Technical University of Crete, Greece
School of Electronic and Computer Engineering

Distributed Event Detection using the STORM System

Vasiliki Manikaki

Thesis Committee:
Professor Antonios Deligiannakis (Supervisor)
Professor Minos Garofalakis
Professor Vasileios Samoladas

Chania, October 2014

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, in some networks such as sensor networks, transmitting information draws additional power, which is not desirable because usually the sensors have limited battery power and their power source may be irreplaceable. Subsequently, it is not recommended to send all available information to a central node because this transmission will consume a substantial amount of power. For these reasons, one of the most significant limitations of sensor networks is the need to reduce energy consumption. The geometric method was proposed relatively recently and 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, the geometric method is implemented using the real-time distributed computation framework named Storm, for distributed event detection. A topology implementing the geometric method has been constructed using the components provided by the Storm framework, allowing scalable processing of data acquired from nodes and monitoring of certain functions. Finally, the system also allows runtime addition of new functions that can be monitored.

Πολυτεχνειο Κρητης

Τμημα Ηλεκτρονικων Μηχανικων και Μηχανικων Υπολογιστων

Κατανεμημένη Ανίχνευση Γεγονότων με χρήση του Συστήματος **STORM**

Βασιλική Μανικάκη

Εξεταστική Επιτροπή

Καθ. Αντώνιος Δεληγιαννάκης (Επιβλέπων)

Καθ. Μίνως Γαροφαλάκης

Καθ. Βασίλειος Σαμολαδάς

Χανιά, Οκτώβριος 2014

Περίληψη

Κατανεμημένη ανίχνευση γεγονότων είναι η διαδικασία εντοπισμού συγκεκριμένων περιοχών ενδιαφέροντος σε εισερχόμενα δεδομένα που είναι διαθέσιμα σε κάποιο αριθμό κατανεμημένων κόμβων. Η παραδοσιακή προσέγγιση για την ανίχνευση γεγονότων προϋποθέτει την κεντρική συλλογή και επεξεργασία δεδομένων, το οποίο δεν είναι πρακτικό για διάφορους λόγους. Πρώτον, επειδή οι κόμβοι μπορεί να είναι πάρα πολλοί, η κεντρική συλλογή της πληροφορίας δεν είναι πάντα εφικτή. Αυτό μπορεί να συμβαίνει γιατί ο όγκος της πληροφορίας που πρέπει να σταλεί να είναι πάρα πολύ μεγάλος, οπότε να μην υπάρχει το απαιτούμενο bandwidth για αυτή τη διαδικασία. Δεύτερον, σε κάποια δίκτυα όπως τα δίκτυα αισθητήρων, η μετάδοση πληροφορίας απαιτεί μεγάλη ενέργεια, το οποίο δεν είναι επιθυμητό επειδή συνήθως οι αισθητήρες έχουν περιορισμένη μπαταρία και η πηγή ενέργειάς τους μπορεί να είναι αναντικατάστατη. Επομένως, δεν προτείνεται να μεταδοθεί όλη η πληροφορία σε κάποιον κεντρικό κόμβο γιατί αυτή η μετάδοση θα καταναλώσει πάρα πολύ μεγάλη ενέργεια. Για αυτούς τους λόγους, ένας από τα πιο σημαντικούς περιορισμούς στα δίκτυα ασύρματων αισθητήρων είναι η απαίτηση για χαμηλή κατανάλωση ενέργειας. Η γεωμετρική μέθοδος προτάθηκε σχετικά πρόσφατα και επιτρέπει σε ένα δίκτυο κόμβων τον ακριβή έλεγχο με κατανεμημένο τρόπο αν η τιμή μιας σύνθετης συνάρτησης, ακόμα και μη γραμμικής, υπολογισμένη πάνω σε δεδομένα των κόμβων, βρίσκεται πάνω ή κάτω από ένα συγκεκριμένο όριο. Επομένως, σύνθετα γεγονότα μπορούν κατανεμημένα να εντοπιστούν εάν εκφραστούν ως συνάρτηση παρακολούθησης ορίου. Η γεωμετρική μέθοδος θέτει ένα σύνολο από τοπικούς περιορισμούς σε κάθε κόμβο και επιτυγχάνει να περιορίσει την ανάγκη για επικοινωνία μεταξύ των κόμβων για όσο ικανοποιούνται οι περιορισμοί.

Σε αυτή την εργασία, η γεωμετρική μέθοδος υλοποιείται χρησιμοποιώντας το κατανεμημένο υπολογιστικό σύστημα Storm που λειτουργεί σε πραγματικό χρόνο. Μια τοπολογία κατασκευάστηκε χρησιμοποιώντας τα συστατικά του συστήματος, επιτρέποντας την επεκτάσιμη επεξεργασία δεδομένων των κόμβων και τον έλεγχο συγκεκριμένων συναρτήσεων. Τέλος, το σύστημα επιτρέπει την προσθήκη νέων συναρτήσεων που μπορούν να ελεγχθούν κατά τη διάρκεια εκτέλεσης του προγράμματος.

Acknowledgements

This thesis is the end of my long journey in obtaining my degree. This degree isn't the result of one person job. It 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 diploma thesis.

Away from work, special thanks go to my best friend for the past 7 years, Fani Abatzi, for helping me get through the difficult times, and for all the emotional support, entertainment, and caring she provided. She made my life at TUC a truly memorable experience and her friendship is invaluable to me. I look forward to many more times together.

I would like to cease this opportunity to thank all my friends for making my life easy and enjoyable in this city. They made me forget my tensions at work and filled my days with laughter and unforgettable coffee breaks: Nikos Kofinas (thank you for hosting my bunny "Kefte" when I was unable to), Lefteris Chatzilaris (thanks for your real friendship all these years.), Nikos Mainas, Keimis Perros, Nikos Pavlakis. I really had amazing time with you guys and all the best for your future endeavors.

I also would like to thank all my friends in Heraklion, made me feel at home, while I was away from my actual home. It was a pleasure to share many weekends with Effie, Giannis, Maria, Sterios and Nikolas.

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, who constantly reminded me and demonstrated that all I had to do was say the word and she would be here, if for nothing else than to just make me dinner.
I thank my father Stratos, who through a two hour phone conversation at a crucial time, helped me find the strength to continue my work here and helped me come to the realization that obtaining this degree would not be the first challenge in my life. He is my most enthusiastic cheerleader, he always believed in me even when I did not.
I thank my sister Konstantina for being the sweetest part of my life. Words cannot express my love for her. I could not ask for a better sister and friend.
And a special thank to my mother-in-law Sofia, who was there to make me that dinner when my mother wasn't and who offered unconditional and complete support all these years.

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 2014

To my parents and sister
for their endless love, support and encouragement

and to Michalis Foukarakis
who made me laugh countless times.
Thank you for the support and company during late nights of typing.

Contents

List of Figures

LIST OF FIGURES

Chapter 1

Introduction

Detecting events of interest based on the data available at a number of distributed nodes has long been a topic of interest. Straightforward approaches for this task would require the centralized collection and processing of the data. However, such an approach is infeasible, for a variety of reasons, in recent applications. On one hand, the amount of data that needs to be communicated may exceed the bandwidth capabilities of the central processing node (which quickly becomes the bottleneck), thus making continuous data communication infeasible. In other applications, such as sensor networks, data transmission is a significant factor of energy drain in sensor nodes. The sensor nodes have expensive and usually irreplaceable power sources, which makes the need for lower power consumption even greater. Refraining from a continuous propagation of sensor readings is thus essential for prolonging the lifetime of the network.

Recent efforts have been concentrating on being able to detect events of interest without requiring the continuous central collection of data. Several efforts express the event detection task as a threshold query, where the value of a complex function, computed over the data of the distributed nodes, is monitored and the system is required to know whether this value exceeds (or not) a given threshold. While this task is simple for linear functions, the generic problem of function monitoring for non-linear functions was only recently solved using the geometric method.

The geometric method [1] [2] [3] [4] [5] [6] studies these cases. This approach decomposes the monitoring problem into local constraints that can be imposed on the geographically distributed data streams, thus achieving the desired 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.

1.1 Thesis Contribution

This work is a first attempt to implement the geometric method using the highly scalable distributed real-time computation system called Storm[7]. This implementation allows the scalable processing of data acquired by different nodes, the monitoring of different complex functions and the specification at runtime of the functions of interest to use. The geometric method has been implemented using Storm concepts such as spouts and bolts. This system could be an integral part of a complex event processing (CEP) engine, where the goal is to identify complex patterns and combinations of events.

1.2 Thesis Outline

The thesis is divided into 7 chapters. Chapter 2 includes general information about the components this work is built upon, such as the Storm system, the geometric method and the Hadoop software library. The first two are described in greater detail in chapters 3 and 4 respectively. Chapters 5 and 6 describe implementation details. The first one presents the Storm topology of the implemented system, while the second includes information about the methodology used for the monitoring functions. Finally, chapter 7 concludes the thesis and presents potential future work.

Chapter 2

Theoretical Background

2.1 Storm

Storm is a free and open source distributed real-time computation system. Its creator is Nathan Marz and the BackType (social analytics company acquired by Twitter) team. It became an open-source project on 19th September 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.

- **Extremely robust:** Storm clusters are easy to manage and Storm has been built with novice users in mind.

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

- **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.

The main use cases of Storm are:

1. **Stream processing:** Storm can be used for processing new data streams and updating databases in real-time.

2. **Distributed RPC:** Storm can be used to parallelize an intense query (e.g. a search query) on the fly.

3. **Continuous computation:** Storm can do a continuous query and stream the results into clients in real-time.

Storm will be described in more detail in chapter 3.

2.2 Geometric Method

The geometric method studies the following problem: Each node stores a d-dimensional measurement vector and an arbitrary control function is defined on the average of the measurement vectors collected by the nodes. It is desired to determine when the function's value crosses a certain threshold. The geometric method defines a local constraint for each node and asserts that the control function's value never crosses the threshold as long as these local constraints are satisfied.

The geometric method was first presented in [1]. The monitoring areas used in [1] were spheres (balls), an approach also adopted in this work due to its low computational cost. In addition to this approach, a new one based on a technique called "safe zones" [2] [5] [6] is used in this work. The technique is an improvement of the geometric method and takes advantage of the convexity property of safe zones to achieve greater reduction of transmitted messages. More details about the geometric method implementation and the improved safe zones version are presented in chapter 4.

2.3 Hadoop & Distributed File System (HDFS)

Apache Hadoop[1] is a project that develops open-source software used for distributed computing. Its main advantages are reliability and scalability. The software library it provides includes a framework with simple programming models that enables distributed processing of large data sets across clusters of computers. To ensure scalability, it has been designed to work reliably using one machine for computation but it also works well with thousands. The library assumes that hardware failures of machines are common, so it has been designed to be able to detect and handle failures at the application layer. The Apache Hadoop framework is composed of the following modules:

- **Hadoop Common:** The common libraries and utilities needed by other Hadoop modules.
- **Hadoop Distributed File System (HDFS):** A distributed file system that stores data and provides high-throughput access to them across the cluster of machines.
- **Hadoop YARN:** A resource-management platform (framework) that manages cluster resources and which uses them to schedule users' applications (jobs).
- **Hadoop MapReduce:** A programming model and system (based on YARN) used for large scale data processing in parallel.

In this work, Storm is used for distributed computations in a cluster while Hadoop is used for storing application relevant data in its distributed file system, HDFS. HDFS is written in Java and has a master/slave architecture. HDFS clusters include a single NameNode, a master server whose responsibility is to manage the file system namespace and to regulate file accessibility by clients. The NameNode is accompanied by a number of DataNodes which manage storage attached to the nodes that they run on (usually each node in the cluster includes a single DataNode). HDFS provides a Java API for accessing files, but it is also possible to generate clients in other programming languages.

[1] http://hadoop.apache.org/

Chapter 3

Storm

3.1 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 cluster[1] which coordinates Nimbus and the Supervisors. Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. Zookeeper can also store Nimbus's and Supervisors' state. The cluster architecture is shown in Figure 3.1.

3.2 Main Concepts

Storm includes different types of components that are each responsible for handling specific processing tasks. This section describes these components.

3.2.1 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.

[1] http://zookeeper.apache.org/

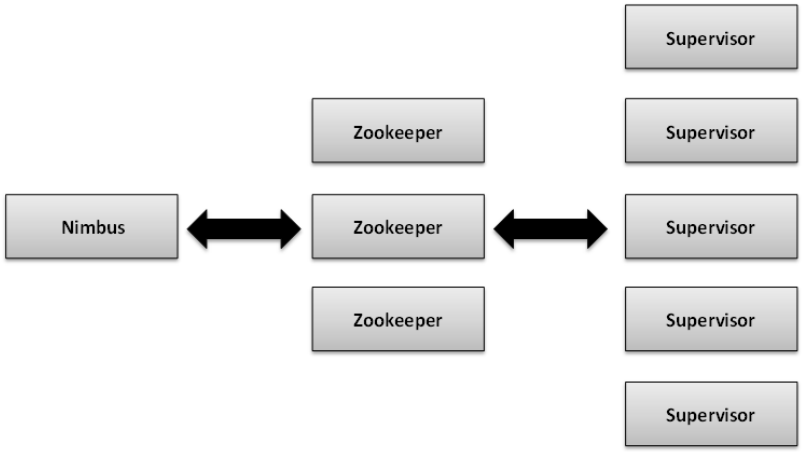

Figure 3.1: Storm cluster architecture.

3.2.2 Spouts

The role of a storm project is to process streams. Stream emission in a topology is achieved using spouts. Spouts read data from external sources(e.g. queue systems, APIs, plain text files, etc.) and transmit them to other components to process them further. Spouts can be configured to emit multiple streams. Figure 3.2[1] shows the behaviour of a spout.

Spout Methods

Spout defines a number of methods for initialization, stream processing and fault-tolerance. Storm calls all these methods on the same thread.

- **open():** This is the initialization method. It is called once when a task for the spout is initialized within a worker on the cluster.

- **nextTuple():** The *nextTuple()* method represents the core of any spout implementation. It is called repeatedly and is responsible for emitting new tuples into the topology if any are available, else it returns immediately. Since all the methods are called on the same thread, *nextTuple()* must not be blocking so that the other methods have a chance to be called.

[1]Figures 3.2, 3.3 and 3.4 were retrieved with permission from https://blog.safaribooksonline.com/2013/06/11/your-guide-to-storm/

- **ack():** This method is called whenever a previously emitted tuple was successfully processed by the tuple's recipient(s).
- **fail():** Whenever a tuple fails to be processed, this method is called to handle the failure. For example, it could put the message back on the queue to be replayed at a later time or terminate the worker if too many failures have occurred.

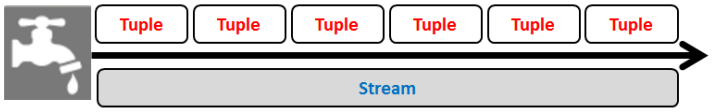

Figure 3.2: The behaviour of Storm Spout.

3.2.3 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. Figure 3.3 shows the behaviour of a bolt.

Bolt Methods

The bolt's main methods are *prepare()* and *execute()*.

- **prepare():** This method is called right before the bolt starts processing tuples to initialize the bolt. It passes in information about the topology.
- **execute():** The main method in a bolt is *execute()*, which is called each time a new tuple is received. Inside this method the bolt has full access to the input tuple's data and can process them appropriately.

3.2.4 Topologies

A specific arrangement of spouts, bolts and their connections is called a *topology* and contains the whole application logic. Figure 3.4 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

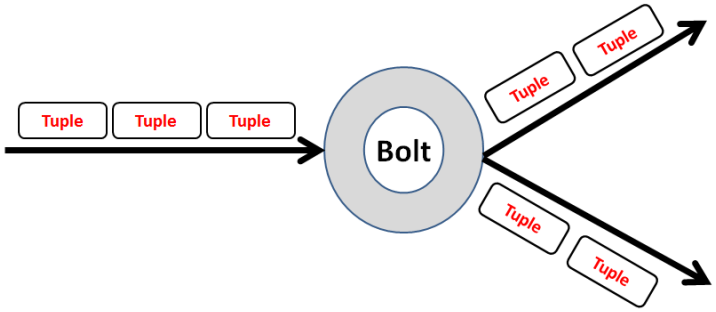

Figure 3.3: The behaviour of Storm Bolt.

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.

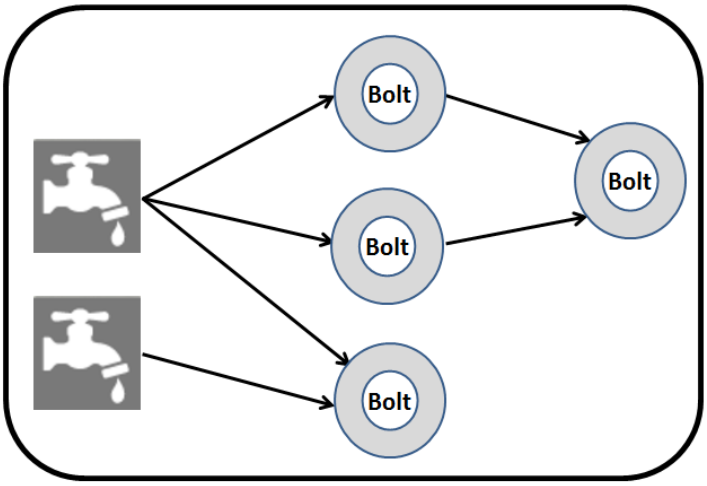

Figure 3.4: Sample Storm Processing Topology that consists of 2 Spouts and 4 Bolts.

3.3 Stream Grouping

A topology needs to define which streams should each bolt 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 shuffle 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 shuffle grouping.

3.4 Storm UI

Storm UI is a web interface that shows information about Storm's cluster and running topologies. The UI can be accessed at http://nimbus host:8080.

Chapter 4

Geometric Method

In this Chapter, a general description of the Geometric Method concepts is presented.

4.1 Problem Formulation

A sensor network of n nodes is considered and each of them receives a data stream. The nodes store a constantly changing d-dimensional vector termed as the *local statistics vector* (LSV) and its value varies over time, however its dimensionality (number of elements) remains constant across all the nodes. The streams optionally have weights with positive values assigned to them. The following notation is used to represent the above:

n	The number of nodes in the network
$S = \{s_1, s_2, \ldots, s_n\}$	The n data streams that the nodes collect.
$P = \{p_1, p_2, \ldots, p_n\}$	The n nodes of the network.
$u_1(t), u_2(t), \ldots, u_n(t)$	The local statistic vectors stored by the nodes.
$w_1, w_2, \ldots, w_n$	The weights assigned to the streams

Table 4.1

The *global statistics vector* is defined as the weighted average of the local statistics vectors and monitoring functions must be expressed over it. These functions are arbitrary functions from the space of d-dimensional vectors to the reals. In the geometric method, it is crucial to determine if the monitored function's output value is greater (or lower) than a predetermined threshold, using as input the global statistics vector. The notation

for these definitions is as follows:

$\overrightarrow{v}(t) = \frac{\sum_{i=1}^{n} w_i \overrightarrow{u}_i(t)}{\sum_{i=1}^{n} w_i}$	The global statistics vector with weighted streams.
$\overrightarrow{v}(t) = \frac{1}{n} \sum_{i=1}^{n} \overrightarrow{u}_i(t)$	The global statistics vector without weights.
$f : \mathbb{R}^d \rightarrow \mathbb{R}$	The monitored function's notation.
$f(\overrightarrow{v}(t)) > r$	The condition that needs to be determined by the geometric method for a specific time t. r is the predefined threshold value.

Table 4.2

For the geometric method to work, all the nodes need to store and maintain three more vectors. First, they store the *estimate vector*, a vector computed whenever the nodes communicate with the coordinator transmitting their local statistics vectors. This communication phase is called a synchronization. Second, they store a delta vector, which is equal to the difference between the current local statistics vector of the node and the last statistics vector that the node has transmitted. Third, they calculate and keep the drift vector. The drift vector is computed based on the estimate vector, the delta vector, along with a possible slack vector, used in some variations of the balancing step used by the coordinator upon a violation of a local constraint by a node (discussed shortly). The notation for these definitions is shown in Table 4.3.

4.2 Geometric Interpretation

As mentioned before, the geometric method determines whether the value of the monitored function with input the global statistics vector is equal or greater than a threshold value. For this to work, the method decomposes the monitoring task into local constraints on streams. The nodes are responsible for verifying if the local constraints imposed on them have been violated. As long as there is no local violation in a node, it has been proven in [1] that the global network state will remain unchanged and there is no need for extra communication between the nodes. Thus, the local constraints imposed on each

$\overrightarrow{v}'_i$	The last statistics vector collected from the node p_i
$\overrightarrow{e}(t) = \frac{\sum_{i=1}^{n} w_i \overrightarrow{v}'_i}{\sum_{i=1}^{n} w_i}$	The estimate vector with weighted streams.
$\overrightarrow{e}(t) = \frac{1}{n} \sum_{i=1}^{n} \overrightarrow{v}'_i$	The estimate vector without weights.
$\Delta \overrightarrow{v} i(t) = \overrightarrow{v}_i(t) - \overrightarrow{v}'_i$	The statistics delta vector.
$\overrightarrow{\delta}_i(t)$	The slack vector sent by the coordinator to node p_i.
$\overrightarrow{u}_i = \overrightarrow{e}(t) + \Delta \overrightarrow{v}_i(t) + \frac{\overrightarrow{\delta}_i}{w_i}$	The drift vector calculated by node p_i when the network has a coordinator which manages balancing of local statistics vectors using slack vectors. Without weights, w_i=1.
$\overrightarrow{u}_i = \overrightarrow{e}(t) + \Delta \overrightarrow{v}_i(t)$	The drift vector when balancing the local statistics vectors is not supported or in the decentralized sensor network case.

Table 4.3

node ensure that the estimate vector will be correct as long as they are satisfied. Moreover, it is observed that the global statistics vector is, at any time, equal to the weighted average of the drift vectors stored by the node:

$$\overrightarrow{v}(t) = \frac{\sum_{i=1}^{n} w_i \overrightarrow{u}_i(t)}{\sum_{i=1}^{n} w_i}$$

This property has the geometric representation shown in Figure 4.1. The global statistics vector lies inside the convex hull of the drift vectors. Using this and the observation (proven in [1]) that the convex hull of drift vectors (such as the one in Figure 4.1) is bounded by the balls defined by the drift vectors and the estimate vector, allows the nodes to easily manage their local constraints.

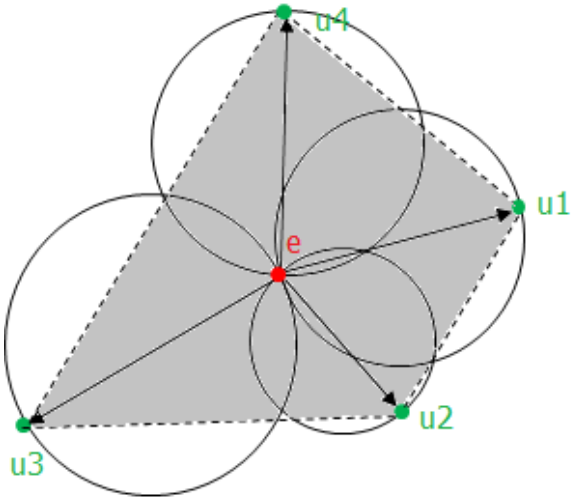

Figure 4.1: The convex hull of the drift vectors is highlighted in gray.

4.3 Local Constraints

To facilitate the visualization of the constraints imposed on the nodes, a coloring scheme is used. The set of vectors that their value on the monitoring function is greater than the threshold are considered to have one color, while the set of nodes for which their value on the function is less than or equal to the threshold have another color. Each node's constraint is to check if the ball defined between its drift vector and the estimate vector (see figure 4.1) is monochromatic, which means that all the vectors contained in the ball have the same color. To determine if that's the case, the only thing required is to calculate the minimal and maximal values of the function in the ball. This calculation is performed locally and requires no communication between nodes as long as the constraints are satisfied. If the latter holds, then it is guaranteed that the function has not crossed the threshold. If not, then the nodes transmit their local statistics vectors and a new estimate vector is calculated and sent back to the nodes. It is notable that other modes of operation by the coordinator do not always require a full synchronization step, but in some cases the coordinator may request the local statistics vectors from only a subset of the nodes, in a balancing effort aimed to reduce communication if local violations by the nodes do not often result in a global violation of the function.

4.4 Safe Zones

In [2] [5] [6], a new method for monitoring distributed data is proposed. It is a generalization of previous research on geometric monitoring. In this approach, the constraints are adapted to the geometric properties of the monitoring function instead of being generic. For each constraint, a concept called a "safe zone" is constructed. The safe zone includes the set of vectors that satisfy the node's local constraints, ensuring that no communication is required as long as the vectors remain inside their safe zones.

According to [2] [5] [6], if the area where the function's values are below the threshold is convex, it is not necessary to construct the sphere (ball). It is sufficient to determine the function's value on the drift vector. On the other hand, if the area is concave, then a convex subset can be defined in which it is investigated if the drift vector lies inside.

Chapter 5

Distributed Event Detection Topology

5.1 Topology

The topology for the system includes a spout implementation that reads input values and sends them to the other components for processing. These spouts from now on will be referred to as "Spouts". In this architecture, two types of bolts are used. "Bolts" are the first layer of processing components and receive input directly from the Spouts. In special cases (detailed in later sections), each Bolt sends data to the next layer of processing, the "Coordinator" bolt. Bolts also receive back messages from the Coordinator and adjust their behaviour accordingly. Figure 5.1 depicts the arrangement of components in the topology as described.

Implementation

In the main class, a TopologyBuilder instance is used which describes how the nodes are arranged and how they exchange data. The different groupings used for the spouts and bolts will be described in their corresponding sections. A Config object is then used to define the topology configuration. This configuration is available to all the nodes inside their initialization method (*prepare()* for bolts, *open()* for spouts), which is called once for each bolt or spout task. Finally, the topology is created using *createTopology()* and submitted to the cluster with the *submitTopology()* method. It is possible to run the topology on the local machine without submitting it to a cluster by using Storm's LocalCluster class. This mode has been extensively used for development, testing and debugging because it was the easiest way to see all topology components working together. Afterwards, it was tested in an actual cluster with some changes to adapt to the different environment.

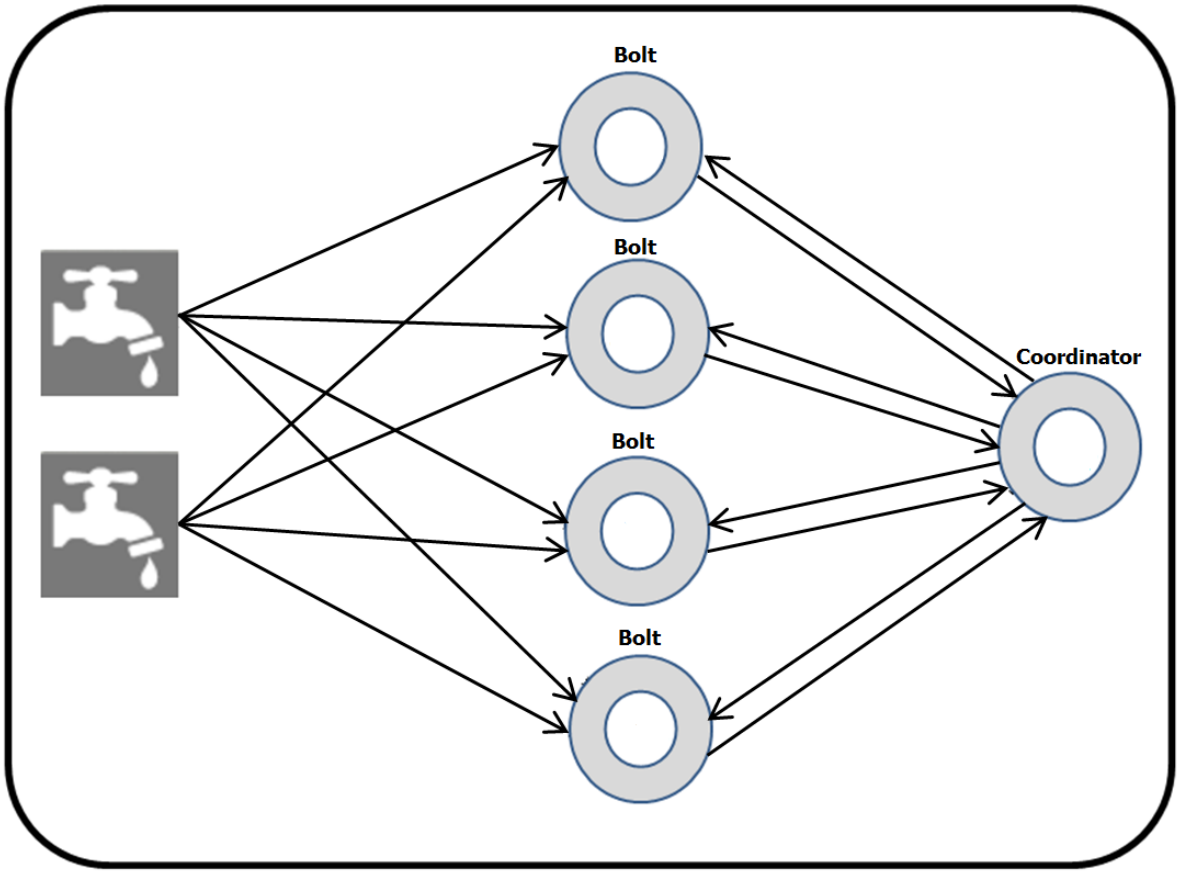

Figure 5.1: Distributed Event Detection Topology.

5.2 Spout

Spouts are the entry point of the topology and are responsible for reading input data and transmitting them to the Bolts. Spouts can read data from files or other data streams in real time. In this topology, each Spout reads from its own text file. The files have a specific format and each file line corresponds to one tuple of data that needs to be sent to the Bolts. The most common tuple includes the actual data to be processed by the Bolts, while other types of tuples are used as commands to the Bolts for deleting nodes or introducing new functions to the system. More details about input files and tuples will be described in the Section 5.3. Spouts currently send their data sequentially one after the other with a set delay between data emissions from different spouts.

Implementation

Spout is a class that implements the IRichSpout interface provided by the Storm framework. Every Spout reads from its own file which is defined in its *open()* method, the first method that is called by any spout. The parameters this method receives are the TopologyContext, the configuration object (Config) and the SpoutOutputCollector object. The TopologyContext contains information about this task's place within the topology, including the task id and component id of this task, input and output information, etc. The configuration object is created in the topology definition (see Section 5.1 Implementation). Finally, the SpoutOutputCollector object is stored within the class and allows emitting the data that will be processed by the Bolts. The *nextTuple()* method represents the core of any spout implementation. Storm calls this method in order to request that the spout emits tuples to the output collector. In this method, the next line from the input file is read. The line is split using "|" as a separator. Then the Spout translates the different values into the appropriate tuple and emits the tuple to the corresponding Bolt(s). For example, an input file line may look like this:

$$3|Delete$$

In this case, the Spout detects that a Delete operation needs to be sent to the appropriate Bolt and emits the correct tuple. In this topology, it has been defined that the spouts emit their tuples in order and the Spout that has priority to emit has to wait at least *emitInterval* seconds after the previous spout has emitted its tuple. The parameter *emitInterval* is defined in the Spout class.

5.3 Bolt

The Bolt's main responsibility is to implement the Geometric Method to monitor specific functions. The role of Bolts in the system is threefold. First, they receive data from the Spouts and process it appropriately. Second, they detect local constraint violations on the nodes based on the information sent from the Spouts and then send the events to the Coordinator to inform it about the violations. Third, they send their computed local statistics vectors to the Coordinator when requested. In the first case, there are three different types of input:

- **New Value:** This is the most common tuple received by the Bolts. It includes information about which node sent the data (Node ID), the data value itself and the function to use for data processing (see Section 6.1). If this Node ID is encountered for the first time, then it is assumed that it originates from a newly added node and it is added to the system. The Spouts use fields grouping for sending their data so that tuples with the same Node ID always go to the same bolt. Each tuple also includes a timestamp describing the exact time it was sent.

- **New Function:** This tuple informs the Bolt that a new function has been introduced to the system and is now available for monitoring. It includes information about the name of the function, the location where its implementation code is located (e.g. the name of a .jar file) and a few parameters used to initialize the function. The Spouts use all grouping for sending this tuple since all the Bolts are required to know about this addition to the system.

- **Delete Node:** It contains information about the Node ID to delete as well as timestamp information. Optionally, it includes a Function ID to indicate that the Node ID will be deleted only in relation to this specific function instead of being removed completely from the system. The "Spouts" use fields grouping for sending these tuples.

Implementation

Bolt is a class that implements IRichBolt. The Bolt class body includes the behavior described above. Each one has a hash map of registered functions and keeps an instance of the function object it currently monitors. These functions may change dynamically according to the tuples received. The most important method in the Bolt class is *execute()*, which is called once per tuple received. This method analyzes the tuple's data and behaves accordingly. There are five different cases, each corresponding to a different stream ID received from the Spouts or the Coordinator:

- **NewFunction:** The NewFunction tuple always comes from the Spouts. It enables the addition of a new function to the Bolt's hash map. The values included in the tuple are the function name (i.e., VarianceFunction), its threshold value and a boolean variable named *thresholdEqualityCondition*. The tuple also includes the location of the function's implementation class (typically a .jar file). The combination of the function's name, the threshold value and the *thresholdEqualityCondition* is used as the key for storing the function object to the hash map. An example follows. A Spout reads the following line from its input file:

  $$newFunction|storm.AverageFunction|AverageFunction.jar|90.0|true.$$

  The string "newFunction" indicates that a new function is going to be added to the system. The Spout creates a tuple with stream ID equal to "NewFunction" and inserts "storm.AverageFunction" as the function name, "AverageFunction.jar" as the location for the implementation, "90.0" as the threshold value and "true" as the value of *thresholdEqualityCondition*. Then it emits the tuple. The Bolt receives it, searches for the function implementation inside AverageFunction.jar or the topology's jar and creates an AverageFunction object. It then puts that object inside its hash map, using "storm.AverageFunction_90.0_true" as the key. This procedure is described in detail in Section 6.3.1.

- **DeleteNode:** This tuple comes from the Spouts and informs the Bolt that it needs to delete one of the nodes that its registered functions manage. The Node ID contained in the tuple is used for removing the Node from the system. If a Function ID is not specified in the tuple, then the node is deleted from all the functions, else it is deleted only from the function that corresponds to the given ID. Every successful deletion is handled the same way as a local violation. More specifically, in the case where a node is deleted for a specific function, a single local violation occurs, while in the case where the node is deleted for all the functions, local violations occur for each one.

- **Value:** As described previously, Bolts receive data values from the Spouts. This tuple includes the ID of the node that sent the value and the function id this value corresponds to. The function id should be of the form FunctionName_threshold_boolean and a function with these parameters must have already been defined and added to the Bolt's hash map from a previous NewFunction tuple. For every new value tuple a Bolt receives from a Spout, it checks if a local violation has occurred. If one is detected, it sends the timestamp of the tuple that caused the local violation to the Coordinator. If no violation was detected, it continues normally with the next tuple. Tuples that refer to a Node ID which the Bolt encounters for the first time are treated as local violations.

- **SendLSV:** At some time after a local violation has occurred, the Bolt will receive this message from the Coordinator requesting its data. In this case, the Bolt calculates the sum of the local statistics vectors (LSVs) of all the nodes that it manages and sends it along with the number of nodes to the Coordinator.

- **GlobalV:** This tuple is sent to all the Bolts from the Coordinator and includes the estimate vector that the Coordinator had calculated previously. The Bolts store this vector.

5.4 Coordinator Bolt

The Coordinator's main responsibility is to coordinate what happens with local violations. It occasionally receives a timestamp from a Bolt as an indication that a local violation has just occurred and was detected by the Bolt. Once this timestamp is received by the Coordinator, it sends it back to all the Bolts, requesting them to send their sum of LSV arrays in order to compute the new estimate vector. Once it receives all the respective LSV arrays as well as the number of nodes managed by each Bolt, it calculates the estimate vector in the following way: It adds the sum of LSV arrays received and divides it by the sum of number of nodes managed by each Bolt. The new estimate vector is then sent back to all the Bolts.

Chapter 6

Methodology and Implementation

This chapter describes the methodology used in this work and provides more implementation details about handling the monitoring functions. It is divided into three sections. 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.

6.1 Functions

As previously mentioned, the system implements the geometric method to monitor functions. In this section, these functions are described in more detail.

6.1.1 General Functions

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 available implemented function classes. The UML class diagram of Figure 6.1 shows the implemented hierarchy.

Function

At the top of the hierarchy is the abstract Function class. The variables and data structures of the Function class are presented first and the method definitions follow.
To implement the geometric method, some arrays are required. These are stored in the function class inside several hash maps and are described below. Table 6.1 presents the data types that the hash maps store.

Figure 6.1: Function Class Hierarchy

Hash Map Name	Hash Map Value	Description
lastValues	Tuple[ ]	The last Y tuples sent by a Spout
counts	Integer	The number of tuples received for each Node ID
lastSent	Float[ ]	The last LSV sent
dV	Float[ ]	The difference between LSV and lastSent
driftVector	Float[ ]	The sum of the estimate vector and the dV for each Node ID
LSV	Float[ ]	The Local Statistics Vector of the Node ID

Table 6.1

The *LSV* hash map is used by the functions monitored with the geometric method. Each function defines it in a different way. A hash map called *lastValues* is used, in which the last Y tuples sent by a Spout are stored. Y is a constant defined when first running the topology. Each time a new tuple needs to be stored in the tuple array of *lastValues* for a specific node, it is placed in the last cell of the array and the previous values are shifted one cell to the left. If the array had Y values already, then the least recent one (in the first cell) is removed from the array. This hash map is updated whenever a new tuple is received from the Spout. Intuitively, it is expected that this hash map will hold information that will be valuable when dealing with high input rates and past data (and timestamps) for each Node ID will need to be remembered. The hash map *counts* holds the number of tuples received for each Node ID. This is also updated whenever a new tuple is received from the Spout. The *lastSent* hash map is updated in two different cases. In the first case it is updated once a local violation is detected by a bolt. The Local Statistics vector of the Node ID which caused the violation and contacted the coordinator is stored. In the second case the Local Statistics vectors of all Node IDs managed by a bolt when the Coordinator requests the Bolt's data are stored. The difference between *LSV* and *lastSent* is stored in the *dV* hash map. This is calculated each time *LSV* or *lastSent* is updated. The sum of the estimate vector and the *dV* for each Node ID is stored in the *driftVector* hash map. This is updated whenever either the estimate vector or the *dV* hash map change values.

Apart from these hash maps, the Function class includes a number of important variables. First, the *threshold*, which is the value used to compare the function's value with to determine if there is a local violation. The *thresholdEqualityCondition* boolean variable is used in the cases where the function's value is equal to the threshold to

determine if there is a local violation (see also Section 6.2). Finally, the class keeps the latest estimate vector sent by the Coordinator bolt in the *estimate* variable.

Since Function is an abstract Java class, it contains both abstract and non-abstract methods. The former are required to be implemented by subclasses, while the latter provide common operations used by all subclasses. The non-abstract methods are:

- **void insertVal(Tuple newValue):** This method inserts the value tuples sent by the spouts to *lastValues*, shifting its values left, keeping the last Y values. It also updates the *counts* hash map which holds the number of value tuples received for each NodeID. This method is called whenever a new message is received from a spout. Then, it is checked if a local violation occurs.

- **void updateLastSent():** This method is called whenever the Coordinator bolt requests from all Bolts to send their sums of LSV arrays. Whenever a Bolt sends its sum, it stores the LSV vector of each node in the *lastSent* hash map.

- **void updateLastSent(int NodeID):** This method is called whenever a local violation is detected. It updates the *lastSent* hash map value corresponding to the NodeID which caused the violation with the latest LSV vector.

- **void updateDV():** This method updates the dV hash map each time *lastValues* or *lastSent* is updated. (dV = lastValues - lastSent)

- **Float[] sum():** This method calculates and returns the sum of LSV arrays to be sent to the Coordinator bolt when requested.

- **void setEstimate(List¡Float¿ newEstimate):** This method stores the estimate vector received from the Coordinator bolt.

- **void setDriftVector():** This method calculates the new drift vector after either *estimate* or dV change values. (driftVector = estimate + dV)

- **boolean deleteNode(int NodeID):** Deletes the node with the given Node ID. The node is removed from all the hash maps stored by this function.

The abstract methods are:

- **boolean hasLocalViolation(int NodeID):** This method returns true if the latest value received from the node with the given Node ID causes a local violation.

- **void updateLSV(int NodeID):** This method is called whenever a new tuple is received and updates the LSV array of the node that sent the value according to the function subclass that implements the method.

- **void updateLSV():** This method is called from all the Bolts, before they calculate their sums. This is done to update the LSV arrays for all the nodes with the latest estimate.
- **float f(Float[] vector):** This method returns the value of the function on the given vector. Its implementation clearly depends on the function being monitored.

Ball Function

This is a direct abstract subclass of Function and implements the ball technique described in Section 6.2.1. It provides an implementation for the *hasLocalViolation()* method. The rest of Function's abstract methods are implemented by subclasses.

Safe Zone Function

This is a direct abstract subclass of Ball Function and implements the safe zone technique described in Section 6.2.2. It derives from Ball Function instead of Function because, in certain cases, local violations are detected using the ball technique. It stores an array of Safe Zone objects called *safeZones* that models the safe zones required by the technique. Subclasses that want to use the safe zone technique have to implement the following abstract method:

- **void computeSafeZones():** This method computes the safe zones that correspond to the implementing function subclass and stores their information in the *safeZones* array.

6.1.2 Implemented Sample Functions

Average Ball Function

This is a subclass of Ball Function, in which the objective is to determine whether a two-dimensional estimate vector has a squared L2 norm lower/greater than a given threshold. In this function, the LSV hash map stores the last two values that have been received for each Node ID. 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 6.2

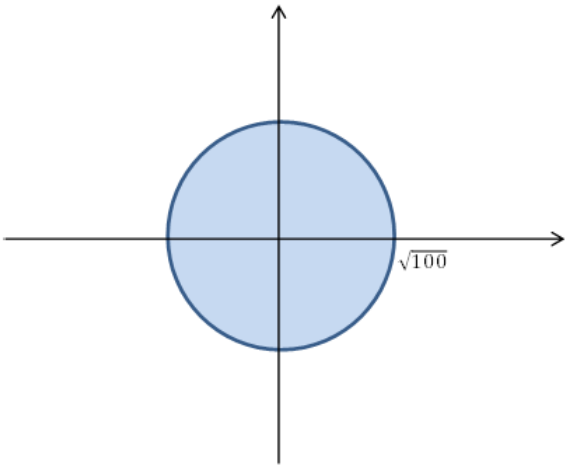

Figure 6.2: Graph of function Average (Threshold = 100)

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.

Variance Ball Function

This is a subclass of Ball Function, In this function, the LSV hash map 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 6.3

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.

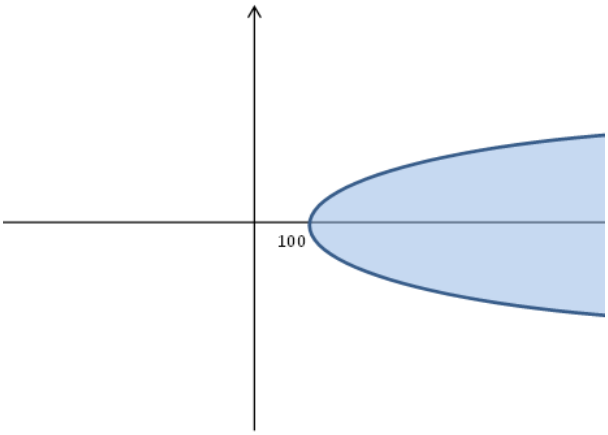

Figure 6.3: Graph of function Variance (Threshold = 100)

6.2 Local Violation Detection

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

6.2.1 Ball Technique

In this method, each Bolt constructs a ball which is centered at:

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

has a radius of:

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

and applies a grid inside the ball. The local constraint on each stream is set as follows: each Bolt 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:

$$\begin{gathered}(\text{f(estimate)} < \text{T} \;\&\&\; \text{f(all ball's point)} \geq \text{T}) \;||\\ (\text{f(estimate)} \geq \text{T} \;\&\&\; \text{f(all ball's point)} < \text{T})\end{gathered}$$

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:

$$(\text{f(estimate)} \leq \text{T} \;\&\&\; \text{f(all ball's point)} > \text{T}) \;||$$
$$(\text{f(estimate)} > \text{T} \;\&\&\; \text{f(all ball's point)} \leq \text{T})$$

If there is at least one ball's point that violates this constraint then the Bolt detects a local violation and emits a notification to the Coordinator bolt.
In the following figures it is investigated if there are local violations for the average function. The first two cases (Figures 6.4 and 6.5) 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 6.6 and 6.7 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.

6.2.2 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) ≥ 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 6.8, 6.9, 6.12, 6.13). 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 6.11 and 6.10). 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 6.15 and 6.14).

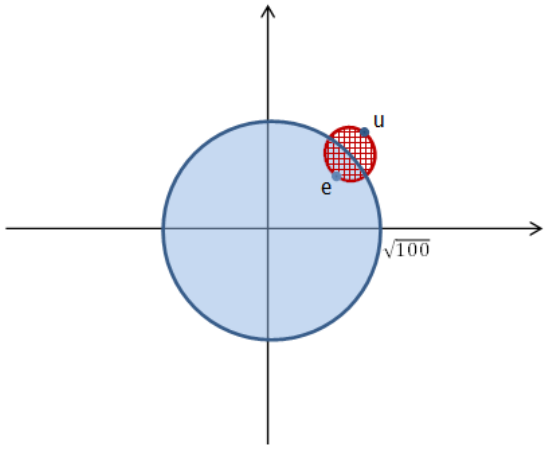

Figure 6.4: Local Violation in Average Function - Ball Technique

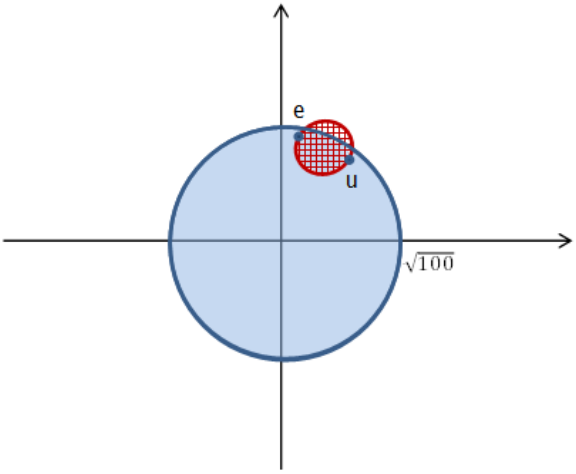

Figure 6.5: Local Violation in Average Function - Ball Technique

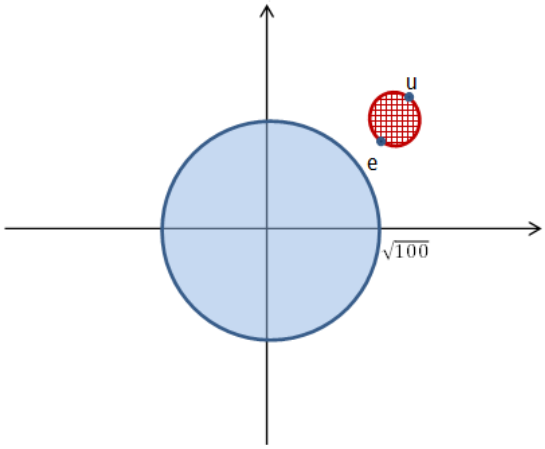

Figure 6.6: No Local Violation in Average Function - Ball Technique

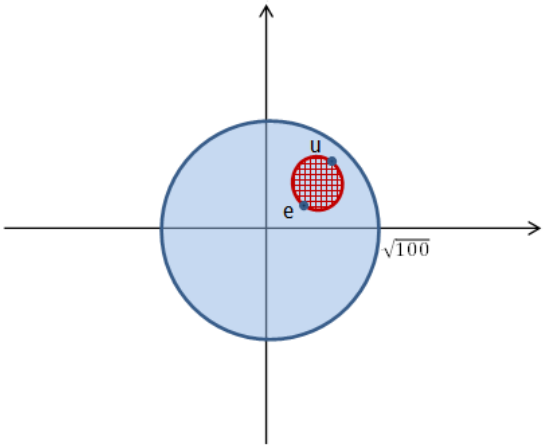

Figure 6.7: No Local Violation in Average Function - Ball Technique

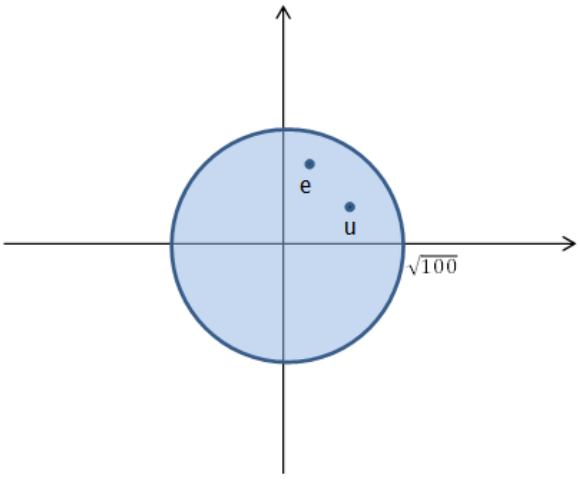

Figure 6.8: No Local Violation in Average Function - Safe Zone Technique.

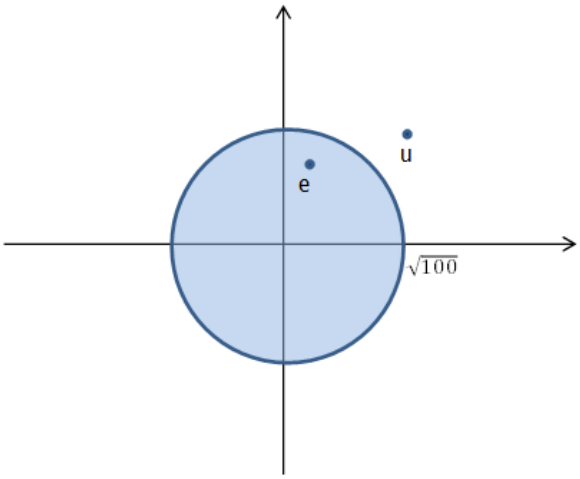

Figure 6.9: Local Violation in Average Function - Safe Zone Technique.

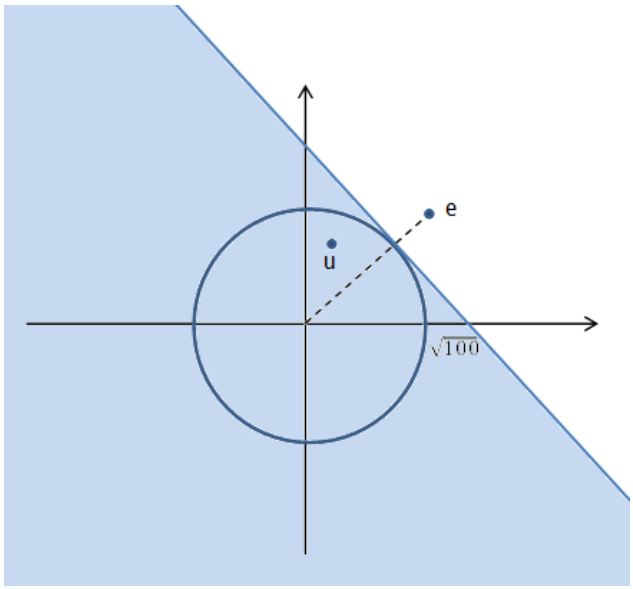

Figure 6.10: Local Violation in Average Function - Safe Zone Technique.

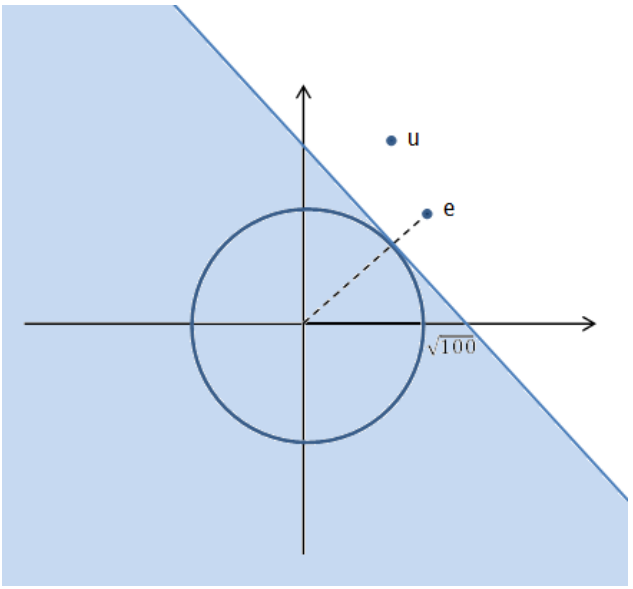

Figure 6.11: No Local Violation in Average Function - Safe Zone Technique.

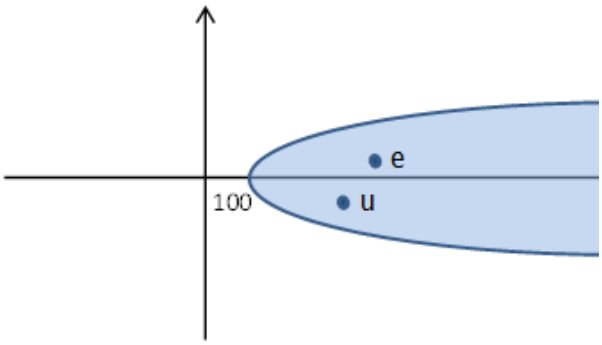

Figure 6.12: No Local Violation in Variance Function - Safe Zone Technique.

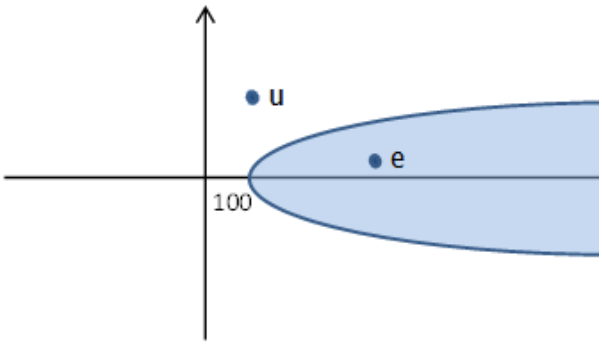

Figure 6.13: Local Violation in Variance Function - Safe Zone Technique.

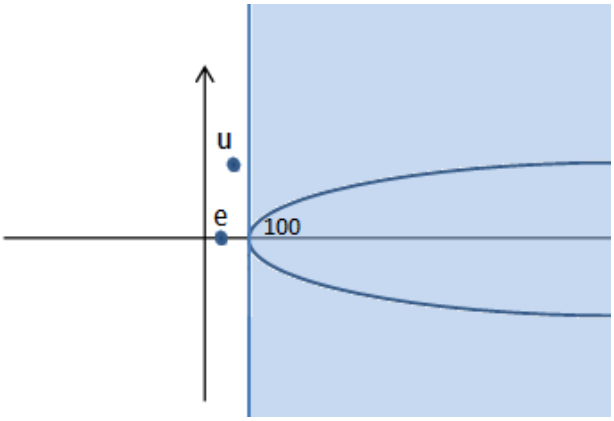

Figure 6.14: No Local Violation in Variance Function - Safe Zone Technique.

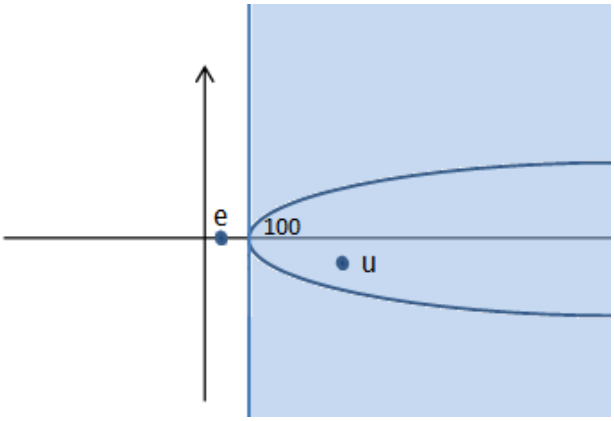

Figure 6.15: Local Violation in Variance Function - Safe Zone Technique.

6.3 Changing Functions Dynamically

An important component of the implemented topology enables the monitoring of an unlimited number of functions. Previous sections described two of those functions, the average function and the variance function. This section describes and explains the methodology deployed in this work regarding the dynamic monitoring of new and existing functions.

6.3.1 Adding a Function

To add a new monitoring function to the system, an implementer needs to do the following:

- **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.

- **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.

- **Put the .jar file inside an HDFS folder accessible to all cluster machines:** The new .jar file needs to be accessible by the system, which means that all the machines in the cluster are required to have access to the function implementation inside the .jar. To achieve this, the implementer needs to put the file in HDFS, since the cluster machines all have access to it. The file has to be copied to an accessible folder which has to be defined and be known in advance, so that the system will know where to look for it.

- **Provide the corresponding input to the Spouts so that they can inform the Bolts about the new addition and load the new class:** To introduce the new function to the system, the Bolts need to load the function class and keep an instance of it in their hash map of monitored functions. To do that, they need to be informed by the Spouts that a new function is available and where its implementation is. As already described in Section 5.3, the input file may contain lines that describe the addition of a new function. The line starts with the word "newFunction", followed by the fully qualified name of the implemented class, the name of the .jar file, the threshold value and the ThresholdEqualityCondition boolean variable, all separated by the '|' symbol. Section 5.3 contains an example of such an input file line.

The behavior of the system regarding this functionality is as follows:

1. A Spout reads an input file line starting with "newFunction", processes its information and sends it to all the Bolts. This way, all of them are informed about the existence of the new function and can proceed accordingly.

2. The Bolts receive this tuple and check if they already have a hash map entry for this function, taking into account the threshold value and the boolean variable.

3. If there is already a function with the same name, threshold value and boolean variable value, then nothing more is required. The Bolts can already monitor this function with these parameters.

4. If no such entry exists, then they check if the function's class has already been loaded (e.g. submitted along with the topology).

5. If the class has already been loaded, then go to step 11.

6. If the class hasn't been loaded, the Bolts search the machine's temporary folder (e.g. /tmp for Unix systems) for the .jar file with the provided name.

7. If the .jar file exists, then go to step 10.

8. If there is no .jar file in the temporary folder, then the Bolts look in HDFS for the .jar file with the provided name. If the file is not located in an accessible folder in HDFS, then there is no way to add this function to the system and future values for this function will be discarded since it is not possible to process them.

9. After the .jar file is found in HDFS, the Bolts copy it to the respective machine's temporary folder of their local file system for easy access from the Bolts.

10. The Bolts load the class from the .jar file.

11. The Bolts create an instance of the class using the provided values as constructor arguments.

12. The new function instance is added to the hash map of the Bolts, using as hash map key, the combined value of its name, threshold and boolean value (e.g. `AverageFunction_90.0_false`).

From this point forward, this function implementation is available to the system and the function can be monitored.

6.3.2 Selecting a Function

After adding functions to the system using the above method, the Bolts are able to monitor them. The selection of function to be monitored is defined by the incoming tuples. As seen in Section 5.3, an input file line that contains a value has the following format:

$$2|78|Value|\texttt{AverageFunction\_90.0\_true}$$

The last part of the line (`AverageFunction_90.0_true`) describes the function to monitor with its specific parameters. The Bolt that receives this tuple checks its hash map for the entry that has the same key as this value. If the key exists in the hash map, then the value (78) is used along with the function to determine if a local violation has occurred. This way, whenever a new value tuple is received by a Bolt, it will know which function is required exactly to investigate for local violations.

Chapter 7

Conclusion

The ultimate goal of distributed event detection for distributed nodes is to efficiently gather and process data in order to identify and handle the desired events. To reduce bottlenecks due to bandwidth issues in central nodes collecting data and to preserve energy drain in sensor networks, a method to decrease the required communication between the nodes is essential. One of the approaches proposed toward this purpose is the geometric method. By inducing local constraints on each node, the method allows the nodes to avoid transmitting data to other node as long as the constraints (e.g. a function's value doesn't cross a specific threshold) are satisfied.

In this thesis, the geometric method was implemented using the Storm computational system. First, the geometric method and its concepts was presented, along with the safe zones approach, an improvement upon the geometric method. Then, the Storm system and its components were described in detail. A Storm topology includes several components such as spouts and bolts. Spouts are used for reading from data input streams and emitting data tuples to bolts. Bolts on the other hand receive this data and process it, emitting their own streams if necessary to other bolts. The implemented system included a Storm topology with spouts that read from input files values corresponding to sensor data and other commands and bolts that used this data to monitor whether several defined functions crossed the defined threshold. A coordinator bolt was used to ensure that the bolts maintain the latest estimate vectors used in their calculations. Two different functions were monitored, the average and the variance function, using two different methods for monitoring, the ball technique and the safe zone technique. Finally, an important characteristic of the system was the ability to add and select which functions to monitor at runtime.

7.1 Future Work

This work could be extended and improved in various ways. One of the limitations of the implemented system is the assumption that the spouts are emitting their tuples in order and each one waits for its turn to send its available data to the bolts. Currently they are bounded by the *emitInterval* parameter, which defines a period of time between each spout's emissions. While this is an acceptable way to organize the data inflow and avoid conflicts and inconsistencies, it hinders the ability of the system to quickly process work with a large amount of fast incoming data. To eliminate this limitation, apart from removing the *emitInterval* constraint, a way to synchronize the incoming data inside the Bolt implementation will be required. In that case, the system will be able to handle any type of input, regardless of amount or speed of inflow.

Another possible extension to the system relates to the type of input read by the Spouts. Currently, the Spouts read from plain text files. There are other, more advanced types of external data input sources that can be used for this purpose. For example, a streaming API like Twitter can be used or a queuing broker like Kafka[1] or Kestrel[2]. In any case, receiving input from more advanced sources will greatly improve the usability of the system.

[1] http://kafka.apache.org/
[2] https://github.com/twitter/kestrel

References

[1] Sharfman, I., Schuster, A., Keren, D.: A geometric approach to monitoring threshold functions over distributed data streams. ACM Transactions on Database Systems (TODS) **32**(4) (2007) 23 1, 4, 14, 15

[2] Sharfman, I., Schuster, A., Keren, D.: Shape sensitive geometric monitoring. In: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM (2008) 301–310 1, 4, 17

[3] Burdakis, S., Deligiannakis, A.: Detecting outliers in sensor networks using the geometric approach. In: Data Engineering (ICDE), 2012 IEEE 28th International Conference on, IEEE (2012) 1108–1119 1

[4] Sagy, G., Keren, D., Sharfman, I., Schuster, A.: Distributed threshold querying of general functions by a difference of monotonic representation. Proceedings of the VLDB Endowment **4**(2) (2010) 46–57 1

[5] Keren, D., Sagy, G., Abboud, A., Ben-David, D., Schuster, A., Sharfman, I., Deligiannakis, A.: Geometric monitoring of heterogeneous streams. (2014) 1, 4, 17

[6] Keren, D., Sharfman, I., Schuster, A., Livne, A.: Shape sensitive geometric monitoring. Knowledge and Data Engineering, IEEE Transactions on **24**(8) (2012) 1520–1535 1, 4, 17

[7] Nathan Marz: Website of the storm project. Available online: `http://www.storm-project.net`. 2