TECHNICAL UNIVERSITY OF CRETE

DOCTORAL THESIS

An adaptive complex event processing system over Storm

Author:
George PROUNTZOS

Supervisor:
Professor Antonios DELIGIANNAKIS

A thesis submitted in fulfillment of the requirements for the degree of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering

Thesis Committee
Professor Antonios Deliggiannakis
Professor Minos Garofalakis
Professor Vasilis Samoladas

October 30, 2017

"If you try to fail, and succeed, which have you done?" — "

George Carlin

Abstract

Distributed complex event processing is a method of tracking, analyzing, processing and detecting specific events or patterns of events that may occur in event data streams from various distributed sources. Every node within a network does in-situ processing from its sources. This way of handling large sets of data whose sources may be geographically scattered, has a couple of advantages over centralizing the data into a single node and process them there. The amount of information is potentially vast and limitations of the available bandwidth render this approach impractical. Thus, a distributed approach is implemented for communication efficiency. Additionally, having a single node do all the processing and synchronizations of the network creates a Single Point of Failure (SPOF) making the system unreliable when real-time processing is important.

For these reasons, an architecture of an in situ complex event processing is developed using the Apache Storm primitives. The architecture detects patterns of interest on incoming event data streams over a number of nodes in the network in real-time. The patterns, events and network agents are specified from a user in a file and the system is designed to support multiple such queries as well as to adapt to any significant changes in the network.

Περίληψη

Κατανεμημένη επεξεργασία σύνθετων γεγονότων είναι μια μέθοδος ανίχνευσης, ανάλυσης, επεξεργασίας και εντοπισμού συγκεκριμένων γεγονότων ή μοτίβων γεγονότων που εμφανίζονται σε ροές πληροφορίας απο ποικίλες κατανεμημένες πηγές. Κάθε κόμβος εντός δικτύου πραγματοποιεί επί τόπου επεξεργασία στα γεγονότα των πηγών του. Αυτός ο τρόπος χειρισμού μεγάλου όγκου δεδομένων με γεωγραφικά διεσπαρμένες πηγές έχει αρκετά πλεονεκτήματα σε σχέση με την συγκέντρωση και επεξεργασία των δεδομένων σε έναν κόμβο. Η ποσότητα της πληροφορίας είναι δυνητικά τεράστια και οι περιορισμοί στο bandwidth καθιστούν την μέθοδο της συγκέντρωσης ανέφικτη. Έτσι, για αποδοτικότητα στην επικοινωνία υλοποιήθηκε η προσεγγιση της κατανεμημένης επεξεργασίας. Επιπλέον, έχοντας μόνο εναν κόμβο για συγχρονισμό του δικτύου και επεξεργασία δημιουργείται κόμβος αποτυχίας SPOF καθιστώντας το σύστημα μη αξιόπιστο οταν η επεξεργασία σε πραγματικό χρόνο είναι σημαντική.

Για τους λόγους αυτούς, αναπτύχθηκε μια αρχιτεκτονική επί τόπου επεξεργασίας σύνθετων γεγονότων χρησιμοποιώντας το Apache Storm . Η αρχιτεκτονική εντοπίζει πρότυπα ενδιαφέροντος στις εισερχόμενες ροές δεδομένων γεγονότων σε πραγματικό χρόνο σε πολλαπλούς κόμβους δικτύου. Τα πρότυπα, γεγονότα και πράκτορες στο δίκτυο ορίζονται απο το χρήστη και το σύστημα σχεδιάστηκε ώστε να υποστηρίζει πολλαπλά ερωτήματα και να προσαρμόζεται σε σημαντικές αλλαγές στο δίκτυο.

Acknowledgements

The journey of my studies has been long and would not be possible if not for a number of people. First of all, I wanna thank my family for their love, support and trust throughout this journey. Additionally, I need to thank *Stayros, Stefanos, Dimitris, Giwrgos, Michalis, Filipos, Giwrgos, Dafni, Aliki, Nikos, Giwrgos, Iwanna, Giannis, Giwrgos, Vaggelis, Orestis, Marilena* and of course *Nasia* for their patience and support.

I want to thank specifically my advisor, professor Antonios Deligiannakis for his guidance, availability and the professionalism he displayed during the conduction of the thesis. I also want to thank Vasiliki Manikaki for her help and the committee, professors V. Samoladas and M. Garofalakis for reading and evaluating this thesis.

Contents

List of Figures

List of Abbreviations

CEP	**C**omplex **E**vent **P**rocessing
EPN	**E**vent **P**rocessing **N**etwork
EPA	**E**vent **P**rocessing **A**gent
API	**A**pplication **P**rogramming **I**nterface

Chapter 1

Introduction

1.1 Thesis Overview and contribution

Nowadays, there are many systems and application domains that demand timely processing of new data and involve internetworking of a number of physical devices. Big data systems, Internet of Things platforms, environmental sensor networks are some examples that output infinite sequences of measurements or data that need to be processed and respond quickly to events. This data consists of information about various requests, responses, states updates etc. and need to be monitored and processed so that the interesting and useful part can be detected and used for actions, statistics, predictions, conjectures or conclusions that are of interest.

Complex event processing is used for processing event data in data streams and detecting efficiently interesting situations in real time. In this way the processing workloads are being shared between the nodes, a single point of failure (SPOF) is avoided and the communication needed between them is reduced to minimum. In this thesis, the FERARI project was used for these reasons. It's a Flexible Event Processing for Big Data Architecture which is a distributed streaming platform that can provide efficient complex event processing. It uses the Proton engine, a scalable, integrated platform which supports the development and maintenance of both event driven and complex event processing applications. This powerful CEP engine has been implemented on top of the streaming cloud platform Apache Storm. Each node of the network runs a Storm installation along with the FERARI topologies and has event processing agents assigned to it. It processes in-situ the incoming data and by using the geometric method it monitors locally the receiving events whether constraints and conditions are satisfied or violated.

This work makes an attempt to add adaptive behavior to CEP processing in cases where the FERARI system detects changes regarding event appearance probabilities, network latencies, the query or the agents. The contribution of this thesis is the design of an architecture and an algorithm development to make the CEP processing system more flexible by adapting to various changes and being able to continue functioning smoothly in the most effective way.

1.2 Thesis Outline

The thesis is structured in a bottom-up way where every chapter covers a lower level of the system's design architecture than its next.

Chapter 2 describes some background concepts that are used throughout the thesis. These are the complex events and the computing framework of Apache Storm.

Chapter 3 analyses the engine which processes the complex events of the system both as standalone and implemented on top of the cloud platform, Storm.

Chapter 4 introduces the FERARI implementation and the modifications needed to make it adaptive when circumstances demand some changes to the system.

Chapter 5 concludes the thesis.

Chapter 2

Background

This chapter introduces some concepts which are used throughout the thesis. The complex events and the computation framework are presented in this section.

2.1 Complex Events and Complex Events Processing

Events are objects that represent data and notify about activities or changes of states. The same activity or change of state can be represented by more than one event object. The events can be simple or complex. The simple events are atomic instances and are fed into the system for processing. They are called primitive events. Event processing is a method of tracking and analyzing (processing) streams of information (data) about things that happen (events) and deriving a conclusion from them. **Complex event processing (CEP)** combines data from multiple sources to infer events or patterns that suggest more complicated circumstances. The complex events consist of other complex or primitive events and are derived by the CEP engine if certain event patterns are met [3], [1], [10]. Figure 2.1 shows how this is done.

These patterns are the rules that are expressed in the form of **queries** and submitted in the CEP engine in order to detect the complex events. The overall structure of a query of a complex event language is:

Query clauses

`FROM` <input stream >
`PATTERN` <pattern structure>
`WHERE` <pattern matching condition>
`WITHIN` <sliding window>
`HAVING` <pattern filtering condition>
`RETURN` <output specification>

The PATTERN, WHERE, WITHIN clauses can define completely a query pattern [11]. The **PATTERN** clause declares the structure of a pattern and

uses constructs like *AND* where all the involving events need to take place, *SEQ* where the involving events need to be sequential, *OR* where any of the events need to occur. The **WHERE** clause contains value-based predicates to define the events relevant to the pattern. The **WITHIN** clause specifies a time window over the pattern in which the events are relevant ignoring the ones out of it. It is important for a query to be as accurate as possible so that all complex events of interest are detected and will not be missed.

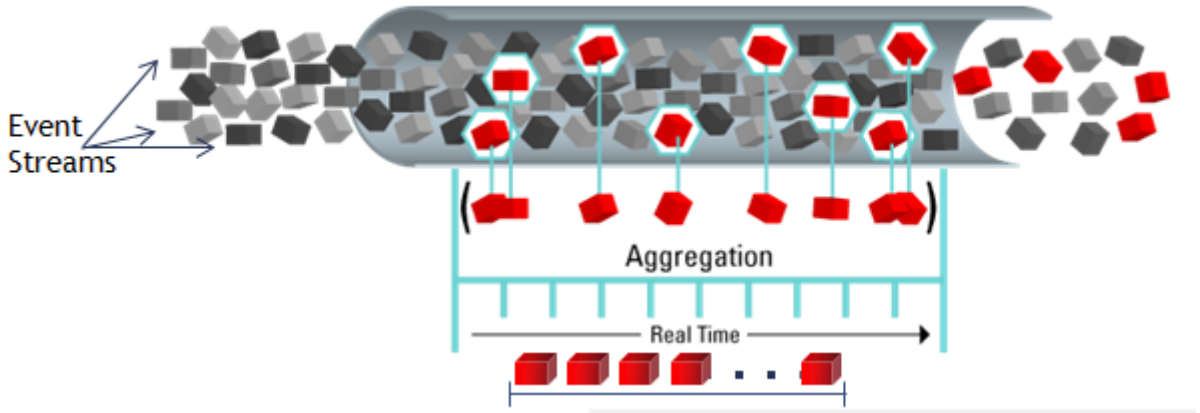

FIGURE 2.1: CEP is aggregating event streams and detecting patterns.

CEP is applied in a wide variety of fields, like health for pre detecting deceases, operational intelligence (OI) to provide insight into business operations by running query analysis against live feeds and event data, financial services for potential fraud detection and more.

2.2 Apache Storm

2.2.1 Project information

Apache Storm is a free distributed real time stream processing computation framework [2], [8]. Initially it was developed majorly by Nathan Marz and the team Back Type in the programming languages Clojure and Java. Storm was released in 2011 as an open source project and currently is used by many companies and organizations like Twitter and NaviSite. It is widely used for a number of reasons:

- It is scalable. Storm topologies run across a cluster of computers in parallel. New machines can be added at any given time in the cluster and can be used for the topology's tasks immediately. Storm offers commands for adjusting the parallelism in the running topologies during processing or hints can be given when setting the components. Storm's inherent parallelism means it can process very high throughputs of messages with very low latency.
- It is fault-tolerant. In cases when machines of the clusters fail then their workers will be assigned to another node and if the workers die the daemon supervisor will restart them. Even if the master daemon Nimbus dies, the workers will still continue to function.
- It guarantees data processing. Storm is able to keep the tuples anchored to its input tuple or more tuples (for joins or aggregations) and track the tuple tree through the topology with "acker" tasks. Any tuples or messages that fail to ack will be replayed.
- It can be used with any programming language. Spouts and bolts can be designed by any language as there are adapters for languages like python, JavaScript, Perl.
- Easy to use. Storm clusters are easy to deploy as there are out of the box configurations suitable for production. There are just three abstractions: bolts, spouts, topologies that are easy and straightforward to program and can be run in local mode for testing.
- It is free and open source. It has a large growing ecosystem of libraries and tools to use in conjunction with Storm including many features and utilities.

2.2.2 Storm Architecture

The architecture of Storm consists primarily of 2 basic types of nodes, the Nimbus and the Supervisor.

Nimbus is a daemon which is run by the master node, it's the central component of Storm and its job is to run the topology, distribute the code around the cluster, assign tasks to the machines of the cluster and monitor for any failures.

Supervisors are daemons which are run by the slave nodes, they listen to Nimbus waiting for workers and tasks to be assigned to them or stop them. The Supervisors communicate with the Nimbus through the distributed messaging framework, the **Zookeeper**, which is the one monitoring and storing the state of the stateless daemons. These can be seen in the Figure 2.2.

Each *machine* in the cluster can run one or more worker processes for one or more topologies. Each *worker process* can run one or more *executors* that are the bolts or spouts of a specific topology. The *tasks* perform the actual

data processing and their number for a component is immutable. This is illustrated in Figure 2.3.

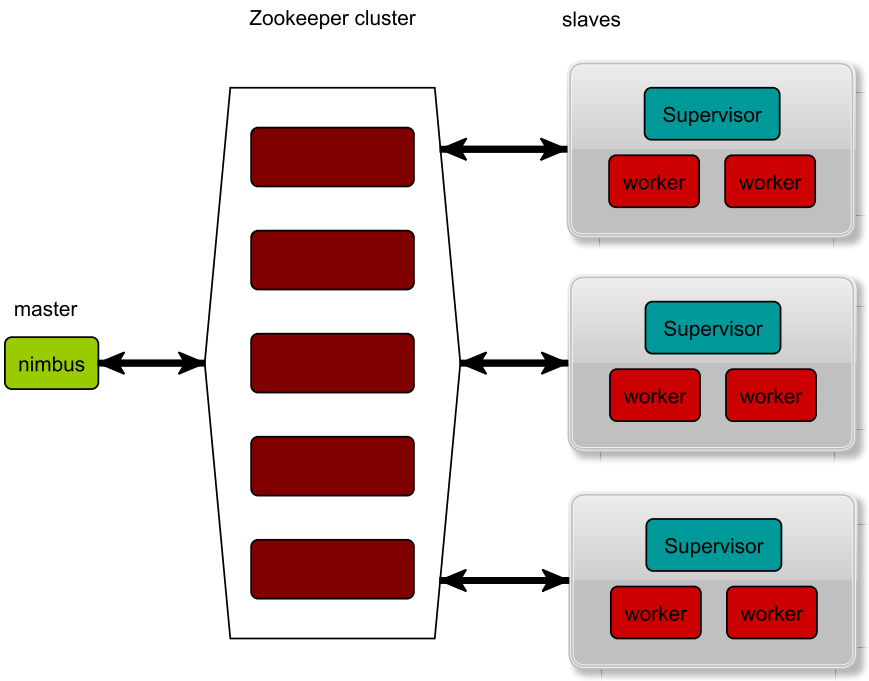

FIGURE 2.2: Master node and slave nodes communicate through Zookeeper

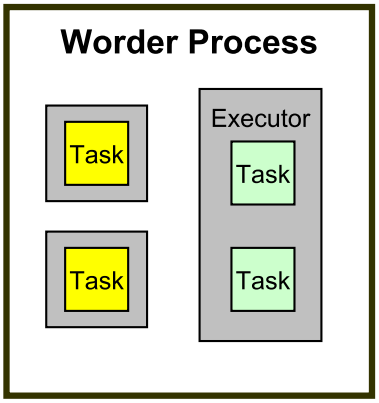

FIGURE 2.3: Tasks and executors of a worker process

2.3 Basic Concepts

Storm's main concepts are the bolts, spouts and streams. In this section these Storm primitives will be discussed.

2.3.1 Spout

A *spout* is the main source of streams in a topology. The spouts are responsible for accepting raw data from external sources like queues, files, streaming APIs etc. and emit them into the topology through one or more streams for processing. Additionally, the spout give each tuple a message id, so that after it has been processed, storm calls the ack () or fail () method passing the message id to identify the tuple it is referring to for replay.

The core interface for implementing spouts is the ISpout. The major methods of the interface that need to be implemented are:

- The **open()** method, which is called once when the spout is initialized in the worker and here the external sources (i.e. distributed file systems, databases, files) are defined.
- The **nextTuple()** method, which is called when Storm requests the next tuple for processing.
- The **ack(messageID)** and **fail(messageID)** methods, which are called when Storm has fully processed or failed to process the tuple with this message ID.
- The **OutputFieldsDeclarer** method, which declares the multiple streams the spout can emit tuples to and the SpoutOutPutCollector, which specifies the stream for the tuples to be emitted to.

2.3.2 Bolt

The *bolt* is the basic processing component in a topology. They take as input the tuples emitted by the streams they have subscribed to and produce tuples as output. The bolts can process the tuples in any way like filtering, transformations, aggregations etc. Then the output tuples can be emitted to other bolts for further processing. New threads can be launched from within bolts for asynchronous processing because the collector is thread-safe. The core interface for implementing bolts is the IBolt. The major methods of the interface that need to be implemented are the following:

- *The cleanup()* method is called when the component is shutting down.
- *The execute()* method is called when a tuple is available after it has been emitted from a spout and is responsible for the processing of the tuple.
- *The prepare()* method is called once when the bolt is initialized in the worker.

Both bolts and spouts can implement the **declareOutputFields()** where the output stream ids and fields are declared.

2.3.3 Streams

A stream is an abstraction in Storm. It is a virtual channel through which the tuples are emitted from spouts or bolts to other bolts of the topology. A topology example using streams can be seen in Figure 2.4. They are defined by the *declareOutputFields()* method with a schema that names the fields of the tuples that are streamed.

The bolts subscribe to the streams they will be receiving input tuples from. In order for the streams to be defined, stream groupings are used. The built-in groupings that Storm offers are:

- Shuffle grouping: Tuples are randomly distributed across the tasks in such a way that each one receives equal number of tuples.
- Fields grouping: The stream is partitioned by the fields specified.
- Partial Key Grouping: It works like the fields grouping in a more efficient way in some situations.
- All grouping: The stream is replicated across all the bolt's tasks.
- Global grouping: The entire stream goes to a single task of the bolt.
- None grouping: It is similar to the shuffle grouping.
- Direct grouping: The producer of the tuple decides which task of the consumer will receive this tuple. A bolt can get the task ids of its consumers by either using the provided TopologyContext or by keeping track of the output of the emit method in OutputCollector (which returns the task ids that the tuple was sent to).

- Local or shuffle grouping: If the target bolt has one or more tasks in the same worker process, the tuples will be shuffled to balance them between the tasks.

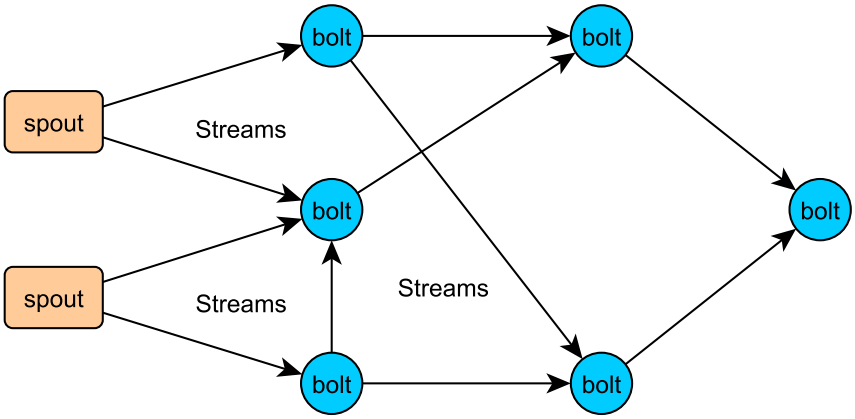

FIGURE 2.4: Example of bolts, spouts and streams in a Storm topology.

2.4 Redis

Redis is an open source NoSql, in-memory key value data structure store which is used as a database, cache and message broker. It has outstanding performance and is highly scalable. A Redis server supports many data structures and features like: transactions, Pub/Sub, Lua scripting. In this project the Publish/Subscribe paradigm is used.

Senders, the publishers, are not programmed to send their messages to specific receivers, the subscribers, but the latter express interest in one or more channels and only receive messages that are of interest without knowledge of what(if any) publishers there are. This feat allows greater scalability and more dynamic network topology and is implemented by SUBSCRIBE, UNSUBSCRIBE and PUBLISH. Lastly, clients may subscribe to glob-like patterns in order to receive all the messages sent to channel names matching a given pattern. [9]

Chapter 3

Related projects

This chapter covers the central CEP engine that will be used in the system, Proton and the Apache Storm implementation of it, which is Proton on Storm.

3.1 Proton

Proton, known as Proactive Technology Online and developed by IBM, is a scalable integrated platform that supports event-driven and complex event processing applications. It is reactive not just to single events but also to situations which are essentially specific patterns of events that take place within a dynamic time window. It enables the application to detect and react to any custom pattern without knowledge of the primitive event, supports various types of contexts and offers a complete event processing operator set like join, aggregate, absence operators. [7]

A Proton project consists of events, consumers/producers, contexts, expressions and event processing agents (EPAs). It receives raw, primitive events and by applying the patterns within a context on the events, the engine generates derived events and takes actions or reports to the EPN consumers if needed. This can be seen in Figure 3.1. The core concepts are described in this section.

3.1.1 Event Processing Network

The event processing network (EPN) is an abstraction which basically is the whole query. It is written in files in JSON format, describe the flow of the events among the EPAs and define the EPAs, the consumers, producers, contexts and events.

3.1.2 Events

The *events* belong to event classes which describe the event structures that the proton engine has to recognize. They enter the engine during runtime carrying information about occurrences in the system's domain. They can have user defined attributes. Additionally, there are the *derived events* that are derived from the EPAs and have the same characteristics as the input events. The built-in attributes of the events are :

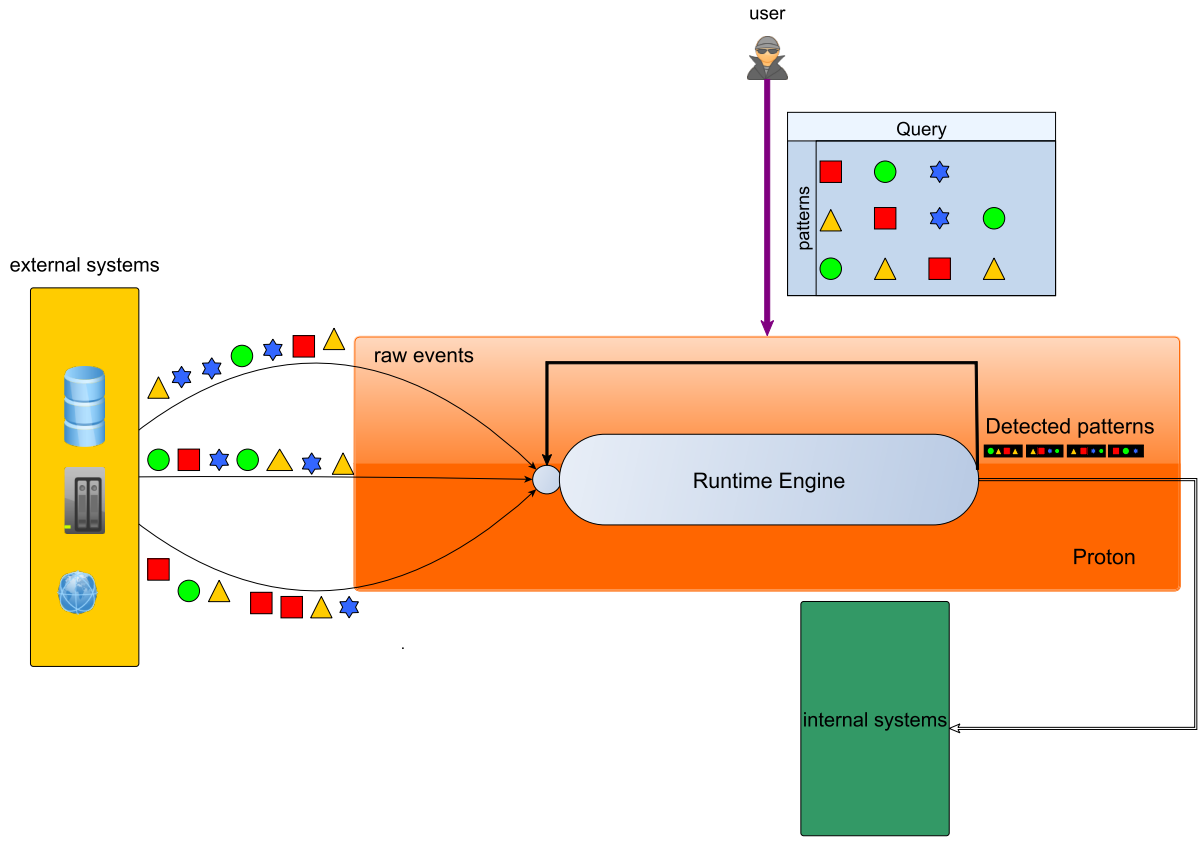

FIGURE 3.1: The core CEP engine: Proton.

- Name: It is the name of the event class and the only one that must be provided. The other attributes can be given default values.
- OccurrenceTime: This attribute of Date type is the time the event occurred and can be assigned by the source.
- DetectionTime: The time the event is detected by the engine and is the count of milliseconds passed since January 1, 1970 UTC.
- Duration: The time interval the event occurred.
- Certainty: The probability that event may occur.
- Cost: The cost of the event. The engine constructs this attribute and sets its value to 0.0 if its omitted or left empty.
- Annotation: An annotation about the event. The default value is an empty string.
- EventId: A string identification of the event. If omitted of left empty, the value to the attribute is set to an auto-generated identifier.
- EventSource: The name of the source of the event. The default value is an empty string

3.1.3 Producers

The producers are responsible for feeding the CEP engine with the events that occur. They have various parameters to customize them according to the needs of applications, i.e. format of the input events, delimiters used to separate the event attributes, polling interval between two input events and more.
A producer definition includes these adapters:

- Files: The events are read from a given file. The file adapters can be set as timed file adapters where the input events enter the system based on their OccurrenceTime attribute.
- REST: The events enter the system through a rest client with GET commands from an external REST service periodically. The REST adapters can be parameterized by setting the URL of the REST service, the ContentType and the PollingMode for batches or single instances of events.
- Custom: There can be custom mechanisms added to the adapter framework in order to read the events.

3.1.4 Consumers

The consumers are responsible for handling the derived events that are generated in the event processing network. Similarly to the producers, they have various parameters to customize them and the consumer adapters include:

- Files: The events are written in a file.
- REST: The derived events are sent to an external REST service with POST commands and can be parameterized by defining the URL of the service, the ContentType and the AuthToken.
- Custom: There can be custom mechanisms for handling the derived events.

3.1.5 Contexts

A context determines when a particular event processing agent is relevant. Each agent can be running multiple context instances simultaneously. There are three types of contexts:

3.1.5.1 Temporal Contexts

A temporal context defines a time window in which the agent is relevant. There are two types of temporal contexts, initiators and terminators.

Initiators The Initiators denote when the temporal contexts start. There can be multiple initiators, so we need a correlation policy which determines what happens in the case a new temporal context is opened when another one is already open with the values "add" or "ignore". They can be one of three types:

- Startup: The temporal contexts are open when the event processing agent is initially defined.
- Event initiator: The event that acts as the initiator of this context.
- Absolute time: The exact time of the temporal context is initialized.

Terminators The Terminators denote when the temporal contexts end. There can be multiple terminators and one or more terminators can terminate one more temporal context instances. They can be one of four types:

- Event terminator: The event that acts as terminator for this context.
- Absolute time: The exact time the temporal context is terminated.
- Relative time: The temporal context ends after the specified has passed since the initiation time of the temporal context.
- Never ends: The temporal context is never terminated. If no terminator is needed this one is chosen.

Additionally, the terminators can be set to terminate or discard a temporal context. If a temporal context is discarded, the event instances that have accumulated during this temporal context are discarded and no detection occurs.

3.1.5.2 Segmentation Contexts

A segmentation context defines a semantic equivalent that groups events which have a set of attributes in common. It can either be an attribute or an expression and has a unique name and a collection of participant events.

3.1.5.3 Composite Contexts

A composite context consists of one or more temporal or segmentation contexts and an instance of a composite context is open if all the specified contexts of it are matched.

3.1.6 Event Processing Agents

The Event processing agents are the entities that are responsible for the pattern detection and the generation of the derived events from the input events, which can be derived events themselves. The EPAs have these characteristics:

- Unique name
- EPA type
- Contexts
- Participating events
- Derived events
- Cardinality and Evaluation policies.

In more detail:

- They have a unique *name* to be identified in the event processing network.
- The core property of every agent is its *operator type*. The operator type of the EPA constructs the pattern based on the input events and can be one of these:
  - Basic operator: This is a stateless operator that works as a filter on the input events and detects the patterns that meet the defined conditions and thresholds.
  - Join operators: The pattern is detected if the participant events arrive in the exact order of the operands (Sequence) or in any order (All).
  - Absence operator: The pattern is detected if the operand events have not arrived during the time window of the context.
  - Aggregation operator: The Aggregate EPA is a transformation agent that applies functions on a number of input events and computes values. These values can be used in the derived events or as a condition in the basic operator.

  - Trend operation: The Trend EPA is an agent that traces values of a specific attribute over time on series of events. It detects increment, decrement or stable patterns among a minimum specified number of event instances and operates only on a single event type.

- In addition, EPAs have their *contexts* which can be temporal, or composite containing a segmentation context.

  As defined temporal context is the time duration in which the patterns or events are relevant, and composite context containing segmentation determines whether the events, that are fitting or not the context, are considered input events. The EPA participant events are partitioned according to the values or expressions defined by the segmentation context.
  The operators basic and absence do not use EPA segmentation contexts since they do not correlate between operands.

- The *participant events* are the input events to the EPA and the operands to the aforementioned operators. They are comprised of some properties which are the **name** of the event, a **condition** which is a filter expression that rejects the events that do not satisfy it and the **consumption policy** which allows the event to be reused or reprocessed later in the same pattern.
  The trend operands and aggregation operands and operators have some more properties. The most frequently used operand property of the aggregation operator is the Instance Selection Policy which decides what happens when multiple events of the same event occur. The quantifier can be set to three options: First, Last or Override.
  If it is set First/Last then only the First/Last event of the operand is selected. If it is set to Override then all previous events are lost and only the last is available, in contrast where the other events are buffered.

- The *evaluation policy* determines when the event is calculated and reported. There are two available modes:

  - In the **immediate mode**, a pattern is detected and reported when a new input event instance occurs (given that the conditions of the pattern are satisfied).
  - In the **deferred mode**, a pattern is detected and the composition process is performed when the context is terminated.

  The operators: all, sequence, aggregation, trend use the evaluation policy attribute.

- The **cardinality policy** determines whether a pattern can be detected once or multiple times in a context. If the policy attribute is set to:

  - **Single**, then the pattern can be detected only once during the context.

- **Unrestricted**, then the pattern is calculated any times its conditions are satisfied during a context.

The operators: all, sequence, aggregation, and trend use the cardinality policy attribute.

3.2 Proton On Storm

Proton on Storm is the standalone Proton, implemented using the Apache Storm primitives. It runs as a Storm topology using bolts and spouts for the queues, the contexts and the EPAs, and constitutes the core CEP engine. The components can be seen in Figure 3.2.

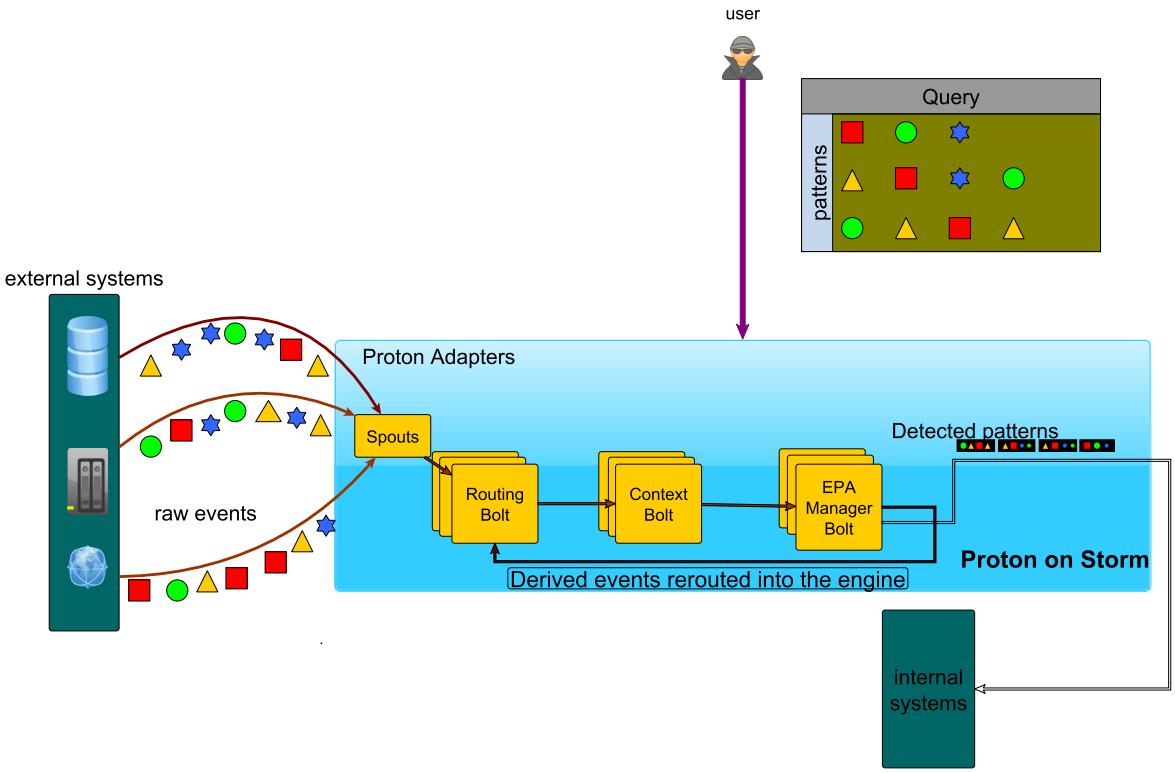

FIGURE 3.2: Proton On Storm architecture

The user defines the producers for feeding the raw input data from external systems and they are translated to the input adapters which direct the data to the **Spouts**.

The **Routing bolt** determines the routing metadata, the context and agent name, of the incoming events and adds them to the event tuple. The Routing bolt runs multiple independent parallel instances. Then the metadata are grouped using STORM options and the tuples are emitted to the context service so that events with same metadata can be processed together.

The **Context bolt** controls the context service of the system. It adds the relevant context partition id to the tuple, manages the lifecycles of the contexts, the initiators and terminators of partitions according to events or timers specified and segmenting the input events into collections so that they are processed in a batch. Then the tuples are field grouped on their context partition and agent name and will be sent to relevant EPA instance.

The **EPA Manager bolt** is the last component of the topology. This is the EPA manager where the EPAs instances are fed the events that are relevant according to their context partition and they are processed for pattern

matching and/or creating derived events, if relevant. The derived events are emitted back to the routing bolt.

Chapter 4

An adaptive complex event processing system over STORM

4.1 FERARI Architecture

FERARI is an architecture which uses the CEP engine Proton On Storm for real time processing of large volumes of event data over distributed topologies in a network. The system needs to be able to process continuous streams of event data in a flexible and scalable way keeping the communication cost to a minimum. In this section, the overall architecture of sites and network is described. [6] [5] [4]

4.1.1 Site Architecture

Each site in the network runs an Apache Storm topology with independent nimbus/supervisors daemons and their own tasks. The topology involves, apart from the Proton On Storm which serves as the CEP engine of the system, other components which ensure the communication required and the proper functionality of operators.
The components of the FERARI topology are presented below. An overview of the topology can be seen in Figure 4.1.

4.1.1.1 Input Spout

The **Input Spout** is the component which feeds the system with input data. This data may come from any source, like a database, a file, HTTP Post method. The Input spout's main job is to receive the information, create the primitive events out of it and emit them to the processing engine. Additionally, it uses the plan generated by the optimizer to build the façade be able to parse the events.

4.1.1.2 PushAndPull spout

The **PushAndPull spout** receives events that are pushed by the other sites of the network through the communicators and then pass them to the event processing engine. The routing bolt of the coordinator site generates pull requests for event types which are sent from the communicator to all relevant sites. The time machines of the sites that produce events of that type receive

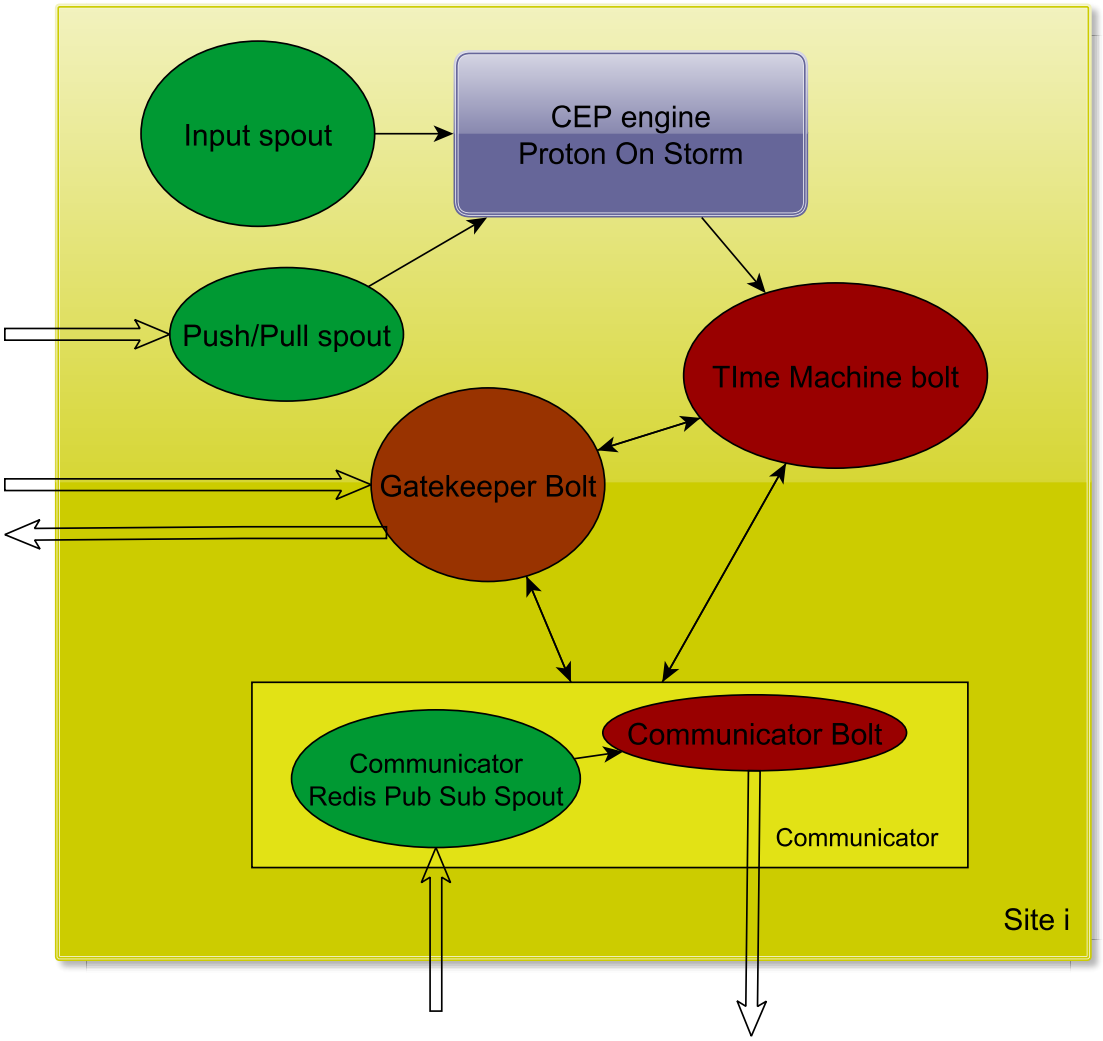

FIGURE 4.1: FERARI topology in a node.

the request and put these types of event in push mode for the requested time window, which is by default infinite.

4.1.1.3 Time Machine

The **Time Machine** bolt's main task is to use the detection timestamps and buffer the tuples for a determined amount of time and according to their type. Every time a coordinator asks a request the node to provide data, the communicator of the node will ask for the tuples from the time machine, if they are available in this time window. When available, it will forward the requested events that took place in the requested timespan to the coordinating site's PushAndPull spout. The Time machine can function in two ways. The first is the "play" way where it buffers the data it receives and saves the time stamp of the last tuple. When a local violation or a global that involves the node occurs, the node enters a violation resolution state. In this state

a "pause" message is sent to the time machine and it functions in "pause" mode sending buffered tuple if they are requested until the violation is resolved. After the violation resolution the time machine gets in "play" mode again. Furthermore, statistics are stored in time machine which can be requested and used by the Optimizer to create new plans for sites. Lastly, in this component are stored values sent from the gatekeeper about the monitoring function.

4.1.1.4 Communicator bolt

The **Communicator bolt** is the component of the system responsible for the communication with the other nodes of the network. Since the Storm installations are independent in the sites, a Redis server is used to provide inter-node communication.

The main job of the communicator is to send and respond to event requests. The other nodes' operators may request events that are relevant and they will be asking for this event data, so their communicators will send pull requests. Additionally, the communicator plays a part in violation resolutions. When a violation occurs in the network, the communicator sends a message and informs the time machine to enter "pause" mode until the violation has been resolved and the communicator messages time machine to enter "play" mode.

4.1.1.5 Communicator RedisPubSubSpout

This is the spout of the communicator. Its job is mainly to wait and listen to specific channels and respond to messages accordingly. The RedisPubSub spout receives the configuration files as well as the plans from the optimizer which are vital for the initialization of the other components of the node

Additionally, it receives the pull and push requests from the other nodes, and messages concerning the coordinator like threshold values, violation resolution or reports. If the node is a coordinator, then the relevant messages are sent to the Gatekeeper bolt.

4.1.1.6 GateKeeper bolt

The **GateKeeper bolt** is the component that takes mostly part in detecting and resolving violations. It observes the monitoring function and if it is out of bounds of the local threshold set by the GM coordinator, then the coordinator is notified through the communicator for resolution. If the node has the GM operator, then it is its job to resolve any local or global violations, compute and reassign new thresholds to the nodes so that the number of violation gets reduced.

4.1.2 Initialization and setup

In this section, the initialization and setup of the system will be described. For the system to start working and processing, it needs to be provided information about the network, the event streams and the query itself. This information is processed by another component of the system, the Optimizer.

Optimizer The Optimizer is a Storm topology on its own. It is installed in a node of the network but it does not affect the functionalities of the rest architecture. It consists of 2 storm primitives, which are Optimizer bolt and Optimizer spout. The optimizer's job is to process the information given to him, calculate and generate plans for each site. This information is given to the optimizer spout externally through a socket, REST service, files. Then it is passed to the Optimizer bolt where the actual methods for plan generation take place and the configuration properties of the CEP engine components are set.

The optimizer needs two kinds of information, information about the network, and information about the queries. These parameters for the optimizer are essential for the plans to be generated.

4.1.2.1 Network parameters

The network parameters are information about the nodes and what type of events they receive. Specifically, they describe the interconnection between the sides of the nodes and the latency between them. In addition, they include the types of the events along with their frequencies in the node. This data is not static and may change while the topology is running, but it is necessary for the initialization and the first optimal plans.
They are inserted in the system through a .csv file and the format is:

`siteName` ;(name of the site); `links` (name of sites the first site is linked) ; (latency between the sites) ; `events` ;(name of the events) ; (frequencies of the events).

An example of a .csv file is:
`siteName` ;cent1;`links`; site4; 14;site1;10;`events`; CallPOPDWH; 0.1;
`siteName`;site1;`links`; site2; 12;site3; 13;`events` ;CallPOPDWH; 0.1;
`siteName` ;site2;`events` ; CallPOPDWH; 0.09;
`siteName` ;site3; `events`; CallPOPDWH; 0.09;
`siteName`;site4; `events`; CallPOPDWH; 0.15;
The graph for the network of the above example is shown in Figure 4.2.

4.1.2.2 Queries

The system needs to know how and what events is going to process. So, the optimizer is fed with a set of queries defined by the user. It comes in JSON format and generally describes the EPN and the EPAs as described in section 3.1. Among others, it contains information about the names of the possible

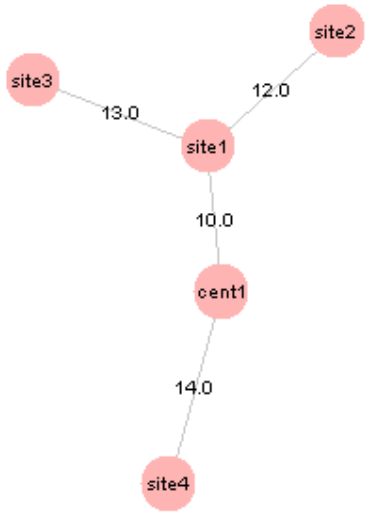

FIGURE 4.2: Network graph.

input events, derived or not, the types of the EPAs, the contexts, the policies, the types and expressions of the derived events.

After the query JSON file and the network parameters are received, the optimizer parses them and breaks down the query into sub queries. Afterwards, the optimizer generates optimal plans for each site to answer these sub queries and sends them to each site separately. The EPN also is checked whether it includes the GM operator. If it does, its parameters and weight vectors are set. The format of an EPN are described in a JSON file and can be seen in Figure 4.3.

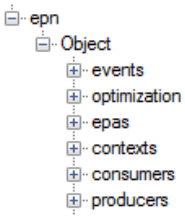

FIGURE 4.3: Format of JSON file.

4.2 An adaptive complex event processing system over STORM

In the previous section, the necessary parameters for setup and initialization of the system were described. Those are queries, the inter-site communication latencies, the names and frequencies of the events for each node. However, these parameters may vary and change while the processing takes place. When this happens, the system needs to adapt to the new parameters while ensuring the smooth and proper functioning of the system.

For the system to start working we need to install and run the Apache Storm topology in each and every one of the sites. For the sites to start processing events, the network parameters and the set of queries from an external source are required. After this information is fed to the system, the optimizer runs the necessary algorithms to generate a set of optimal plans for each site in the network. Afterwards, the optimizer sends the plans to the communicators of all the corresponding nodes of the network and the sites begin processing input events.

While the sites are working and process all the incoming events, a new set of queries could be sent to optimizer, inter-site latencies may change. Additionally, the optimizer may detect a change in the event frequencies the sites are receiving. Every event passes through the routing bolt, so it is chosen to accommodate the method UpdateStatistics, which updates the hashmaps that include the event types and their counters. Then the maps are emitted to the time machine and they become accessible to the optimizer who decides whether new plans are required.

In case these incidents occur, the optimizer has to take into account the changes and adapt to them by generating new plans for the sites. The new plans will be transmitted through Redis to the other sites, which will start initializing and configuring the new topology with the new parameters. Meanwhile the old topology needs to stay active and keep processing the events until the contexts are terminated. Then the old topology is killed and only the new one remains active in the site.

4.2.1 Architecture

The platform, in order to achieve this adapting behavior to new parameters, needs extra functionalities and components. One of them is a new topology with a single storm primitive, called AdaptationSpout. The Adaptation spout can be considered a distinct, cut off from the underlying distributed architecture entity that can communicate through Redis signals with it. It can be seen in the Figure 4.4.

The other component is another topology running in parallel with the FERARI architecture in each site and has a single Storm component as well, called Input spout. This spout's main task is to feed the FERARI topology with the primitive raw data. It receives or reads the data from an external source and then publish it in a Redis channel. The spouts in the active and

potentially multiple FERARI topologies subscribe to these channels and get the event data. It can be seen in Figure 4.5.

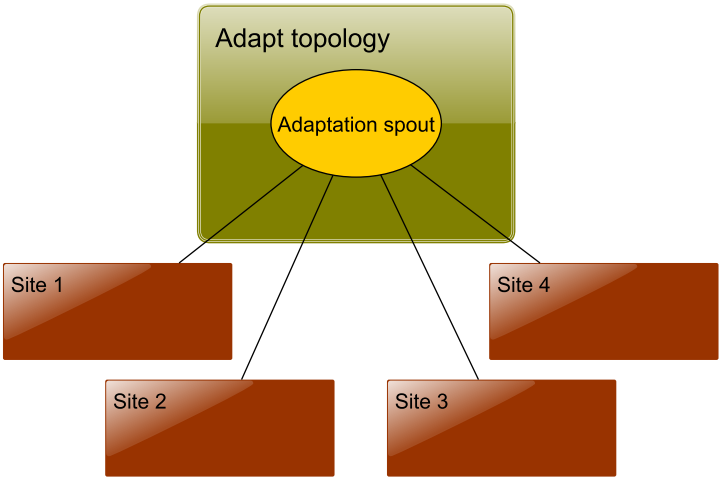

FIGURE 4.4: Adaptation spout communicating with all FERARI topologies.

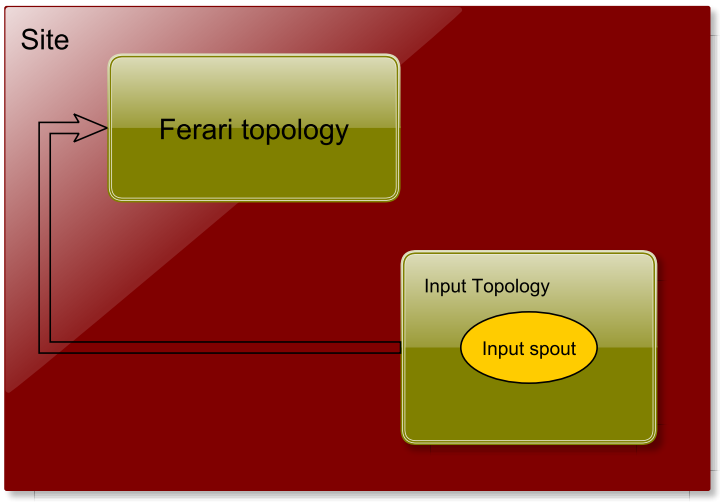

FIGURE 4.5: Input spout sending raw data to the FERARI Topology.

In more detail, the Adaptation topology is the first topology that starts running.

In the open() method, the necessary channels that will be used to communicate with other components of the distributed architecture are defined and initialized. Additionally, the sources of the files containing the network parameters and the set of queries are defined in this method.

The spout waits until the Json file is received and the parameters are set. The spout then parses the network parameters file and finds the central node of the graph, which will accommodate the optimizer. It then begins the process of creating, starting and submitting the two topologies in every site described in the network parameters file.

When the parameters change and new plans have to be generated, the spout is notified and repeats this process. The optimizer in the first topology is passed the parameters, creates the new plans which are transmitted to the sites. When the new topology with the new plans is re ready to run, the adaptation bolt is notified so that it can begin the process of killing the old topology.

It is necessary for the spout to keep track of the active topologies running at sites and the LocalCluster() objects that are used to start and kill them, so there is a structure keeping this data as well as the topology id.

4.3 The adaptation workflow

The work flow of the adaptation process will be described in this section, using a simplified mobile fraud detection scenario. In this example, for the needs of a mobile network several antennas with geographical distance between them have been set up. These antennas represent the sites of the distributed architecture and the nodes in the network graph. Each cell tower occupies a big range in which the customers can make calls using the corresponding antenna. The calls are the input data along with their properties, which involve attributes the calling number, duration, time occurrences, charge amount and their event name is CallPOPDWH. The system is receiving the call events in the input spouts and is trying to detect potential deceitful actions. The flow of actions is as follows.

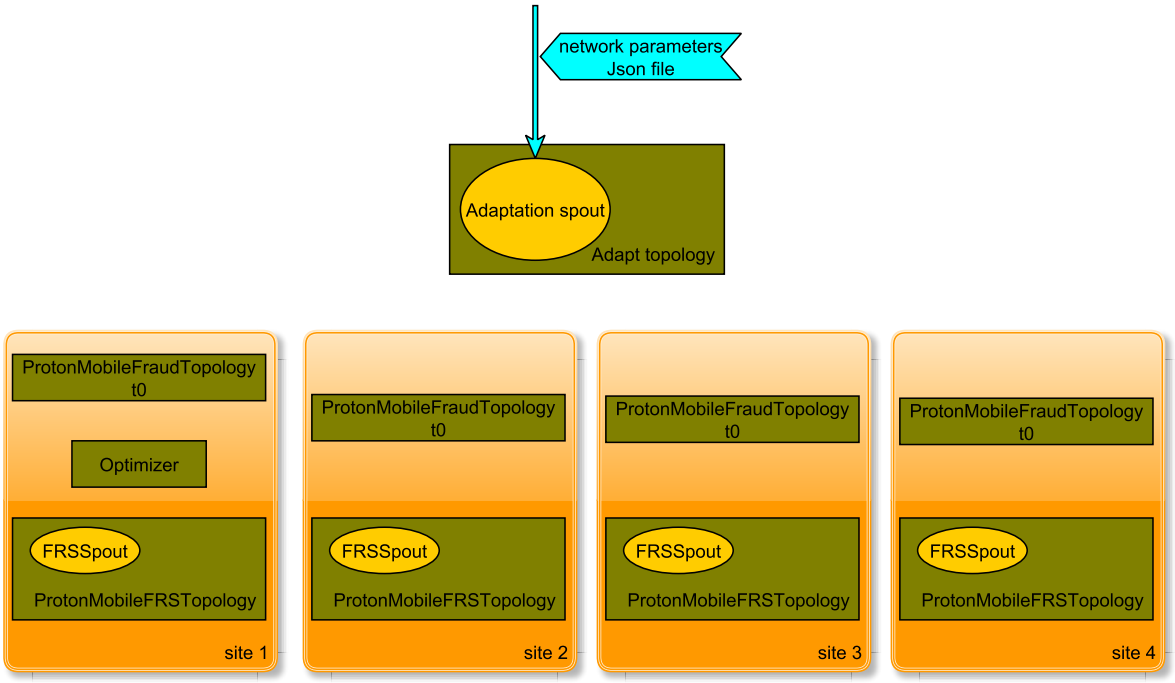

FIGURE 4.6: Network parameters and Json file reaching the Adaptation spout

The adaptation spout is waiting for the network parameters and the Json file. When they are both available, the spout creates the topologies ProtonMobileFraudTopology with topologyID: t0 and ProtonMobileFRStopology at all sites and the structure that keeps the information about the topologies in the clusters is updated.

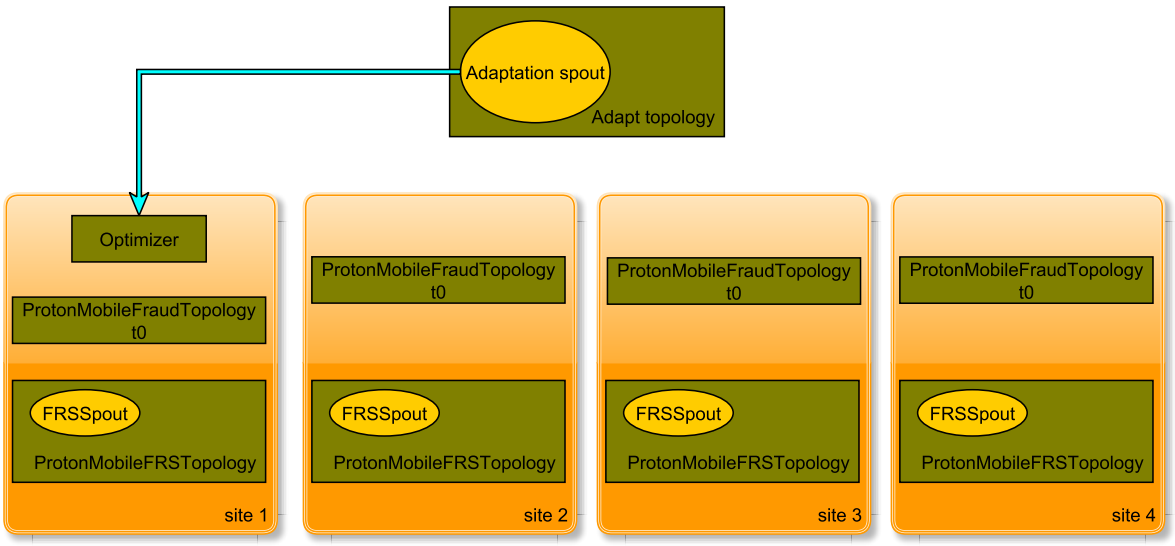

FIGURE 4.7: Adaptation forwards the parameters to optimizer.

Adaptation spout sends the parameters to the optimizer. He parses the EPN and the network parameters and runs a number of algorithms to generate the plans for each site which are written in Json files.

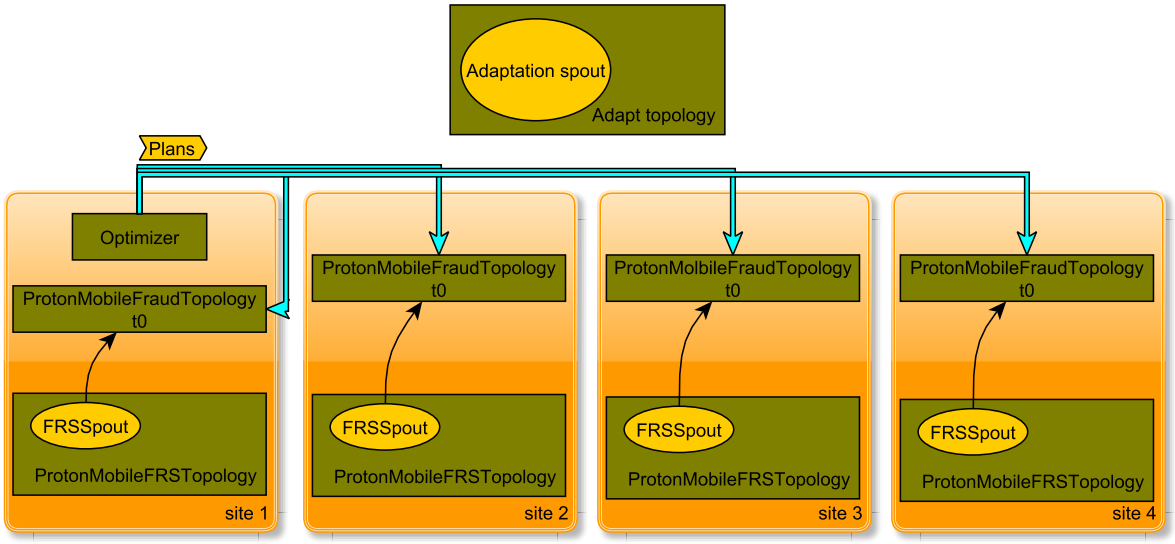

FIGURE 4.8: The optimizer sends the plans to the sites

The JSON files are sent to the communicators of the sites and then forwarded to Proton On Storm. The Routing, Context and EPAManager bolts need them for initialization as the EPAs, Contexts and event definitions are described in them. The files are also necessary to the Input Spout of the ProtonMobileFraudTopology because it needs to know the event definitions.

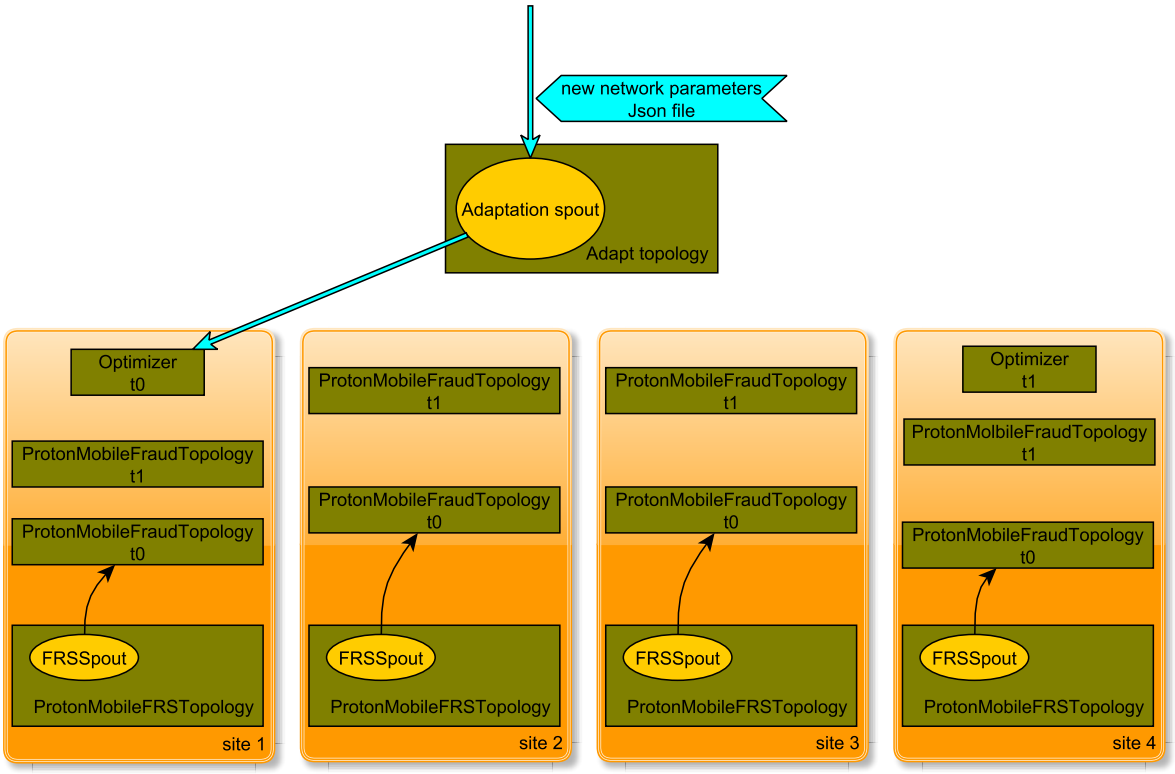

FIGURE 4.9: New parameters reach the Adaptation spout

ProtonMobileFRStopology is receiving the primitive events from external sources and propagates them to the Input spout of the ProtonMobileFraud-topology for process. While the plans and queries are executed, new network parameters or set of queries may be available and new plans need to be generated. The Adaptation spout gets notified, creates new ProtonMobileFraud topologies with topologyID:t1 and sends them to optimizer at topology with topologyID=t0.

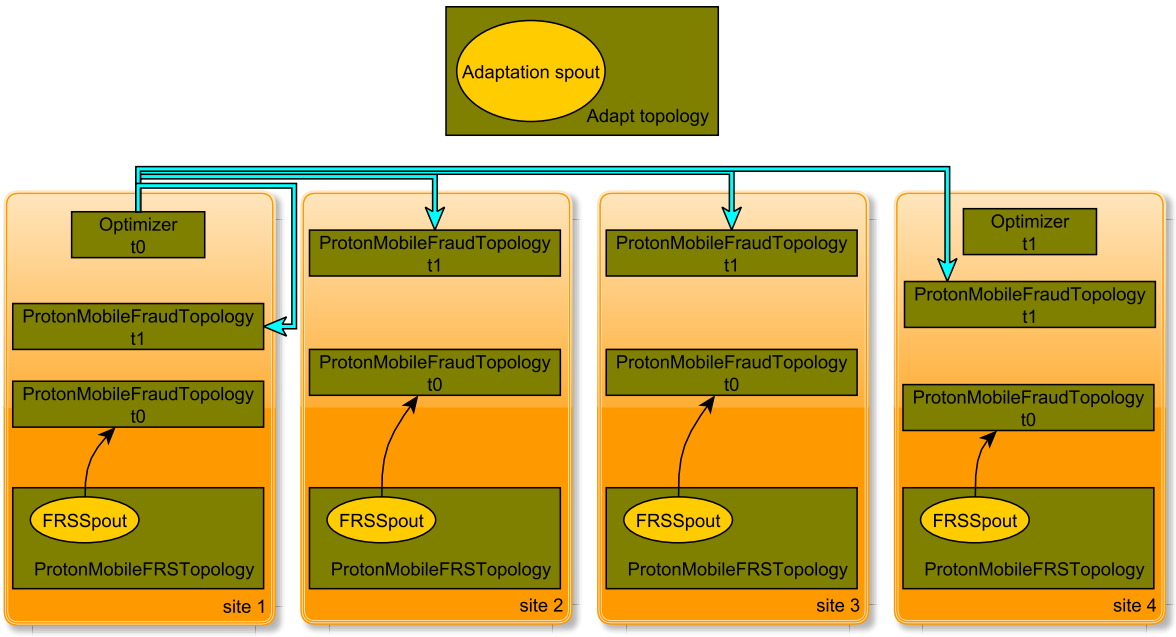

FIGURE 4.10: Optimizer sends the new plans to the new topologies .

New plans are generated and transmitted to the newly created ProtonMobileFraudTopologies and then forwarded to CEP engine's components with topologyID t1.

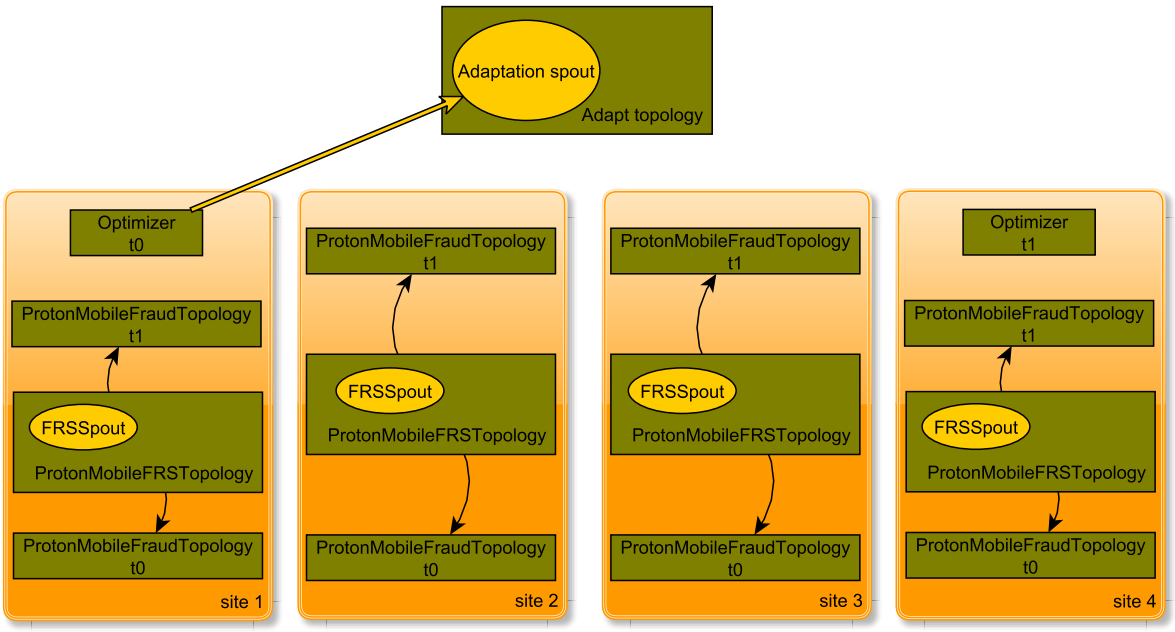

FIGURE 4.11: Adaptation spout is notified about the status of the new topologies.

When t1 is ready and processing adaptation spout is notified. It waits for the temporal contexts of t0 to be terminated. Until the contexts end, the topologies run in parallel in the sites.

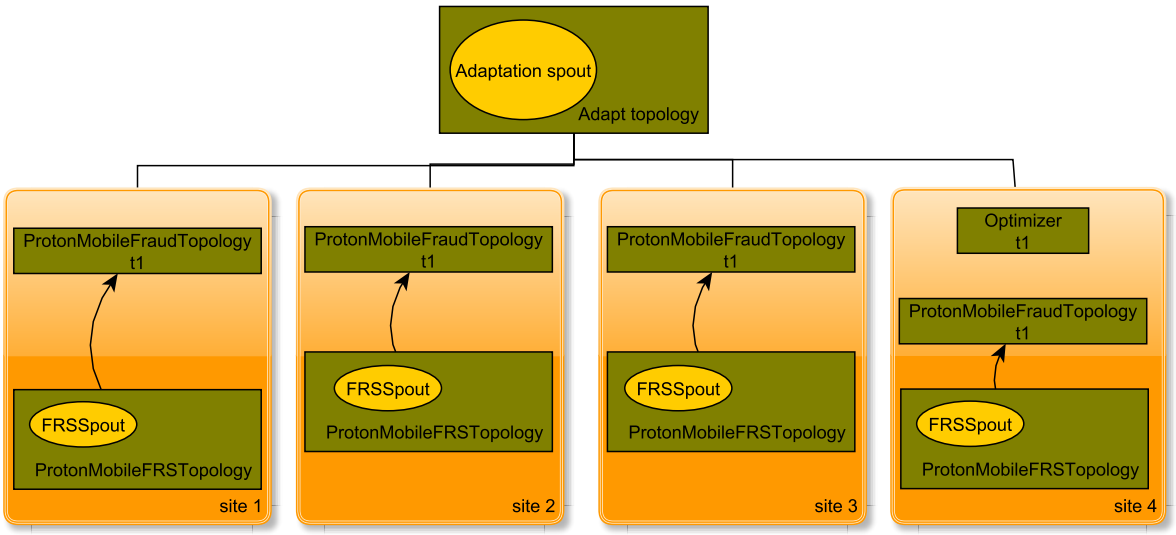

FIGURE 4.12: Adaptation spout kills the old topologies.

When the contexts are terminated the adaptation spout signals the sites to kill the topology with topologyID=t0, leaving the topology t1 running.

4.4 Implementing the adaptive behavior

The system needs to be able to adapt to the new parameters generated by the optimizer. When the new plans are available there may be derived events that haven't been processed, requests from the coordinator may have not been answered yet. For those reasons complex events may be lost, so it is necessary for the system to wait for all the temporal contexts of the first plans to be terminated.

The topologies with different plans have to run in parallel until the contexts are over. For this reason, the classes Routing bolt, RedisPubSubSpout, PushAndPullSpout, Optispout, CommunicatorBolt, CommunicatorState and TimeMachineState that receive messages have one more attribute in their constructor which is their topology id. This parameter prevents inter-node messages from ending up in the wrong topology in the same as this could cause malfunctions at the initialization of the nodes, events to be .

Additionally, a new type message is required which will notify the AdaptationSpout about the state of the newly created topology so that the previous can be killed safely without losing events. The FRSSpout uses the Redis server's features as it utilizes the Publish/Subscribe messaging paradigm to publish the events to channels. In this way every active topology can be receiving all the input raw data.

Lastly, the optimizer can be considered an entity like the Adapt topology, cut off from the system. If necessary, the architecture can be easily modified to keep the optimizer in a single site without the need to kill him, or even create a separate topology for him.

4.5 Results

The Mobile fraud example was executed in the system. In the example, the topology with topologyID=0 and topologyID=2 had node site1 as micro-coordinator and topology with topologyID=1 had node cent1. The results of the fraud example be seen in this section. It can be seen that the system adapts and forces multiple topologies run in parallel for some time before they are killed and allow the last one to keep running.

4.5.1 cent1

cent1 received CallPOPDWH at 1420063200000in topologyID: 0
cent1 received CallPOPDWH at 1420063227000in topologyID: 0
cent1 received CallPOPDWH at 1420063242000in topologyID: 0
cent1 received CallPOPDWH at 1420063246000in topologyID: 0
cent1 received CallPOPDWH at 1420063251000in topologyID: 0
cent1 received CallPOPDWH at 1420063255000in topologyID: 0
cent1 received CallPOPDWH at 1420063278000in topologyID: 0
cent1 received CallPOPDWH at 1456938481000in topologyID: 0
cent1 received CallPOPDWH at 1420063287000in topologyID: 0
cent1 received CallPOPDWH at 1420063290000in topologyID: 0
cent1 received halfExpensiveCalls at 1504638707654in topologyID: 0
cent1 received LongCallAtNight at 1504638707689in topologyID: 0

cent1 received halfFrequentEachLongCall at 1504638707707in topologyID: 0
cent1 received halfExpensiveCalls at 1504638707691in topologyID: 0
cent1 received halfFrequentLongCalls at 1504638707736in topologyID: 0
cent1 received CallPOPDWH at 1420063296000in topologyID: 0
cent1 received CallPOPDWH at 1420063246000in topologyID: 1
cent1 received halfFrequentLongCalls at 1504638708515in topologyID: 0
cent1 received halfExpensiveCalls at 1504638708572in topologyID: 0
cent1 received LongCallAtNight at 1504638708578in topologyID: 0
cent1 received halfFrequentEachLongCall at 1504638708580in topologyID: 0
cent1 received CallPOPDWH at 1420063251000in topologyID: 1
cent1 received LongCallAtNight at 1504638709175in topologyID: 1
cent1 received halfExpensiveCalls at 1504638709175in topologyID: 1
cent1 received halfFrequentEachLongCall at 1504638709176in topologyID: 1
cent1 received halfFrequentLongCalls at 1504638709176in topologyID: 1
cent1 received CallPOPDWH at 1420063302000in topologyID: 0
cent1 received CallPOPDWH at 1420063255000in topologyID: 1
cent1 received CallPOPDWH at 1420063278000in topologyID: 1
cent1 received CallPOPDWH at 1420063310000in topologyID: 0
cent1 received CallPOPDWH at 1456938481000in topologyID: 1
cent1 received CallPOPDWH at 1420063287000in topologyID: 1
cent1 received CallPOPDWH at 1420063319000in topologyID: 0
cent1 received CallPOPDWH at 1420063290000in topologyID: 1
cent1 received CallPOPDWH at 1420063296000in topologyID: 1
cent1 received CallPOPDWH at 1420063323000in topologyID: 0
cent1 received CallPOPDWH at 1420063302000in topologyID: 1
cent1 received CallPOPDWH at 1420063310000in topologyID: 1
cent1 received CallPOPDWH at 1420063333000in topologyID: 0
cent1 received CallPOPDWH at 1420063319000in topologyID: 1
cent1 received CallPOPDWH at 1420063323000in topologyID: 1
cent1 received CallPOPDWH at 1420063343000in topologyID: 0
cent1 received CallPOPDWH at 1420063333000in topologyID: 1
cent1 received CallPOPDWH at 1420063343000in topologyID: 1
cent1 received CallPOPDWH at 1420063363000in topologyID: 0
cent1 received CallPOPDWH at 1420063363000in topologyID: 1
cent1 received CallPOPDWH at 1420063374000in topologyID: 1
cent1 received CallPOPDWH at 1420063374000in topologyID: 0
cent1 received CallPOPDWH at 1420063376000in topologyID: 1
cent1 received CallPOPDWH at 1420063376000in topologyID: 0
cent1 received CallPOPDWH at 1420063384000in topologyID: 1
cent1 received CallPOPDWH at 1420063384000in topologyID: 0
cent1 received CallPOPDWH at 1420063395000in topologyID: 0
cent1 received CallPOPDWH at 1420063395000in topologyID: 1
cent1 received CallPOPDWH at 1420063409000in topologyID: 0
cent1 received CallPOPDWH at 1420063409000in topologyID: 1
cent1 received CallPOPDWH at 1420063409000in topologyID: 1
cent1 received CallPOPDWH at 1420063415000in topologyID: 1
cent1 received CallPOPDWH at 1420063425000in topologyID: 1
cent1 received CallPOPDWH at 1420063428000in topologyID: 1
cent1 received CallPOPDWH at 1420063438000in topologyID: 1
cent1 received CallPOPDWH at 1420063453000in topologyID: 1
cent1 received CallPOPDWH at 1420063454000in topologyID: 1
cent1 received CallPOPDWH at 1420063479000in topologyID: 1
cent1 received CallPOPDWH at 1420063479000in topologyID: 1
cent1 received CallPOPDWH at 1456938482000in topologyID: 1
cent1 received CallPOPDWH at 1420063493000in topologyID: 1
cent1 received CallPOPDWH at 1420063493000in topologyID: 1
cent1 received CallPOPDWH at 1420063453000in topologyID: 2

cent1 received CallPOPDWH at 1420063454000in topologyID: 2
cent1 received CallPOPDWH at 1420063494000in topologyID: 1
cent1 received CallPOPDWH at 1420063479000in topologyID: 2
cent1 received CallPOPDWH at 1420063479000in topologyID: 2
cent1 received CallPOPDWH at 1420063495000in topologyID: 1
cent1 received CallPOPDWH at 1456938482000in topologyID: 2
cent1 received CallPOPDWH at 1420063493000in topologyID: 2
cent1 received CallPOPDWH at 1420063495000in topologyID: 1
cent1 received CallPOPDWH at 1420063493000in topologyID: 2
cent1 received CallPOPDWH at 1420063494000in topologyID: 2
cent1 received CallPOPDWH at 1420063495000in topologyID: 1
cent1 received CallPOPDWH at 1420063495000in topologyID: 2
cent1 received CallPOPDWH at 1420063495000in topologyID: 2
cent1 received CallPOPDWH at 1420063520000in topologyID: 1
cent1 received CallPOPDWH at 1420063495000in topologyID: 2
cent1 received CallPOPDWH at 1420063520000in topologyID: 2
cent1 received CallPOPDWH at 1420063524000in topologyID: 1
cent1 received CallPOPDWH at 1420063524000in topologyID: 2
cent1 received CallPOPDWH at 1420063545000in topologyID: 2
cent1 received CallPOPDWH at 1420063545000in topologyID: 1
cent1 received CallPOPDWH at 1420063557000in topologyID: 2
cent1 received CallPOPDWH at 1420063557000in topologyID: 1
cent1 received CallPOPDWH at 1420063575000in topologyID: 2
cent1 received CallPOPDWH at 1420063575000in topologyID: 1
cent1 received CallPOPDWH at 1420063580000in topologyID: 1
cent1 received CallPOPDWH at 1420063580000in topologyID: 2
cent1 received CallPOPDWH at 1420063583000in topologyID: 1
cent1 received CallPOPDWH at 1420063583000in topologyID: 2
cent1 received CallPOPDWH at 1420063588000in topologyID: 2
cent1 received CallPOPDWH at 1420063588000in topologyID: 1
cent1 received CallPOPDWH at 1420063590000in topologyID: 2
cent1 received CallPOPDWH at 1420063590000in topologyID: 1
cent1 received CallPOPDWH at 1420063604000in topologyID: 2
cent1 received CallPOPDWH at 1420063604000in topologyID: 1
cent1 received CallPOPDWH at 1420063609000in topologyID: 1
cent1 received CallPOPDWH at 1420063609000in topologyID: 2
cent1 received CallPOPDWH at 1420063618000in topologyID: 1
cent1 received CallPOPDWH at 1420063618000in topologyID: 2
cent1 received CallPOPDWH at 1420063627000in topologyID: 2
cent1 received CallPOPDWH at 1420063627000in topologyID: 1
cent1 received CallPOPDWH at 1420063643000in topologyID: 2
cent1 received CallPOPDWH at 1420063643000in topologyID: 1
cent1 received CallPOPDWH at 1420063653000in topologyID: 2
cent1 received CallPOPDWH at 1420063657000in topologyID: 2
cent1 received CallPOPDWH at 1420063669000in topologyID: 2
cent1 received CallPOPDWH at 1420063680000in topologyID: 2
cent1 received CallPOPDWH at 1420063683000in topologyID: 2
cent1 received CallPOPDWH at 1420063709000in topologyID: 2
cent1 received CallPOPDWH at 1456938488000in topologyID: 2
cent1 received CallPOPDWH at 1420063726000in topologyID: 2
cent1 received CallPOPDWH at 1420063735000in topologyID: 2

4.5.2 site1

site1 received CallPOPDWH at 1420063213000in topologyID: 0
site1 received CallPOPDWH at 1420063227000in topologyID: 0

site1 received CallPOPDWH at 1420063243000in topologyID: 0
site1 received CallPOPDWH at 1420063244000in topologyID: 0
site1 received CallPOPDWH at 1456938483000in topologyID: 0
site1 received CallPOPDWH at 1420063246000in topologyID: 0
site1 received CallPOPDWH at 1420063252000in topologyID: 0
site1 received CallPOPDWH at 1420063270000in topologyID: 0
site1 received CallPOPDWH at 1420063285000in topologyID: 0
site1 received CallPOPDWH at 1420063290000in topologyID: 0
site1 received CallPOPDWH at 1456938483000in topologyID: 1
site1 received CallPOPDWH at 1420063290000in topologyID: 0
site1 received CallPOPDWH at 1420063246000in topologyID: 1
site1 received CallPOPDWH at 1420063252000in topologyID: 1
site1 received CallPOPDWH at 1420063293000in topologyID: 0
site1 received CallPOPDWH at 1420063270000in topologyID: 1
site1 received CallPOPDWH at 1420063285000in topologyID: 1
site1 received CallPOPDWH at 1420063293000in topologyID: 0
site1 received CallPOPDWH at 1420063290000in topologyID: 1
site1 received CallPOPDWH at 1420063290000in topologyID: 1
site1 received CallPOPDWH at 1420063329000in topologyID: 0
site1 received CallPOPDWH at 1420063293000in topologyID: 1
site1 received CallPOPDWH at 1420063293000in topologyID: 1
site1 received CallPOPDWH at 1420063345000in topologyID: 0
site1 received CallPOPDWH at 1420063329000in topologyID: 1
site1 received CallPOPDWH at 1420063345000in topologyID: 1
site1 received CallPOPDWH at 1420063359000in topologyID: 0
site1 received CallPOPDWH at 1420063359000in topologyID: 1
site1 received CallPOPDWH at 1420063371000in topologyID: 0
site1 received CallPOPDWH at 1420063371000in topologyID: 1
site1 received CallPOPDWH at 1420063379000in topologyID: 1
site1 received CallPOPDWH at 1420063379000in topologyID: 0
site1 received CallPOPDWH at 1456938489000in topologyID: 1
site1 received CallPOPDWH at 1456938489000in topologyID: 0
site1 received CallPOPDWH at 1420063388000in topologyID: 1
site1 received CallPOPDWH at 1420063388000in topologyID: 0
site1 received CallPOPDWH at 1420063393000in topologyID: 0
site1 received CallPOPDWH at 1420063393000in topologyID: 1
site1 received CallPOPDWH at 1420063397000in topologyID: 1
site1 received CallPOPDWH at 1420063397000in topologyID: 0
site1 received CallPOPDWH at 1420063399000in topologyID: 0
site1 received CallPOPDWH at 1420063399000in topologyID: 1
site1 received CallPOPDWH at 1420063399000in topologyID: 0
site1 received CallPOPDWH at 1420063399000in topologyID: 1
site1 received CallPOPDWH at 1420063403000in topologyID: 1
site1 received CallPOPDWH at 1420063403000in topologyID: 0
site1 received CallPOPDWH at 1420063416000in topologyID: 1
site1 received CallPOPDWH at 1420063416000in topologyID: 0
site1 received CallPOPDWH at 1420063434000in topologyID: 0
site1 received CallPOPDWH at 1420063434000in topologyID: 1
site1 received CallPOPDWH at 1420063443000in topologyID: 1
site1 received CallPOPDWH at 1420063443000in topologyID: 0
site1 received CallPOPDWH at 1420063445000in topologyID: 1
site1 received CallPOPDWH at 1420063445000in topologyID: 0
site1 received CallPOPDWH at 1420063448000in topologyID: 1
site1 received CallPOPDWH at 1420063448000in topologyID: 0
site1 received CallPOPDWH at 1420063453000in topologyID: 1
site1 received CallPOPDWH at 1420063453000in topologyID: 0
site1 received CallPOPDWH at 1420063455000in topologyID: 1

site1 received CallPOPDWH at 1420063455000in topologyID: 0
site1 received CallPOPDWH at 1420063456000in topologyID: 1
site1 received CallPOPDWH at 1456938493000in topologyID: 1
site1 received CallPOPDWH at 1420063443000in topologyID: 2
site1 received CallPOPDWH at 1420063456000in topologyID: 1
site1 received CallPOPDWH at 1420063445000in topologyID: 2
site1 received CallPOPDWH at 1420063448000in topologyID: 2
site1 received CallPOPDWH at 1420063483000in topologyID: 1
site1 received CallPOPDWH at 1420063453000in topologyID: 2
site1 received CallPOPDWH at 1420063455000in topologyID: 2
site1 received CallPOPDWH at 1420063492000in topologyID: 1
site1 received CallPOPDWH at 1420063456000in topologyID: 2
site1 received CallPOPDWH at 1456938493000in topologyID: 2
site1 received CallPOPDWH at 1420063494000in topologyID: 1
site1 received CallPOPDWH at 1420063456000in topologyID: 2
site1 received CallPOPDWH at 1420063483000in topologyID: 2
site1 received CallPOPDWH at 1420063527000in topologyID: 1
site1 received CallPOPDWH at 1420063492000in topologyID: 2
site1 received CallPOPDWH at 1420063494000in topologyID: 2
site1 received CallPOPDWH at 1420063551000in topologyID: 1
site1 received CallPOPDWH at 1420063527000in topologyID: 2
site1 received CallPOPDWH at 1420063551000in topologyID: 2
site1 received CallPOPDWH at 1420063551000in topologyID: 1
site1 received CallPOPDWH at 1420063551000in topologyID: 2
site1 received CallPOPDWH at 1420063564000in topologyID: 1
site1 received CallPOPDWH at 1420063564000in topologyID: 2
site1 received CallPOPDWH at 1420063566000in topologyID: 1
site1 received CallPOPDWH at 1420063566000in topologyID: 2
site1 received CallPOPDWH at 1420063566000in topologyID: 1
site1 received CallPOPDWH at 1420063566000in topologyID: 2
site1 received CallPOPDWH at 1420063568000in topologyID: 2
site1 received CallPOPDWH at 1420063568000in topologyID: 1
site1 received CallPOPDWH at 1420063569000in topologyID: 1
site1 received CallPOPDWH at 1420063569000in topologyID: 2
site1 received CallPOPDWH at 1420063578000in topologyID: 1
site1 received CallPOPDWH at 1420063578000in topologyID: 2
site1 received CallPOPDWH at 1420063587000in topologyID: 1
site1 received CallPOPDWH at 1420063587000in topologyID: 2
site1 received CallPOPDWH at 1420063591000in topologyID: 2
site1 received CallPOPDWH at 1420063591000in topologyID: 1
site1 received CallPOPDWH at 1420063591000in topologyID: 1
site1 received CallPOPDWH at 1420063591000in topologyID: 2
site1 received CallPOPDWH at 1420063591000in topologyID: 1
site1 received CallPOPDWH at 1420063591000in topologyID: 2
site1 received CallPOPDWH at 1420063595000in topologyID: 1
site1 received CallPOPDWH at 1420063595000in topologyID: 2
site1 received CallPOPDWH at 1420063595000in topologyID: 1
site1 received CallPOPDWH at 1420063595000in topologyID: 2
site1 received CallPOPDWH at 1420063598000in topologyID: 2
site1 received CallPOPDWH at 1456938497000in topologyID: 2
site1 received CallPOPDWH at 1420063605000in topologyID: 2
site1 received CallPOPDWH at 1420063607000in topologyID: 2
site1 received CallPOPDWH at 1420063617000in topologyID: 2
site1 received CallPOPDWH at 1420063618000in topologyID: 2
site1 received CallPOPDWH at 1420063619000in topologyID: 2
site1 received CallPOPDWH at 1420063648000in topologyID: 2
site1 received CallPOPDWH at 1420063657000in topologyID: 2

Chapter 5

Conclusion

This thesis is focused on developing an adaptive complex event processing architecture capable of processing large volumes of data over distributed topologies with the ability to adapt to various changes. It is built on the Apache Storm framework using spouts as data sources, bolts for data manipulation and streams for the in-between them communication.

In the thesis we propose the necessary modifications to the existing FERARI architecture in order to support the adaptation of the complex event processing based on FERARI's optimizer. The system is designed to adapt to scenarios when new queries are inserted and significant fluctuations on the event frequencies or inter-node latencies occur.

We introduced a new daemon-like topology that supervises the active topologies in each site and an intra-node topology. These entities enables the parallel processing of multiple topologies, corresponding to different plans that the FERARI optimizer generated, and the timely termination of topologies making, thus, the system even more flexible and scalable.

Bibliography

[1] Jagrati Agrawal et al. “Efficient pattern matching over event streams”. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 147–160.

[2] Quinton Anderson. *Storm real-time processing cookbook*. Packt Publishing Ltd, 2013.

[3] Opher Etzion, Peter Niblett, and David C Luckham. *Event processing in action*. Manning Greenwich, 2011.

[4] Gennady Laventman (IBM) Eliezer Dekel (IBM) Inna Skarbovsky (IBM) Antonis Deligiannakis (TUC) Izachak Sharfman (TECHNION) Tomislav Krizan (PI) Daniel Trabold (FHG) Michael Kamp (FHG) Sebastian Bothe (FHG). “Flexible Event pRocessing for big dAta aRchItectures”. In: *Architecture Definition*. IBM. 2015, p. 51.

[5] Ioannis Flouris et al. “Complex event processing over streaming multi-cloud platforms: the FERARI approach”. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM. 2016, pp. 348–349.

[6] Ioannis Flouris 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*. ACM. 2016, pp. 2093–2096.

[7] IBM Research – Haifa. “IBM Proactive Technology Online User Guide”. In: (2016).

[8] Jonathan Leibiusky, Gabriel Eisbruch, and Dario Simonassi. *Getting started with storm*. " O’Reilly Media, Inc.", 2012.

[9] Tiago Macedo and Fred Oliveira. *Redis Cookbook: Practical Techniques for Fast Data Manipulation*. " O’Reilly Media, Inc.", 2011.

[10] Nicholas Poul Schultz-Møller, 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*. ACM. 2009, p. 4.

[11] Eugene Wu, 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*. ACM. 2006, pp. 407–418.