NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

GRADUATE PROGRAM
COMPUTER NETWORKING

MASTER THESIS

A STORM architecture for Fusing IoT data

Dimitrios A. Zampouras

Supervisor: **Hadjieftymiades Stathes,** Associate Professor

ATHENS

FEBRUARY 2018

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΔΙΚΤΥΩΣΗ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αρχιτεκτονική σύντηξης δεδομένων IoT στο STORM

Ζαμπούρας Α. Δημήτριος

Επιβλέπων: **Χατζηευθυμιάδης Ευστάθιος,** Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΦΕΒΡΟΥΑΡΙΟΣ 2018

MASTER THESIS

A STORM architecture for Fusing IoT data

Zampouras A. Dimitrios
A.M.: M1407

Supervisor: **Hadjieftymiades Stathes,** Associate Professor

EXAMINATION COMMITTEE: **Dimitrios Varoutas,** Associate Professor

February 2018

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αρχιτεκτονική σύντηξης δεδομένων IoT στο STORM

Ζαμπούρας Α. Δημήτριος
Α.Μ.: Μ1407

ΕΠΙΒΛΕΠΩΝ: **Χατζηευθυμιάδης Ευστάθιος,** Αναπληρωτής Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Δημήτριος Βαρουτάς,** Αναπληρωτής Καθηγητής

Φεβρουάριος 2018

ABSTRACT

This master thesis presents a framework built on top of Apache Storm's real-time distributed processing system that enables any experimenter to define abstract topologies from a pool of supported algorithms. Firstly a survey of existing processing engines is presented and then the implementation logic for the design of this framework. This master thesis aims to provide a complete system that is configurable with the use of specific DSL files that can be of use to any experimenter that would like to analyse incoming streams without specific programming knowledge.

SUBJECT AREA: Stream processing

KEYWORDS: YAML topologies, Apache Storm, Contextors, Fusion engine, Apache Flux

ΠΕΡΙΛΗΨΗ

Σκοπός της διπλωματικής εργασίας ειναι η δημιουργία μιας βιβλιοθήκης βασισμένη στο κατανεμημένο συστημα επεξεργασίας ροών σε πραγματικό χρόνο Apache Storm με σκοπό την δυνατότητα περιγραφής γράφων επεξεργασίας με την χρήση ενός συνόλου υποστηριζόμενων αλγορίθμων. Αρχικά αναφέρονται τα σύστηματα επεξεργασιας δεδομένων μεγάλης κλίμακας και στην συνέχεια αποτυπώνεται η προσέγγιση για τον σχεδιασμό της βιβλιοθήκης. Με την χρήση αρχείων περιγραφής ενδιάμεσης γλώσσας YAML στοχεύουμε στην υλοποίηση ενός συστήματος το οποίο θα μπορούσε να φανεί ιδιαίτερα χρήσιμο σε ερευνητές οι οποίοι θέλουν να αναλύσουν ροές δεδομένων χωρίς τις απαραίτητες προγραμματιστικές γνώσεις.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Επεξεργασία ροών δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: YAML τοπολογίες, Apache Storm, Contextors, Fusion engine, Apache Flux

CONTENTS

LIST OF FIGURES

1. BIG DATA PROCESSING SYSTEMS

The vision of an Internet of Things (IoT) has attracted the interest of researches and practitioners aiming to deliver innovative applications that improve aspects of our daily lives. Although, over the past decade, advances in hardware development, sensing capabilities and IoT software architectures have triggered important technical and commercial successes, challenges and opportunities persist as we move towards generic and scalable approaches for composing and interoperating existing IoT functionality. [1]

By 2020, industry analysis estimates that 25 billion devices will be connected to mobile networks worldwide. Interconnected devices brimming with intelligence will monitor and support almost any aspect of our daily lives in order to enhance our interaction with the world around us. More and more houses and cities even adopt the Internet of Things paradigm by creating networks of connected devices measuring temperature, carbon dioxide emissions, electricity consumption and much more.

To cope with the increase of connected devices that will be part of IoT, a new level of wireless internet connectivity will be required. 5G is the next generation of wireless networks and it is envisioned to make devices able to emit at a higher data rate and cellular coverage will expand significantly. While still being in an early age of the 5G revolution, a lot of potential can be expected from such networks in terms of interconnected devices. The higher transmission data rates will enable mobile devices to send more bit of information to other devices, gateways, or cloud based infrastructures. Also the latency of the 5G networks is expected to be just a single millisecond, many times faster than 4G, and the increased reliability factor is something particularly important for industrial and mission critical IoT applications.

Millions of interconnected devices and applications even now are continually generating a large amount of data that has high dimensionality and complex in structure. This kind of data generated from such systems is what we call big data, which heralds the era of massive automatic data collection and analysis. Traditional data analysis and storage techniques are inadequate to cope with the velocity and size of this data. [2]

Big data is best understood when considering some of its properties:

- *Volume* is the most obvious property of big data. Data is being generated every second from a multitude of sources.
- *Velocity* refers to the pace at which data flows into a system. Any user interaction can produce output and information from numerous sources and the rate of flow produced of those sources might be really fast.
- *Variety* that refers to the diversity of the data transmitted from the interconnected devices.

Therefore big data are data sets produced by millions of devices, so voluminous and complex that traditional data processing application software proves to be inadequate to deal with them. Big data essentially challenges the every aspect of data manipulation from its collection, storage and analysis.

Frequently the term big data tends to refer to the use of predictive analytics, user behaviour analytics and all kinds of other advanced analytics methods that are designed to able to cope with extreme datasets.

Many tools exist that address the various characteristics of big data:

- Data processing tools that are used to perform some form of calculation and extract intelligence out of a data set
- Data transfer tools that are used to gather and ingest data into the data processing systems.
- Data storage tools used to store data sets during various stages of processing.

Data processing tools fall into two primary approaches depending on how data is being processed and treated when inserted into a data processing system for analysis:

- *Batch processing*. Technically batch processing is the execution of a series of jobs in a data processing system on a set or group of inputs, rather than a single input [3]. Batch processing is being used more frequently on inputs that we are not interested to analyse urgently, but there have been exceptions to this logic. Batch processing has some benefits mainly derived from its approach on handling incoming data, such as reduced system load(instead of repeating the execution for each input, the execution handles much more), it avoids idling computer resources, and keeps a high rate of utilization. The concept of batch processing is being shown on Figure 1.

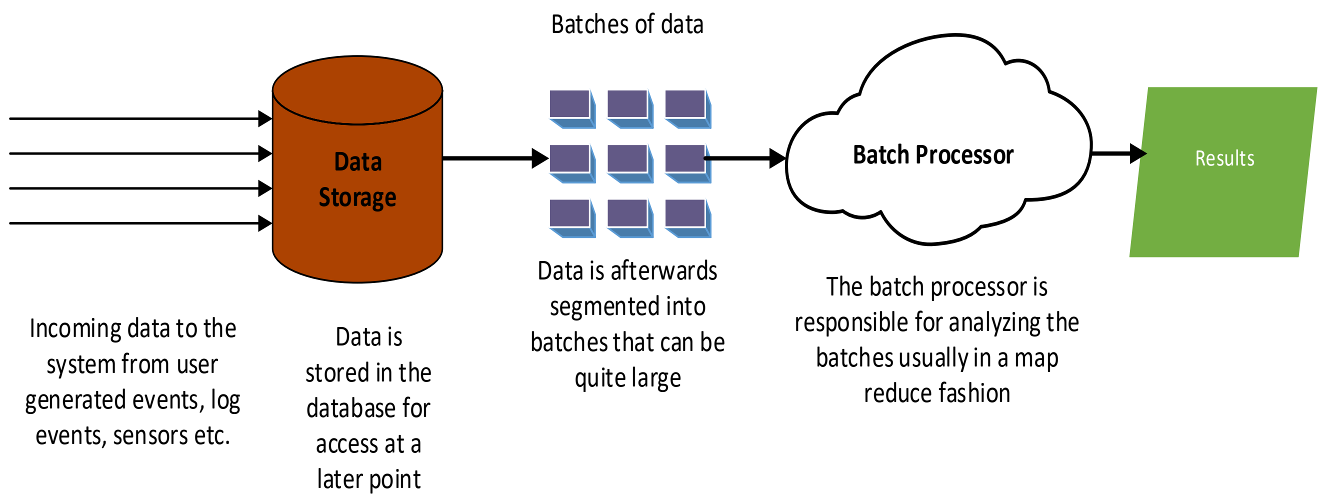
Batches of data
Data Storage
Batch Processor
Results
Incoming data to the system from user generated events, log events, sensors etc.
Data is stored in the database for access at a later point
Data is afterwards segmented into batches that can be quite large
The batch processor is responsible for analyzing the batches usually in a map reduce fashion

Figure 1 Batch processing systems.

- *Stream processing* on the other hand is the exact opposite from batch processing in terms of data handling. A stream processing system continuously acts upon the incoming data and we require immediate availability of the results. A stream processing system sometimes might directly receive input from the sources while in a batch processing system the input is being stored in large files which are then divided into chunks for processing. Therefore unlike a batch processing system a stream processing system is somewhat online and continuous way of analysing the incoming data. However stream processing isn't limited to working on one data point at a time. Windows are being used that contain multiple data points in many cases, which sometimes sacrifices some of the speed and availability of results because it adds an extra layer of latency over working on a single data point. Usually in streaming systems we keep the processing limits within milliseconds, seconds and minutes at most. The concept of most Stream processing systems is depicted on Figure 2.

On the next pages the most known tools for data processing are being presented.

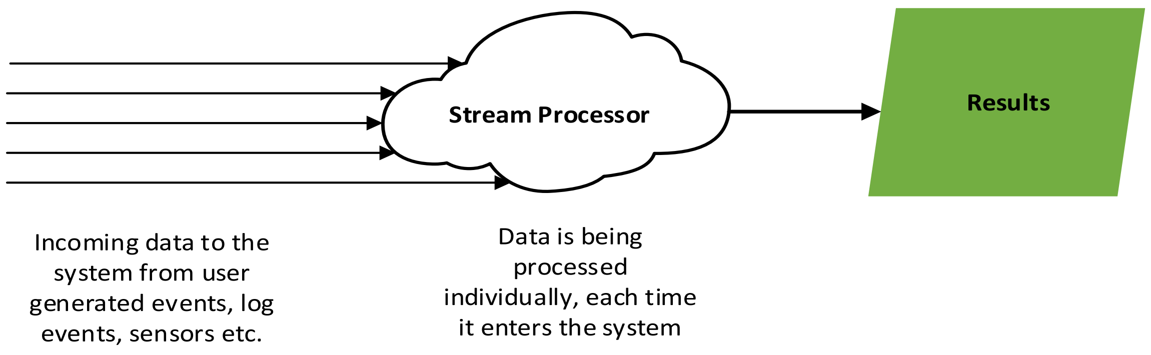
Stream Processor
Results
Incoming data to the system from user generated events, log events, sensors etc.
Data is being processed individually, each time it enters the system

Figure 2 Stream processing systems

1.1 Apache Hadoop

Hadoop is the most known batch processing system. Hadoop is based on a job model called map – reduce. Map reduce is a programming model with an associated implementation for processing and generating large data sets. Any users using batch processing tools and specifically Hadoop specify a map function that processes key-value pairs to generate intermediate key/value pairs and a reduce function that is responsible for merging all intermediate values. Programs written in this functional style are automatically provisioned and executed on large clusters of commodity machines.

One of Hadoop's responsibilities is to take care of the details of partitioning the input data, schedule the execution across a set of machines, handle machine failures and manage the intercommunication between them [4]. Hadoop is designed to scale from single servers to thousands of machines, each offering local computation and storage. High availability is handled by Hadoop's library in order to deliver a highly available service on top of a cluster of computers. [5]

The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS),which is a distributed system able to deliver data with high throughput that handles replication and storage faults. Map reduce systems and Hadoop as well, highly leverage the notion of moving the computation to the data(instead of sending large files through the network) and uses the processing power of the system that has the data stored. Hadoop also uses Hadoop YARN, a framework for job scheduling and cluster resource management.

The basic course of action that Hadoop undertakes is mostly the same with the previous figure. First the incoming data is stored on the distributed filesystem HDFS. Once batches of data are created a MapRedure process runs over each batch. [6]

1.2 Apache Spark

Apache Spark is hybrid processing framework as well. It uses Hadoop's YARN resource manager. Spark uses a system that allows caching of intermediate (or final) results in memory, an ability that is highly useful for processes that continuously and repeatedly run over the same data sets and can make use of the previous calculations stored in memory. Apache Spark ecosystem of libraries is being depicted in Figure 3.

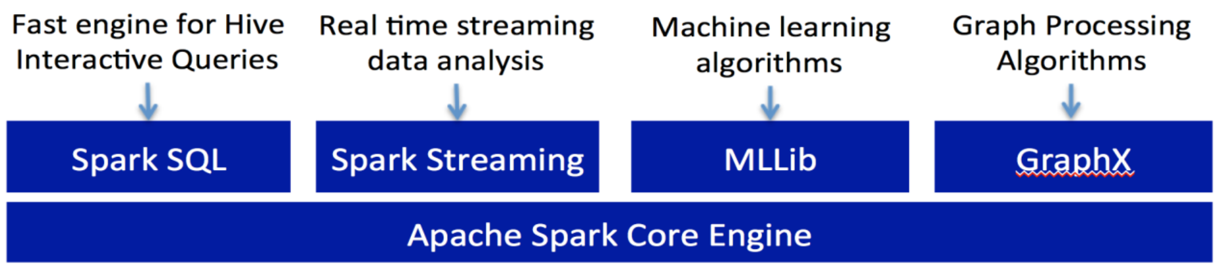
Fast engine for Hive Interactive Queries
Real time streaming data analysis
Machine learning algorithms
Graph Processing Algorithms
Spark SQL
Spark Streaming
MLLib
GraphX
Apache Spark Core Engine

Figure 3 Spark ecosystem

Therefore Spark is speedier that Hadoop because of the way it processes data, via in memory computation and processing optimization. It has a resilient distributed dataset format (RDD) that gives spark the ability to transparently store data in memory and send to disk only what's important or needed [7].

Instead of operating in a MapReduce fashion, it operates on the data set on one fell swoop. The discrete steps of a map reduce job would be first to read the data from the cluster, then perform an operation, write the results perform the next operation etc. Spark, on the other hand, completes the full data analytics operations in-memory and almost real-time. Spark can be deployed in a variety of ways, provides native bindings for the Java, Scala, Python, and R programming languages, and supports SQL, streaming data, machine learning, and graph processing [8][9].

The structure of Spark when deployed in a cluster is shown on the Figure 4. The Driver program as shown is an application executed that is responsible for creating the SparkContext, which is responsible for coordinating the existing multiple client processes. Typically this SparkContext is connected to a Cluster Manager, especially when deployed on a cluster. Spark supports many Cluster Manager types such as the included Spark Standalone, Apache Mesos and Apache Hadoop YARN. The Cluster Manager is responsible for providing executors to applications as soon as a SparkContext has been created [10].

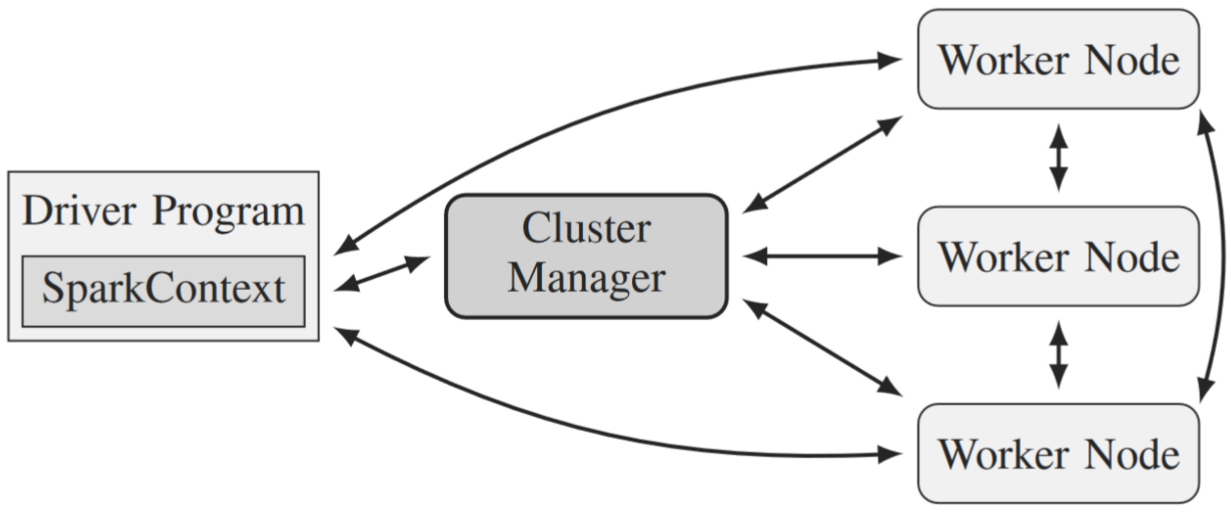

Figure 4 Spark cluster architecture

A worker node in the Spark system is essentially an executor process that is responsible for any computations. Spark also provides a stream processing library by essentially performing micro-batch processing and state management.

Apache Spark is not so similar to Apache Storm real time stream processing system since it approaches each stream, or incoming flow of data as a micro-batch. Therefore it is more similar to Storm's Trident framework.

1.3 Apache Samza

Apache Samza is a distributed real-time stream processing framework that uses Apache Kafka for messaging and Apache Hadoop YARN to provide resource management. It was developed by LinkedIn in order to create a system that would provide quicker results than a Hadoop system, since immediate analysis and computation was needed on the data. Samza's architecture is developed to run on containers for executing jobs [11].

Samza is built on top of Kafka's messaging system and is composed of three basic components: [12] [13]

- *A streaming layer* that is responsible for providing partitioned streams that are replicated and durable
- *An execution layer* that is responsible for scheduling tasks across the machines
- *A processing layer* that is responsible for processing the input stream and applying transformations

The architecture of Apache Samza is shown on the figure below (Figure 5).

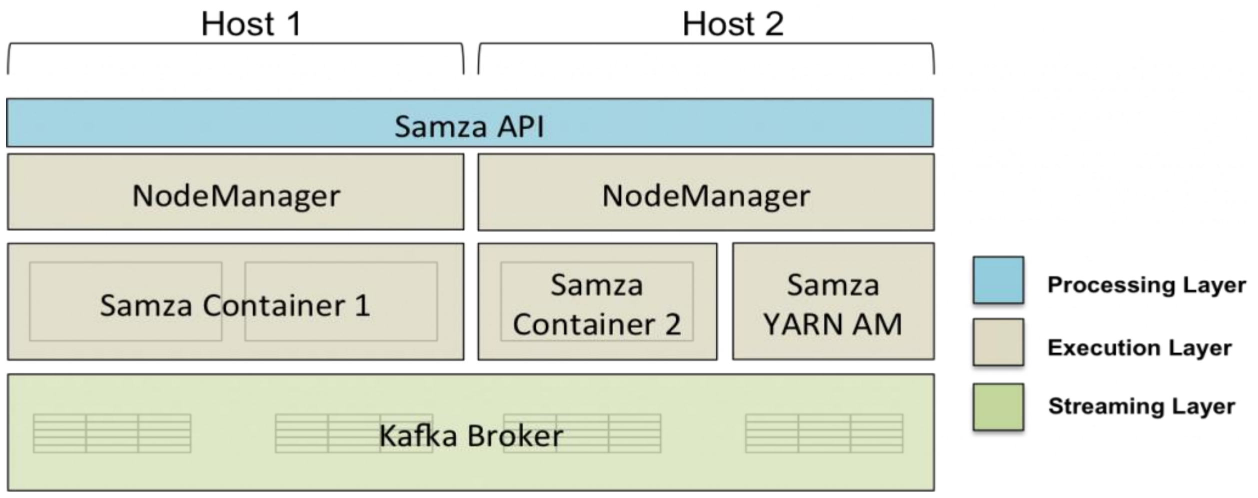

Figure 5 Apache Samza layers and architecture

Apache Samza also provides fault tolerance by restarting any containers that fail and then resuming processing on the streaming data.

1.4 Kafka Streams

Kafka Streams API is a library by Kafka that can perform stream processing on top of Kafka's messaging queue. It builds upon important stream processing concepts such as properly distinguishing between event time and processing time, windowing support, and simple yet efficient management and real-time querying of application state. It provides an API that can do stateful stream processing (not microbatch). The main goal for this library of stream processing was to provide a processing framework without the additional operational complexity of other processing systems.

In Figure 6 the Kafka Streams architecture is being depicted. Kafka Streams is a library that can be used for stream processing but also for handy transformations on the fly in order to emit to new topics, more suitable to the needs of the application [14].

A unique feature of the Kafka Streams API is that the applications you build with it are normal applications. These applications can be packaged, deployed, and monitored like any other application, with no need to install separate processing clusters or similar special-purpose and expensive infrastructure [15].

Much like other processing systems, Kafka-streams use the notion of a topology, or a processing graph with the use of two kinds of processors: the *Source Processor* and the *Sink Processor.* A source processor is a type of stream processor that produces an input stream from one or many Kafka topics by consuming messages from them and forwarding them to the next, down-stream processors.

The Sink Processor is a processor that does not send data to any next processor, but is responsible for sending the received transformed messages to a specified Kafka topic. Essentially a Sink Processor is a terminating node. On Figures 6 & 7 the architecture and a topology graph is being depicted.

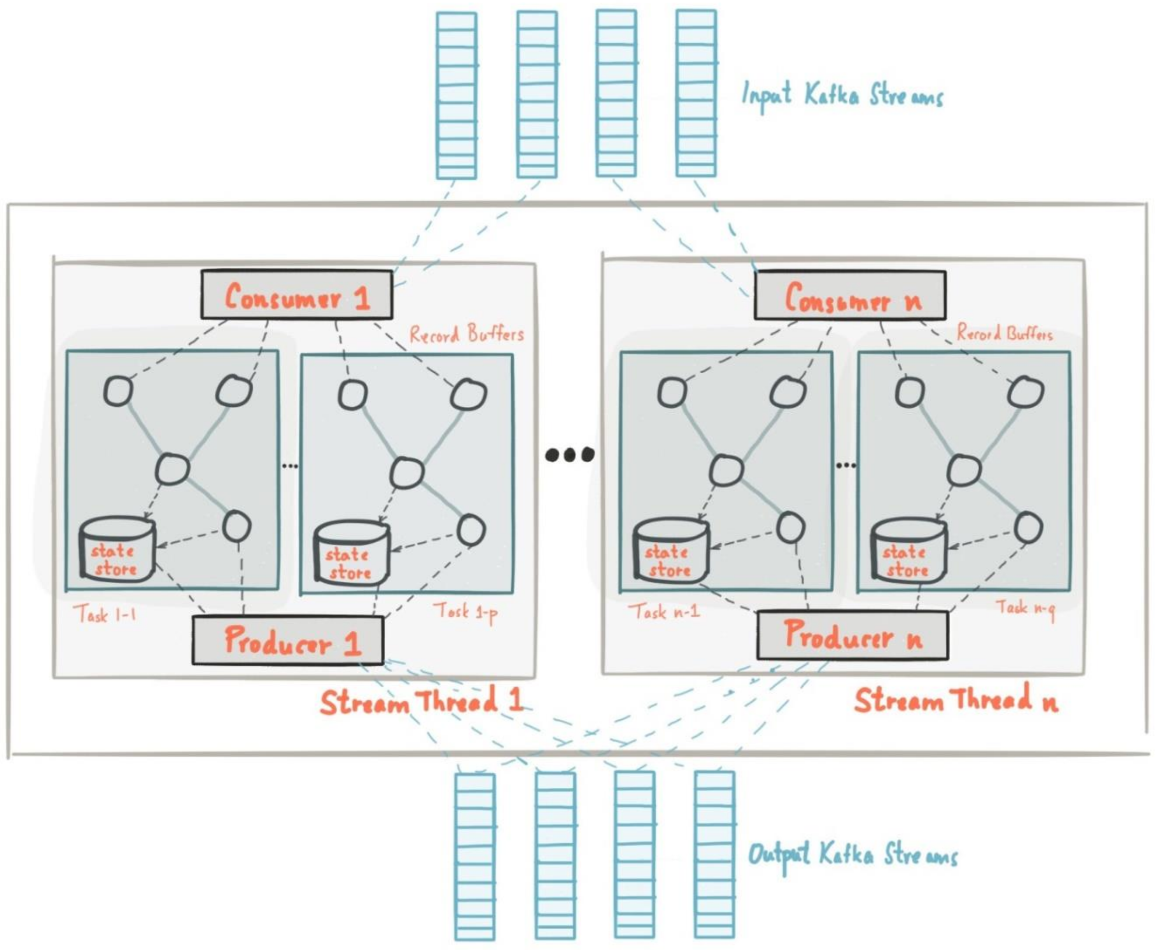

Figure 6 Kafka Streams architecture

Many more extra features are present in this library that essentially can leverage Kafka messaging service to be a stream processing engine that is simple to use and easy to deploy. [16] [17]

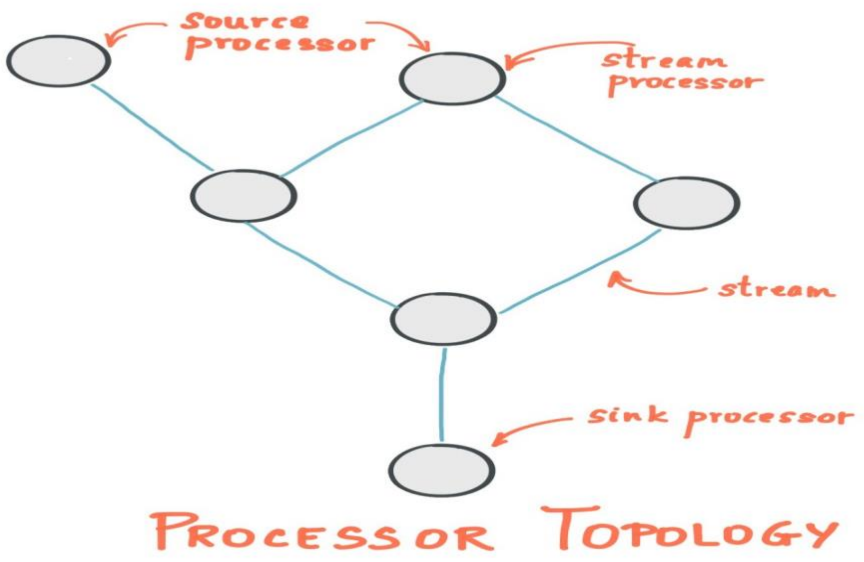

Figure 7 Topology graph

1.5 Apache Flink

Apache Fink is yet another open source system for distributed stream and batch processing that has support for scaling and offers guarantees for an exactly-once-processing of the received inputs. The root of Apache Flink is StratoSphere, an open source research project for big data analytics. It is stateful and fault tolerant and can recover easily from failures and can process large a large volume of information with low latency.

Flink makes use of the master-worker pattern to implement its architecture. It consists of two different element types, a Job Manager, the master, and one or more Task Managers, the workers. The Job Manager is responsible for receiving assignments for clients and is responsible to assign tasks and track the execution state of any of the workers. A heartbeat mechanism is used to ensure that the workers are available and no failures are present. The Task Manager is responsible to execute the tasks assigned by the Job Manager and exchange information between workers as needed. Each Task Manager also provides a number of processing slots to the cluster, which are used for parallelizing tasks. The number of slots can be configured, however it is recommended to use the same slots as the number of CPU cores in every Task Manager machine.

It allows users freely process events from one or more streams, and use consistent fault tolerant state. In addition, users can register event time and processing time callbacks, allowing programs to realize sophisticated computations.

The basic building blocks and Flink's approach to stream processing are *streams* and *transformations.* Conceptually a *stream* is a (potentially never-ending) flow of data records, and a *transformation* is an operation that takes one or more streams as input, and produces one or more output streams as a result [18].

When executed, Flink programs are mapped to streaming dataflows that consist of streams and transformation operators. Each flow starts with one or more sources and ends in one or more sinks. Again, at this system the main concept is the creation of workflow topologies that represent the logic of the application and are depicted as directed acyclic graphs (DAGs). Special forms of circles are permitted with some special constructs. An example topology is being shown at Figure 8.

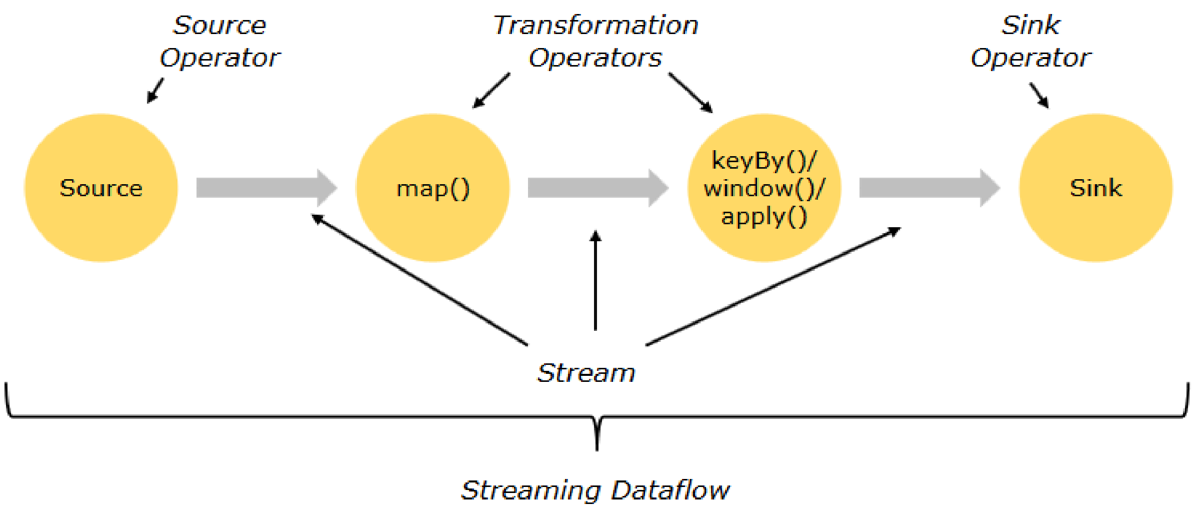

Figure 8 Dataflow topology graph

Apache Flink is really close to Apache Storm and essentially provides stream processing in a similar fashion. Apache Flink has internal support for batch processing which Storm does not.

We continue by describing the Apache Storm stream processing system, its architecture and its components. Apache Storm is currently one of the most prevalent stream processing systems with a quite active community.

2. APACHE STORM

Apache Storm is a free and open source distributed real-time computation system. Storm can handle and process reliably unbounded streams of data [19]. Similar to what Hadoop used to do for batch processing, Apache Storm does for unbounded streams of data in a reliable manner. Storm is exceptionally fast with the ability to process over millions of records per second per node on a cluster of moderate size [20].

Generally Storm is a tool that fits in the big data toolbox and offers a way of processing incoming sources of information using incremental functions, something that most batch big data systems such as Hadoop can't, and offers a variety of fundamental properties that make it an attractive option. First of all Storm can be applied in a wide variety of cases and works well with multiple existing big data technologies. Secondly, it offers scalability since it provides an easy way to break down work using a number of threads, using multiple JVM's only by changing some configuration. Additionally it guarantees that incoming data will surely be processed, and is robust. Last but not least, Storm offers the way to implement many components in many programming languages, can be run entirely on a Java Virtual Machine (JVM) or on a cluster making development on this system easier [21].

2.1 Storm Core Concepts

Each of Storm's internal components is responsible for carrying out a specific task in Storm. The work is delegated to different types of components that are each responsible for a specific processing task.

The main component responsible for delivering data into the platform for processing is a spout. Each spout then passes the data to a component called a bolt, which applies some kind on operation on the incoming data, a transformation in some way. Each bolt can pass the processed information to another bolt for further processing, can stop the line of processing by being the last bolt and presenting the end results or can even store in some database the refined data [22].

The main data type that is being transferred between spouts and bolts, between bolts also is what Storm calls a *tuple*. A tuple is the object that is responsible for encapsulating any data being sent between the components (from now on when referring to the word components of Storm we mean that it's either a spout or a bolt).

Starting from a spout and the following bolts that are chained essentially we describe to storm a logical graph of computation which is the logic of how we would like to process any incoming data by chaining bolts and spouts. Therefore starting from spouts and then connecting them with bolts a directed acyclic graph (DAG) is being created with each node being a component of Storm. This is called a storm topology and is basically the logic of our real-time application [23].

Typically spouts can be found only at the beginning of such graph, since they are responsible for fetching the data into our system. Following are the bolts which as already stated perform some kind of computation on each incoming data. In Figure 9 an abstract topology graph is being depicted with Spouts (valves) and Bolts (water drops).

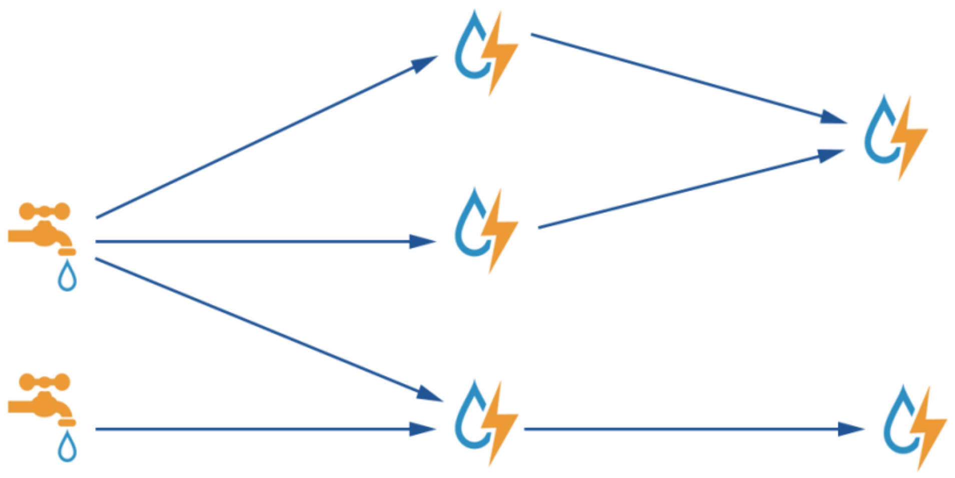

Figure 9 Storm topology - Spouts and Bolts

By packaging and supplying the topology to the Storm cluster, two components of Storm are being used to deploy and run the topology: The master node and worker nodes. The master node is being called Nimbus, whose task is to distribute effectively the code around the cluster, by assigning tasks to each worker node while at the same time monitoring their throughput and failures. Next to the master node, Storm contains a cluster of Apache Zookeeper nodes. Those nodes are essentially offering coordination of the distributed resources and applications as a service to the Storm ecosystem. Zookeeper service essentially knows the state of a topology as well as the state of the master and Supervisor nodes (worker and Supervisor nodes are explained below). Zookeeper also acts as a transmitter of all communication between the other two stateless node types.

Worker nodes run a daemon called Supervisor, which is responsible for executing its assigned task, a portion of the topology. In order to keep an overview about the state of all Supervisor daemons, a heartbeat mechanism is used. Consequently, each Supervisor periodically sends a heartbeat signal to the master node as well as information about possibly free resources. The main task of a Supervisor is spawning worker processes based on the instructions it gets from the Nimbus and checking the condition of the created workers using again a heartbeat mechanism. In case any failure occurs the Supervisor is responsible for restarting the failed worker process.

Each worker spawns a Java Virtual Machine (JVM) and is responsible for executing a part of a topology. Simply put, a topology is not a new concept, is has already been mentioned in previous systems and essentially the idea is the same; a topology is defined as “a directed graph where the vertices represent computation and the edges represent the data flow between computation components”. A topology in Storm can run on many worker nodes across different machines or workers [24].

The system components of Storm are being depicted in Figure 10 below.

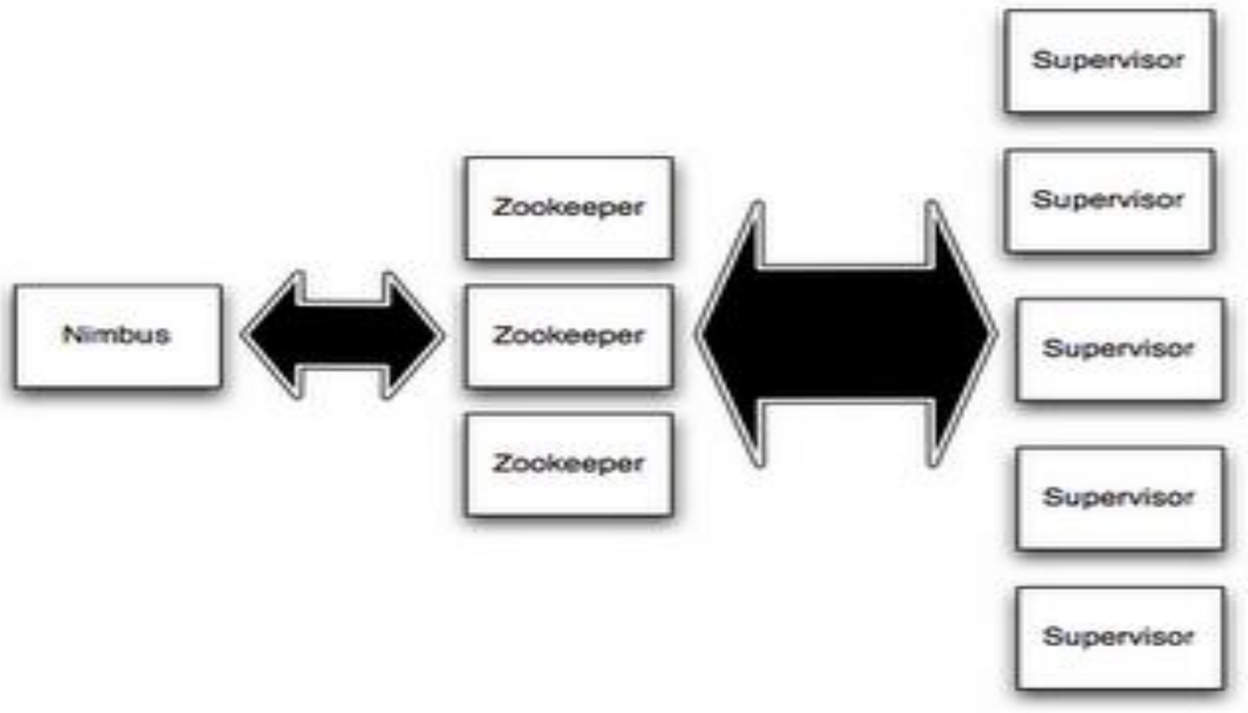

Figure 10 Master and worker nodes

Since Storm offers the ability to replay any failed data that was not correctly processed by the cluster and in order to orchestrate the worker nodes apache Zookeeper is being used for state management. The use of Zookeeper enables the worker nodes to be stateless and this gives them the ability to restart and attempt to overcome any failure without affecting the entire health of the system.

There are mainly two ways we can deploy a Storm topology depending on our needs:

- *Local Mode*, where the entire Storm topology is being run on a single JVM therefore on a single machine. This mode is mainly being used for development and debugging because it provides the easiest way to see all the components of our described topology working together on a single machine.
- *Remote Mode*, where we have to submit our created topology to the Storm cluster, which is then being passed to the Nimbus master node in order to be disseminated into smaller tasks into the worker machines. Remote mode is equivalent to the production mode of our application, meaning that we won't be able easily to check every machine and debug the topology in case of failures, but this deployment will be faster and able to cope with extremely large workloads. It is considered best practice to any system under development to be tested on a local deployment before shipping the application to production.

Both deployment modes are supplementary and needed in order to create faultless and robust applications. Each mode needs different configuration and packaging of the application to be able to run. For example when deploying in local mode the entire Storm library must be packaged alongside the application creating a fat – jar or uber – jar as it is being called in order to run on the JVM of our machine. On a cluster this is not needed. The developer of the application needs to see through this to make it possible, otherwise problems occur.

Following is the description of each component of Storm in much more detail.

2.1.1 Topologies

A topology is essentially a graph of computation where each node represents some individual computation and the edges connecting the graph nodes essentially represent the data being passed between them. This topology is declared by defining the connection for each component participating in the graph and thus forming a directed graph. Connections are being formed between pairs of components(i.e. a spout sending data to more than one bolt will be participating in two connections, two streams by defining a type of grouping) by having each component connected to other components a graph is being created starting from spouts. Figure 11 shows a topology of spouts and bolts.

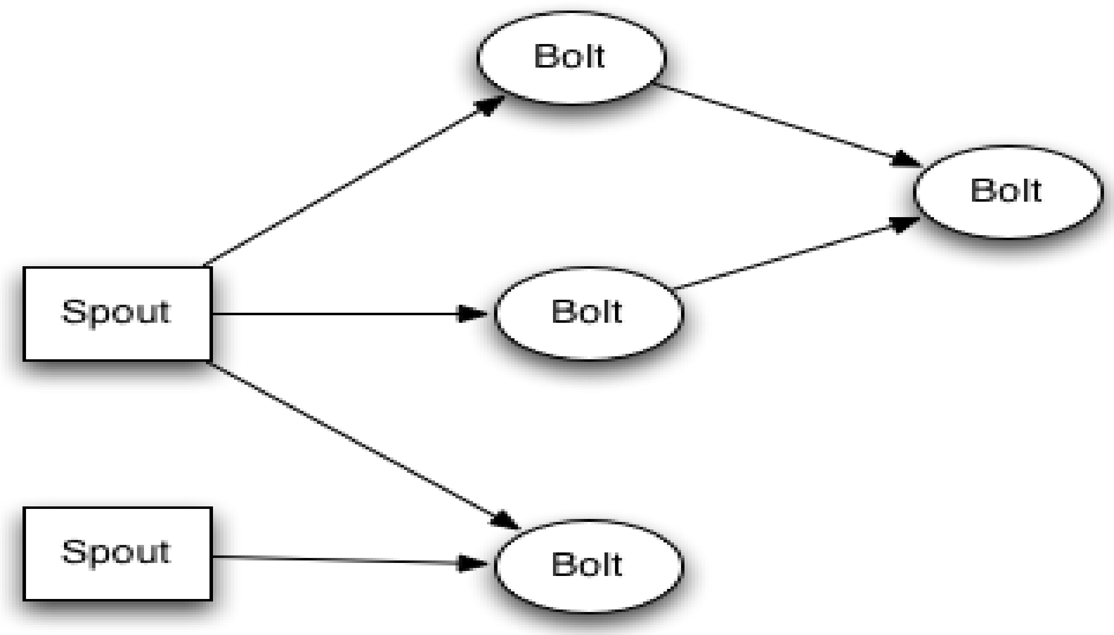

Figure 11 Example of a computational graph - A topology

After the topology has been created in java code as explained above it is then being shipped to two available modes of execution, remote or local. All it is needed to declare a topology is a java main class or essentially any class that can produce an object of apache.storm.Topology. Many aspects of the topology can be tweaked via a configuration object available that is being included in the description of the topology. Using this configuration class the developer can override existing configuration, but also can insert his own objects, and retrieve it at each component of the topology since this configuration object is being passed to every component participating in the computational graph.

2.1.2 Tuples

Each node participating in the topology graph sends data between one another in the form of tuples. When a tuple is being sent from a component to another component we say that this tuple is emitted. A tuple is essentially a wrapper of objects, which defines an ordered list of objects, or values and each value is assigned a name. To put it simply a Tuple is a list that can be accessed as a map as well, but it is not a map structure. Therefore we can access each value by name or by its index. Each element contained in the tuple can be of any type since essentially treats any contained value as a java.lang.Object and Storm provides mechanisms for accessing the values within this list by using helper functions such as *getValueField(String)*, which accesses the list map-like trying to get a value by its fieldName or *getValue(int)* to get a value by its positional index. Extra helper functions exist that resolve to all of the primitive types by casting value such as *getInt(int)*, which fetches the value and casts it to the corresponding primitive contained in the function name. The structure of a Storm tuple is being shown on Figure 12 [25] [21].

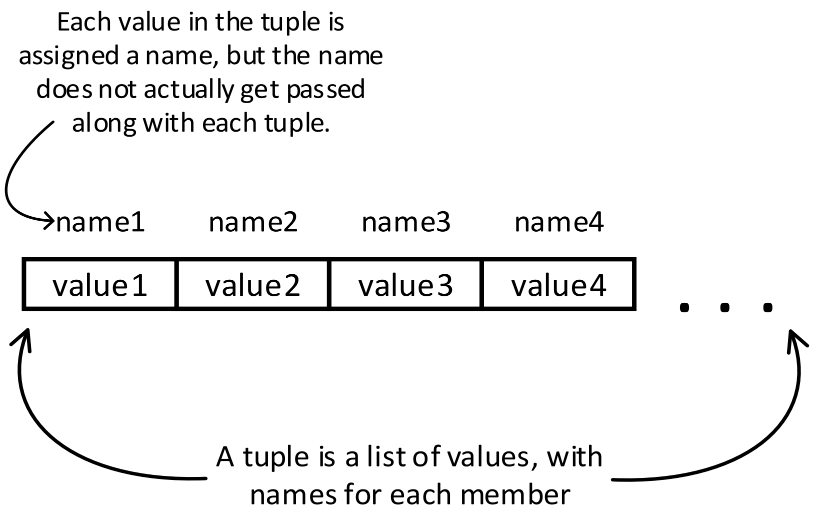
Each value in the tuple is assigned a name, but the name does not actually get passed along with each tuple.
name1
name2
name3
name4
value1
value2
value3
value4
. . .
A tuple is a list of values, with names for each member

Figure 12 A Storm tuple

Since the contained values of the tuple are essentially java Objects are dynamic and they don’t need to be declared. But Storm needs to know how to able to serialize and deserialize each value in order to be able to send them between the nodes of a distributed topology. Primitive types are already supported but for a custom object or type a custom serializer might be needed to be registered. When a custom serializer is needed but is not present Storm falls back to standard Java serializer.

One important aspect of Storm’s tuples is that they are dynamic and there is no need for defining any type. As Nathan Marz, one of the founders of Storm has stated, this technique of dynamic tuples lifts off many of the complexity and annotations needed to type-check each contained value as happens with other big data processing systems.

This is based on the approach that since the creator of the stream is the user and therefore he is the creator of each tuple on every stream, he should know how each tuple defined at each step and what dynamic types and fields does it contain [26].

2.1.3 Streams

The core abstraction in Storm is the *stream*. A stream is an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion. Streams are defined with a schema that names the fields in the stream's tuples. The basic components that are responsible for doing any kind of stream transformations are spouts and bolts.

Stream-wise a spout is responsible for fetching data into the system, and bolts are responsible for applying some sort of computation, but both are also responsible for creating streams, that are the edges of the topology graph and essentially describe the a path from one node to the other. To define a stream all we need is to give it a name or id(otherwise the name “default” will be used) and define which values and fieldnames each tuple emitted by this stream is going to contain. Therefore each component of a Storm’s topology receives one or many streams connections, computes and transforms it and emits to one or many streams.

The declaration of the stream happens on each component with the OutputFieldsDeclarer object which handles new stream definitions with fields and has convenience methods for declaring a single stream without specifying an id. In this case, the stream is given the default id of "default" [26] [23].

2.1.4 Spouts

A *spout* is responsible for fetching data that in the topology. Usually the incoming data would be from external source of information more frequently from a message queue, but the external source and how the data are being fetched into the system is completely left out to the user for implementation. Therefore spouts can get data from many sources and Storm provides some connectors for the most used and frequent sources of external information(for example connectors for kafka, drpc spouts, kestrel queue, feed spout connector etc.). Spouts are not responsible for processing any of the incoming data. They simply act as a source of streams, reading from a data source and emitting tuples to the next type of node in a topology which is the bolt.

Each spout can be configured to emit to many streams as long as we call the method *declareStream* on the on the OutputFieldsDeclarer and specify the stream we would like to emit to and declare the field labels its tuples are going to contain. Emitting each tuple is possible by calling the method *emit* on the *SpoutOutputCollector* object and specifying one of the declared streams. The tuple emitted on each stream should match the number of fields declared for this stream.

The main method on a spout component is the emit method. When this method is called Storm is requesting that the Spout emit tuples to the output collector. This method should be non-blocking, so if the Spout has no tuples to emit, this method should return [27].

2.1.5 Bolts

Unlike Spouts, whose purpose is to fetch the data into the system, bolts are responsible for receiving tuples, performing some kind of computation or transformation and then optionally to emit a new tuple to one or many output streams. Bolts can do anything from filtering, functions, aggregations, joins, communicate with databases and more.

Bolts can do simple stream transformations. Doing more complex stream transformations often requires multiple steps and as a consequence multiple bolts. It is considered best practice when defining tasks for bolts to assign to them a small task rather than more complex ones and create more bolts that handle portion of the task.

Bolts generally can emit to more than one stream. To do so all we need to do is to declare multiple streams with the use of *declareStream* method of the *OutputFieldsDeclarer* and specify to which stream we would like to emit to using the emit method on the *OutputCollector* .

The main method in bolts is the *execute* method which takes in as input a new tuple. Bolts emit new tuples using the *OutputCollector* object like spouts do. Bolts must call the *ack* method on the *OutputCollector* for every tuple they process so that Storm knows when tuples are completed.

There are multiple implementations of Bolt's base class in apache Storm and each one is offering a different type of Bolt behavior depending on its task. Therefore there are bolts that connect to specific types of databases like MongoDB, bolts that make it easier to interact with a Hadoop YARN filesystem and numerous others.

Each of the bolts described so far are bolts that receive one tuple at a time, even when getting tuples from multiple streams, and for each tuple arriving the *execute* method is being called.

There is however the need in such stream processing systems to support windowing since it's one of the most frequently used processing methods on streams of data. Storm provides windowing support so that the user can configure a sliding or tumbling window based on some time or count configuration. The user therefore can specify a windowed bolt that will fire up its execute method only when the window configuration specifies to do so, essentially applying transformations to an array of tuples, those that were received from the previous invocation of the window up until the next. The core structure of the windowed bolts is not the tuple, as was the case with simple bolts. A windowed bolt's execute method receives a *TupleWindow* object that is actually a list of tuples. Again Storm provides helpful functions to support operations within the windowed bolt such as *getNew()* which is a method that returns the tuples that were not in the previous invocation of the window, and other methods as well. What is also really helpful is that a time based window can be configured to measure the time from a timestamp field present in the each tuple arriving in the window and not only on the time that the system received the tuple. This can be used to assert time deltas of the actual data on the stream at the time the timestamp was taken [28].

2.1.6 Stream Groupings

A stream grouping is a way of defining how the tuples are sent between instances of spouts and bolts as shown in Figure 13. As we have already mentioned Storm is a scalable system among many other things and when defining the parallelism for a component there are some cases where we would like to differ our way of emitting tuples so that the policy for sending tuples might change as we see fit. For example if we would like to count the occurrence of a word in a specific bolt of our topology and we would like many instances of the bolt there are two approaches we might think of.

One would be to have every bolt instance count the occurrence of each word but since there are many instances of each bolt and each instance of the bolt receives many words, there might be counts of the same word on different instances. So after we count the occurrences of the same word for each bolt instance, then we should give all those counts of the same word to someone who will add them up producing the final count of the word. This is much like a map reduce job. The policy or grouping of sending tuples to each instance of a component at random is being called *shuffle grouping*. This type of grouping is used whenever we don't care of who the recipient might be because each instance of the bolt is able to perform the job. Using a shuffle grouping will guarantee that the tuples will be distributed randomly to the recipients guaranteeing that each instance will receive a relatively equal number of tuples.

Another approach would be to keep sending any occurrence on the same word to only one recipient. When receiving a word we haven't seen before we could randomly choose one instance to receive this tuple but from now on any subsequent same words that will show up are going to be sent to the same instance of the bolt. This type of grouping is called fields grouping. Using this approach the presence of a reducer, someone who would receive all the word counts in order to sum them up is not needed. This type of grouping is very useful in situations like these.

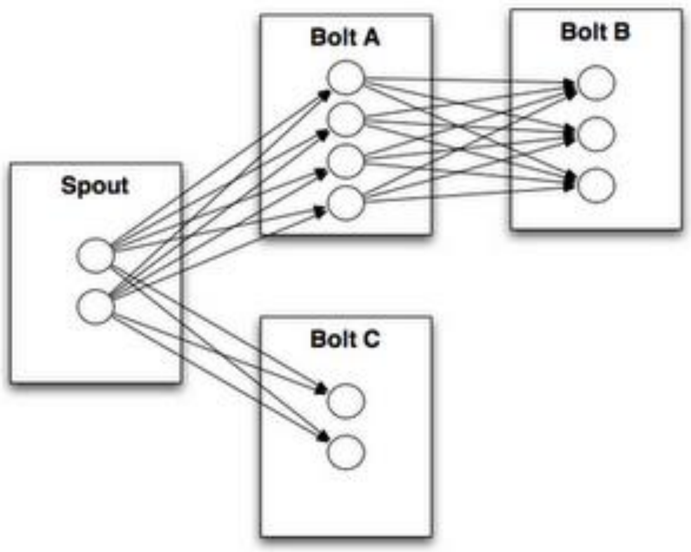

Figure 13 Shuffle grouping and parallelism in Storm

Other types of grouping exist covering many situations where a different approach is needed when emitting tuples from one component to another. These are listed below:

- Fields grouping**,** where he stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, but tuples with different "user-id"'s may go to different tasks.

- *Partial key grouping*, where the stream is partitioned by the fields specified in the grouping, like the Fields Grouping, but are balanced between the downstream bolts, which provides better utilization of the resources especially when the incoming data is skewed. By using this type of grouping in certain scenarios, we could improve throughput and latency and even achieving reduced imbalance by orders of magnitude and in some cases even to 45% [29].

- *All grouping*, where the stream is replicated across all the bolts tasks.

- *Global grouping*, where the entire stream goes only to a single one of the bolt's tasks(the one with the lowest id)

- *None grouping*, when we don't care how the stream is group which is equivalent with the shuffle grouping with plans to modify it so that bolts with none groupings wiil execute in the same thread as the bolt or spout they subscribe from (when possible).

- *Direct grouping*, where the producer of tuples of the stream that is grouped this way, decides which task of the consumer will receive this tuple. This is only available for "direct" streams, streams that we have not given them any name. This kind of grouping requires the use of different emit methods such as *emitDirect*. A bolt can get the task id of its consumers by using the topology context.

- *Local or shuffle grouping,* where if the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. Otherwise,it acts like a normal shuffle grouping [23][30].

2.1.7 Parallelism in Storm

Storm has support for scaling the components of a topology and this is one of the great advantages it provides. When designing a topology Storm offers many ways you can tune the scaling options it provides. In any case if we choose not to change any of the turning knobs of scaling the topology will still work, but it will run more like in a linear way. This could be desirable in some cases, not any application of Storm should be scaled, after all this is dictated by our workload. In any case if the incoming streams send a really large volume of data, then scaling is essential for handling all those data.

2.1.8 Parallelism Hints

Storm offers when declaring a component as part of a topology the ability to provide a hint, on how many instances of this component should the Storm consider to create.

```
builder.setSpout(“word-spout”, new WordSpout(), 4);
```

While the *setSpout* method can be called with only two parameters, when we would like to indicate our level of parallelism we should call the same function with an extra argument, a number, indicating how many same instances of this component we would like Storm to create for us. Otherwise the default would be one. The same parallelism hint can be provided in the bolt declaration process:

```
builder.setBolt(“count-bolt”, new WordCount(), 10 );
```

It is evident that the more instances of a bolt we provide, the more quicker and efficient our topology will be when handling increased workload. At the same time the complexity of our application increases and more intricate grouping other than shuffle grouping might be required. It is always important to have in mind, especially when designing a topology, that it might need to scale at some point and even when we have indicated to

Storm that we would like only one instance per component, we should always consider that this might change, with the simple addition of an extra number. Therefore it is important to always have in mind that each component should be able to work well in a parallel fashion.

2.1.9 Executors and tasks

Up until now we have discussed about components and how can they be scaled using the parallelism hint. However these components have to run somewhere and those are the worker nodes. The worker node as we have stated is responsible for executing a portion of the topology, some subset of the spouts and bolts, which executes on its JVM. A worker node is responsible for running executors which are essentially a thread of execution on the JVM. Executors therefore are threads that are being spawned by a worker process. Keep in mind that worker nodes might also be executing other executors that belong to other topologies deployed in the storm cluster. Each worker node is not necessarily using all of its resources for executing a portion of one topology deployed. Figure 14 further explains this.

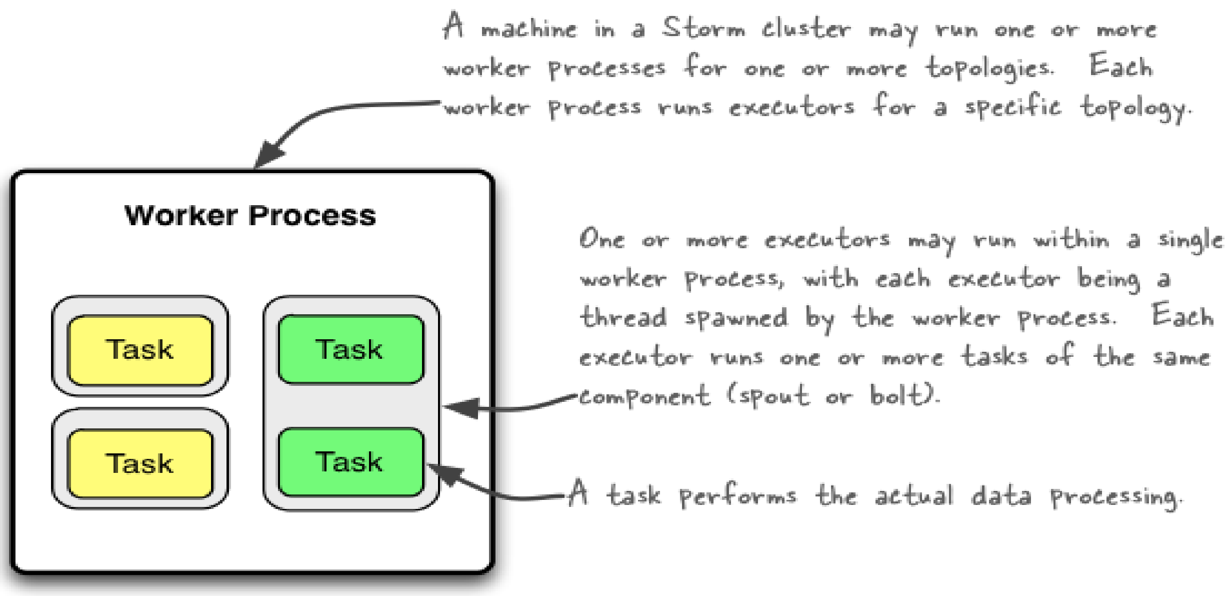
A machine in a Storm cluster may run one or more worker processes for one or more topologies. Each worker process runs executors for a specific topology.
Worker Process
Task
Task
Task
Task
One or more executors may run within a single worker process, with each executor being a thread spawned by the worker process. Each executor runs one or more tasks of the same component (spout or bolt).
A task performs the actual data processing.

Figure 14 Worker processes, executors and tasks

Tasks are essentially the instances of spouts and bolts running within a thread of execution, on the executor. Therefore tasks perform the actual data processing. Each component implemented in the topology code is being executed as many tasks across a cluster. The number of tasks for a component is always the same throughout the lifetime of the topology, Storm doesn't support auto scaling of tasks, but the number of executors (threads) might change over time. At almost any time the number of executors will be less or equal to the number of tasks. By default the number of executors and tasks is set to be the same (Storm will run one task per thread) but this can be varied. [31][21][32]

When specifying the parallelism hint we actually vary the initial number of executors of a component.

To sum up what we have stated so far:

- Each worker process can run one or more executors, essentially threads, where each executor is a thread spawned by the worker process
- Each executor runs one or more tasks of the same component

We can increase the number of workers (JVMs) by specifying it on the configuration class provided by Storm:

```
TopologyBuilder builder = new TopologyBuilder();
Config config = new Config();
config.setNumWorkers(2);
…
```

Adding new workers comes at a cost: additional overhead for a new JVM. This example adds an additional worker without additional executors or tasks, but to take full advantage of this feature, Storm developers must add executors and tasks to the additional JVMs [33]. We can increase the parallelism by increasing the number of executors with the parallelism hint:

```
Config config = new Config();
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(MY_SPOUT_ID, mySpout);
builder.setBolt(MY_BOLT1_ID,myBolt1,2)
        .shuffleGrouping(MY_SPOUT_ID);
builder.setBolt(MY_BOLT2_ID,myBolt2)
        .shuffleGrouping(MY_SPOUT_ID);
```

This code sample assigns two executors to the single, default worker for the specified topology component, MyBolt1, as the following figure(Figure 15) illustrates.

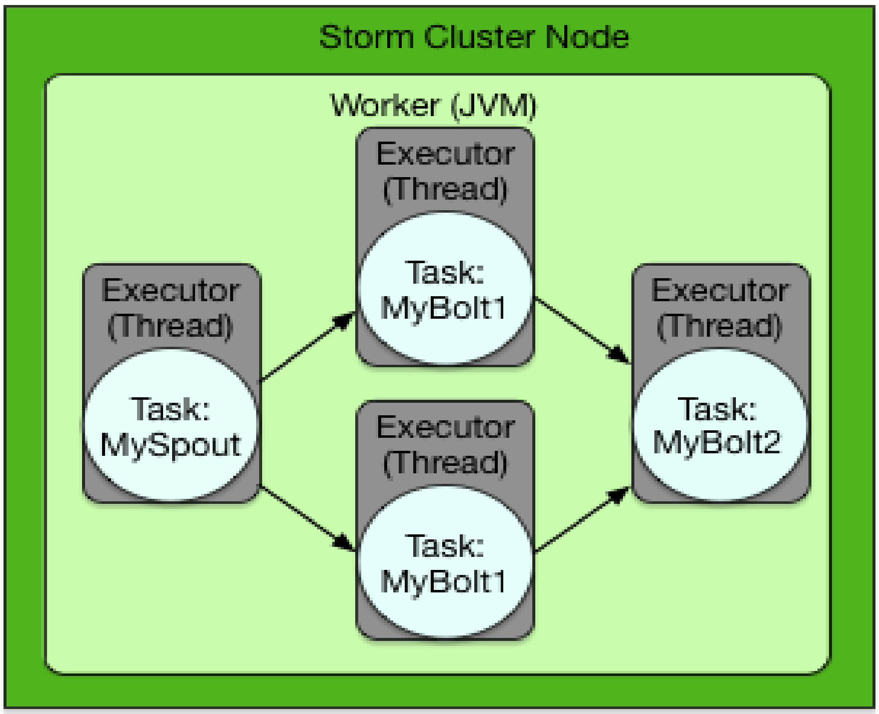

Figure 15 Defined parallelism

The number of tasks can also be increased with the method *setNumTasks(int)* which is available for spouts and bolts:

```
Config config = new Config();
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(MY_SPOUT_ID, mySpout);
builder.setBolt(MY_BOLT1_ID,myBolt1)
               .setNumTasks(2)
               .shuffleGrouping(MY_SPOUT_ID);
builder.setBolt(MY_BOLT1_ID,myBolt2)
               .shuffleGrouping(MY_SPOUT_ID);
```

This code sample assigns two tasks to execute MyBolt1, as the following figure illustrates. This added parallelism might be appropriate for a bolt containing a large amount of data processing logic. However, adding tasks is like adding executors because the code for the corresponding spouts or bolts also changes.

The defined parallelism of this code section is depicted on Figure 16.

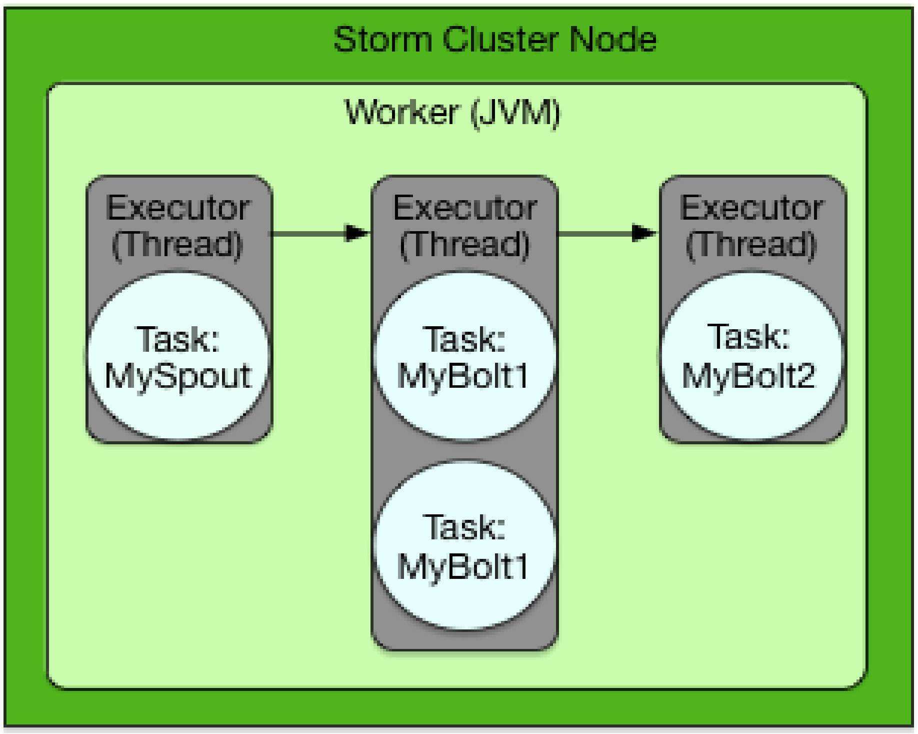

Figure 16 Increased number of tasks

And if we try to put them all together:

```
Config conf = new Config();
conf.setNumWorkers(2); // use two worker processes
topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2); // set
parallelism    //hint to 2
topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
               .setNumTasks(4)
               .shuffleGrouping("blue-spout");
topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
               .shuffleGrouping("green-bolt");
StormSubmitter.submitTopology("mytopology",conf,topologyBuilder.create
Topology() );
```

The derived topology of this code section is being shown below on Figure 17.

Storm also offers the feature to change the number of worker processes and of executors without restarting the cluster or the topology. This feature is called rebalancing. There two ways to rebalance the topology, from the Storm web UI, or the Storm CLI tool.

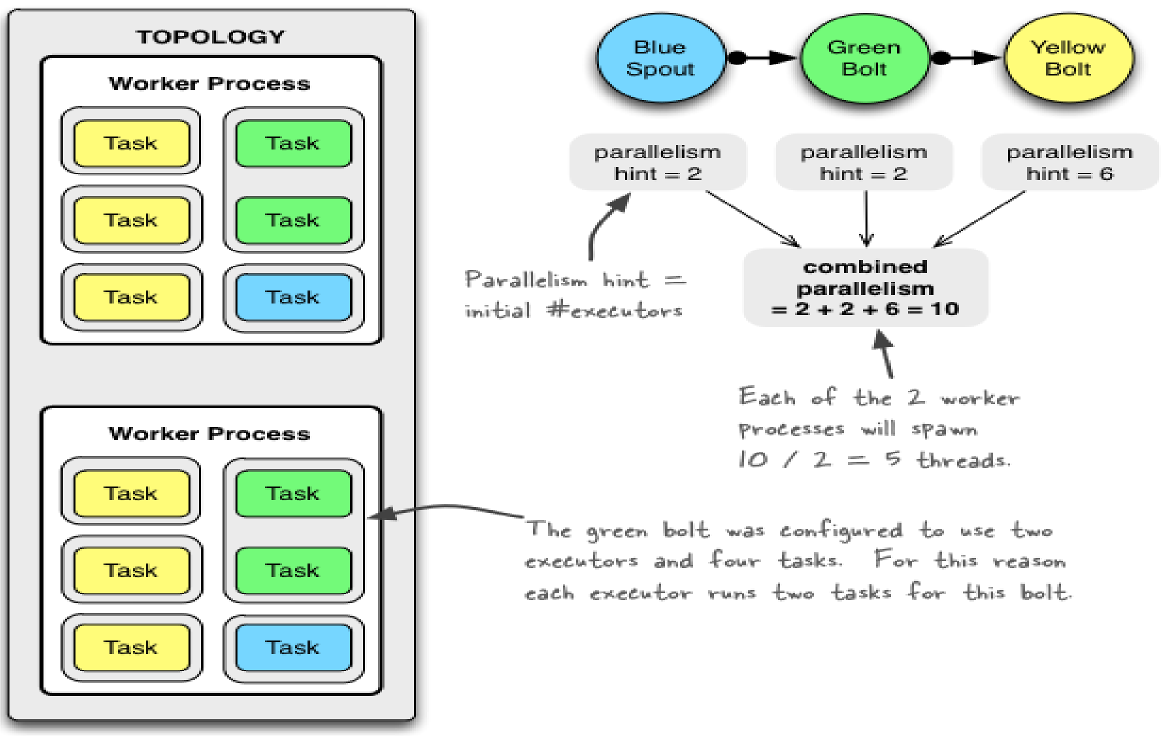
TOPOLOGY
Worker Process
Task
Task
Task
Task
Task
Task
Worker Process
Task
Task
Task
Task
Task
Task
Blue Spout
Green Bolt
Yellow Bolt
parallelism hint = 2
parallelism hint = 2
parallelism hint = 6
Parallelism hint = initial #executors
combined parallelism = 2 + 2 + 6 = 10
Each of the 2 worker processes will spawn 10 / 2 = 5 threads.
The green bolt was configured to use two executors and four tasks. For this reason each executor runs two tasks for this bolt.

Figure 17 Derived parallelism

Generally Storm is a really powerful stream processing engine with the ability to fine tune many aspects of the system making it really appealing to big data industries. It offers an integrated and versatile system and approach that can handle stream processing in a real-time fashion. In the computing industry such a tool can be used effectively to monitor real time feed from various sensor platforms and networks, such as Twitter or Facebook.

The complexity to understand and express a way to process incoming data in such systems many times is derived from the way those systems were designed. In order for all those systems to be able to offer a really adaptive way to work over the incoming data sometimes makes them too complex for users to understand. Users with a different scientific background experience this at much greater extend.

With the upcoming revolution of Internet of Things that inevitable produce a really large volume of information a set of tools combined with a stream processing system could potentially provide a nice abstraction from the internal turning knobs of such a system. Data scientists could benefit from such a tool that could simplify and present a configurable and more human understandable way to work with incoming data, but other users as well.

We aim to provide such an abstraction over Storm and to implement a framework so that the user would be able to choose from a set of defined algorithms, essentially a workflow, how he would like the system to analyze the incoming information, without the need to know any programming languages. Usually when performing stream or event analysis there are certain algorithms you would like to use in order to process such information.

Our goal would be for the user to describe a topology, a processing workflow, much like the topology described in Storm by using a DSL language for chaining algorithms of his choice.

In the next sections we will describe how such a similar system was previously designed, what was it based upon and how eventually we started our implementation of this framework as well as the main problems we encountered.

3. FUSION BOX AND CONTEXTORS

This section is about the implementations and the creation of the fusion box, an engine implemented to enable processing and consolidating data from heterogeneous sources. This engine enables the integration and interpretation of different types of data sources, with the use of multiple algorithmic flows according to these sources. This platform aims to shift the complexity that comes when coping with heterogeneous streams , perform complex operations efficiently, enable the customization of the middleware processing according to specific requirements of a wide spectrum of application scenarios and allow the creation of complex workflows.

To enable this engine to adapt and uphold all those requirements a workflow processing engine has been built around basic blocks named contextors that perform autonomous algorithmic steps.

The theory of contextors (Coutaz and Rey 2002) describes a contextor as an abstract functional unit which defines some outbound data and a single outgoing flow, which is a product of the contextor’s functional core. The core of the contextor is what essentially differentiates on contextor from another by defining its behaviour; its logical core. The core of a contextor consists of an implemented algorithm that is used for transforming the inbound flows to a specific outbound glow. This outbound flow then is being used as an input to a next contextor(s) and thus enables the creation of complex workflow structures. A contextor is being shown on Figure 18.

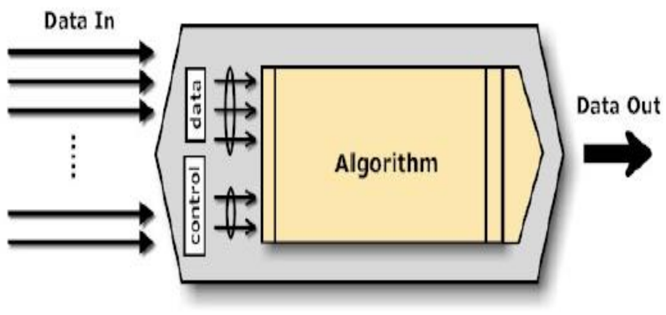

Figure 18 A contextor

The Fusion Box architecture is being based to the provisioning of the necessary middleware services in order to support the full lifecycle of a contextor. The Fusion Box therefore acts as a sandbox for deploying and running complex data processing workflows that are being composed of multiple contextors. The Fusion Box also provides all the necessary infrastructure services to enable a) the exchange of information between the contextors through a messaging framework b) the dynamic deployment and provisioning of workflows c) The dispatching of the output produced by the workflows either in the same FusionBox or outside of it.

The network module of the FusionBox handles information flows from external data sources that have been selected as inputs to a specific data processing workflow. Essentially the network module is responsible for providing the data sources needed to the workflows and to the contextors.

The data processing engine (DPE) comprises the application layer of the FusionBox and provides the appropriate runtime environment for all workflows. As already stated the structural structural elements of the workflow graph are instances of the FusionBox contextor, where a specific algorithm is encapsulated and runs within each instance.

The operation of each contextor is completely event based, and is triggered by the reception of a new data element on any of its data-in channels.

There a number of similarities between the FusionBox engine and how Apache Storm handles data processing. A contextor in its abstract form can be correlated with a bolt in Storm in a logical sense, since both of them are the logical computational unit that support the framework. A bolt is not the same as a contextor, each bolt in storm is essentially a thread running, while a contextor is a java bean. This gives Storm the ability to scale up its number of threads corresponding to a specific bolt definition but at the same time a more sophisticated data management framework is required for supporting thread-like computational units(fields grouping etc.). Stateless algorithms or execution logic can greatly be enhanced by Storm's approach.

Spouts in storm are essentially responsible for feeding any kind of data into the system much like the Network Module of the Fusion Box does. A spout's implementation is being left to the user as he can effectively define and program the logic of the spout and how data is being delivered in the system.

A workflow in the FusionBox is essentially similar with a Storm topology; both describe the connection between abstract logical units and create a graph.

Given those similarities the purpose of this Master thesis is to create the required framework on top of Storm's existing one and investigate possible ways that a similar engine can be constructed and at the same time exploit Storm's ecosystem and tools that offers for a more robust and scalable Fusion Box engine. [34] [35]

4. STORM AS A DATA FUSION ENGINE

Storm gives a competent framework for creating complex topologies and at the same time offers the ability to fine tune all the parts of a topology, has the ability to scale and cope with demanding data streams.

The goal of this thesis is the implementation of a framework on top of storm in order to describe and deploy topologies of predefined algorithms enclosed in bolts. This framework will be responsible to provide an abstraction on top of storm's existing framework in order to give the ability to the user of the system (e.g. a data scientist) to define and execute such topologies without the need to program any of storm's internal components(such as bolts and spouts).

To achieve such an abstraction there are some steps that need to be implemented that will enable this system to create abstract topologies.

1. An easy way to describe the topology via a description file.
2. A complete set of algorithms that a scientist could find useful when having to process any number of data streams.
3. The implementation of a framework on top of storm that can handle such generic topologies.

4.1 Apache Storm Flux

Apache Storm Flux project [36] is a framework for creating and deploying Apache Storm topologies with more ease.

Currently the way a user can create a Storm topology is by creating a java object, a topology object where all the connectivity of the spouts and bolts is being described. After the topology object has been created the user can choose to deploy it on a Storm cluster, or run it locally in order to test the topology. However this class file is hard coded and any configuration change cannot be made without having to recompile the file and possibly the entire application which is something that is time consuming and can lead to any number of errors.

To help automate this process Flux is essentially a framework responsible for creating topologies with the use of description files in YAML syntax. The way it approaches this is by relying heavily on reflection in order to build the desired topology object which then submits to the cluster for deployment. Flux framework therefore offers some unique features to the Storm ecosystem such as:

1. Easily configure and deploy Storm topologies (Both Storm core and Microbatch API) without embedding configuration in your topology code
2. Support for existing topology code
3. Define Storm Core API (Spouts/Bolts) using a flexible YAML DSL
4. YAML DSL support for most Storm components (storm-kafka, storm-hdfs, storm-hbase, etc.)
5. Convenient support for multi-language components
6. External property substitution/filtering for easily switching between configurations/environments (similar to Maven-style ${variable.name} substitution)

Furthermore Flux can actually be described more appropriately as a YAML bean engine that can create any kind of object. An example topology is being shown below:

```
name: "yaml-topology"
config:
  topology.workers: 1

# spout definitions
spouts:
  - id: "spout-1"
    className: "org.apache.storm.testing.TestWordSpout"
    parallelism: 1

# bolt definitions
bolts:
  - id: "bolt-1"
    className: "org.apache.storm.testing.TestWordCounter"
    parallelism: 1
  - id: "bolt-2"
    className: "org.apache.storm.flux.wrappers.bolts.LogInfoBolt"
    parallelism: 1

#stream definitions
streams:
  - name: "spout-1 --> bolt-1" # name isn't used (placeholder for
logging, UI, etc.)
    from: "spout-1"
    to: "bolt-1"
    grouping:
      type: FIELDS
      args: ["word"]


  - name: "bolt-1 --> bolt2"
    from: "bolt-1"
    to: "bolt-2"
    grouping:
      type: SHUFFLE
```

This YAML file describes a topology that comprises of a spout that sends tuples to two bolts. Each YAML file that can be used for flux has some main components:

- name : the name of the topology being described,
- components: some beans that will be used from the spouts/bolts by being passed from a constructor or a simple method(not being shown here)
- spouts: The spout definition section
- bolts: The bolt definition section
- streams: The stream definition section where the connection the spouts and bolts is being described.

This YAML description file describes a topology where a spout sends words and two receiving bolts that receive each emitted word, one for counting how many times each word has appeared and another bolt that is responsible for logging each word that arrives.

The alternative way to create such a topology would have been in a java file:

```
public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        TestWordSpout testWordSpout = new TestWordSpout();
        builder.addSpout(“spout-1”,testWordSpout,1);
        TestWordCounter testWordCounter = new TestWordCounter();
        builder.addBolt(“bolt-1”, testWordCounter,1)
                    .fieldsGrouping(“spout1”,newFields(“word”));
        LogInfoBolt logInfoBolt = new LogInfoBolt();
        builder.addBolt(“bolt-2”,logInfoBolt,1)
                    .shuffleGrouping(“spout-1”);
        StormTopology topology = builder.buildTopology(…)
        topology.run();
}
```

Any extra needed modification (e.g. changing one spout, changing the fields grouping etc.) would require recompilation of the class and potentially of the entire project.

Flux also offers much flexibility to the user by giving him the ability to combine many YAML DSL files in order to compose all the components of the topology as well as the ability to reference variables from property files and environmental variables from other YAML files as well:

With the following dev.properties file:

```
kafka.zookeeper.hosts: localhost:2181
```

You would then be able to reference those properties by key in your .yaml file using ${} syntax:

```
 - id: "zkHosts"
    className: "org.apache.storm.kafka.ZkHosts"
    constructorArgs:
      - "${kafka.zookeeper.hosts}"
```

In this case, Flux would replace *${ kafka.zookeeper.hosts }* with *localhost: 2181* before parsing the YAML contents.

Furthermore Flux can support constructor initialization and method invocation by referencing the method to be called and passing arguments, either simple or complex objects. Property setting is also supported.

Arguments to a class constructor can be configured by adding a contructorArgs element to a component. ConstructorArgs is a list of objects that will be passed to the class' constructor. The following example creates an object by calling the constructor that takes a single string as an argument:

```
- id: "zkHosts"
    className: "org.apache.storm.kafka.ZkHosts"
    constructorArgs:
      -"localhost:2181"
```

Or a complex object using the reference[ref] tag:

```
components:
  - id: "stringScheme"
    className: "org.apache.storm.kafka.StringScheme"


  - id: "stringMultiScheme"
    className: "org.apache.storm.spout.SchemeAsMultiScheme"
    constructorArgs:
      - ref: "stringScheme" # component with id "stringScheme"  #must
be  declared above.
```

Flux example of setting properties:

```
  - id: "spoutConfig"
    className: "org.apache.storm.kafka.SpoutConfig"
    constructorArgs:
      # brokerHosts
      - ref: "zkHosts"
      # topic
      - "myKafkaTopic"
      # zkRoot
      - "/kafkaSpout"
      # id
      - "myId"
    properties:
      - name: "ignoreZkOffsets"
        value: true
      - name: "scheme"
        ref: "stringMultiScheme"
```

Configuration methods are similar to Properties and Constructor Args; they allow you to invoke an arbitrary method on an object after it is constructed. Configuration methods are useful for working with classes that don't expose JavaBean methods or have constructors that can fully configure the object. Common examples include classes that use the builder pattern for configuration/composition.

The following YAML example creates a bolt and configures it by calling several methods:

```
bolts:
  - id: "bolt-1"
    className: "org.apache.storm.flux.test.TestBolt"
    parallelism: 1
    configMethods:      # public void withFoo(String foo);
      - name: "withFoo"
        args:
          - "foo"
      - name: "withBar"   # public void withBar(String bar);
        args:
          - "bar"
      - name: "withFooBar" # public void withFooBar(String  #foo,
String bar);
        args:
          - "foo"
          - "bar"
```

Arguments passed to configuration methods work much the same way as constructor arguments, and support references as well.

Also Flux supports a configuration section that simply a map of Storm topology configuration parameters that will be passed to the org.apache.storm.StormSubmitter as an instance of the org.apache.storm.Config class:

```
config:
  topology.workers: 4
  topology.max.spout.pending: 1000
  topology.message.timeout.secs: 30
```

Flux supports also the ability to specify a class that returns a topology object for deployment.

For our purposes, Flux seems to be an excellent choice since it gives the ability to the user to describe topologies or change aspects of existing ones via a description file. Furthermore Flux's ability to construct objects/components has a significant impact on how we would like our system to behave. However there are some details that need to be fine grained in order to make Flux a topology engine in such a way that the experimenter would not have to write any code or recompile any of the project's files.

4.2 Advancing the framework

By using Flux we can describe the topology and therefore abstract the way the topology is being created so that it's not so tightly connected with the project itself, it can change without having to change or recompile our entire project.

Continuing our system design we can now create a set of algorithms that will essentially be bolt classes and then use the Flux framework to build almost any kind of topology. One approach would be to create each algorithm as an extension of the bolt class and therefore each time we would like to chain some algorithms together all we would have to do is chain the algorithms.

While this approach was considered at the beginning it was evident that the code used for each bolt in order to operate was quite repetitive. Therefore the approach was changed so that every bolt would be essentially a wrapper for the algorithm. It was pretty clear that any algorithm would require little or no knowledge of the bolt's context of execution. Any information needed from the bolt to the algorithm can be supplied by having all the algorithms to comply to an interface that will regulate the information exchange between them.

Therefore each bolt will act more like a generic bolt that will be able to hold or wrap an algorithm inside. The bolt will be solely responsible for receiving messages and emitting them to the appropriate next bolt or bolts according to the topology. The algorithm on the other hand will be responsible for applying any kind of logic or transformation to the incoming message or messages and supplying the payload to the bolt for transmission. Furthermore the bolt will give the algorithm its chance to perform any necessary initialization.

All these conventions have been derived from experimenting with the storm framework and its limitations.

The current design supported for each algorithm and bolt is being shown by the following classes:

```
public interface IAlgorithm {
  void prepare();
  void Values execute(Tuple incomingTuple);
}
```

```
public  class GenericBolt implements IRichBolt, IAlgorithm {
    IAlgorithm algorithm;

  public void setAlgorithm(IAlgorithm algo) {
      this.algorithm  = algorithm;
  }

  @Override
  public void prepare(Map map,
                    TopologyContext topologyContext,
                             OutputCollector outputCollector) {
       this.algorithm.prepare();
  }

  @Override
  public void execute(Tuple tuple) {
     values = algorithm.executeAlgorithm(tuple);
     emit(values);
  }
```

The prepare method gives actually the chance to the algorithm to implement all necessary initializations at the prepare method of the bolt.

This is needed since storm essentially requires that any bolt and spout prior to topology creation is serializable, therefore each of its members. In order to provide more flexibility to the programmer it provides a prepare method where any initialization of the non serializable objects is to happen. That is why we provide the same method to the enclosed algorithm.

However according to the kind of bolts(single input bolts vs Windowed bolts) this abstraction can change according to the wrapper bolt. In case we have to use windowed bolts the implementation changes respectively by having a GenericWindowedBolt that implements the IWindowedAlgorithm interface with similar methods(input changes).

While someone can argue that some algorithms can work in both ways, windowed or by single value, setting this enforces a cleaner approach and gives the programmer the ability to implement both versions of the algorithm if needed without having to resort to solutions like reflection to accommodate both variations of the algorithm's behavior at the same class file.

4.3 Stream definition, grouping, declarer

In all the above topologies described we were able to describe a way so that the topologies can be configured and expressed in a YAML DSL using Flux. However we would like to give to the experimenter the ability to connect all spouts and bolts regardless of what the input/output would be of each algorithm.

The tuple is the main data structure in Storm. A tuple is a named list of values, where each value can be any type [37].

Storm has some limitations when emitting tuples from a spout to a bolt or from a bolt to another. Every spout or bolt before the prepare stage, before the topology creation that is, needs to declare the outgoing fields or "labels" for each spout or bolt participating to the topology. Since a tuple is essentially a list of named values array every bolt would have to know how many fields is it going to receive and what will be their name, in order to be accessible from the tuple wrapper class. An example is being shown below:

```
@Override
public void declareOutputFields( OutputFieldsDeclarer declarer) {
    declarer.declare(stream, new Fields(“id”, “value”, “timestamp”);
}
```

This is a declare method example being used in bolts to declare the format of the outgoing fields. This describes that the emitted tuple will contain three fields named "id", "value" and "timestamp" with their corresponding values. The next bolt receiving this tuple will be able to get all of the three values by calling the method getFieldByName(String fieldname) .

Furthermore Tuples are dynamically typed – the types of the fields do not need to be declared and also have helper methods like getInteger and getString to get field values without having to cast the result [37].

Every spout or bolt, when emitting and receiving tuples, checks that the tuple conforms to the format declared from each emitter. Whenever a spout or a bolt emits something different on each tuple (e.g. a field more) then an exception is being thrown.

A stream is an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion as we have already discussed. Streams are defined with a schema that names the fields in the stream's tuples. By default, tuples can contain integers, longs, shorts, bytes, strings, doubles, floats, booleans, and byte arrays.

Every stream is given an id when declared. Since single-stream spouts and bolts are so common, OutputFieldsDeclarer has convenience methods for declaring a single stream without specifying an id. In this case, the stream is given the default id of "default" [23].

Whenever having to declare the outgoing fields we also can specify a specific stream to emit the tuples. Apart from the default stream we can have multiple streams which we will have to name. In the method declareOutputFields we must specify every stream that this component is emitting. In the case of a single stream the default id can be used. A component emitting to many streams can choose which tuples to emit to any of those streams according to how we would like our component to behave and what kind of fields will be emitted to each stream.

Therefore apart from declaring the outgoing fields for each stream we can choose the conditions and the behavior of our component. For example a bolt declaring two streams, can choose to emit all the values that surpass a threshold to a stream while forwarding the other values to another.

```
public void declareOutputFields( OutputFieldsDeclarer declarer) {

    declarer.declare(“over-threshold”,
                 new Fields(“id”, “value”, “timestamp”);

    declarer.declare(“under-threshold”, new Fields(“id”,”value”);

}


public void exectute(Tuple tuple) {

   int value = tuple.getInt(1); // dynamically resolved type,
           // we know that at the second position is the “value”
          // and is an integer

  if( value > threshold)
          collector.emit(“over-threshold”,
            new Values(tuple.getString(0), value,
                     tuple.getLong(2));

  else collector.emit(“under-threshold”,
            new Values(tuple.getString(0), value);

}
```

In this example not only the bolt sends messages to two recipients, but also splits the stream according to some threshold and changes the fields of each message according to the stream that the message is being emitted to.

In order for a component to be able to send messages to more than one streams, this must be described also at the stream definition section of the topology creation. For example:

```
#stream definitions
streams:
  - name: "spout-1 --> splitter-bolt"
    from: "spout-1"
    to: "splitter-bolt"
    grouping:
      streamId: “stream-1”
      type: SHUFFLE

  - name: "splitter-bolt --> over-threshold-bolt"
    from: "splitter-bolt"
    to: "over-threshold-bolt"
    grouping:
      streamId: “over-threshold”
      type: SHUFFLE

  - name: "splitter-bolt --> under-threshold-bolt"
    from: "splitter-bolt"
    to: "under-threshold-bolt"
    grouping:
      streamId: “under-threshold”
      type: SHUFFLE
```

As the example shows the stream definitions actually describe that each component might have more than one streams that it can emit therefore there must be compliance between what is declared at the stream and topology definition and what is declared at the component declare method. This is left to the programmer.

In our scope however we would like a way to abstract this definition so that when a user describes a topology with any kind of fields, that contract must be kept without the need to explicitly program and change each connected component every time the user describes a topology. A solution that seems a little bit far-fetched is to write to the component class files the declaration of fields and streams every time the user specifies a topology graph which is something error prone and requires recompiling the entire framework and suite to incorporate the stream to fields declaration compliance.

A nice solution would be to extend the implementation of bolts and spouts and give a set of functions that enable to register stream and field definitions to the class. By calling those functions with our desired parameters each component can simply add the stream and fields definition. Flux enables us to call methods with parameters therefore a class with a method declareStreamWithFields(String streamId, String … fields) could be called from the YAML DSL files and the corresponding class would register the declaration as follows:

```
public class GenericBolt implements IRichBolt, IAlgorithm {
  public Map<String, List<String> outgoingFields>
                       streamFieldsMap;


  public void declareStreamWithFields( String streamId,
                         String …fields) {
       //assume initialization of Map has been performed in the prepare
method
       streamFieldsMap.put(streamId, Arrays.asList(fields));
   }


  public void declareOutputFields(
                OutputFieldsDeclarer declarer) {
          streamFieldsMap.forEach( stream , fieldStrings)
                -> declarer.declareStream(stream,
                        new Fields(fieldStrings)));
       }
}
```

The creator of the YAML file now is responsible for defining the outgoing fields and streams for each component and as a consequence is also responsible for complying to the described topology and stream definition.

```
name: "yaml-topology"
config:
  topology.workers: 1


# spout definitions
spouts:
  - id: "spout-1"
    className: "org.apache.storm.testing.TestWordSpout"
    configMethods:
        - name: “declareStreamWithFields”
          args:
              - “stream-1”
              - “word”
    configMethods:
         - name: “declareStreamWithFields”
           args:
              - “stream-2”
              - “word”
          parallelism: 1

# bolt definitions
bolts:
  - id: "bolt-1"
    className: "org.apache.storm.testing.TestWordCounter"
    # no declare method here, this bolt doesn’t send anything to anyone
    parallelism: 1

  - id: "bolt-2"
    className:  "org.apache.storm.flux.wrappers.bolts.LogInfoBolt"
    # no declare method here, this bolt doesn’t send anything to anyone
    parallelism: 1

#stream definitions
streams:
  - name: "spout-1 --> bolt-1"
    from: "spout-1"
    to: "bolt-1"
    grouping:
      streamId: “stream-1”
      type: FIELDS
      args: ["word"]

  - name: "bolt-1 --> bolt2"
    from: "bolt-1"
    to: "bolt-2"
    grouping:
      streamId: “stream-2”
      type: SHUFFLE
```

By incorporating those changes we see that the complexity has shifted from the programmer that implements the classes, to the user that creates the topology since he is responsible for calling the appropriate methods. That’s a complexity that can become confusing and cumbersome to the user, since he must know at each component which methods to call and also what kind of algorithm is enclosed in each bolt.

If we provide to the user a suite of algorithms by having him to specify the fields definitions, we subsequently impose to the user to have knowledge of each algorithms input and output, since he has to declare each set of fields on each node of the topology graph(the output of one algorithm becomes input to the other).

We would like to have a way to automate this procedure so that the user only has to know what an algorithm does and not what it should be emitting to the next. The user is responsible to create a topology with a logical meaning, and by that it is implied that all algorithms should be able to connect with each other, as far as storm is concerned, but the logical chaining of algorithms cannot happen with each and every algorithm because that would lead to a topology without any logical meaning.

For example an algorithm that classifies an incoming value and forwards the value to an algorithm that computes the median of many incoming values would not have much meaning. This is something that the user must avoid, however the framework on top of the Storm should not, in order to create a really versatile framework. Consequently if we would like to ensure a logically valid topology we could implement a set of rules indicating the possible logical pairs that each algorithm can chain to.

By inspecting the set of algorithms that have been implemented so far, we can see that each algorithm performs a transformation to the incoming fields and produces the outgoing fields. Some algorithms check if a value has breached a threshold and detain the value, others compute if there has been a value that can be characterized as an outlier, others detect if there has been a surge of values towards some upper or lower boundaries etc. Each algorithm subsequently applies a transformation to the fields. For example the CUSUM algorithm that detects if there has been a surge in the stream values simply adds another field to the tuple emitted indicating such an incident. The FieldFilter algorithm basically removes any unwanted fields with their values from a stream, reducing the values a tuple contains. In the first case all we would have to do is add one more field to the declarer while in the latter we would have to specify the remaining fields that have not been removed from the algorithm to the declarer.

This leads us to the realization that we could implement a way to specify the fields of each bolt by taking into consideration the incoming fields (from the previous algorithm or spout) and applying the transformation of the current algorithm and declaring the new fields if the bolt has someone to send them to (this is specified to the topology creation).

Each algorithm will have to implement an interface that will provide a function that the incoming fields to the bolt, and subsequently to the algorithm will be passed and the algorithm will declare the transformed fields and use them as incoming fields for the next algorithm.

```
public interface FieldTransformer {

    //applies a transformation to the incoming fields
    //returning the new fields

    Public String[] transformFields( String incomingFields);

}
```

Now every algorithm interface IAlgorithm and IWindowedAlgorithm needs to implement the FieldTransformer interface. Each algorithm depending on the implementation can remove fields, change them completely or do nothing on the incoming fields.

What we would like now is to have a manager that will be responsible to pass any field relevant information to the bolts according to the topology described before the topology creation. Since Flux is responsible for creating the bolts and spouts we could implement this functionality inside Flux.

However changes must be done to the Flux code so that it can incorporate this extra functionality. At the same time we would like not to break the existing functionality as well. Flux uses description files, essentially class files that describe the core elements of each Storm component (config, spouts, bolts, stream definitions and components – essentially beans). We could extend those existing models to create our own models and insert a set of functions so that they can resolve the field declaration.

For this purpose a class called FusionBoltDef that extends the existing BoltDef (Bolt Definition class of Flux) has been created and this way we can declare fusion-bolts inside the Flux YAML DSL. Those bolts are somewhat different from plain Flux bolts in the sense that they try to resolve the field and stream definitions accordingly to what we have described so far.

```
public class FusionBoltDef extends BoltDef {

    public String[] fields = null;

    public String[] getFields() {

        return fields;

    }

    public void setFields(String[] fields) {

        this.fields = fields;

    }

}
```

Every FusionBoltDef is essentialy a description file for creating FusionBolts which is an interface that all bolts of our system must comply to:

```
public interface FusionBolt {
    public void setFields(boolean terminalNode, String ...fieldNames);
    public void addOutgoingStreamName(String streamName);
    public String[] getOutgoingFields();
}
```

Essentially what we are trying to do is to model the communication between the GenericBolt and the enclosed, algorithm whatever this algorithm might be.

This enables us after the bolts have been created in Flux to use the information described from the stream definitions in order to do depth first search of the topology graph starting from spouts and continuing with the bolts. At each step of the recursion if we find a bolt, let’s say bolt-1 that connects with another bolt(bolt-2), we acquire the outgoingFields from the parent of bolt-1 and insert them at bolt-1 where the algorithm of bolt-1 has a predefined transformation on them and returns the outgoing fields. It is important to remember that we need to declare only the outgoing fields, therefore bolt-2 will have to declare its outgoing fields only if there is another connection starting from bolt-2 to another chained bolt. The same procedure happens at the streams declared at the described topology. They’re being passed inside the bolt and to the declarer of the component.

We now have the ability to create complex topologies without the need for the user to specify incoming and outgoing fields, our extended Flux engine takes care of that.

While any bolt and algorithm can vary its approach to different values of the stream which is something the programmer can easily implement, when providing the abstraction we aim to, it goes without saying, that an algorithm that selectively decides in which stream to emit each incoming value is something that cannot be easily implemented at a generic algorithm unless this is the nature of the algorithm.

Since the connection of each bolt and subsequently of the algorithm is expressed externally at the topology creation(one bolt might have input from many streams) we cannot incorporate different behaviour according to the output streams of the algorithm. That means that the algorithm will not differ its behaviour when emitting to many streams. Imagine a thresholding algorithm that has to mutate its behaviour according to how many streams it must send the tuple. This cannot be implemented on a compiled class unless we change the code or else the behaviour of the algorithm by injecting code and recompiling the class.

In a scenario where we would like an algorithm to emit let’s say numerical values that befall to a specific range to specific streams, e.g. an algorithm that maps integers from (-100,0] (0,100) we can have the same effect by chaining the same algorithm twice and changing its parameters. For this specific example we would have to split the stream by specifying two recipients of the stream essentially copying the values and the one would have to apply a threshold to keep values below of equal to zero and then another with values bigger than -100. The same would have to happen at the other stream respectively by creating an algorithm of threshold over zero and another that with a threshold less than 100.

4.4 Defining Spouts

As we continue our search for an abstraction of our framework we need to find a way to declare spouts so that they can consume messages from almost any queue the user might need to. Right now kafka and mqtt consumers are available and more consumers will be implemented.

One of the main questions and problems that arise when deciding how to approach the framework is how can we instruct the spout to send to multiple streams, which is something partially solved from the approach we used when we were designing the fusion bolts, and how will we give the user the ability to split the incoming messages according to his needs. Mqtt consumer handles incoming messages as a sequence of characters, while a kafka consumer treats incoming messages as a byte array. The kafka consumer provided by Storm has a more complex approach since it handles the complete functionality of the kafka message queue, however it provides a way to define a serialization scheme that the user can specify or extend according to his needs, that essentially handles the message serialization.

We would like to uphold some kind of consistency between the spout definitions of both queues so that the definition would be similar. When describing a message consumer configuration aspects, that appear the same, to any kind of message queue are the connecting host, a port, a message topic in which we are interested and a message format or scheme so that we can interpret the consumed messages. Since that kafka consumer specifies such information and configuration at a configuration class which is then passed into the kafka spout, the mqtt configuration has been altered to have the same look so that in the YAML DSL file appear somewhat similar. However we could not avoid making new definition files for each consumer in the Flux files. Therefore two new kinds of spouts configs have been created mqtt-config and kafka-config as shown below:

```
mqttconfig:
  - id: "mqtt-config"
    className: "flux.model.extended.MqttSpoutConfigDef"
    brokerUrl: "tcp://localhost:1883"
    topic: "health_monitor/blood_pressure"
    clientId: "hello"
    regex: ","

spouts:
  - id: "blood-spout"
    className: "consumers.MqttConsumerSpout"
    constructorArgs:
      - ref: "mqtt-config"
```

```
kafkaconfig:
  - id: "kafka-config"
    className: "flux.model.extended.KafkaSpoutConfigDef"
    regex: ","
    zkHosts: "localhost:2181"
    topic: "health"
    zkRoot: "/health"
    clientId: "storm-consumer"
```

```
spouts:
  - id: "kafka-spout"
    className: "consumers.FusionKafkaSpout"
    constructorArgs:
      - ref: "kafka-config"
```

Both mqtt-config and kafka config appear almost similar to the user while at the same time hiding the complexity of the underlying implementation. For example this is what it would look like creating a kafka consumer with vanilla Flux:

```
- id: "keyValueSchemeasMultiScheme"
  className: "org.apache.storm.kafka.KeyValueSchemeAsMultiScheme"
  constructorArgs:
    - ref: "fusionScheme"

- id: "zkHosts"
  className: "org.apache.storm.kafka.ZkHosts"
  constructorArgs:
    - "localhost:2181"

- id: "spoutConfig"
  className: "org.apache.storm.kafka.SpoutConfig"
  constructorArgs:
    # brokerHosts
    - ref: "zkHosts"
    # topic
    - "health"
    # zkRoot
    - "/health"
    # id
    - "storm-consumer"
  properties:
    - name: "bufferSizeBytes"
      value: 4194304
    - name: "fetchSizeBytes"
      value: 4194304
    - name: "scheme"
      ref: "keyValueSchemeasMultiScheme"


spouts:
  - id: "kafka-spout"
    className: "org.apache.storm.kafka.KafkaSpout"
    constructorArgs:
      - ref: "spoutConfig"
```

Clearly the complexity of defining a kafka consumer has been greatly reduced.

Another issue we have with the consumers is that we have to specify a way to interpret the incoming message. Any incoming message, apart from complex types that we don't support for numerous reasons, will consist of mainly primitive types such as strings and numbers. Given that any incoming message can be handled as a string message and then can be split to its corresponding parts. For example a message in the format "sensor-1, 80.0, 90" is a message that can be split every time a comma appears. Therefore the regex for this message format is ",". That requires that each configuration (either mqtt or kafka) defines a regex pattern so that every message can be split to its parts.

Furthermore even when a message has been split we would like to know its type. This is imperative since each tuple handles any contained value as an object in order to avoid complex annotations from the part of the programmer and that has a consequence that each contained value loses its static java type. To put it more bluntly since the programmer has constructed the tuple, he is responsible to know how to access each field.

The main goal of the framework is to avoid the heavy use of reflection on each incoming tuple since this would make the system much more complex and slow if we have to check the type of each field before we take any action. This approach will lead to a complex implementation and poor maintainability. However in marginal cases we would like to give to the ability to do so, to whoever decides to extend the framework and add new algorithms. In order to support this approach the values included inside each tuple will have to be resolved to their dynamic type despite the fact that the tuple wraps them as an object. This is where we require some extra information from the user to supply the class of each contained value on each incoming message apart from the field's labels of each value. Each consumer now contains a class mapper that maps each field to its corresponding class and creates an instance of this class (java.lang.String for Strings, java.lang.Integer for Integers etc.). Using this approach each tuple has a list of objects but each type has been instantiated to its correct type and now we can use reflection on it if is needed (instance of and other functions). The augmented config class now looks like this:

```
mqttconfig:
  - id: "mqtt-config"
    className: "flux.model.extended.MqttSpoutConfigDef"
    brokerUrl: "tcp://localhost:1883"
    topic: "health_monitor/blood_pressure"
    clientId: "hello"
    regex: ","
    fields:
      - "id"
      - "value"
      - "timestamp"
    classes:
      - "java.lang.String"
      - "java.lang.Double"
      - "java.lang.Long"
```

Respectively the kafka-config has been changed to incorporate this functionality as well.

Let's dive a little bit deeper into the mapper functionality:

```
public class OutputFieldsClassMapper implements Serializable {


protected List<ClassConverter<?>> converters = null;
protected List<Class> classes = null;
protected String regex = null;
protected String[] classNames;



…

private void resolveClassConverters(String[] classes) {
    converters = new ArrayList<>();
    for (String clazz : classes) {
        switch (clazz) {
            case "java.lang.Integer":
                converters.add(
                   (ClassConverter<Integer>) Integer::valueOf);
                break;
            case "java.lang.Double":
                converters.add(
                    (ClassConverter<Double>) Double::valueOf);
                break;
            case "java.lang.Float":
                converters.add(
                    (ClassConverter<Float>) Float::valueOf);
                break;
            case "java.lang.Long":
                converters.add(
                    (ClassConverter<Long>) Long::valueOf);
                break;
            default:
                converters.add(
                    (ClassConverter<String>) value -> value);
        }
    }
}


}
```

Each OutputFieldClassMapper class needs input the class names in String which then resolves to their corresponding Class object by using the class.ForName function. Since we have imposed out restrictions that we only handle primitive types all it needs to do is implement a set of converters that will interpret the value to their corresponding class type.

The ClassConverter is an interface that contains only one function:

```
public interface ClassConverter {
    T convertToObject(String value);
}
```

This class is being instantiated on runtime and creates the implementation of how the String value is going to be resolved.

Generally the problem of resolving the object to its corresponding class type at runtime is a problem that it is being addressed with this kind of approach any time it is needed. A thresholding algorithm that essentially compares numbers needs to know what kind of numbers it is going to compare. The converter class comes handy in this type of situation again requiring some user supplied information to help the process. What is significant here is that we have a set of converters instantiated once at runtime at the topology creation (at the prepare method of each component, spout or bolt) and we don’t have to check for every incoming tuple and value what is its corresponding type. This methodology is quicker and much more robust but requires more user interaction with the system.

To sum up with this strategy now the spouts can handle unwinding simple messages and resolving them to their corresponding type. The approach we used on the previous chapter about declaring fields and streams is also being used at the spouts with one minor setback: The kafka – spout does not support natively the ability to send to many streams. This can be tackled however until this matter is resolved (from the storm team) with another approach: we can chain any spout to a bolt and delegate the responsibility to emit to many streams with the use of the NoAlgorithm (actually an algorithm that does nothing) but the wrapping bolt can handle sending the tuples to multiple streams.

Unfortunately this was a setback that was impossible to overcome by extending the kafka storm classes since there were being used many inner protected classes and extending this functionality was not possible. An approach using reflection has been considered and will be left to future work.

5. ALGORITHMS EMPOWERING THE FUSION ENGINE

A set of algorithms has been provided with the first version of the Fusion engine in order to give the user some choices regarding the setup of his topology and in order to provide some insight of how the framework should be designed. Apart from utility algorithms like median, max, min etc more complicated algorithms have been implemented such as Cusum and Shewhart that belongs to control charts algorithms, a Bayesian network algorithm and other algorithms that are responsible for merging streams according to the specifications provided by the user(time, sliding and tumbling window). This is not the complete set of algorithms the framework is going to support but new algorithms can be added once we create the underlying framework to support bolt intercommunication and message relaying in a generic fashion.

Following, is a brief description of each algorithm used.

5.1 Shewhart algorithm

This algorithm is also called control chart or Shewhart chart (after Walter A. Shewhart) or process behavior charts and is a statistical process tool that is mainly being used if a process is in state of control. More specifically this algorithm is designed to monitor the process mean and standard deviation for deviations from stability by assuming a normal distribution across the values of the process [38].

By analyzing the control chart we have indications that the process is currently under control and every variation of the values is coming from known sources that are common to the process, therefore no corrections or changes to the control of the process are needed. However if a value that deviates a lot from the current process mean and standard deviation appears then the chart will indicate a sudden shift or change of the process and then correctional actions will have to be taken [39].

In order to achieve this, the Shewhart control chart has a baseline and upper and lower limits that are symmetric about the baseline. Measurements that are outside the limits are considered to be out of control. The baseline for the control chart is the accepted value, an average of the historical check standard values. The upper (UCL) and lower (LCL) control limits are computed by these formulae:

$$UCL = current\_value + k * process\_standard\_deviation$$

$$LCL = current\_value - k * process\_standard\_deviation$$

An example is being shown on Figure 19.

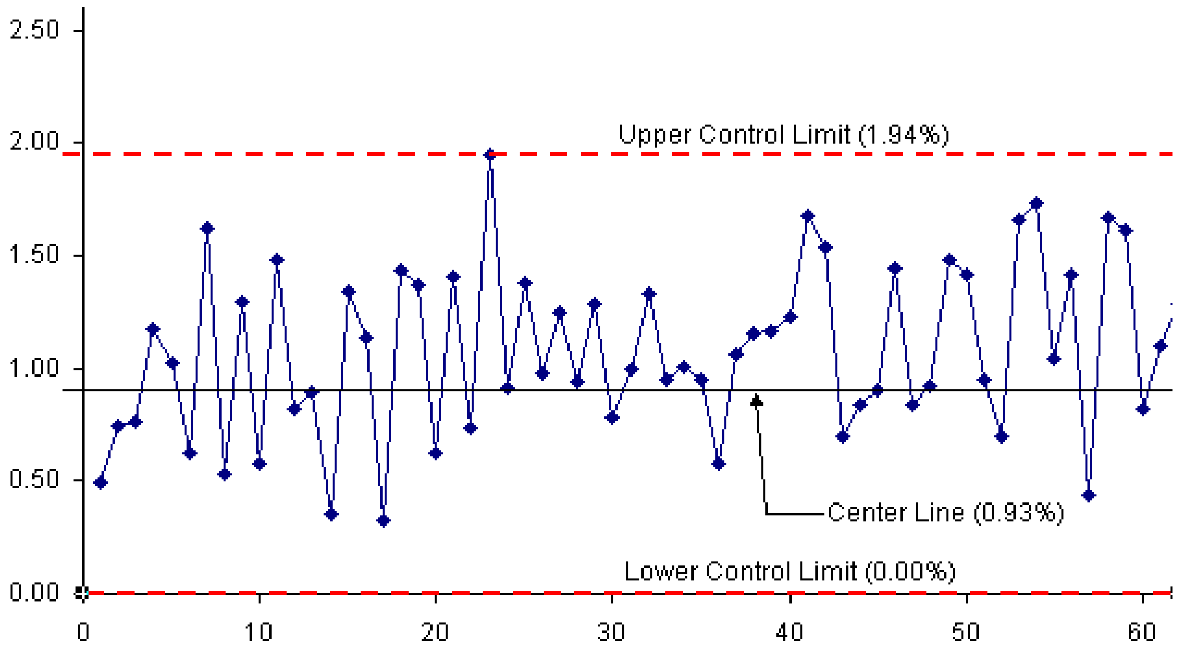

Figure 19 Shewhart example

The value k can be set to 2 in which case approximately 5% of the measurements are likely to produce out-of-control signals. To produce out-of-control signals that are egregiously out of control, the k value can be set to 3 that corresponds to approximately 1% of the total measurements that can produce out-of-control signals [40]. An algorithm showing the steps of the Shewhart control chart follows:

```
ALGORITHM 2. Shewhart Control Chart
Input: univariate time series $x_t$, tightness $k$
Output: detection signal $s$

$\bar{x}_0 \leftarrow 0$;
$\sigma_0 \leftarrow 0$;
$t \leftarrow 1$;
while ( $true$ )
    $\bar{x}_t \leftarrow \bar{x}_{t-1} + \frac{x_t - \bar{x}_{t-1}}{t}$;
    $\sigma_t \leftarrow \sqrt{\frac{1}{t}\left((t-1)\cdot\sigma_{t-1}^2 + (x_t - \bar{x}_t)(x_t - \bar{x}_{t-1})\right)}$;
    $UCL_t \leftarrow \bar{x}_t + k \cdot \sigma_t$;
    $LCL_t \leftarrow \bar{x}_t - k \cdot \sigma_t$;
    if (( $x_t > UCL$ ) or ( $x_t < LCL$ )) then
        $s \leftarrow 1$;
    else
        $s \leftarrow 0$;
        end
        $t \leftarrow t+1$;
end
```

This algorithm is being offered in the Fusion engine and can be used in numerous ways when exploring data incoming from streams. It can detect the presence of outliers so that we can choose to completely ignore those values as errors in the process or we can choose to interpret them as values that require corrective actions. Either way this algorithm applies a transformation to the incoming fields by emitting one extra field called “shewhart” and displays a distinct value of $\{-1,0,-1\}$ for indicating respectively a breach on the lower control lever, no breach, or a breach in the upper control level and the incoming value along with any extra information.

This algorithm decides the upper and lower control limits based on some previous historical data. This means that this algorithm can work with a single incoming value of every tuple, or with a list of tuples or a window. Both versions of this algorithm have been created and it is up to the user to modify them correctly. Windowed algorithms are parameterized externally from the bolt’s context and not at an algorithmic level. Therefore this windowed version of the algorithm could work if the user were to define a tumbling window of values, or a time window.

A windowed version of the algorithm produces again the same, an extra field that indicates if there has been a breach with an option to emit the entire received window along with the extra field [40].

5.2 CUSUM Algorithm

CUSUM algorithm (cumulative sum control chart), like Shewhart algorithm belong to the statistical quality control family of algorithms performs a sequential analysis technique and is ideal for monitoring change detection. The CUSUM (cumulative sum) is used to track the variation of a process. It is a method that is able to detect small shifts in the process’ mean. [41]

The cumulative sum (CUSUM) algorithm attempts to detect a change on the distribution of a time series with respect to a target value at real-time. Specifically, we consider a univariate time series consisting of data values collected over time and a target value μ for this data stream. CUSUM involves the calculation of positive and negative changes ($P$and N, respectively) in the time series x_t cumulatively over time and it compares these changes to a positive and a negative threshold ($thresh^+$ and $thresh^-$, respectively). Whenever these thresholds are exceeded, a change is reported through the above-detection and below-detection signals (s^+ and s^-, respectively) while the cumulative sums are set to zero. In order to avoid the detection of non-abrupt changes or slow drifts, the algorithm takes into consideration tolerance parameters for positive and negative changes (k^+ and k^-, respectively) [42].

The input parameters for the CUSUM algorithm are the following:

- the target value μ
- the above-tolerance value k^+
- the below-tolerance value k^-
- the above-threshold value $thresh^+$

- the below-threshold value $thresh^-$

The output parameters for the CUSUM algorithm are the following:

- the above-detection signal $s^+ \in \{0,1\}$
- the below-detection signal $s^- \in \{0,1\}$

```
Cumulative Sum (CUSUM)
Input: univariate time series $x_t$, target value $\mu$, above-tolerance $k^+$, below-tolerance $k^-$, above-threshold $thres^+$, below-threshold $thres^-$
Output: above detection signal $s^+$, below detection signal $s^-$
$P \leftarrow 0$;
$N \leftarrow 0$;
$t \leftarrow 1$;
while ( $true$ )
    $s^+ \leftarrow 0$;
    $s^- \leftarrow 0$;
    $P \leftarrow \max\left(0, x_t - \left(\mu + k^+\right) + P\right)$;
    $N \leftarrow \min\left(0, x_t - \left(\mu - k^-\right) + N\right)$;
    if ( $P > thres^+$ ) then
        $s^+ \leftarrow 1$;
        $P \leftarrow 0$;
        $N \leftarrow 0$;
    end
    if ( $N < -thres^-$ ) then
        $s^+ \leftarrow 1$;
        $P \leftarrow 0$;
        $N \leftarrow 0$;
    end
    $t \leftarrow t + 1$;
end
```

The algorithm assumes that the arrived time series follow a normal distribution. In order the algorithm to work properly, the tolerance and threshold parameters should be tuned in a way that determines what an actual change is for a specific time-series [43].

This tuning can be performed by following these steps:

- Start with large $thresh^+, thresh^-$ values.

- Choose k^{+}, k^{-} parameters to half of the expected change, or adjust them such that P, N are zero more than half of the times.
- Then set the $thresh^{+}, thresh^{-}$values so that the required number of false alarms or the required delay for detection is obtained.
- If faster detection is sought, try to decrease k^{+}, k^{-} values.
- If fewer false alarms are desired or changes that do not make sense are detected, try to increase k^{+}, k^{-} values.

The upcoming Figure 19 illustrates an example of the CUSUM algorithm over a sensor stream where two changes (a positive and a negative one) are detected. The target value for x_t is set to $\mu = 0.5$, the tolerance values are set to $k+= k-= 0.3$ and the threshold values are determined to $thresh^{+} = thresh^{-} = 1.3$. Specifically, the next figure presents the original sensor data and the time steps where a change is detected by the algorithm.

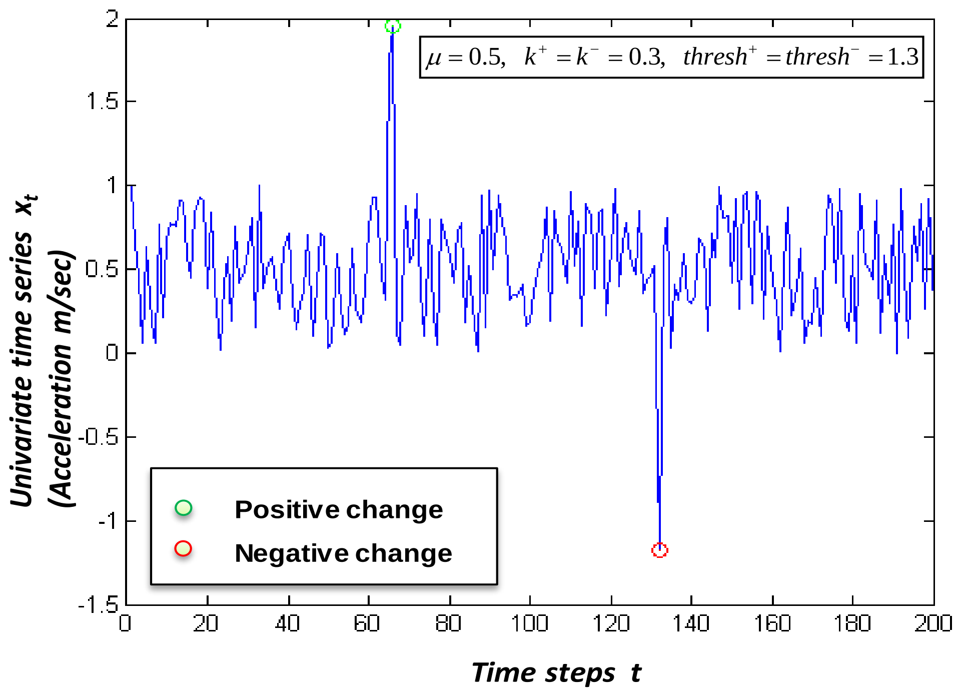

Figure 20 CUSUM on stream values

The next figure (Figure 20) depicts the cumulative sums of positive and negative changes over time for x_t . Obviously, in case of multivariate sensor data, CUSUM should be applied to each variable separately.

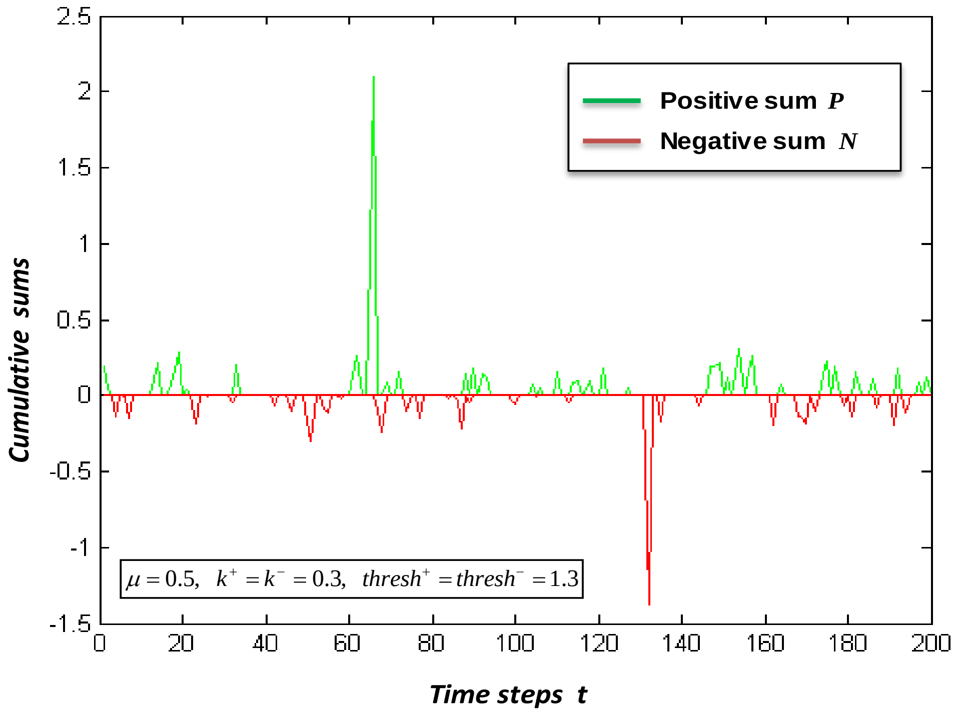

Figure 21 CUSUM chart

5.3 Bayesian Networks

A Bayesian network, or a belief network is a probabilistic directed acyclic graphical (DAG) model, a type of statistical model that represents our limited view of a an uncertain domain. Each node of the network represents a random variable while the edges connecting the nodes according to our understanding of the domain represent the conditional dependencies among the nodes [44].

Each conditional dependence in the graph is being estimated by using probabilistic and statistical estimation methods and mainly Bayes theorem (alternatively Bayes' law or Bayes' rule):

$$P(A \mid B) = \frac{P(B \mid A) * P(A)}{P(B)}$$

Each node in the directed acyclic graph represents the states we would like to model, usually drawn in a circle and given a name, and the set of edges that are essentially the connection between the random variables (or the nodes). Each edge corresponds to a direct dependence among the two connected nodes, or roughly speaking that an edge from node A to node B indicates that variable A influences variable B. Node A then is considered parent of node B, or its ancestor and node B is considered child, or descendant of node A. Each node participating in the graph must have a kind of relationship with another node, and not with itself (a node cannot be its own ancestor or its own descendant), thus ensuring the acyclic nature of the graph [45][46].

Essentially a Bayesian network is a network that represents the causal probabilistic relationship among a set of random variables, their conditional dependences, and it provides a compact representation of a joint probability distribution.

Bayesian networks are not exact representations of a situation that we are trying to model, in fact our knowledge of the situation is incomplete but we can be certain that causality plays an important role.

An example of a Bayesian network for illustration is being shown below on Figure 22.

In this example we would like to know if our family is at home before trying the doors. When someone leaves the house the outdoor light is on, but sometimes the light is on whenever there is a guest at the house. Also the dog is being put on the back yard when nobody is home, but there might be the case that the dog has a bowel problem therefore is being put outside. If the dog is outside probably he is going to bark but we might get confused by another dog's bark.

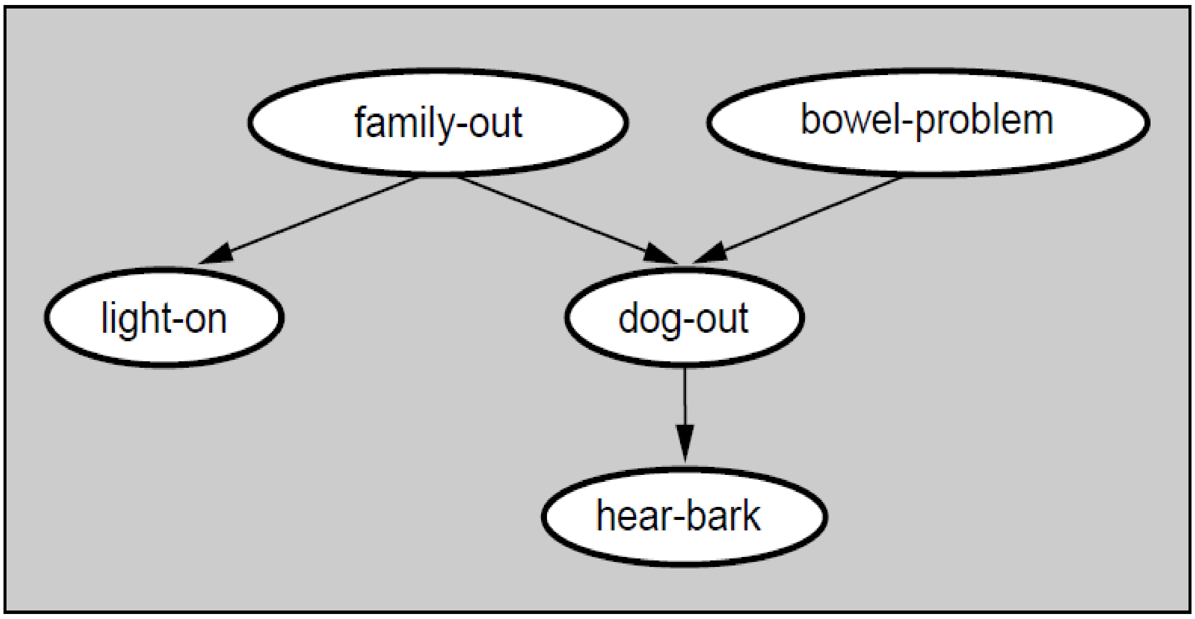

Figure 22 Bayesian Network - Example

This is essentially our model of a situation and by providing probabilities to any node on the graph we can ask the network to provide us the probabilities of some events given our evidence. For example what would be the probability that nobody's home given that the light is off and the dog outside [46]. This methodology is called *inference*.[47]

There are a number of ways we can inference probabilities in Bayesian networks and usually we prefer an approximate inference of the network since it's not so hard or heavy computationally to do so especially if we have a large graph that has many nodes.

In the scope of this system a library for inference on Bayesian networks is being used so that the user can use such abstract models to gain a more concrete possibility of an outcome of interest.

This library is called Jayes [48] and is open licensed and under the auspices of the Eclipse project. This library does not however provide the ability to train the network by feeding it data. It is left to the user when creating the nodes and its descendants or ascendants to provide the probability distribution of each event.

We have integrated this library on the Fusion engine on a single value algorithm that requires as input on each step a subset of the model nodes as evidence and outputs the node we would like to inference upon.

For example a user connects some sensor streams such as humidity, temperature, rain and foggy levels and would like to calculate the possibility of a fire incident or a hurricane. By providing the conditional probabilities to the network, he can then infer the possibility of such outcomes and apply preventive measures.

However in order for this algorithm to work some kind of classification of values needs to be applied before the data is being fed to the Bayesian network. A temperature stream is a stream of continuous numerical values while the Bayesian network requires some kind of characterization of the evidence. When creating the Bayesian network discrete values of state are being given for each node (e.g. temperature -> {low, medium, high}). Therefore a classification of incoming values is needed for the Bayesian network. This sequence of algorithms expected ultimately imposes some kind of logic recipe as previously discussed. One solution to this problem would be to integrate the classification algorithm to the Bayesian network but this is left for future work.

5.4 Stream Mergers

Stream merging is an important aspect of any stream computing engine. When incoming streams are being processed in a system there is many times the requirement for them to merge in order to process a certain subset of their value together or to join those streams. Storm provides many ways to group or join streams, such as windowed bolts where you can choose to merge an unbounded stream of data into finite sets based on some criteria such as time. A window can be conceptualized as an in memory table in which values are being added and removed based on a set of policies [49].

Storm provides two kinds of window functionalities that the user can implement: sliding and tumbling windows [50].

Each window is being specified with the following two parameters:

1. Window length
2. Sliding interval

In a *sliding window*, tuples are grouped within a window that slides across the data stream according to a specified interval. A time-based sliding window with a length of ten seconds and a sliding interval of five seconds contains tuples that arrive within a ten-second window. The set of tuples within the window are evaluated every five seconds. Sliding windows can contain overlapping data; an event can belong to more than one sliding window.

In a *tumbling window*, tuples are grouped in a single window based on time or count. A tuple belongs to only one window. For example a time-based tumbling window with a length of five seconds would be firing up at the end of the fifth second interval evaluating all contained values that arrived during the beginning of this window's count up to the fifth second (0,5]. After that the window would fire up again at the 10th second (5,10] evaluating any received tuples during this durations, none of the windows overlap and the same tuple cannot be present on both windows. Each segment represents a distinct time segment.

Storm supports the configuration of such windows based on count and time. A window (sliding or tumbling) can be configured to fire up on every x incoming elements or on every x seconds. Furthermore the time evaluation can be extracted from the elements of the streams or stream if a timestamp field is present which has to be specified to the window manager. This way we can group or evaluate windows of data that for us are considered discrete time events (e.g. every one second).

Based on this we have created a merger algorithm that can merge the contents of 1 to N streams based on time. This algorithm simply gets a window configured by the user and merges all incoming values into an outgoing tuple that contains information about the stream or streams that they came from and the actual values.

This significantly changes the way this fusion engine handles input and output so far, nevertheless it greatly needed when processing streams. The transformation imposed by this algorithm has led to consider alternative approaches on how this engine should handle incoming and outgoing bolt data.

Those were some of the main algorithms contained in the Fusion engine package and surely more are to come. Algorithms such as min/max, threshold, fields filtering and others are not being mentioned since they are considered utility algorithms.

6. FUSION TUPLES

The main issue that we have encountered when designing the framework was that by chaining algorithms, with each algorithm applying a transformation on the incoming fields at each step, compatibility issues arose between the chained algorithms.

Storm although it does enforce specific type of data structure being passed between the components in the system it does not stop the programmer from creating his own. As we have already mentioned Storm's basic data type is a Tuple, which is essentially a wrapper of any kind of object that addresses elements in a sequence or by name. A tuple actually is a list of objects that can be accessed as a map as well. Clearly the preferred way of Storm would be to avoid any complex data structures being passed into the topology since it is a real time processing engine and at those systems speed is of the essence. Apart from that, Storm offers various types of grouping that would become useless where more complex structures to be used. In our situation however, with the plethora of algorithms we would like to support it is difficult to support extreme transformations of fields between all algorithms.

For example the merger algorithm completely transforms the incoming messages producing a map of streams to values since it is responsible for merging streams. This is a complete overhaul of the enclosed objects in the tuple; up until this algorithm an array of elements was sufficient, but after this algorithm we now have an enclosed map inside the tuple and essentially only one field declared in the tuple: this map. Those kinds of transformations potentially can be the source of many issues and bugs when designing a generic framework that can essentially take care of connecting any algorithm.

Although passing complex data structures can reduce the speed of the system we might consider trading this speed for this kind of flexibility. Therefore given any algorithm that applies a transformation in the incoming fields the data structure we have implemented is able to hold values of many streams as well as information about them.

Therefore a map of stream names which point to a list of values and a map of metadata for each stream, which point to field metadata, has been created. This is called a fusion tuple and can potentially solve all data aggregation and transformation functions that can happen during the flow of the topology.

With the use of such structure we can emit a single array from a stream, essentially what we called before a tuple, but many as well. At the same tuple we can insert many or one value from each stream. For accessing the values included we must consult the metadata info about the stream. The example that follows shows the data structure:

```java
public class FusionTuple {
    Map<String, List<Values>> valueMap;
    Map<String, List<Meta>> metaMap;

}
```

The value class is a list of Objects and the Meta class is a class containing a triplet that indicates the field name and the class of a contained object in a stream.

For example if we had two streams(stream-1, stream-2) with fields [“id”, “value”, “timestamp”] and [“id”, “value”, “timestamp”, “max”] we can now populate this data structure to hold values of both streams. The metadata of each stream would indicate what their field names are and what their class is.

Of course we could imagine another data structure that could hold this kind of information and many more might be chosen to be tested in case we were to fully adopt this approach. Note that at this point if we choose to work with any kind of more complex structure such as this we lose the ability to do grouping on the fields, which is a handy aspect of storm. Right now simple tuple messages are supported and this kind of messaging as well, mainly for testing how these two approaches could work in terms of speed.

Slight modifications to each algorithm have been made to support this kind of data structure and we compare the speed of the system using both ways. Notice that now that we use a complex data structure serialization and deserialization should be used to avoid concurrent modifications of the same object (if storm is deployed on a local cluster a local transfer is being used between bolts- threads) [51].

At the next figure (Figure 23) we measure the average time needed for each object to be parsed using the pattern we have specified and also we measure the average construction time of each fusion tuple object. There were 10 rounds conducting the same operations and in each round 200.000 objects were being created up a total of two million. After the java optimizer kicks in we see a significant time decrease for each object creation. This chart shows that the approach for creating fusion tuples which contain also some metadata about what is the stream name that produced this object, its fields etc, effectively doubles the processing time for each tuple when compared with the simple tuple approach which is only encumbered with the parsing phase.

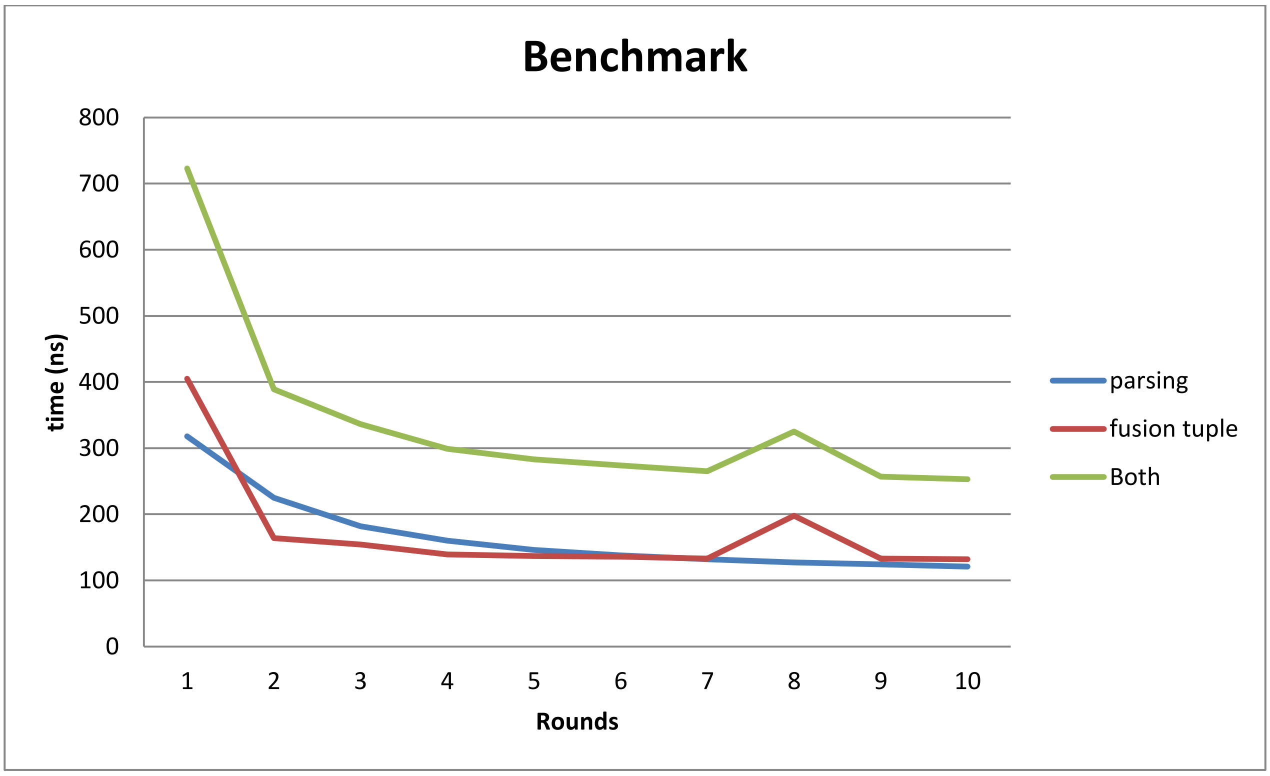

Figure 23 Parsing time and fusion tuple creation time

7. TOWARDS A MORE HOLISTIC APPROACH

One of the main problems encountered when designing and implementing the framework on top of Storm is the data connection among the algorithms implemented. This is major issue that can be approached in many ways.

Any algorithm by default is data-centric and by this we mean that when a generic stream is inserted into the system, some extra information is needed for each algorithm to operate. CUSUM algorithm for example needs to know the field that is going to monitor and calculate the upper and lower cumulative sums. This is solved by indicating to each algorithm one, or many, depending on the algorithm fields of interest.

The main issue arises when data is forwarded from one algorithm to another. As we have already stated each algorithm applies a transformation to each incoming tuple. Some algorithms might be called to work on each incoming tuple, while others might need a set of values to work on. Although the set of algorithms we have worked on so far is limited, we should take into consideration that even more algorithms will be added and those algorithms might require even more unique transformations to the data format they require in order to operate correctly.

This greatly creates the need for a more general approach in order to be able to incorporate any algorithm without requiring any extra changes to the underlying framework.

So far we have seen two approaches:

1. The main approach, where we use simple variables wrapped inside of a tuple. In this approach the problem was that there was a lot of effort maintaining the fields declared and forwarding them across the declarer of each participating bolt. Furthermore it becomes really perplexing when using this scheme to implement Stream merging and other much more complex transformations to the streams. This would require for anyone implementing a new algorithm to essentially know all the formats emitted by each algorithm, that would make sense to be chained with his algorithm, and the same applies to every algorithm following.

2. The Fusion tuple approach, where we have designed a flexible but large data structure to emit at each step. This data structure potentially can withstand many algorithm transformations but is has an increased space complexity. Subsequently it creates much more load to the system and sometimes this data structure might contain simple values that could be emitted using the main approach. Additionally this approach disables some of Storm’s handy features such as fields grouping that will definitely prove to be useful when deploying to clusters and scaling the topology.

While both approaches could be used with their downsides this is an interesting topic of discussion on how could we implement a structured data communication between the algorithms so that we could minimize the impact of both of those approaches.

One approach would be to keep the data structures as simple as possible and leveraging Storm’s functionality while at the same time going to more complex data structures as the need arises.

If we could characterize each stream according to what kind of data structures it contains we could implement a mapping of algorithms to specific stream types. This would essentially create a set of families of algorithms with their corresponding incoming stream type (i.e. normal tuples) vs more complex streams and types contained (multi-value stream, after a merger algorithm). This approach is being shown on figure 24.

Therefore, one solution that might be possible would be to fine grain the communication between the algorithms at an early stage of the topology where the streams are not joined and fairly simple by using Storm's approach, by passing simple variables inside the tuples and as the need arises we could advance to more complex schemes. Each algorithm depending on its data-in scheme could be allocated on sets of families that could belong to.

Another approach which can take advantage of simple and complex data schemes is inspired from the adapter design pattern (sort of). Instead of enforcing the creation of a common "language" or families of "languages" (stream with specific data types) we could make each algorithm "adapt" to the next algorithm chained. To do so each algorithm should be able to encapsulate at least one "data translator" or "formatter" whose sole purpose is to transform each tuple, or windows of tuples, to the format that the next algorithm is able to understand.

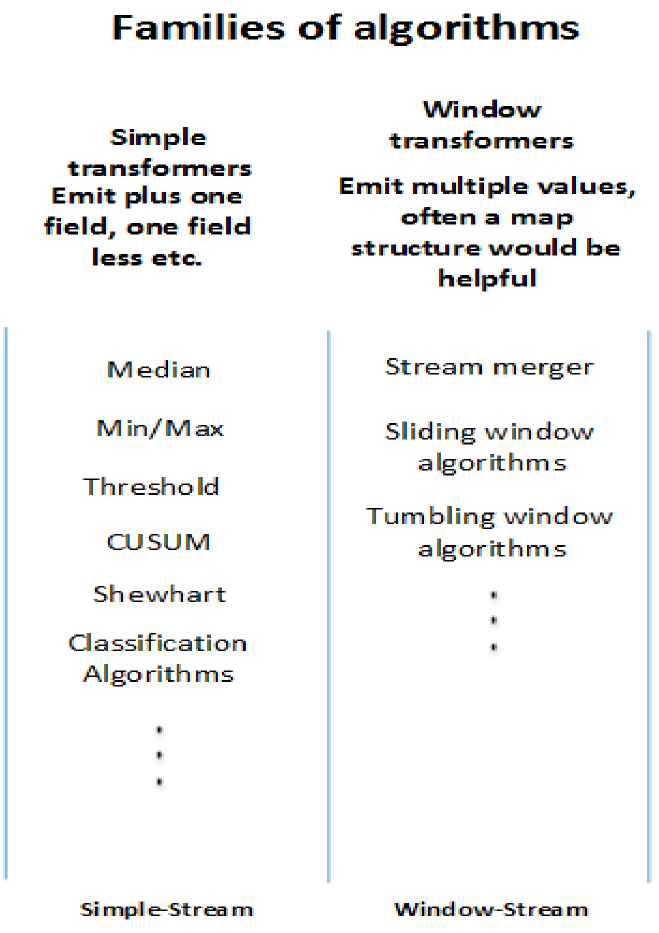
Families of algorithms
Simple transformers
Emit plus one field, one field less etc.
Window transformers
Emit multiple values, often a map structure would be helpful
Median
Min/Max
Threshold
CUSUM
Shewhart
Classification Algorithms
Stream merger
Sliding window algorithms
Tumbling window algorithms
Simple-Stream
Window-Stream

Figure 24 Classification of algorithms and streams

To facilitate this change when implementing an algorithm we must implement a formatter as well, responsible for structuring each incoming value(s) to the preferred way of the algorithm. Each algorithm chained with another will use the formatter of the next algorithm. After the first algorithm has finished its work on a tuple, or on a tuple window, those values will be offered to the formatter(the formatter of the next algorithm) in order to transform them to the format that the next algorithm expects. This approach can support the use of simple schemas but more complex ones as well. An example is being shown on Figure 25. Each formatter that the algorithm holds is the respective formatter specified by Algorithm A, B or C.

However, transforming each tuple with the use of formatters further adds to the complexity of the system. Furthermore the use of formatters should not be abused; when someone decides to implement a new algorithm, he should consider first using the existing formatters already implemented rather than opting for a new one that will create really complex structures.

At this point it was made really clear from the design of the framework that there is no magic solution or a way we could make each algorithm work regardless of what data structure it expects. We have proposed some approaches that we consider that could be able to solve this connectivity issue, however this is a work in progress. Other approaches more complex will be proposed and hopefully by comparing each one the optimal will be selected.

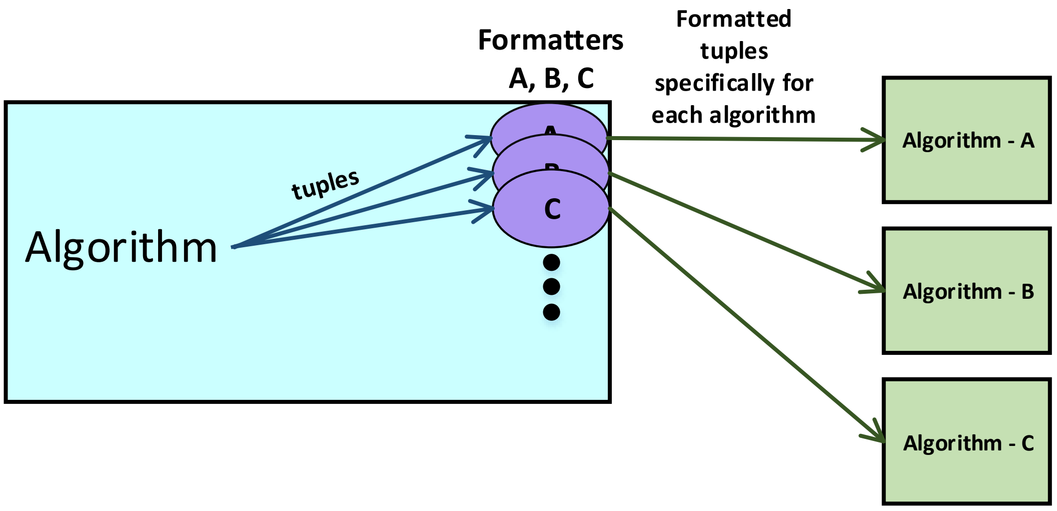

Figure 25 Algorithms and formatters

8. EXPLORING STORM'S FULL POTENTIAL

Up to now we have visited the storm internals in terms of message passing, topology creation all in the scope of making algorithms able to connect with each other in a generic way. However Storm is much more; Storm is a distributed system that is scalable. Every bolt is essentially a thread running and we can scale the number of threads that each bolt can have. Scaling however without taking into consideration in our case what is the enclosed algorithm can have fatal results. Algorithms that are stateful cannot be easily scaled to a parallelism bigger than one; but that always depends on the topology we have created.

Algorithms such as CUSUM and Shewhart that depend on the previous values for historical data cannot be scaled since Storm when having many instances of a bolt – therefore of an algorithm in our case, will emit tuples to the most suited instance by using a round robin approach while taking in consideration various aspects of the thread, such as its load etc. That means that if we have for example a CUSUM algorithm with parallelism of five, the sequential values arriving at the previous component will not be emitted to the same instance of the CUSUM algorithm therefore we will definitely end up with data that are not logically consistent.

There are some approaches that can be used to solve each problem but they are most dependent on the algorithm enclosed and the topology graph. Other algorithms that are stateless can be scaled without fear of ending up with inconsistent data such as thresholding algorithms, Bayesian networks etc.

9. CONCLUSION

Storm so far has proven a viable solution for what we have envisioned. The many turning knobs and parts it has offer great versatility so that a framework can be built on top of it. In our case this versatility can be a problem when we would like to shield the user from creating a topology that would not have much meaning or could lead to potentially incorrect conclusions.

There many aspects that still need to be addressed so that this framework could be considered ready to be used and this is a work in progress. Each time we encounter a new algorithm that we would like to integrate could potentially break the existing design and be forced to readapt the entire framework to a more general approach. There are also many open issues that need to be resolved such as parallelism and deploying topologies on a cluster and even more will arise.

Until now this generic framework seems to be working fine and shows great potential for growth and could become the fusion engine we have envisioned. Future work on this fusion engine would certainly require the design of a graphical user interface where the user would just drag and drop algorithms in order to create the desired topology. Logical validation of a topology would be something greatly needed in order to help inexperienced users with the algorithm suite that we would offer. The same could happen with parallelism hints.

Overall this is a work in progress that shows great potential and was really interesting to try to transform an existing streaming engine into something much more.

ABBREVIATIONS - ACRONYMS

UOA	University of Athens
IoT	Internet of Things
5G	Fifth generation mobile networks
4G	Fourth generation mobile networks
HDFS	Hadoop Distributed File System
YARN	Yet Another Resource Negotiator
RDD	Resilient Distributed Datasets
SQL	Structured Query Language
API	Application Programming Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
JVM	Java Virtual Machine
UI	User Interface
CLI	Command Line Interface
DSL	Domain Specific Language
DPE	Data Processing Engine
YAML	YAML Ain't Markup Language
Args	Arguments
CUSUM	Cumulative Sum Control Chart

REFERENCES

[1] L. Atzori, A. Iear, and G. Morabito, “The Internet of Things: A survey,” *Computer Netowrks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] A. Praveena and B. Bharanti, “A survey paper on big data analytics,” *2017 International Conference on Information and Embedded Systems*.

[3] “Batch processing - Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/Batch_processing. [Accessed: 14-Feb-2018].

[4] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] “Apache Hadoop.” [Online]. Available: http://hadoop.apache.org/. [Accessed: 14-Feb-2018].

[6] “Apache Hadoop 2.9.0 – HDFS Architecture.” [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. [Accessed: 14-Feb-2018].

[7] “Hadoop MapReduce vs. Apache Spark - DZone Big Data,” *dzone.com*. [Online]. Available: https://dzone.com/articles/apache-hadoop-vs-apache-spark. [Accessed: 14-Feb-2018].

[8] I. Pointer, “What is Apache Spark? The big data analytics platform explained,” *InfoWorld*, 13-Nov-2017. [Online]. Available: https://www.infoworld.com/article/3236869/analytics/what-is-apache-spark-the-big-data-analytics-platform-explained.html. [Accessed: 14-Feb-2018].

[9] “Five things you need to know about Hadoop v. Apache Spark | InfoWorld.” [Online]. Available: https://www.infoworld.com/article/3014440/big-data/five-things-you-need-to-know-about-hadoop-v-apache-spark.html. [Accessed: 14-Feb-2018].

[10] “Apache Spark Architecture Explained in Detail.” [Online]. Available: https://www.dezyre.com/article/apache-spark-architecture-explained-in-detail/338. [Accessed: 14-Feb-2018].

[11] “How LinkedIn Uses Apache Samza.” [Online]. Available: https://www.infoq.com/articles/linkedin-samza. [Accessed: 14-Feb-2018].

[12] “Apache Samza, LinkedIn’s Framework for Stream Processing,” *The New Stack*, 07-Jan-2015.

[13] “Apache Samza,” *Wikipedia*. 14-Oct-2017.

[14] J. Kreps, “Introducing Kafka Streams: Stream Processing Made Simple,” *Confluent*, 10-Mar-2016. [Online]. Available: https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/. [Accessed: 14-Feb-2018].

[15] “How to Use the Kafka Streams API - DZone Big Data.” [Online]. Available: https://dzone.com/articles/kafka-streams-more-than-just-dumb-storage. [Accessed: 14-Feb-2018].

[16] “SoftwareMill blog: Kafka Streams - how does it fit the stream processing landscape?,” *SoftwareMill - Scala, Big Data, Java, Cloud*. [Online]. Available: https://softwaremill.com/kafka-streams-how-does-it-fit-stream-landscape. [Accessed: 14-Feb-2018].

[17] “Apache Kafka,” *Apache Kafka*. [Online]. Available: https://kafka.apache.org/10/documentation/streams/core-concepts. [Accessed: 14-Feb-2018].

[18] “Apache Flink 1.4 Documentation: Dataflow Programming Model.” [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.4/concepts/programming-model.html. [Accessed: 14-Feb-2018].

[19] “Apache Storm.” [Online]. Available: http://storm.apache.org/. [Accessed: 14-Feb-2018].

[20] “Storm, distributed and fault-tolerant realtime computation”, Narthan Marz, Twitter, Presentation

[21] T. S. Allen, M. Jankowski, and P. Pathirama, “Storm Applied,” Manning Publications.

[22] J. Leibuisky, G. Eisbruch, and D. Simonassi, *Getting Started With Storm*. O’Reilly Publications.

[23] “Apache Storm Concepts.” [Online]. Available: http://storm.apache.org/releases/current/Concepts.html. [Accessed: 14-Feb-2018].

[24] G. Hesse and M. Lorenz, “Conceptual Survey on Data Stream Processing Systems,” in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015, pp. 797–802.

[25] J. Ankit, *Mastering Apache Storm*. Pakt Publishing.

[26] “Apache Storm - Tutorial.” [Online]. Available: http://storm.apache.org/releases/current/Tutorial.html. [Accessed: 14-Feb-2018].

[27] “ISpout (Storm 0.9.6 API)” [Online].Available: https://storm.apache.org/releases/0.9.6/javadocs/backtype/storm/spout/ISpout.html#nextTuple--]. [Accessed: 14-Feb-2018].

[28] “IWindowedBolt (Storm 1.0.4 API).” [Online]. Available: https://storm.apache.org/releases/1.0.4/javadocs/org/apache/storm/topology/IWindowedBolt.html. [Accessed: 14-Feb-2018].

[29] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, “The power of both choices: Practical load balancing for distributed stream processing engines,” in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 137–148.

[30] “Stream Groupings - Hortonworks Data Platform.” [Online]. Available: https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.6.4/bk_storm-component-guide/content/storm-stream-groupings.html. [Accessed: 14-Feb-2018].

[31] “Understanding the Parallelism of a Storm Topology.” [Online]. Available: http://storm.apache.org/releases/1.1.1/Understanding-the-parallelism-of-a-Storm-topology.html. [Accessed: 14-Feb-2018].

[32] “Understanding the Parallelism of a Storm Topology - Michael G. Noll.” [Online]. Available: http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/. [Accessed: 14-Feb-2018].

[33] “Parallelism - Hortonworks Data Platform.” [Online]. Available: https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.6.4/bk_storm-component-guide/content/storm-parallelism.html. [Accessed: 14-Feb-2018].

[34] C. Kolski and J. Vanderdonckt, *Computer - Aided Design of User Interfaces III*. Springer Publishing, 2002.

[35] Joelle Courtaz and Gaetan Ray, *Foundations for a theory of contextors*.

[36] “Flux.” [Online]. Available: http://storm.apache.org/releases/2.0.0-SNAPSHOT/flux.html. [Accessed: 14-Feb-2018].

[37] “Tuple (Storm 0.9.6 API).” [Online]. Available: https://storm.apache.org/releases/0.9.6/javadocs/backtype/storm/tuple/Tuple.html. [Accessed: 14-Feb-2018].

[38] S. H. Steiner, P. Lee Geyer, and G. O. Wesolowsky, “Shewhart control charts to detect mean and standard deviation shifts based on grouped data,” *Qual. Reliab. Engng. Int.*, vol. 12, no. 5, pp. 345–353, Sep. 1996.

[39] “Control chart,” *Wikipedia*. 06-Dec-2017.

[40] “2.2.2.1. Shewhart control chart” [Online]. Available: http://www.itl.nist.gov/div898/handbook/mpc/section2/mpc221.htm. [Accessed: 14-Feb-2018].

[41] “CUSUM” *Wikipedia*. 20-Jun-2017.

[42] “The CUSUM algorithm a small review”, Pierre Granjon, 2012

[43] “CUSUM Anomaly Detection”, MLAB, Kinga Farkas, 2016

[44] “Bayesian network” *Wikipedia*. 30-Dec-2017.

[45] “Bayesian Networks”, Ben-Gal I., Encyclopedia of Statistics in Quality & Reliability, Wiley & Sons (2007)

[46] E. Charniak, “Bayesian Networks Without Tears: Making Bayesian Networks More Accessible to the Probabilistically Unsophisticated” *AI Mag.*, vol. 12, no. 4, pp. 50–63, Nov. 1991.

[47] “Weng-Keen Wong, Oregon State University © Bayesian Networks: A Tutorial Weng-Keen Wong School of Electrical Engineering and Computer Science Oregon” [Online]. Available: http://slideplayer.com/slide/10631929/. [Accessed: 14-Feb-2018].

[48] “Jayes | Code Recommenders” [Online]. Available: http://www.eclipse.org/recommenders/jayes/. [Accessed: 14-Feb-2018].

[49] “Understanding Sliding and Tumbling Windows - Hortonworks Data Platform” [Online]. Available: https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.6.0/bk_storm-component-guide/content/storm-windowing-concepts.html. [Accessed: 14-Feb-2018].

[50] “Windowing Support in Core Storm” [Online]. Available: http://storm.apache.org/releases/2.0.0-SNAPSHOT/Windowing.html. [Accessed: 14-Feb-2018].

[51] “Serialization” [Online]. Available: http://storm.apache.org/releases/current/Serialization.html. [Accessed: 14-Feb-2018].