architecture's event entry point. Its task is to parse event definitions generated from remote data sources, create the corresponding primitive events and propagate them to the Proton CEP engine.

4.2.0.2 TimeMachine Bolt

The TimeMachine bolt acts as a buffer component for the architecture that stores three different kinds of information:

- Events received from the CEP engine. These events are potentially requested by the CEP micro-coordinator sites and this component's task is to forward them to the requested sites.

- lsv values that are produced from the GateKeeper. When a GM operator is deployed in the system, the TimeMachine is required to buffer the lsv values for each monitoring function.

- Statistics. The TimeMachine stores some statistics that the Optimizer requires for producing new more optimal plans for the sites.

Additionally, the TimeMachine maintains a state for each monitoring function that holds one of two values, either "pause" or "play", which enables it to pause and replay the data that feed the GateKeeper in case of a violation occurrence.

4.2.0.3 GateKeeper Bolt

The GateKeeper bolt is a component that is used only in cases where the GM operator is involved. This component is responsible for detecting local violations and initiating local violation resolution procedures. Practically, the role of this bolt is to investigate whether the monitoring function at the local node is above or below the local threshold previously received from the GM micro-coordinator.

The events that the GateKeeper receives come from the TimeMachine bolt. The GateKeeper's task is to verify that the monitoring function computed on the event data does not cross the threshold value and as long as the local threshold has not been passed, no output is produced. If the received values surpass the threshold, the GateKeeper reports the lsv value to the Communicator bolt, which proceeds to negotiate with the GM micro-coordinator.

The GM operator is placed at a specific node (GM micro-coordinator). In this node, the role of the GateKeeper bolt is twofold. Apart from the previously described functionality, this bolt is also responsible for resolving local violations and detecting when they result in a global constraint violation. Additionally, as time passes, the coordinator part of the GateKeeper bolt adjusts the local threshold for each node. The nodes in which local violations are more frequently detected, receive a higher threshold value, which generally reduces the number of local violations for these nodes and the communication load.

4.2.0.4 Communicator Bolt

The Communicator bolt is the only component of the intra-site architecture with the ability to send messages to other nodes. To achieve that, it includes a Redis sender. Redis is the tool in the architecture that is used for inter-site communication.

The first responsibility of the Communicator bolt is to interact with the GM micro-coordinator, i.e. to resolve local violations and to participate in the resolution of violations that originated at other nodes. When the Communicator is invoked, either due to a local violation reported by the GateKeeper, or due to a lsv request sent by the Coordinator, it sets the TimeMachine to the "pause" state. Once the violation recovery procedure has been completed, the Communicator changes the state of the TimeMachine to the "play" state and the GateKeeper continues to receive aggregate update events starting at the value held by the time pointer at the TimeMachine when it first transitioned to the "pause" state.

Due to its unique ability to communicate with the other sites of the network, the Communicator's second responsibility is to request events from other nodes that are relevant to the requesting node's installed operator. Furthermore, it propagate events that are required from other nodes in order to execute query plans and in order to manage overlapping pull requests for the same event types, it is equipped a structure reserving the pull requests.

4.2.0.5 PushAndPull Spout

The PushAndPull spout is the component that receives pushed events from other sites (through Redis) and feeds them to the Proton CEP engine.

4.2.0.6 RedisPubSub Spout

The RedisPubSub spout (also known as Communicator spout) is the main listener component of the inter-site communication. It receives messages from different sources and sends them through Storm to the appropriate bolts for further processing. The RedisPubSub spout is the component that receives the initialization messages coming from the query Optimizer which include all the necessary information for the initialization of the node. Once the Communicator spout receives this information, it forwards it to all the components that need it for their initialization. Furthermore, in GM source nodes, this spout also listens to all the messages coming from the GM micro-coordinator, i.e. LSV requests, violation resolutions and new threshold values. If a node is also a GM micro-coordinator site, then its RedisPubSub spout registers an additional Redis listener in order to receive the messages destined for the GM micro-coordinator, i.e. local violation reports and LSV values. These messages are then forwarded to the coordinator part of the GateKeeper bolt. Finally, this spout is the component that receives event requests from other nodes sent from their Communicator bolt, concerning the push/pull paradigm, as described in section 6.2.

4.2.0.7 Routing Bolt

The Routing bolt is the first component of the Proton CEP engine developed in IBM Haifa and is responsible for routing all events to the CEP processing bolts. All the events either primitive coming from sources, derived coming from the EPAManager bolt or pushed events coming from other sites pass through this bolt. The Routing bolt has been upgraded to support the push/pull functionality, the collection of statistics and query plan adaptivity.

4.2.0.8 Context Bolt

The Context bolt implements the context service functional component of Proton. It receives input from the Routing bolt and is responsible for managing of context's lifestyle.

4.2.0.9 EPAManager Bolt

The EPAManager bolt is the third and last component of the CEP topology. This bolt, after receiving events from the Context bolt, performs pattern matching and if it detects a derived event it routes it back to the Routing bolt.

4.3 System setup

In the previous section, the architecture and system components have been described. The setup of the network and the sites' role (source or coordinating sites) in it are defined by the Optimizer, which supervises the whole network and is described in the following sections.

4.3.1 Optimizer

The Optimizer has its own Apache Storm topology that consists of the Optimizer spout and the Optimizer bolt, as illustrated in Figure 4.4. The basic focus of the Optimizer is to provide a set of optimal plans and subsequently supervise the plans' performance. The query Optimizer needs two basic sets of information as input parameters in order to execute the plan generation algorithms; the network parameters and the set of queries. Those two pieces of information, which is a priori knowledge of the network, is fed into the query Optimizer through external communication and read by the Optimizer spout. The external communication can take whichever form is suitable (e.g., Sockets, REST communication, etc.). Once this information is received by the Optimizer spout, it is sent through Storm messages to the Optimizer bolt of the query Optimizer where all the algorithms for plan generation are executed. In the next section, the parameters required from the Optimizer for network setup are described.

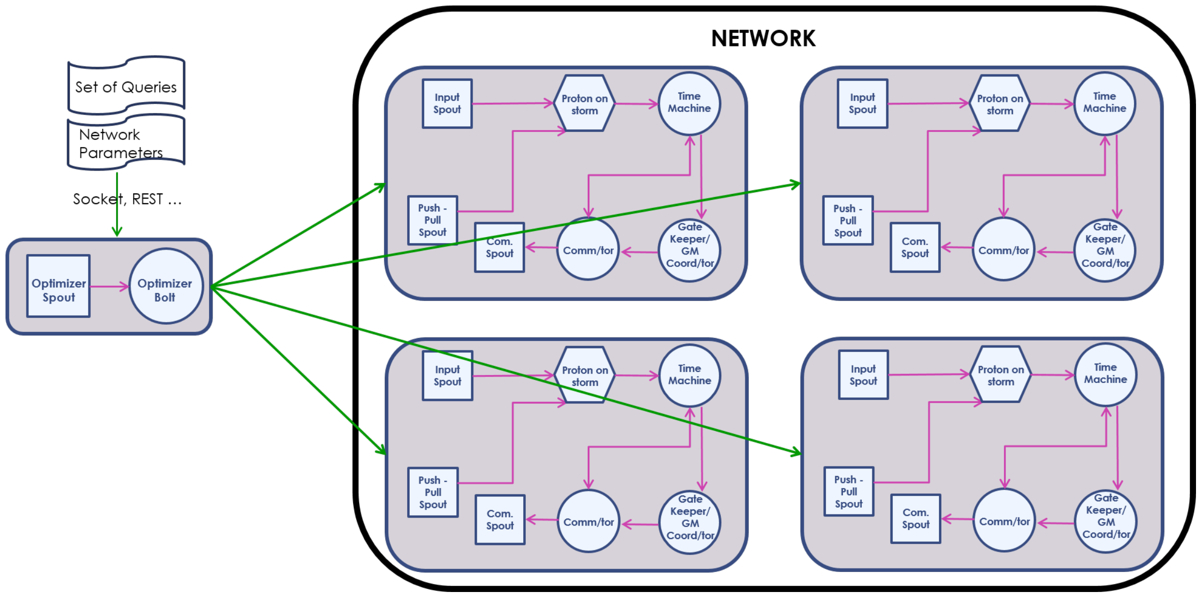

Figure 4.4: Query Optimizer's Work Flow.

4.3.2 Optimizer Input Parameters

The Optimizer input parameters can be the result of two different programs running outside of the desired network and are sent to the Optimizer spout with the aforementioned external communication method. The Optimizer expects all parameters to have been received before the plan generation process begins, namely the network parameters which give an overall picture of the network distributed sites, as well as their input event streams and additionally the set of queries that the user needs to impose upon the event streams. At this point, the Optimizer can create an outline of the network (a network graph) and an outline of the query set (an event detection graph), two necessary structures for the plan generation process.

4.3.2.1 Network Parameters

The network parameters are divided into two sections. The first section includes information concerning each site and its connectivity with all other sites (measured in latency) and provides an overview of the underlying distributed network. The second section contains the type of the incoming events for each site along with their respective frequency and provides a first view of event distribution across the distributed sources. Although both communication latencies and event frequencies may vary or change significantly over time, it is necessary for the plan generation algorithms to have a first view of the parameters upon which they should optimize performance.

The network parameters file is a csv file with one line per site and the following format:
`siteName` ; *nameOfTheSite* ; `links` ; *anotherSiteName* ; *interSiteLatency* ; `events` ; *eventName* ; *evenNameFrequency.*
The keywords `sitename,` `links` and `events` cannot be altered whereas the rest are variables and are set accordingly. For the purposes of demonstration, the following csv file will produce the simple star-topology network depicted in Figure 4.5:
Network parameters csv example file:
siteName ; site1 ; events ; CallPOPDWH ; 0.04;
siteName ; site2 ; events ; CallPOPDWH ; 0.09;
siteName ; site3 ; events ; CallPOPDWH ; 0.09;
siteName ; site4 ; events ; CallPOPDWH ; 0.15;

siteName ; cent1 ; links ; site4 ; 14 ; site3 ; 13; site2 ; 12 ; site1 ; 11 ; events ; CallPOPDWH ; 0.1;

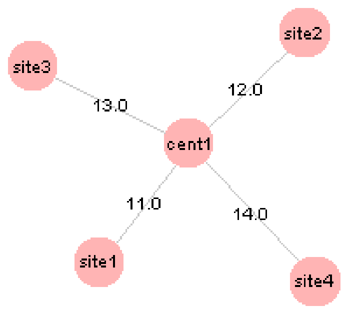

Figure 4.5: Network Graph Example.

4.3.2.2 Set of Queries

The set of queries is typically a JSON file. This input JSON file contains the description of an EPN (Event Processing Network), which consists of all the submitted queries in the form of EPAs (Event Processing Agents). Each EPA has its own input and output (derived) events as well as the derivation logic (meaning the operator and the temporal and/or segmentation context of derivation). The events (raw and derived) are specified along with every attribute they carry. All the existing contexts in the various EPA's are also specified in the input JSON file as well as the optimization parameters based on which the plan generation algorithms will run. The basic format of the input JSON file is as follows:

```
“epn”: {
        “optimization”: { },
        “events”: [ ],
        “epas”: [ ],
        “contexts”: {
            “temporal”: [ ],
            “segmentation”: [ ],
            “composite”: [ ]
        }
}
```

4.3.3 Plan Generation

The plan generation process is the query Optimizer’s main task. More specifically, the Optimizer creates a globally optimal (given the optimization objective) distributed plan for answering the given queries. This globally optimal plan is then broken down to each site’s plan, subsequently transformed to JSON files ready to be shipped to each site for system setup. Additionally, if a GM operator exists within the input EPN, it is transformed to a configuration file which is also sent to each site. The main work of the Optimizer is finalized with the shipment of an EPN JSON file for each site along with the GM configuration files (if a GM operator is specified) through Redis communication channels.

4.4 Dynamic Node Manipulation

The capability to extend the network of nodes with new sites has been implemented. Furthermore, the system has been built to handle node or communication failures. It covers the case where a node has been requested from other remote nodes to send some events and due to network transient failure it is unable to do so for a time.

4.4.1 Registering New Nodes

Laboratory setups assume that the number of sites participating in the monitoring task is fixed and known in advance. However, the system enables the extension of the network by adding new nodes. To achieve this, the Optimizer spout is always listening for input concerning the insertion of new nodes to the network. This input is an extension of the network parameters file as described in section 4.3.2.1. It contains all the required information regarding the new node in relation to the existing nodes. An example network parameters file for a new node named cent3 is shown below:

siteName ; cent3 ; links ; cent1; 25 ; events; CallPOPDWH; 0.0009 ;
The file is then sent to the Optimizer bolt. The Optimizer bolt, after receiving this file and producing new plans, it sends them to the whole network for setup as described in section 6.1.

4.4.2 Managing Communication Failures

Since the Communicator component is able to recognize if communication issues exists due to Redis being used for inter-site communication, it has the capability to delay sending requested events until the problem has been fixed. It includes a queue structure that buffers the postponed events and their timestamp of the moment they would have been sent if there was no communication problem. When the communication issue has been resolved, the Communicator empties its queue of events and sends them to the requested nodes in the order they were stored. The messages that carry the events include the time delay that occurred due to the communication failure so that the receiving nodes can be made aware that there was a delay for these events.

Chapter 5

Geometric Monitoring Implementation

This chapter describes in more detail the geometric approach for function monitoring over a distributed system. The first three sections describe the implementation details about handling the monitoring functions. The first section describes the monitoring functions, their class hierarchy and their methods and attributes. The second section presents the methodology for detecting local violations. Finally, the third section describes the ability of the system to dynamically add new functions and select which function to monitor at runtime.

In the fourth section of this chapter, the node's setup for supporting the GM operator is described. Finally, in the last section the GM operator work flow is presented through an example, which poses a distributed counter problem that aims to detect all the subscribers of a cellular network that have made more than a given amount of calls in the last n hours. The method used by the TimeMachine bolt which handles the time delays, as well as the two implemented methods that allow the GM Coordinator to distribute the global threshold in a dynamic way are described at the end of this chapter.

5.1 Function Hierarchy

To allow the dynamic addition and monitoring of new functions, the function class hierarchy has been designed appropriately, focusing on facilitating the extension of the

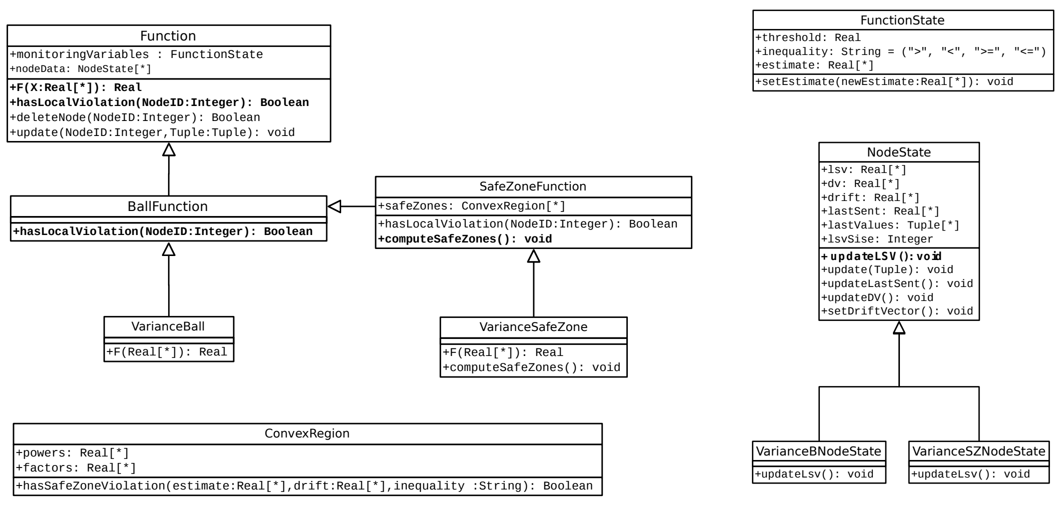

Figure 5.1: Function Class Hierarchy.

available implemented function classes. The UML class diagram of figure 5.1 shows the implemented hierarchy.

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 “$>$”, “$<$”, “$\leq$”, “$\geq$”. 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 the architecture, either at the most general `Function` class, or at the two abstract classes `BallFunction` and `SafeZoneFunction`.

The most general Class (`Function`) 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 the following information for the node:

- 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, 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.

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. 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.

Any function that wants to use the original technique with the spheres simply needs to:

1. Create a subclass of `BallFunction` that provides the code for the `F` method
2. Create a subclass of `NodeState` that provides the code for the `updateLSV` method

The `SafeZoneFunction` class inherits the `BallFunction` class to accommodate the case where a function is defined 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. A safe zone 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:

1. Create a subclass of `SafeZoneFunction` and provide the code for the `F` method
2. Create a subclass of `NodeState` that provides the code for the `updateLSV` method
3. 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.

5.1.1 Implemented Sample Functions

5.1.1.1 Average Ball Function

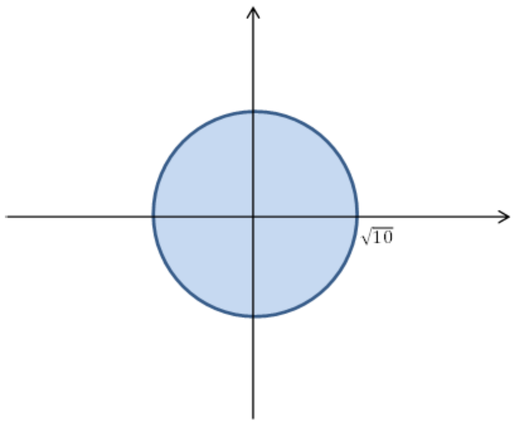

Figure 5.2: Graph of function Average (Threshold = 100)

This is a subclass of Ball Function, in which the objective is to determine whether a two-dimensional estimate vector has an L2 norm lower/greater than a given threshold. In this function, the LSV structure stores the last two values that have been received.

The index (0 or 1) of LSV where the most recent value is stored is alternating between the two indices. For example, the first value received is stored in LSV[0], the second in LSV[1], the third in LSV[0], the fourth in LSV[1] and so on. This function is defined as:

$$LSV[0] * LSV[0] + LSV[1] * LSV[1].$$

Practically, this function tries to determine if the actual average of LSV is inside the sphere with radius $\sqrt{threshold}$. An example graph of the average function is shown in Figure 5.2

5.1.1.2 Average Safe Zone Function

This is a subclass of Safe Zone Function. The only difference from Average Ball Function is the way local violations are detected. This class implements the *computeSafeZones()* method and uses the computed safe zones to investigate for local violations. The implementations of *updateLSV()* and *f()* remain the same as in Average Ball Function.

5.1.1.3 Variance Ball Function

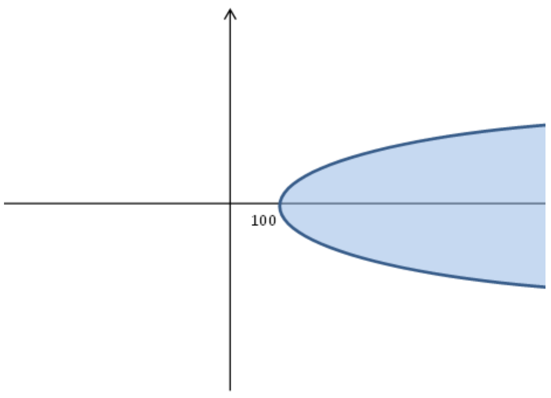

Figure 5.3: Graph of function Variance (Threshold = 100)

This is a subclass of Ball Function, In this function, the LSV structure stores the square of the last value that has been received for each Node ID (LSV[0]) and the last value itself (LSV[1]). This function is defined as:

$$LSV[0] - LSV[1] * LSV[1].$$

An example graph of the variance function is shown in Figure 5.3

5.1.1.4 Variance Safe Zone Function

This is a subclass of Safe Zone Function. Similar to the Average Safe Zone Function, this class implements *updateLSV()*, *f()* and *computeSafeZones()* to detect local violations for the variance function using the safe zone technique.

5.2 Local Violation Detection

Two methods are used to detect a local violation. In both cases, a threshold T is defined.

5.2.1 Ball Technique

In this method, a ball is constructed which is centered at:

$$\frac{estimate + driftVector}{2}$$

has a radius of:

$$\left\| \frac{estimate - driftVector}{2} \right\|$$

and a grid is applied inside the ball. The local constraint on each node is set as follows: each node investigates if the value of the monitored function on the estimate vector and the value of the monitored function on all the ball's points, at that time, are on the same side of the threshold. Depending on whether it is considered a local violation if the function's value is equal to the threshold, there are two cases: In the first case, the function value's equality with the threshold is classified in the region above the threshold. Then, a local violation occurs if the following condition is true:

$$\textit{if}((F(estimate) < T \textbf{ and } F(all\ ball\text{'}s\ point) \geq T) \textbf{ or }$$
$$(F(estimate) \geq T \textbf{ and } F(all\ ball\text{'}s\ point) < T))$$

In the second case, the function value's equality with the threshold is classified in the region below the threshold and a local violation occurs if the following condition is true:

$$\textit{if}((F(estimate) \leq T \textbf{ and } F(all\ ball\text{'}s\ point) > T) \textbf{ or }$$
$$(F(estimate) > T \textbf{ and } F(all\ ball\text{'}s\ point) \leq T))$$

If there is at least one ball's point that violates this constraint then the node detects a local violation and emits a notification to the GM Coordinator. In the following figures it is investigated if there are local violations for the average function. The first two cases (Figures 5.4 and 5.5) show two local violations in which it is apparent that the ball (sphere) is not monochromatic. While the function's value on the estimate vector (f(e)) is below the threshold T, there are some ball points for which the function is over the threshold. In Figures 5.6 and 5.7 there is no local violation because f(e) and the function's value on all the ball's points are on the same side of T.

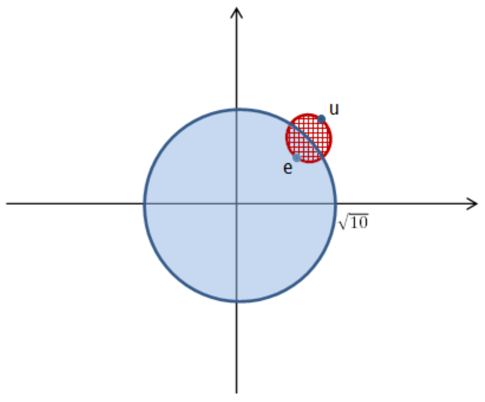

Figure 5.4: Local Violation in Average Function - Ball Technique

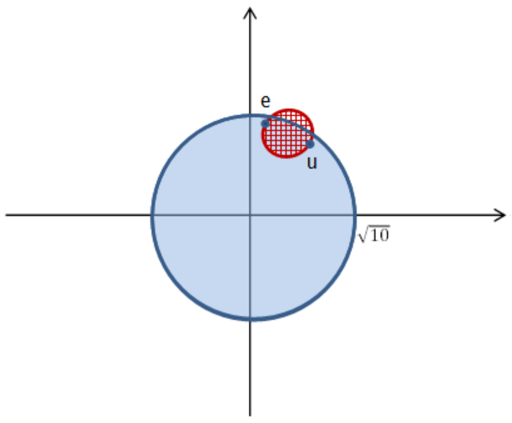

Figure 5.5: Local Violation in Average Function - Ball Technique

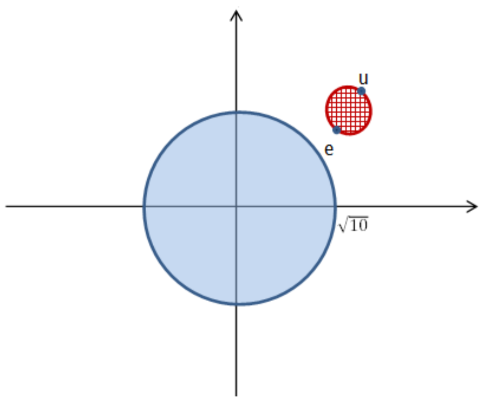

Figure 5.6: No Local Violation in Average Function - Ball Technique

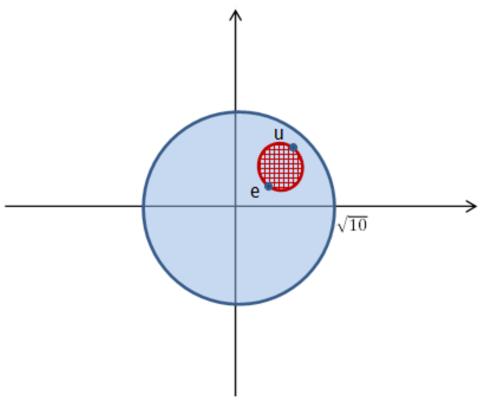

Figure 5.7: No Local Violation in Average Function - Ball Technique

5.2.2 Safe Zone Technique

In this method, safe zones induced by the combination of the monitored function and the threshold value are defined. The area where $f(v) < T$ is denoted as the admissible region, while the area where $f(v) \geq T$ is denoted as the inadmissible region. If $f(estimate)$ lies in the admissible/inadmissible region and that region is convex, then this entire area can be used as a safe zone. Otherwise, one can define a proper subset of the admissible/inadmissible region, covering the location of the estimate vector, that is convex and utilize it as a safe zone. If $f(estimate)$ and $f(driftVector)$ lie on the same side of the safe zone, then there is no local violation. If there aren't any defined safe zones, then the ball technique is used instead.

For the above functions, when $f(estimate)$ lies inside a function area, the function area itself is considered a safe zone (Figures 5.8, 5.9, 5.12, 5.13). In the case where the estimate vector is outside the function area, the safe zones are defined separately for each function. In the average function, the safe zone is defined as the plane perpendicular to the estimate vector that passes through the point $\sqrt{\frac{T*estimte}{||estimate||}}$ (Figures 5.10 and 5.11). On the other hand, in the variance function the safe zone is defined as the plane perpendicular to the x axis that passes through the threshold (The "$Var(X+a) = Var(X)$" property of the variance function allows moving the estimate vector to always have a zero mean after a synchronization, in this way it is only needed to check if LSV[0] has increased by a given max amount) (Figures 5.15 and 5.14).

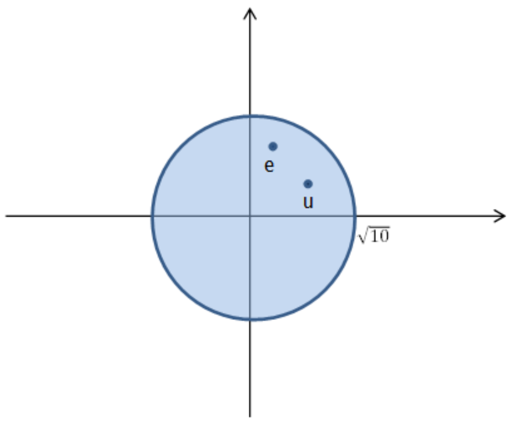

Figure 5.8: No Local Violation in Average Function - Safe Zone Technique.

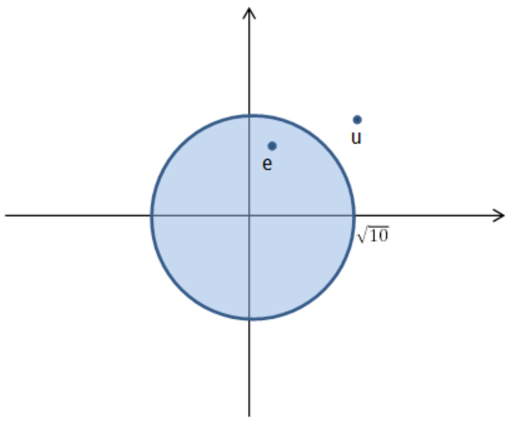

Figure 5.9: Local Violation in Average Function - Safe Zone Technique.

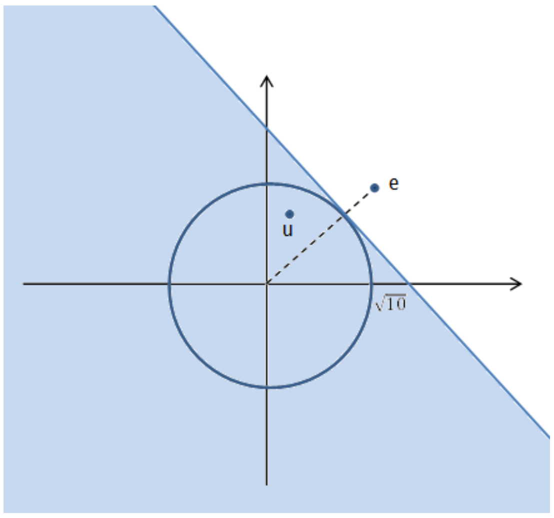

Figure 5.10: Local Violation in Average Function - Safe Zone Technique.

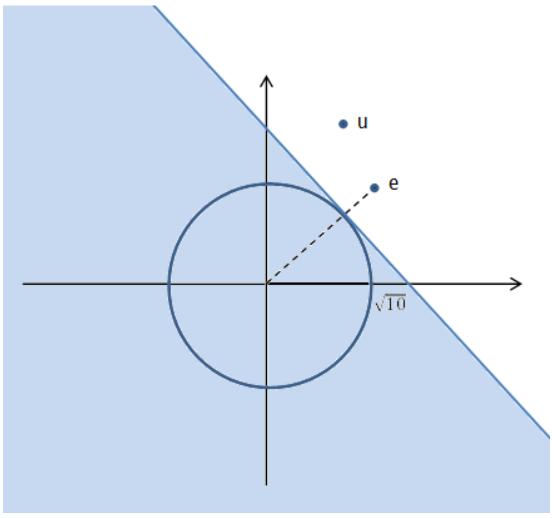

Figure 5.11: No Local Violation in Average Function - Safe Zone Technique.

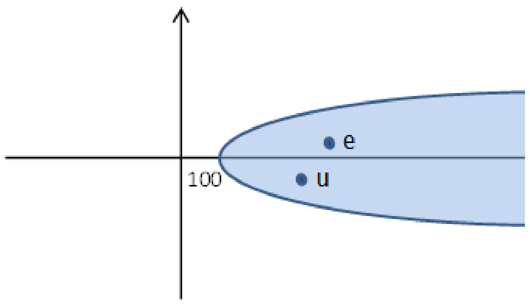

Figure 5.12: No Local Violation in Variance Function - Safe Zone Technique.

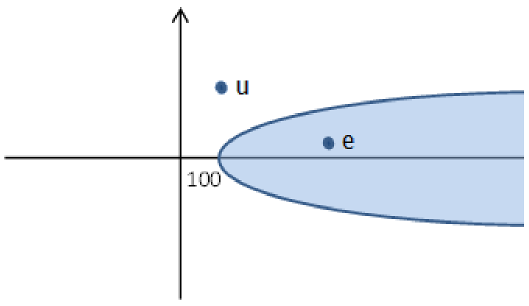

Figure 5.13: Local Violation in Variance Function - Safe Zone Technique.

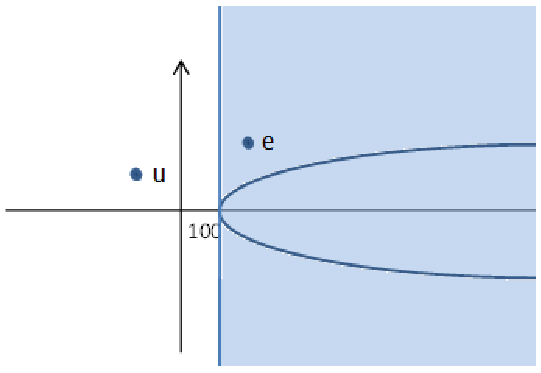

Figure 5.14: No Local Violation in Variance Function - Safe Zone Technique.

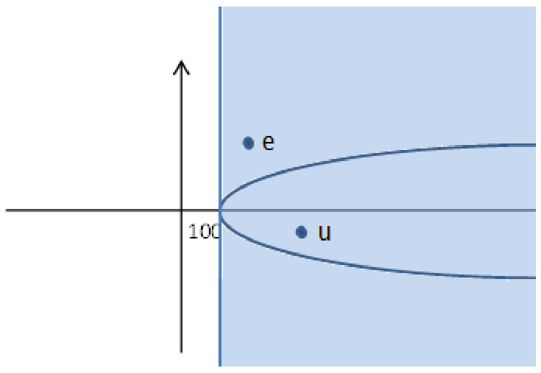

Figure 5.15: Local Violation in Variance Function - Safe Zone Technique.

5.3 Supporting Extensibility to New Monitored Functions

Since one of the goals of this system is to support the detection of distributed events, it was important not to limit the monitoring capabilities of the system to just a set of functions installed during the system deployment, but also to enable the monitoring of an unlimited number of functions. The previous sections described two of those functions, the average function and the variance function. This section will describe and explains the methodology deployed in this work regarding the dynamic loading of .jar files for any new monitored functions from the bolts involved in the distributed event detection process.
To add a new monitoring function to the system, an implementer needs to do the following:

1. **Implement a subclass of Function**: The system provides the `Function` abstract class and uses its abstract methods to perform monitoring. The already implemented Average and Variance functions derive from Function. The bolts keep a hash map of monitoring functions which are all subclasses of Function. This means that if such subclasses of Function are loaded by the system, they can be used and be monitored normally.

2. **Create a .jar file containing the implemented class**: After implementing the subclass(es), the implementer must package them into a .jar file so that they can be loaded dynamically.

3. **Distribute the .jar file to all machines involved**: The new .jar file needs to be accessible by the system, which means that all the machines are required to have access to the function implementation inside the .jar. To achieve this, the implementer needs to put the file in an accessible location which has to be defined and be known in advance so that all the machines will have access to it.

In case a bolt is instructed to monitor a function, it would first check for the required .jar file at a local directory and, if not found there, would then search for it at the global specified location. This process allows the system to continuously add new monitoring functions without having to restart or affect the deployed topology.

5.4 Geometric Monitoring Operator Initialization

The Geometric Monitoring Operator initialization requires the communication of four different components, the Optimizer, the RedisPubSubSpout, the GateKeeper and the TimeMachine. In order, the responsibility of each component is:

- The **Optimizer** sends the configuration file to all the sites that participate in the network. This file includes all the necessary information for setting up the geometric operators that may be involved in the system.
  The parameters included in the GM configuration file are listed below:
  - **globalThreshold**: The global threshold value, which is split into local thresholds, one for each node, such that their sum is equal to the global threshold.
  - **numBillingNodes**: The number of sites that participate in the function monitoring procedure.
  - **equalityInAboveThresholdRegion**: The inequality parameter described in section 5.1
  - **defLSVValue**: When the GM coordinator requests from the site to send their LSV values, it is possible that some sites have not yet received any monitoring data and for this reason they need an default lsv value to transmit to the coordinator. The most common default lsv value is zero.
  - **dynamicThreshold**: It is a boolean variable that determines the threshold distribution strategy between the sites. The two implemented strategies are described in section 5.5.1.2.
  - **functionName**: The name of the monitoring function.
  - **functionLocation**: The location of the implementation of the monitoring function. It can be either a local path or a globally accessible path (e.g., network drive, distributed file system).
  - **isCoordinator**: This parameter determines whether the local site that receives the configuration file has the additional role and functionalities of the GM micro-coordinator. If the parameter is false, then the site acts only as a GM source site.

- The **RedisPubSubSpout** receives the configuration file and forwards it to the other three bolts. The parameters that each bolt requires are different.
- The **GateKeeper** receives information about the monitoring function, the initial threshold value and the default local statistics vector (lsv) value to be used initially since the monitoring is performed at each site. During event processing, the GateKeeper produces additional lsv values that are then sent to the TimeMachine. In the coordinating part of the GateKeeper (GM coordinator), the additional information required is the number of sites that are monitoring the GM and, optionally, if the dynamic threshold functionality will be used so that threshold splitting among the source sites will be performed.
- The **TimeMachine** buffers the received lsv values and is also responsible for pausing and replaying the data in case of local violation occurrence. For this reason it requires information regarding the monitoring function in order to set the relevant structures that buffer the lsv values.

More details about the functionality and work flow of the GM operator will be presented in section 5.5.

5.5 Geometric Monitoring Operator Work Flow

The Geometric Monitoring (GM) operator is different from ordinary complex event queries and requires an elaborate work flow, which will be described in this section. The Optimizer creates geometric monitoring plans and sends them to the GM micro-coordinator sites, which have the GM operator installed. Even though the sites are designated as GM micro-coordinator sites, they still function as source sites. The basic site components (bolts) that participate in the plan are the GateKeeper, the TimeMachine and the Communicator. The typical geometric monitoring work flow, as dictated by the plans, is divided into two steps:

- In the first step, the designated sites continuously monitor the assigned thresholds and detect local violations. When local violations occur, the local statistics vectors of the sites are sent to the coordinating site.

- The second step deals with local violation resolution in the event of violation detection. If it is determined that the local violation is also global, a complex event that designated this fact is generated.

The next section presents a specific example of geometric monitoring operator work flow and describes the messages and events that are propagated throughout the system and the whole procedure in more detail.

5.5.1 The Mobile Fraud Example

The GM monitoring operator work flow example that is going to be investigated in this section concerns the detection of mobile network subscribers with an abnormally high rate of outgoing calls. The example is as follows. A mobile network provider has set up a network of mobile cell towers (i.e. the nodes/sites in the system). The objective for this example is to monitor the number of calls that a customer has made within a predefined time interval, in order to detect potential fraudulent usage, which may be the case if counting the number of calls yields a number that exceeds a given threshold. Due to the fact that customers of the mobile network move around, their mobile phones are connected to different cell towers according to their location. A first simple approach to determine the total number of calls for each customer during the specified time window would be to calculate the call counters of all cell towers to a single central site. However, centralizing the counters requires a network with a prohibitively large communication capacity due to the substantial amount of phones, rendering the simple approach infeasible. The best solution is to apply the previously described in-situ processing semantics which allow for multiple local conditions to be monitored individually at each site. Using this approach, as long as the local constraints are satisfied, no communication is necessary, as all global counts are below the threshold for the monitored time interval and can be omitted. Only in the case where a local threshold is violated an alarm is raised, starting a resolution protocol. This strategy reduces the need for constant communication and eases the requirements on communication capacity.

In more detail, the strategy employs the following methodology:
In order to detect subscribers with an abnormally high rate of outcoming calls, the Optimizer generates a GM configuration file that contains the parameters describing the GM operator. An example of the configuration file is presented below:

```
globalThreshold=50
numBillingNodes=5
equalityInAboveThresholdRegion=false
defLSVValue=0.0f
dynamicThreshold=false
functionName=eu.ferari.examples.distributedcount.function.IdentityFunction
functionLocation=D:/jarakia/IdentityFunction.jar
isCoordinator=false
monitoringObject=LocalCounter
```

According to the information in the configuration file example, the global site threshold is equal to 50 and the number of sites is 5, which means that each site is initially assigned a local threshold of 10. The parameter equalityInAboveThresholdRegion is set to false, meaning that if the local call count is equal to the threshold (10), then no local violation occurs. There is a local violation only when more than 10 calls are counted.

After the initialization phase is finished and all the components have been set up by receiving the parameters they require, the sites can start processing events. In the mobile fraud example, the input data is generated from the interaction of a user's cell phone with the mobile network.

1. The Input spout is fed with the call data and generates events (callData events) which represent these calls. These events are forwarded to the ProtonOnStorm which generates a counter for each phone number, representing the number of events for that client in the time window. These counters are referred to as LocalCounter events. The ProtonOnStorm sends these local counters through CounterUpdate messages to the TimeMachine. Unless the corresponding counter for a phone in the TimeMachine is flagged as paused, its value is then propagated to the GateKeeper as shown in Figure 5.16.

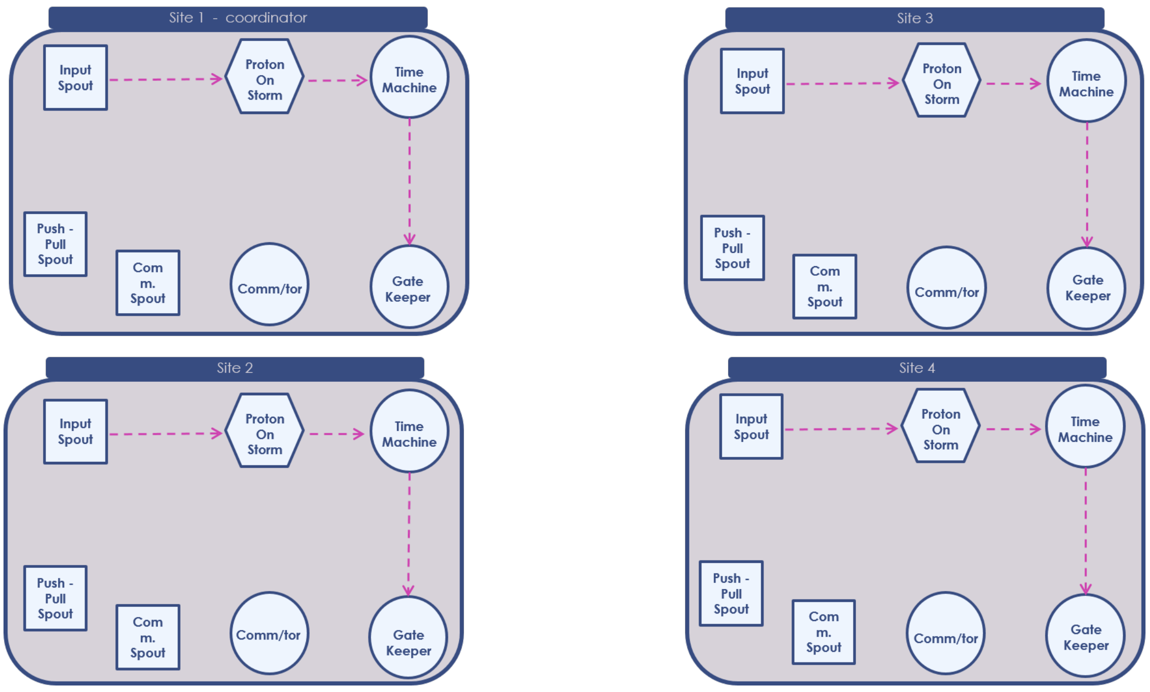

Figure 5.16: Monitoring object generation from input events.

2. The GateKeeper bolt is responsible for investigating if any of the phones has surpassed the number of calls that has been set as the local threshold of the site. For this purpose, the geometric monitoring function that is monitored in this example is the identity function, which has been implemented as a subclass of the `BallFunction` class described in section 5. In this example, at some point one of the sites has detected a local violation for phone 1 (see Figure 5.17). In this case, the GateKeeper passes a message to the Communicator to inform it about the local violation occurence.

   Once the Communicator receives the information about the local violation detection from the GateKeeper bolt, it creates a pause message and forwards it to the TimeMachine so that it pauses event emission for the threshold-violating phone - in this case phone 1. The TimeMachine event emission to the GateKeeper is paused until the violation has been resolved.

3. The TimeMachine, after receiving the pause message from the Communicator and

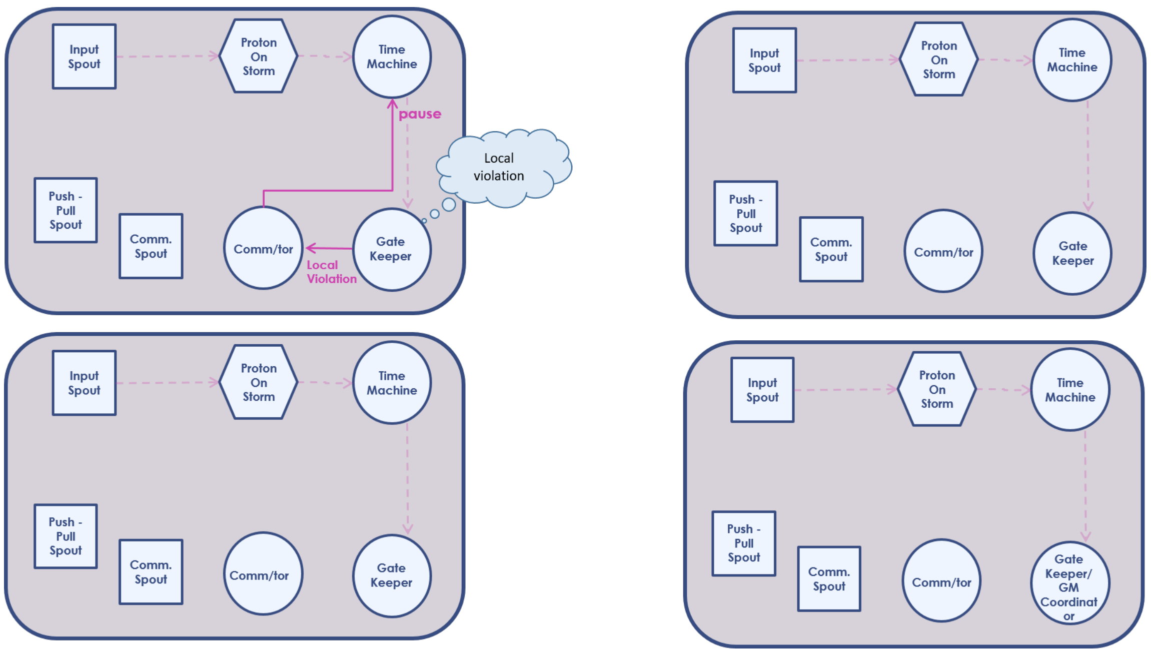

Figure 5.17: Local Violation at Source Site.

pausing the state of phone 1, it sends a pause confirmation message back to the Communicator which contains the lsv value for the counter that caused the local violation, the timestamp that represents the time that this event has occurred and the node in which the violation has occurred. The Communicator, in turn, sends the lsv value that it received from the TimeMachine to The GM Coordinator, informing it about the local violation occurence (see Figure 5.18).

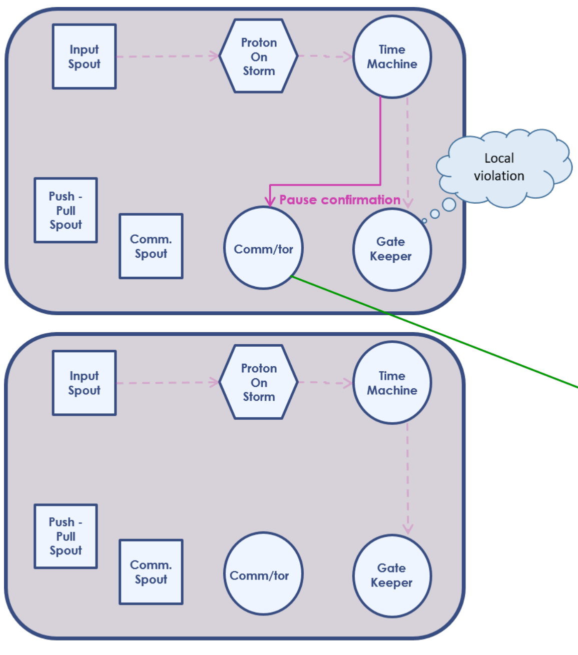

Figure 5.18: The beginning of violation resolution.

4. After the Coordinator is informed about the local violation occurrence of mobile phone 1, the second phase of the geometric monitoring functionality starts. The Coordinator sends a lsv request message to all the sites except from the one that caused the violation and asks for their lsv values for phone 1, at the timewindow that the violation occurred. This message arrives at the Communicator spout (RedisPubSubSpout) of the sites and is forwarded to the Communicator bolt of each of these sites. The Communicator bolt of each site that receives this message, after receiving it, informs the TimeMachine about the local violation for phone 1 by sending a pause message as illustrated in Figure 5.19.

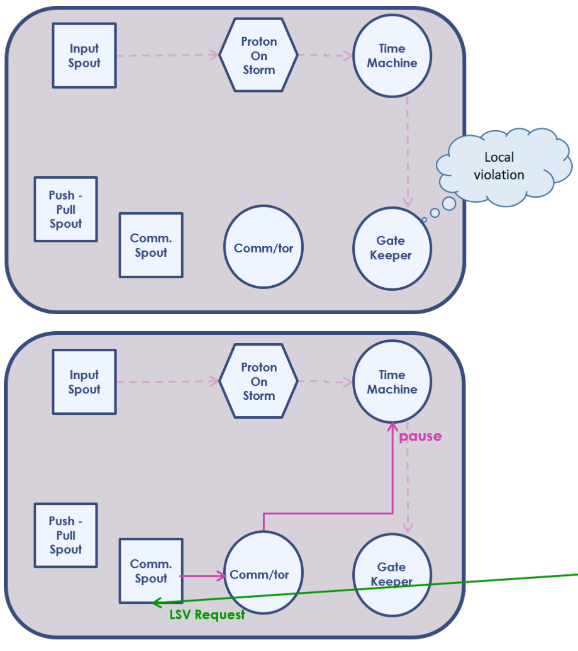

Figure 5.19: Lsv pull request from GM micro-coordinator.

5. When the TimeMachine receives the pause message from the Communicator, it flags the state of phone 1 as "pause" and stops the event emission for this phone. Afterwards, it creates a pause confirmation message that contains the lsv value for phone 1 and for the time window in which the violation occurred and sends it to the Communicator. The Communicator, in turn, sends the lsv that it just received to the coordinating site (see Figure 5.20).

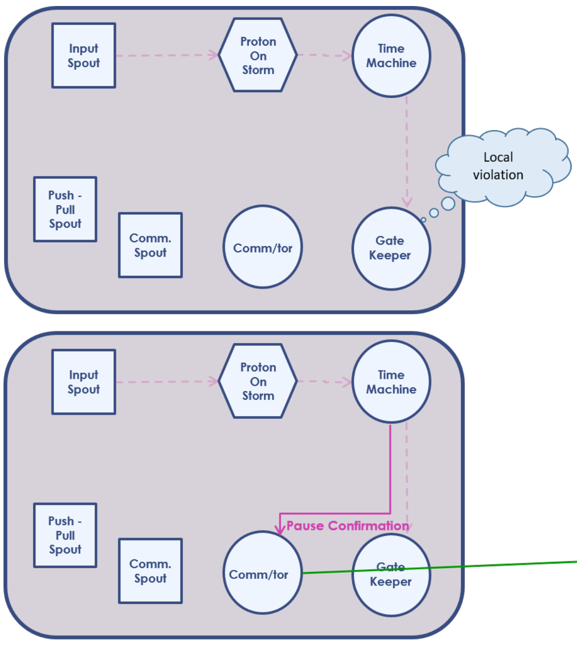

Figure 5.20: Push requested lsvs back to the micro-coordinator.

6. The TimeMachine, after receiving the lsv values from all the sites, it handles the monitoring violation resolution and after calculating a new estimate, it sends it back to them. The sites receive the new estimate through their Communicator spout, which forwards it to the GateKeeper of the site (see Figure 5.21).

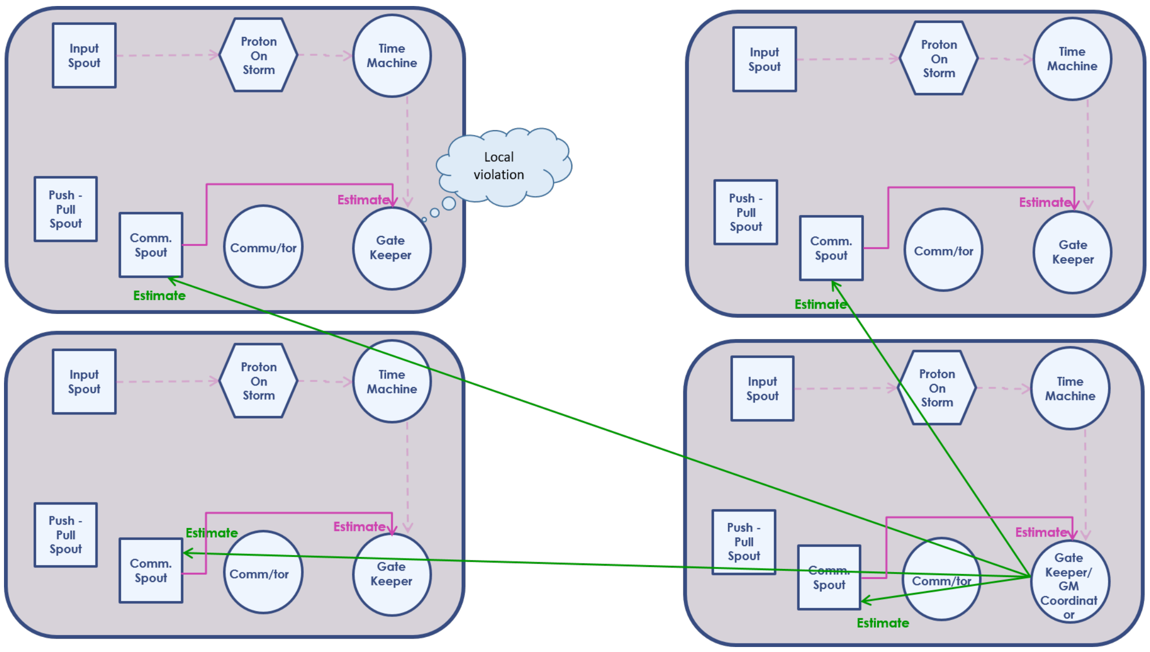

Figure 5.21: Calculate new estimate and send it to source sites.

7. After the violation resolution is finished, the Coordinator calculates new threshold values for each site, which are distributed to the GateKeeper and Communicator bolt of the sites through their Communicator spout. The reception of the new threshold value in the Communicator bolt marks the violation resolution. The Communicator at this point informs the TimeMachine with a play message, so that it will set phone 1 to the “play” state and resume event emission for that phone.

5.5.1.1 Time Machine Mechanism Handling Time Delays

To ensure the correctness of monitoring tasks even through processing and communication delays, the TimeMachine component contains a mechanism for handling these irregularities. The problem can be described with an example. Assuming that at a given time t_1 node n_1 detects a local constraint violation, the Coordinator may be forced to request the local statistics vector value from another node n_2. This request is communicated through the network and there may be communication delays. If the time t_2

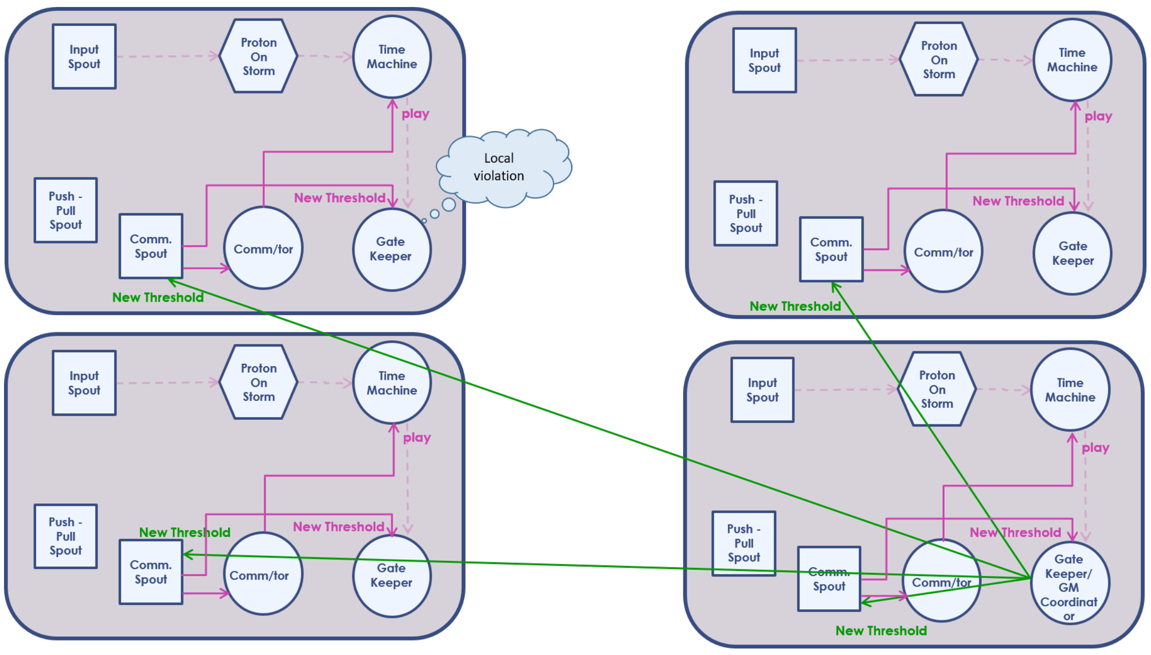

Figure 5.22: Calculate new Threshold and send it to source sites.

that node n_2 receives the request has a significant difference from t_1 due to processing and communication delays, then the value of the local statistics vector that n_2 currently has may be different than the value that it had at time t_1, which it needs to send back. Moreover, after the initial n_1 local constraint violation resolution, it is possible that the local constraints at n_2 have changed. If that is the case, the node n_2 will have to resume all processing activity starting from time t_1. The TimeMachine has been appropriately equipped to handle these cases, even through race conditions that may occur.

For the functionality to work, there are two assumptions. First, the clocks of all the nodes are considered synchronized. Second, there is a maximum time delay for each message, denoted by Δ and measured in seconds. Since the objective is to detect local constraint violations considering that all the events are given timestamps and are processed centrally, the nodes report to the Coordinator both about the occurence of violation events and the exact time they happened. The Coordinator then specifies the time for which it needs the local statistics vector values from the nodes. This time can be as early as 2Δ seconds before the current time. For the nodes to be able to provide the correct value, they store in their TimeMachine component the local statistics vector values of the last 2Δ seconds. In addition to this functionality, the TimeMachine, after violation resolution, is able to resume processing for the events that have occured right after the violation detection for the latest threshold value.

The Coordinator is able to distinguish between conflicting constraint violation occurrences between different nodes to avoid race conditions. To do that, it tries to verify that it is handling the earliest detected threshold crossing, ignoring later ones until the first has been resolved. Similarly, the nodes reply to local statistics vector requests made at the earliest time possible and cancel all later violation resolution processes.

These methods verify the correctness of the event processing procedure by ensuring that the earliest threshold crossings are handled first and all subsequent updates are processed according to the newly determined thresholds.

5.5.1.2 Dynamic Threshold Allocation

The standard threshold allocation procedure divides the global threshold into pieces equal to the number of nodes present in the system, so that their sum is equal to the global threshold. This is the initial default value of the local threshold for each node. After a successful violation occurence and resolution, nodes may receive different updated local threshold values. If the number of nodes is denoted by n, the global threshold by g and the local threshold at each node by d_i, then initially ($d_i = \frac{g}{n}$).

The coordinator is informed by a node about a threshold crossing for client j and receives the local statistics vector value of the node for this specific client. Then, it requests all the local statistics vectors from the other nodes for the same client. When it receives all this information, it calculates the global statistics vector for the client and the new threshold values for the nodes. The local statistics vector for the client j at node i is denoted by c_i^j, the global statistics vector for j is denoted by c^j (which is also the sum of all local statistics vectors) and the value δ^j is defined as $\delta^j = \frac{g-c^j}{n}$. The coordinator calculates the value $c_i^j + \delta^j$ as the new threshold for client j. The sum of the new local thresholds is equal to g. If the value for δ^j is negative (i.e., the global statistics vector for j has crossed the global threshold), then the nodes will notify the coordinator when the monitoring function values go below the new threshold, while if δ^j is positive, this happens when the values go above the new threshold.

The above method is the default implemented method for dynamic threshold allocation. To make better use of the locality of data (mobile calls serviced by cell towers) used in the mobile fraud use case, a different version has been implemented which may detect less overall local violations. A network of cell towers is considered where the objective is to monitor the number of calls for each customer in a specific time period. All the cell towers receive an initial threshold value of zero instead of dividing the global threshold value equally among all the sites. The global threshold is distributed only among the towers that service a specific client's calls. The reason is that, in most cases, a customer will use his/her cell phone around only a few specific cell towers (sites) and most other towers of the network will never service calls for this customer. For each client that performs the first call serviced by a site, a local violation will be detected because the site's initial threshold value is zero. After the violation resolution, the new global threshold value will be shared equally between all the sites that have detected at least one call for this client.

In a more specific example, a network of 4 towers is considered and the global threshold value is equal to 100 (the global threshold value), as depicted in Figure 5.23. A user performs a phone call around cell tower 1 and a local violation is detected, since the local threshold at this site is equal to zero. After the violation has been resolved, tower 1 receives a new threshold value for this customer, which is equal to 100, since tower 1 is the only tower that has detected calls from this client so far (see Figure 5.24). After a while, the user changes location and performs a phone call which is detected by tower 2. Due to tower 2 having a threshold value of zero at this point, a local violation occurs. After its resolution, since this client has made phone calls around towers 1 and 2, the global threshold value is distributed evenly among those two sites, setting their local threshold value equal to 50 each for this client (see Figure 5.25).

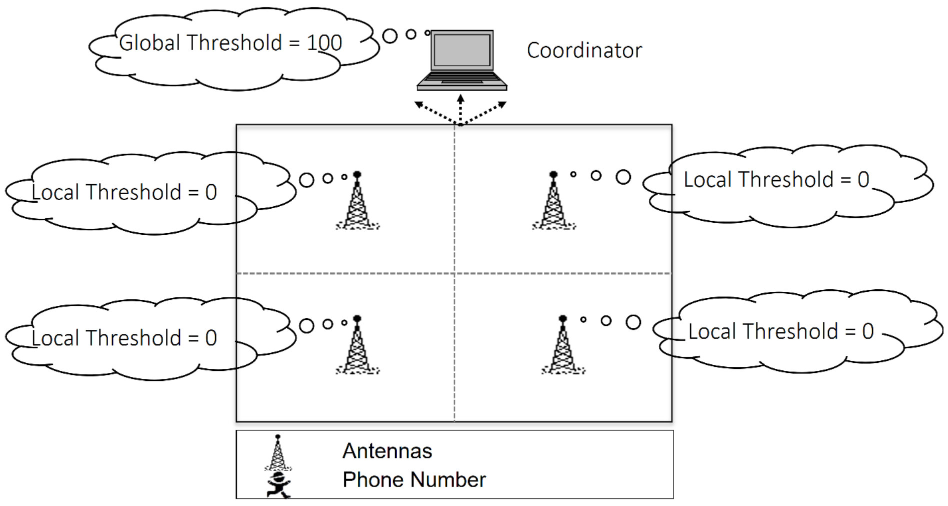

Figure 5.23: Initial Threshold Distribution.

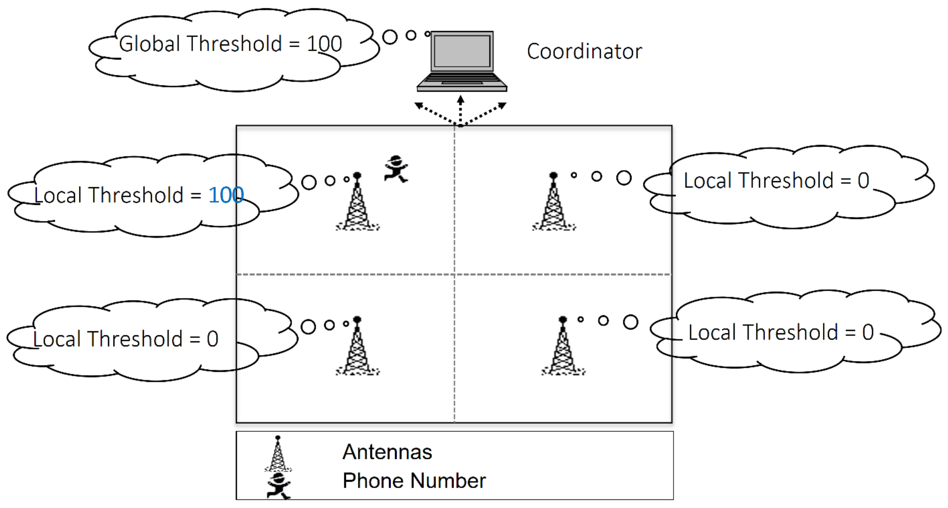

Figure 5.24: Threshold Distribution after the First Local Violation.

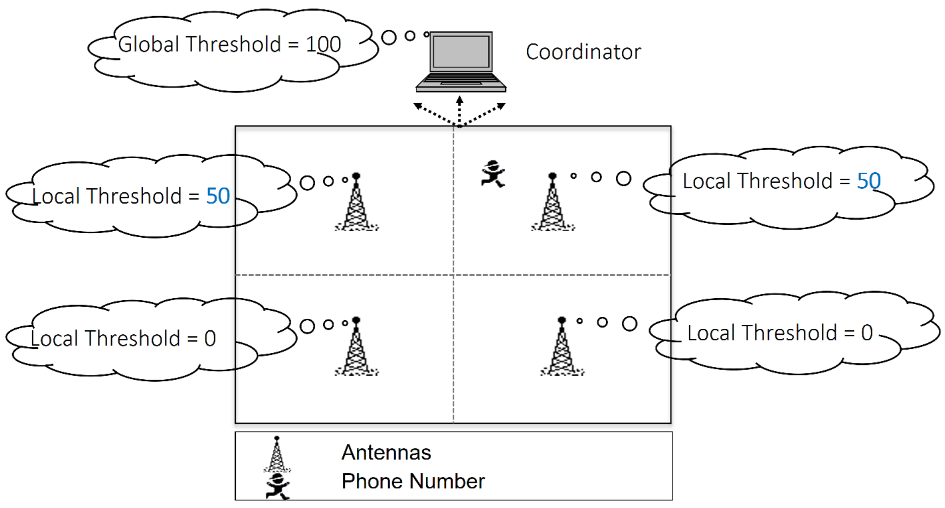

Figure 5.25: Threshold Distribution among Two Sites.

Chapter 6

Complex Event Detection Implementation

This chapter will describe the ordinary complex event processing work flow, including the initial system setup. Moreover, the system capability to adapt the CEP plans is presented.

6.1 Topology Setup

As described in chapter 4.2, each site's Storm topology contains a RedisPubSubSpout spout, which is responsible for receiving the necessary information from the query Optimizer and for shipping it to the rest of the topology's components in order for them to be set.

Once the RedisPubSubSpout spout receives the JSON file, it subsequently forwards it to the Proton on Storm module (CEP Engine) for setup. All three components of the CEP engine, the Routing bolt, the Context bolt and the EPAManager bolt require the JSON file for the initialization of their structures, since it contains all EPN definitions, including definitions for event types, EPAs and contexts.

Apart from the CEP engine, the module that receives as input the stream of events, namely the Input spout, must also receive the JSON file since the knowledge of the exact event definition is necessary for incoming event parsing. With the reception of the JSON file, each topology is ready to receive input events for query evaluation.

6.1.1 Push and Pull Paradigm Initialization

6.1.1.1 Pull Mode

The Optimizer plans form an NFA of multiple states in order to facilitate the monitoring of events in stages and thus avoid unnecessary communication. In this NFA, each state represents the monitoring of a set of events and transitions between states represent the monitoring of a new set of events. A transition occurs after the detection of all the involved events within the specified time window. This transition becomes possible with the split of the involved EPA into multiple connected EPAs with new intermediate derived events marking the state transitions. These events generate pull requests and contain an attribute called "`Intermediate`", which the Routing bolt looks for in order to be able to recognize them.

The Routing bolt, as mentioned in section 4.2.0.7 is a Proton on Storm component which has been modified to support the push/pull functionality. It includes a structure called `EventsToPull` that maps each intermediate event with a set of events to request. Upon detecting an intermediate derived event for the first time, the Routing bolt investigates which EPAs include this event as input and stores the remaining input event types for these EPAs in the `EventsToPull` structure. This way, for each incoming intermediate derived event, the events that need to be requested from other sites are known.

Each time the Routing bolt detects an intermediate derived event, it sends a pull request message to the Communicator, who is responsible for communicating with every involved site and pulling the requested events. The pull request message contains three types of information:

1. The event type that is in push mode.
2. The push receivers (micro-coordinators that need this event type for evaluation).
3. The window duration in which this event is in push mode.

6.1.1.2 Push Mode

The first state in a multi-state NFA formed by the query plan is always in push mode. For the later states of the NFA to be activated, all the events of the previous steps have to be detected within the specified time window. For the initialization of the push functionality, the event description of the events inside the JSON file includes an attribute called `PushToCoordinators`. The Optimizer fills in the `PushToCoordinators` attribute with a string of comma separated values which contain the site names that this particular event is going to be pushed to when detected. When the Routing bolt receives the JSON file during initialization, it sends a pull request message to the local node's Communicator bolt for each event type that contains this attribute. This pull request message is similar to the one described in the previous section, however, the window duration in which the event is in push mode is, by default, infinite.

6.2 Complex Event Processing Work Flow

In order to implement the push/pull functionality and for the system to work properly, the TimeMachine is required to store the events which come from the CEP module. The events with their detection timestamps are sorted and stored in buffers (hash map structures), one for each event type. The TimeMachine is additionally required to implement a time-based mechanism for recovering past events within a timespan and to ship events of interest to coordinating sites, as dictated by the push mode functionality. Pull requests are generated upon state transition of the NFA query plan from the Proton on Storm module and emitted to the Communicator. For forwarding the pull requests, the Communicator contains a structure reserving the pull requests in order to avoid sending time overlapping pull requests for the same event types. The Routing bolt is the most appropriate component for handling the pull requests, since it is the one that receives all the intermediate derived events inside the Proton on Storm topology. The Routing bolt can detect when a derived event that is also a state transition is generated by the CEP engine, so that it can inform the Communicator about the new pull request.

The push and pull work flow is described below.

1. The Input spout of each site receives the events as a stream and forwards them to the Routing bolt of the Proton on Storm CEP module for processing. After processing, the Routing bolt propagates them to the TimeMachine module for buffering purposes. This process is continuous and occurs in every site regardless of possibly assigned coordinating tasks as depicted in Figure 6.1 using dashed lines.

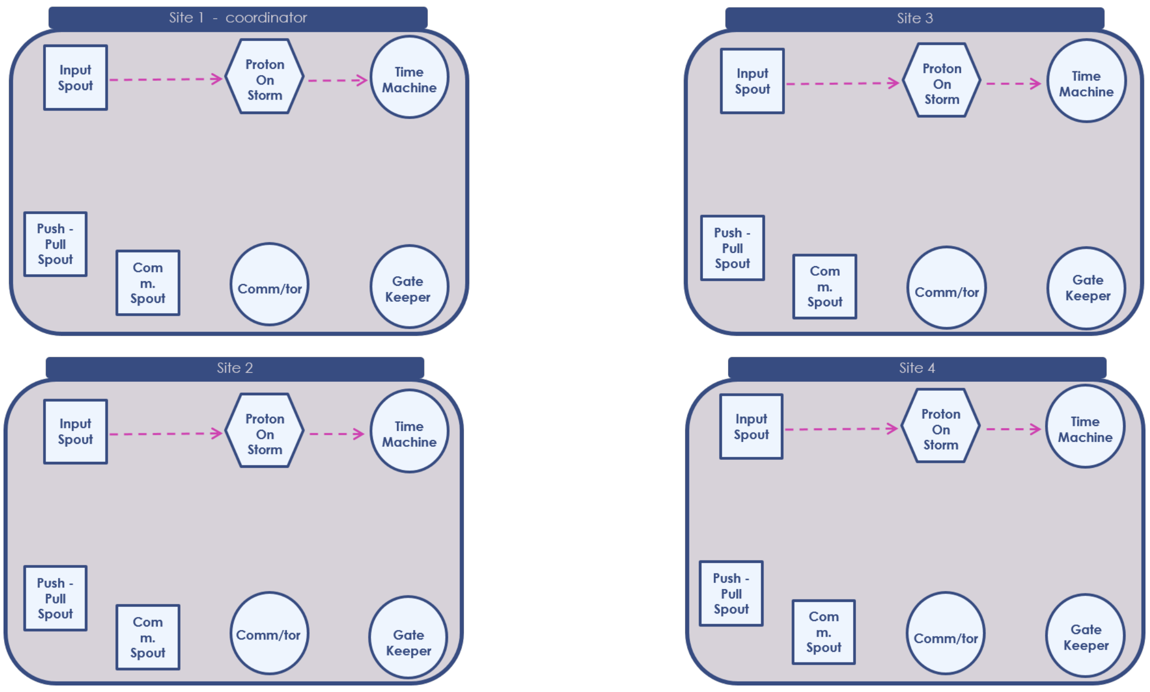

Figure 6.1: Continuous event processing.

2. The site which has been assigned as the coordinator executes the query plans it has received from the Optimizer. For plans that include more than one state, the first state is always in push mode (as described in section 6.1.1.2), while the rest are activated along with the full detection of the events involved in the previous states within the time window. State transitions in the NFA formed by the query plan generate pull requests, which the Routing bolt emits to the Communicator. The Communicator is then responsible for informing all sites (including itself) that produce this event type, that it is interested in such events in a given time window. Then the pull request is sent from the Communicator of the coordinating site to the Communicator of all sites that produce that type of event through their Communication spout (Figure 6.2). Upon arrival, the pull request is sent to the buffering module (i.e. the TimeMachine) which will switch to push mode for the requested events for the requested timespan.

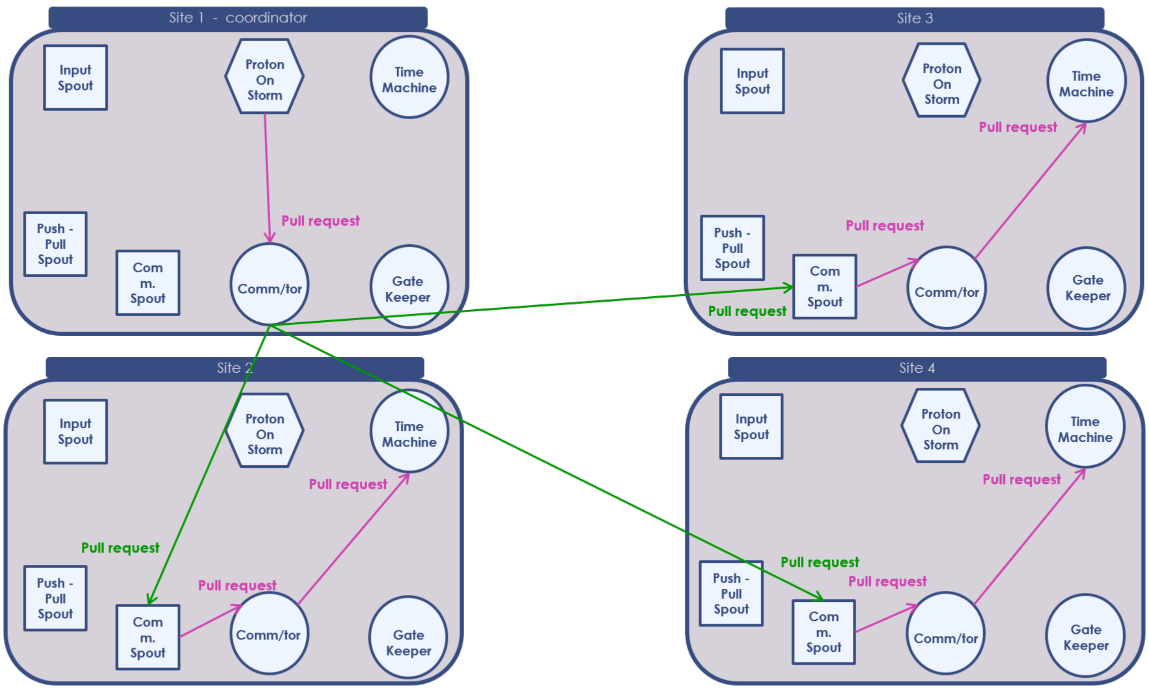

Figure 6.2: CEP micro-coordinator pull request.

3. Once in push mode, the TimeMachine checks whether the requested events have already occurred within the specified timespan and forwards them back to the co-ordinating site's PushAndPull spout through the source's Communicator (Figure 6.3). At the same time, the TimeMachine remains on push mode until the pull request's timespan expires and whenever an event of interest is detected, it is immediately forwarded back to the micro-coordinator.

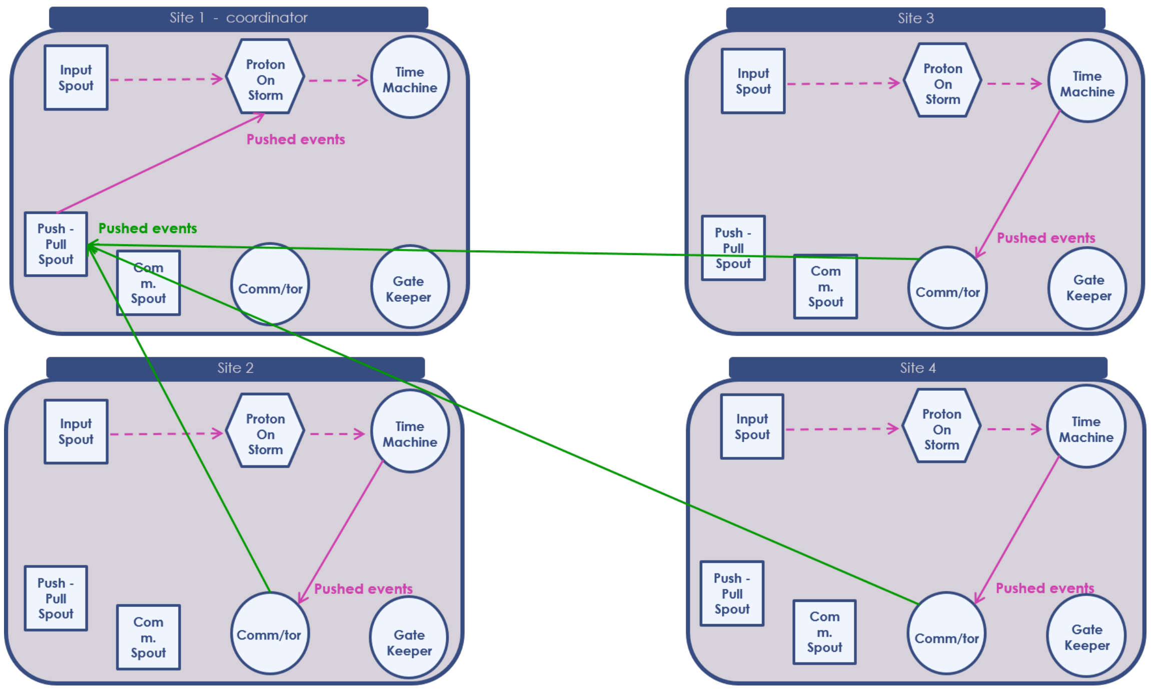

Figure 6.3: Push requested events from sources to micro-coordinator.

6.3 Architectural Injections for CEP adaptivity

As described above in section 4.3.2 the Optimizer requires information regarding the input type events and their frequency in each site as well as each site's connectivity with all other sites in order to generate the most optimal plan for each site. However, it is possible that the plans perform in a sub-optimal manner. The reason for the sub-optimal behaviour of the system may have two origins. The first one is if the latency for the inter-site communication has changed significantly and the second is if there is a drift in the event occurrence frequencies. In both cases, if the gain is significantly higher than the cost of switching the plans, the Optimizer must create new plans and ship them to the topologies.

6.3.1 Statistics Storage

In order for the Optimizer to know when and why to generate new, more optimal plans, the sites periodically collect statistics information and send it to the Optimizer. Each site's Routing bolt has been modified so that the site can collect the required statistics for more efficient plan generation. In this work, a simple event type counter statistic has been implemented, but more statistics can be easily integrated, even using elaborate structures. To support more complex statistics, the `Statistics` class has been made abstract and includes an abstract method called `updateStatistics` which must be implemented by the `Statistics` subclasses. These subclasses can be loaded either statically or dynamically. In the first case, the subclass is available to all nodes of the system before running, while in the second case, its implementation is loaded at runtime using a method similar to the one described in section 5.3.

The Routing bolt needs some information regarding the statistics gathering and forwarding process. This information comes from the Optimizer during topology setup phase through a configuration file that has the following form:

```
statisticsClass=com.ibm.hrl.proton.routing.CountStatistics
jarLocation=null
epoch=1
eType=events
```

The first two parameters represent the statistics class to use for collecting the statistics and location of the .jar file where the class implementation resides. The epoch parameter determines after how many eTypes the statistics will be sent to the Timemachine for buffering. The eType paramater can by default be either events or seconds, but Statistics subclasses can make use of other values.

The Routing bolt is the component in which the statistics are collected since it is the only component in the architecture that has access to all the events that pass through the system (raw, derived and pushed). Then, they are transmitted to the TimeMachine through Storm messages and they are buffered. Once the statistics gathering process has reached a certain amount of input, it is sent to the Optimizer so that it can evaluate the current plan's performance and decide on if new plans are required.

6.3.2 Proton Modifications

For the system to adapt to potential new plan generation, it is necessary to modify the Proton CEP engine. Since intermediate results may have been produced by the micro-coordinator, before the plan is adapted all old temporal contexts must have terminated in order to avoid missing the production of complex events. For that reason, whenever new plans are generated by the query Optimizer, the previous' plan temporal context should remain active in parallel with the new plan until the first terminates.

All relevant Proton structures have been modified to accommodate the new functionality. In more detail, each structure `x` has become a hashmap with keys equal to plan ids and values equal to `x` structures. Furthermore, all Storm messages sent and received by system components include a plan id parameter, which is used by the components as the key for their hashmaps so that they process the correct plans.

With these modifications, the CEP is able to process more than one set of plans in parallel without any limitation on the number of concurrent active plans. After the temporal context of old plans has terminated, the plan id is removed from the hashmaps.

6.3.3 Multiple Plan Support

Similar methodology has been applied to the rest of the architecture to support CEP plan adaptivity and potential processing of plans in parallel. For this purpose, all the messages that participate in intra-site and inter-site communication include the plan id parameter that specifies the plan that the message participates in. Moreover, all the structures that are relevant to the push/pull paradigm or the GM operator have been implemented in a similar way, making them hashmaps with keys equal to the plan ids and values equal to the corresponding data that relates to the plan.

Chapter 7

Conclusion

This thesis has presented a complex event processing system and architecture that enables realtime complex event processing capabilities for large volume event data streams over distributed topologies. The components of the system have been developed using the Storm event processing framework and the communication protocol between the different Storm topologies that comprise the system nodes has been established. The architecture aims to provide communication-efficient methods utilizing in-situ processing, avoiding the need to collect data in a central node.

The architecture supports event processing and monitoring where the monitoring task has been decomposed into local constraints that can be imposed on geographically distributed data streams. The geometric method employed in this scope has been implemented and allows local processing in nodes where communication is required only in cases where the local constraints are violated.

Another implemented method for efficient communication between the system nodes is the adoption of the push/pull paradigm. In this methodology, the nodes send and receive only the events that are required at each point in time instead of blindly sending each possibly relevant event to all potential receiver nodes. Finally, to enable scalability, efficiency and robustness, the system is able to add new nodes to the system, adapt in cases where inter-site communication latency or event occurrence frequencies have changed and recover in case of node failure.

References

[1] Bothe, Sebastian, Vasiliki Manikaki, Antonios Deligiannakis, and Michael Mock. "Towards Flexible Event Processing in Distributed Data Streams." *In EDBT/ICDT Workshops,* pp. 111-117. 2015. 2

[2] Flouris, Ioannis, Vasiliki Manikaki, Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Michael Mock, Sebastian Bothe et al. "FERARI: A Prototype for Complex Event Processing over Streaming Multi-cloud Platforms." *In Proceedings of the 2016 International Conference on Management of Data,* pp. 2093-2096. ACM, 2016. 1

[3] Flouris, Ioannis, Vasiliki Manikaki, Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Michael Mock, Sebastian Bothe et al. "Complex event processing over streaming multi-cloud platforms: the FERARI approach: demo." *In Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pp. 348-349. ACM, 2016. 1

[4] Agrawal, Jagrati, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. "Efficient pattern matching over event streams." *In Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 147-160. ACM, 2008. 5, 6

[5] Mei, Yuan, and Samuel Madden. "Zstream: a cost-based query processor for adaptively detecting composite events." *In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data,* pp. 193-206. ACM, 2009. 5, 6

[6] Schultz-MG'Eller, Nicholas Poul, Matteo Migliavacca, and Peter Pietzuch. "Distributed complex event processing with query rewriting." *In Proceedings of the Third*

ACM International Conference on Distributed Event-Based Systems, p. 4. ACM, 2009. 5, 6

[7] Wu, Eugene, Yanlei Diao, and Shariq Rizvi. "High-performance complex event processing over streams." *In Proceedings of the 2006 ACM SIGMOD international conference on Management of data,* pp. 407-418. ACM, 2006. 5, 6

[8] Zhang, Haopeng, Yanlei Diao, and Neil Immerman. "On complexity and optimization of expensive queries in complex event processing." *In Proceedings of the 2014 ACM SIGMOD international conference on Management of data,* pp. 217-228. ACM, 2014. 5, 6

[9] Akdere, Mert, Ugur Cetintemel, and Nesime Tatbul. "Plan-based complex event detection across distributed sources." *Proceedings of the VLDB Endowment* 1, no. 1 (2008): 66-77. 6, 27

[10] Zhang, Haopeng, Yanlei Diao, and Neil Immerman. "Recognizing patterns in streams with imprecise timestamps." *Proceedings of the VLDB Endowment* 3, no. 1-2 (2010): 244-255. 6

[11] Keren, Daniel, Guy Sagy, Amir Abboud, David Ben-David, Assaf Schuster, Izchak Sharfman, and Antonios Deligiannakis. "Geometric monitoring of heterogeneous streams." *IEEE Transactions on Knowledge and Data Engineering* 26, no. 8 (2014): 1890-1903. 7, 9

[12] Keren, Daniel, Izchak Sharfman, Assaf Schuster, and Avishay Livne. "Shape sensitive geometric monitoring." *IEEE Transactions on Knowledge and Data Engineering* 24, no. 8 (2012): 1520-1535. 7, 9

[13] Lazerson, Arnon, Izchak Sharfman, Daniel Keren, Assaf Schuster, Minos Garofalakis, and Vasilis Samoladas. "Monitoring distributed streams using convex decompositions." *Proceedings of the VLDB Endowment* 8, no. 5 (2015): 545-556. 7, 9

[14] 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. 7

[15] Sharfman, Izchak, Assaf Schuster, and Daniel Keren. A geometric approach to monitoring threshold functions over distributed data streams. *ACM Transactions on Database Systems (TODS)* 32.4 (2007):23. 7, 9

[16] Sharfman, Izchak, Assaf Schuster, and Daniel Keren. “Aggregate threshold queries in sensor networks.” In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. *IEEE International,* pp. 1-10. IEEE, 2007. 7

[17] Keren, Daniel, Izchak Sharfman, Assaf Schuster, and Avishay Livne. “Shape sensitive geometric monitoring.” *IEEE Transactions on Knowledge and Data Engineering* 24, no. 8 (2012): 1520-1535. 7

[18] Keren, Daniel, Guy Sagy, Amir Abboud, David Ben-David, Izchak Sharfman, and Assaf Schuster. “Safe-Zones for Monitoring Distributed Streams.” *In BD3@ VLDB,* pp. 7-12. 2013. 9