November 2017

Facoltà di Ingegneria Informatica

UNIVERSITÀ DEGLI STUDI
DI SALERNO

Facultat d'Informàtica de Barcelona

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

MASTER

IN

INNOVATION AND RESEARCH IN

INFORMATICS

SEMANTIC STREAM COMPUTING FOR LARGE DATA-SET ANALYTICS

Supervisors:

Prof. Pierluigi Ritrovato

Prof. Fatos Xhafa

Candidate:

Andrea Giordano

Academic Year 2016/2017

Abstract

Today, one of the most interesting research fields in IT industries is the quest of software able to deal efficiently with Big Data and taking advantages coming from the use of applications which produce them. Some of them are so performant that are largely used from the most important Big Data dealing companies in the world like Netflix, Facebook and Google. A side aspect but not the least important is tied to the infrastructures generally used in conjunction with these technologies called Cloud computing which exploits the power of several, usually a lot of, physical machines to perform very complex task with high performances outcome.

In this thesis firstly we will study Big Data panorama as a whole, especially about their spread in the IT industries and in healthcare, where data are generated at an extremely high rate and the use of computer science can save or improve lives; moreover, the challenges and a vision of the potential uses of them were rapidly covered together with a view on so-called wearables: new sensors applied on clothes or directly on the skin of people which already are revolutionizing our lives. In this dissertation the *Introduction*, the chapter 2 are dedicated to these subjects.

The aim of this project was to design and develop a system composed of a cluster intended for medical use where the core is a real-time detection of anomalous data originated by wearable sensors. A lot of papers and technical

reports of scientific community were consulted to investigate the State of the Art of Big Data handling technologies: studying their abilities and properties, finding strengths and weakness of each one and comparing them in order to choose the most adequate set of software to achieve the aim of the project.

Therefore, an original architecture was designed assembling several technologies and taking into account requirements and targets to reach: every design choice was justified and a particular attention was paid to assure the essential IT properties of modularity and simplicity without giving up to performances. In this part the sections 5.3 - 5.6 present the involved devices and some contributes brought to already existing tools; additionally, an original approach to handle semantic-enriched data streams is illustrated, taking inspiration from an innovative way submitted at the latest ISWC conference. Furthermore, the document presents the development of a prototype based on the designed system and shows the outcomes of some experiments performed on it: these one concern resource's usage in different operating conditions on the nodes of the cluster. In the chapter 5 are also showed the results obtained using the employed anomaly detection algorithm.

Finally, *Future works* proposes potential improvements adding a semantic enrichment to the system in order to reach higher achievements about medical diagnosis and comprehension of data. The chapter 8 section lists the most celeb existent technologies to do it and shows how their integration with the presented system can provide substantial advantages in Healthcare as well as illustrates obstacles and challenges to overcome by presenting a good starting point to implement further developments.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last years the term "Big Data" was largely used in IT environment. In order to better clarify its meaning we report the Oxford Dictionary definition of Big Data:

> "*Extremely large data sets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behaviour and interactions*"

while US Congress specifies it as [13]:

> "*large volumes of high velocity, complex, and variable data that require advanced techniques and technologies to enable the capture, storage, distribution, management and analysis of the information*"

Hence the use of advanced technologies is essential to deal with Big Data: retrieving data generated with an high rate and then analyze them in real-time or without loss is a complex task often unbearable with a single common server or using typical software. Lately, Big Data dealing techniques are mixed with another raising field of computer science: the Semantic web

technologies. Semantic data analysis adds meanings to raw, and sometimes apparently useless, data pulled out from sensors and other devices that would be otherwise discarded and unused: so, semantic technology allows to multiply the already huge amount of knowledge extracted from the fetched Big Data.

In particular in this thesis we want to investigate the case of use represented by the healthcare, where Big Data handling and semantic technologies help to improve level of medical treatments and to save lives. The current worldwide healthcare situation represents both a big challenge and a great opportunity for IT industries, which operate within Big Data. Definitely, the need for better medical instruments of hospitals and clinics meets the rising technology capabilities of engineering and computer science particularly. Since the healthcare is a massive producer's place of data, it is one of the most important fields for the application of Big Data technologies: in 2011 already US Healthcare generated 150 exabytes of data, so it is reasonable to think that this amount can easily reach the incredible number of some yottabytes [13]. No one non-specialized infrastructure could be able to examine this impressive volume of data in a reasonable timeframe so there is a strong need for new software architectures to handle it. Accordingly to more recent stats in 2015 the 69% of US citizens track their health with various sensors while the same stat in UK reaches the 70% [25]. Moreover, in [26] is reported that the number of healthcare IT devices (without considering wearable sensors) reached 95 millions of units in 2015 whereas a growth up to 646 millions in 2020 is estimated: again, an adequate Big Data infrastructure has to manage a dramatic number of data producers and consumers as well as a big amount of informations. Generally, dealing with Big Data is a very complex task but the difficulties faced by a system raise

when the computation has to be performed with real-time requirements: in this case applications have to fetch, evaluate and in some case store data within few sub-seconds without errors. Most modern hospitals are becoming increasingly instrumented with new technologies: every beds in critical care units have lots of sensors to measure vital signs like heart rate, breath rate, blood pressure and other devices producing large volumes of physiological data. One of the most interesting category of sensing devices is composed of wearable sensors to monitor human signs as well as geographical body position, arms orientation and every secondary information could be extract from these parameters. Typical wearables are wristbands, headsets, smart glasses or even clothes but also smartphone belong to this family: according to [21] just in 2014 15 million wearable smart devices were sold.

The main subjects who would take advantages applying these technologies are the patients: unfortunately, despite the recent improvements, today in a typical hospital scenario health conditions of patients are checked just few times in a day wasting precious time to early react to any symptoms: in fact, hand-made measurement of vital parameters obviously does not permit to do more. Clearly this situation is no more bearable: indeed, preventing and healing diseases beforehand means having the opportunity to free up beds faster and to serve more patients. In [16] is highlighted as the vast majority of data collected by monitoring systems is transient whereas in the other cases they are simply not adequately employed: in fact, these data often are just archived in analogical ways as support for clinicians' decision or for statistical purposes. We talk about petabytes of very useful underestimated - or worse discarded - data.

Big Data technologies can help healthcare not only in "stream" applications but also for "offline" data elaborations, for instance to evaluate the

impact of medicine on long term or in other fields for statistical administrative analysis like medical prescriptions and insurance. Every type of data can be analyzed and used to improve patient's experience. Again, these technology outcomes can take advantage of a vast amount of historical data; they offer hints about which drugs fit better a specific category of people or estimate post-surgery recovery time needed for particular subjects.

The advantages of using Big Data in healthcare are not limited to the single patients: potential benefits include a greater way to perform medical research with more data and studying population's health over years, predicting epidemic spreads and generally improving quality of life.

In addition to the obvious reasons related to the increase of quality life for people and patients in particular, there are also economic advantages deriving from the analysis of Big Data : indeed, as IT industries also Healthcare changes rapidly on time and today hospitals and clinics are not anymore only places of care but also business opportunities for private companies and government; a good-working hospital means patients have good health and, in most cases, this means great incomes for private companies and public entities. According to [16] Big Data technologies in healthcare provide more than 300 billion of dollars in savings for year just in United States; they reduce waste of money and time in research increasing development of faster and more targeted pipelines in drugs and devices.

This thesis wants to propose a solution to deal with Big Data in a healthcare context. In particular the aim is to create a modular system which can be integrated in pre-existing ones or used as starting point to deploy a more complex architecture. The central feature of the system should be able to detect anomalous values from the data flow originating by wearables attached to subjects involved in the system. The data coming from the sensors should

be quickly fetched and analyzed in real-time in order to give the opportunity to trigger reactions in case of emergency; finally they should be stored in database to provide further consultations or elaboration, maybe in offline manner. Furthermore, the system should follow semantic web convention in order to make it adherent to the state of the art of Big Data technologies.

To date a large number of researchers have reported a number of systems to analyze physiological data streams but not so many of them perform pattern detection or online real-time processing. For instance, the scenario described in [30] remotely monitors patients using data from ECG and accelerometers: the application reports to clinicians periods of elevated heart rate filtering expected critical situations, during a run for example.

The remainder of the thesis is organized in following way: firstly in chapters 2 and 3 the illustrated problem and proposed solution to face it are deepened. In chapter 4 the most important technologies currently employed in distributed computing are compared in order to justify the technical choices adopted in design process. Then, chapter 5 provides details about system's implementation, it describes applied technologies and employed technical approaches. Finally, in order to evaluate designed system some test and experimentation are showed in chapter 6 while in chapter 8 further opportunities to extend this architecture are proposed.

Chapter 2

Problem statement

In *Introduction* we have already addressed briefly the problem faced in this thesis; this section want to go into details to deepen characteristics and issues. The challenges concerning Big Data handling do not stop on the amount of data management, there are also problems about the type of data, which are heterogeneous, their speed generation and the search for a way to store this volume of data. As already seen healthcare is a perfect trial field to experiment Big Data technologies because it presents exactly the challenges of Big Data applications: the huge volume of data since in a typical hospital there are hundreds of patients at the same time, then heterogeneous data since there are multiple sensors which produce different kinds of information, data generation rate because in order to read efficiently vital signs these devices have to operate at high frequency and finally veracity since sensors can transmit misleading data due to low batteries or noise or other factors.

Historically scientific community agrees about the definition of 3 major issues - called *V's* - in Big Data analytics. Storing a vast amount of data, around hundreds of terabytes, in a secure way is a challenge of primary importance and surely it represents the first of these *V's*, known as *Volume*.

This issue requests a novel approach to explore new techniques for spreading storage whereas it should also offers efficient data retrieval: some advances in this direction are already guaranteed by the use of cloud computing infrastructures. The second V is derived directly from the former and it is the *Velocity*. Indeed, Big Data need great speed in retrieving, processing and publishing of new data, especially in applications where data incoming rates is extremely high, as in the case of healthcare. The third major issue is the *Variety* because incoming data from different sources can have different format - structured or unstructured - of plaintext or graph-based or also relational-like data: Big Data applications have to handle, link and process them and also it is necessary to use smart technologies for merging all these kinds of data. Actually some researchers, e.g. [19], consider a forth characteristic of Big Data: *Veracity*. It is also called *data assurance* and it is a measurement of credibility of data: this feature is particularly important in some applications where data integrity and reliability is crucial, like in healthcare.

In [20] other Big Data aspects are described. In particular, there are some challenges about Big Data application lifecycle regarding the three phases of data elaboration: acquisition, cleaning and integration. Acquired raw data represent a massive input stream generated by hundreds, thousands or also million of IoT sources. Hence, an important step could be filter them on-fly and extracting useful information from redundant and noisy flow. Data cleaning instead concerns issues about erroneous or uncertain data and it is related to *Veracity* property: addressing this problem requires online error detection and preferably data correction. Finally, data integration is one of the biggest issue in a Big Data scenario. In IoT there are lots of different sources which generate different type of data and organizing and extracting

useful information from them efficiently is challenging.

Most sensors employed in healthcare are wearables which operate at high frequency, even dozens of reads each second [24]. However they do not offer great processing performance so data analysis has to be performed necessarily on remote systems, depending on complexity, amount of data or application requirements. Again, the author of [20] highlights how centralized servers - as much as powerful they may be - cannot handle all these affairs so a distributed system composed of a number of machines is usually required: each node of the system will be responsible for one or more of above tasks and together they contribute to complete the whole job. In this case the main challenge is the *Timeliness* because distributing data across several machines means they are not all available in local so they need to be moved rapidly and with minimal latency. Last, but not least problem is finding a way to execute real-time analysis of data, automatic annotation and integration with existing ontologies: the latter process usually requires appropriate specialized softwares, very efficient codes and well-equipped machines.

Chapter 3

Objectives and project scope

As we have seen, typical Big Data characteristics are high velocity data generation, large variety, veracity and huge data volume: hence, there is a strong need for solutions to analyze these data. Generally, managing large amounts of data is one of the most important challenges in computer science and engineering; the progress in miniaturization technology, the increase of the available bandwidth and the progressive reduction of costs boosted the production of millions of new devices like environment sensors, smartphones, wearables and so on.

Hospitals and clinics are some of the most important data producers thanks to their huge amount of sources: the patients. They represent a fundamental trial field for emerging Big Data technologies. Simultaneously healthcare represents the typical scenario where wearables can be found: they can be employed to monitor physiological parameters possibly raising alarms to warn medical staff without human intervention and potentially saving patients' lives. Reconsidering observation by [16] is evident that many diseases could be detected early and with more benefits if patients were monitored along 24 hours. In [19] is described the example of *Acute Hypotensive*

Episodes, a disease that needs continuous monitoring because it presents sudden critical events with tragic consequences. Moreover, sometimes nurses and clinicians cannot perceive every symptoms because they are silent or not visible to the human eye. Without the right level of technological support, medical practitioners rely exclusively on their experience whereas with a continuous analysis of vital signs, an automated system can help clinicians' work allowing them to focus on more important aspects like posterior diagnosis.

Today, IT industry has to assist healthcare improving medical experience and saving more lives more efficiently and with minor costs. This thesis aims to investigate a particular scenario composed of a lot of sensors worn by a group of patients: these wearables produce unbounded data flows and the purpose is to design an application able to detect abnormal values within them. Data are ingested at high frequency, analyzed and stored safely in a permanent storage to be available for further evaluations and knowledge extraction: all these operations must be executed in real-time so the infrastructure has to face with Big Data challenges. Also, the system has to be scalable according to the number of subjects involved without requiring architectural changes: moreover, it has to integrate the most recent conventions in semantic web context and it must be easily expandable for future improvements allowing potential replacements of passed technologies.

Chapter 4

State of the art

4.1 Batch computing vs Stream computing

Big Data analytics can be performed mainly in two different ways: batch computing and stream computing. Usually these techniques are used in distributed systems like cluster of computers or even with a network of smaller devices but it is possible also imagine an architecture with one single node which has the responsibility of the entire analytical process. Sometimes batch and stream computing are used together in a parallel or serial infrastructure to enhance the powerful of analysis and to extract more relevant information. This is the case of the framework designed in [23] illustrated in Figure 4.1, where two distributed computing systems run simultaneously on top of a shared broker which virtualizes many sensing sources.

Usually we refer to the batch processing as the automatic execution of a complex software or process on a large series of input data, for instance log computation of a long business task at the end of working day. Generally, batch computing involves interrelated data so processing results are obtained aggregating all the input data. Stream processing is slightly different: we

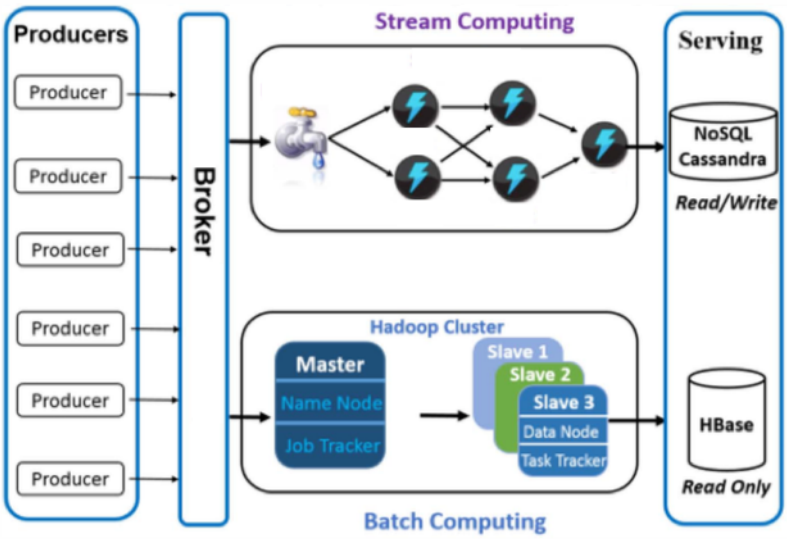

Figure 4.1: *Potential architecture to perform batch and streaming computing simultaneously [23].*

have a continuous stream of data and the crucial idea is to process it strictly in real-time. Obviously, in this situation the processing cannot use every time the entire set of data because it could not exist; instead there is a continuous and infinite data flow of events that happens as the system runs. An acceptable latency in batch computation is in terms of minutes while for stream computation we distinguish between near-real-time and real-time computation: to achieve the latter, the latency must be around sub-seconds. Essentially, in a stream architecture the processing of the current data flow has to be completed before the income of the next flow's event; indeed, in stream computing the emphasis is on the velocity of data more than the elaborated record amount. Sometimes the context drives the choice towards batch or stream computing approach: if the final application requires a real-time (or low-latency) taken decision then the use of a stream computing infrastructure it will be mandatory. Some scientists talk about Big Data Stream Computing as

> "*a system which provides real-time computing, high throughput distributed messages and low latency processing with massively parallel processing architecture.*"[23]

Finally in [2] are described the 8 main characteristics that a RT (i.e. real-time) Stream Processing architecture has to provide to handle Big Data streams. Ideally, these kind of systems have to process data on-the-fly as they arrive and especially pull them when they are generated without any polling mechanism. Another important rule concern the flexibility against potentially imperfections and holes in continuous data flows: they do not have to affect a significant information retrieval from the stream Finally, also the fault tolerance of the system is important: if a processor fails, the system must avoid data loss and should be able to move the process towards others machines to continue it.

4.2 Distributed computing technologies

The recent complexity growth of problems led the IT industries to believe there was a need for a new more powerful processing architecture able to take on with incoming hard challenges and with distributed computing. *Divide and Conquer* technique seems to be the best solving approach, in terms of costs, when there are very large problems with different data types, many records and high throughput; On the other hand, using a parallel methodology raises many other issues about where and how to split the problem efficiently.

Actually, there are many other approaches to parallelize computing like clustering, cloud networking and Grid computing. In this thesis, we will focus on the former: it assumes there are many commodity-hardware computers

with a shared task and same or similar hardware and software equipment. Every nodes in the cluster are coupled with the others using high-velocity linkages (e.g. Gigabit Ethernet). They have to solve an assigned subtask in order to complete the main cluster job: at the end the computed partial results are combined to get the final outcome. Grid computing is quite different from cluster one even through the general approach is similar: the "group" is composed of many machines which are loosely connected each other so usually they do not share info. Generally, these workstations are spread in different geographic places although they participate to solve the same task.

Today clustering computing is a mature technology which offers: horizontal scaling to easily add new nodes in the cluster, fault tolerance properties to safe data in case nodes go down and high performance processing using smart load balancing and splitting the job through the nodes. This scenario has to face several obstacles and it needs solutions which can overtake challenges like the system reliability with hundreds of nodes, data security assurance in every circumstances and the execution of complex jobs proficiently. In the remainder of this chapter, the most important technologies in distributed computing are described.

4.2.1 Apache Kafka

Apache Kafka is one of the most important top Apache framework. It is a publish-subscribe distributed messaging system designed to use in Cluster computing context and a high throughput system thanks to its performance, scalability, and fault tolerance property. Kafka originated as a LinkedIn internal project and later released as open source in 2011 under Apache umbrella. Jay Kreps, one of the creator and main developer in LinkedIn, described Kafka as a central hub of data streams. Indeed, Kafka often has a central

role in computing architecture, both stream and offline, because it represents a safe dock where retrieving, sending, storing data. Moreover, it can exchange data with other machines of the platform without worrying about replication issues or connection management with specific other devices which could be relational databases, NoSql databases or perhaps another distributed environment: Kreps names this problem as *data integration*. In Figure 4.2 a potential simplified scenario where Kafka can be involved is illustrated . It should be noted that there are several technologies that operate with different latencies and very different purposes. Like other publish-subscribe messaging systems, Kafka uses topics to provide an access point for data producers and consumers. In particular, Kafka gives the opportunity to establish and maintain simultaneously a large number of topics: as the data arrive, it stores and replicates them in order to provide offline consumption by topic subscribers so it fits perfectly stream requirements where producers could be faster in data generation respect to consumers in data evaluation.

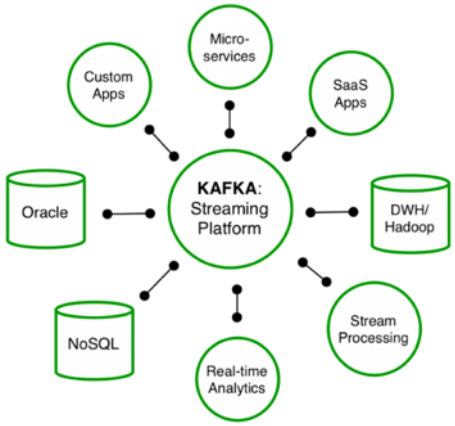

Figure 4.2: *Apache Kafka role in a typical distributed computing system [55].*

Apache Kafka is one of the most important messaging system in high performance computing context mainly for its stability: it presents great

performances even with many TB of stored data and it can bear over 2 million writes each second. One of the main reason to use Kafka respect to its competitors is the large support, documentation and full compatibility with the main distributed computing platforms like Apache Hadoop, Spark, Storm and Flink.

4.2.2 Apache Hadoop

Hadoop is an open source Apache project designed to execute batch jobs in distributed way on large clusters of commodity hardware. The framework is one of the most important and it is historically used from celeb IT companies like Facebook, Google, Yahoo: it offers a scalable and fault tolerant system for Big Data applications. The main Hadoop components are *MapReduce*, which represents a processing paradigm, Apache YARN that provides jobs scheduling and HDFS, a distributed file system. Hadoop is designed to run on commodity hardware and it follows the MapReduce processing model with a master-slave architecture where the master node is named *NameNode*. As data blocks are submitted, Hadoop splits them in smaller pieces and then spread them to slaves. Usually data chunks are replicated in many nodes so Hadoop can guarantee the integrity of data also if a node fails. MapReduce is a programming way designed to face efficiently Big Data on many machines through two steps called *Map* and *Reduce*: in the former phase data are processed while in *Reduce* step a result aggregation is performed. Usually, each slave runs a different MapReduce instance. This programming model has many implementations in different languages, open source or not, performed by Google, Yahoo and Apache. HDFS is the default Hadoop file system but it is used also with other frameworks. Its goal is providing high aggregate throughput moving and spreading the computation where data are located;

other responsibilities are node failure recovery and data replication. HDFS is deepened in one of the following sections.

4.2.3 Apache Spark

Apache Spark is an open source framework designed for cluster computing of Big Data. Its develop started in 2014 at University of Berkeley in order to create a paradigm more powerful and faster than MapReduce: actually Spark paradigm seems to be hundred times faster in some applications [12]. Spark is designed to cover a wide range of workloads such as batch and streaming applications, iterative algorithms and interactive queries. Spark provides an application programming interface based on a data structure called *Resilient Distributed Dataset*, a set of data items distributed over a cluster of commodity hardware. It offers a form of distributed shared memory which facilitates the execution of iterative algorithms and repeated data querying: this allows to reach a significant improvement in latency respect to Hadoop implementations. It should be noted that Spark requires either a cluster manager like Hadoop YARN, and a storage system, for example HDFS or a NoSQL database.

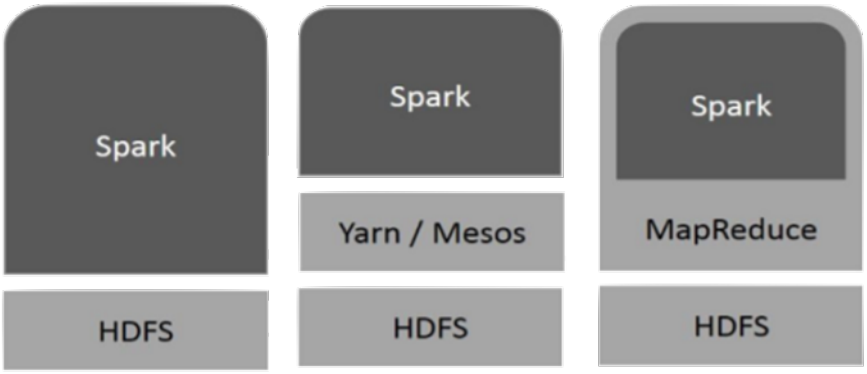

Figure 4.3: *Three ways to implement Apache Spark framework. Starting from the left: Standalone mode on HDFS infrastructure, on top of a YARN manager, on top of a pre-existing distribution based on Map Reduce system [12].*

At foundation of Spark there is a component named Spark *Core* which provides job-dispatching and all cluster management functions through an application programming interface centered on RDD. In order to perform streaming analytics Spark offers *Spark Streaming*, an extension built on top of Spark Core; despite there are other stream processor employable on top of Spark, the Streaming extension is fully integrated in Spark universe and it supports important consumers and brokers like Kafka, Flume and TCP/IP sockets. Actually, *Spark Streaming* does not provide a real-time stream analysis, instead it splits data in very small pieces and performs batch elaboration on them: this technique is called micro-batching and allows, without changes in Spark paradigm, to obtain near-real-time performance.

4.2.4 Apache Storm

The entire IT community agrees to consider Apache Storm as the stream version of Apache Hadoop. Actually, Storm achieves the same objectives of the latter in a distinct scenario and with a different infrastructure topology,

i.e with a different computational model.

Apache Storm is an open source real-time distributed system designed in 2011 to handle Big Data streams and today it is still used by the greatest IT companies in the world, like Twitter and Facebook. It is very appreciated also for the flexibility provided by the many available programming languages. Essentially, Storm provides a set of primitives for real-time computing and exposes an architecture (Figure 4.4) based on two type of nodes: *Bolts* and *Spouts*. As we know, in a streaming scenario there is a continuous unbounded data flow: in Storm these are caught by Spout nodes. As the data arrive they inject continuously in the system key-value pairs called tuples so we can define Spouts as the sources of the framework.

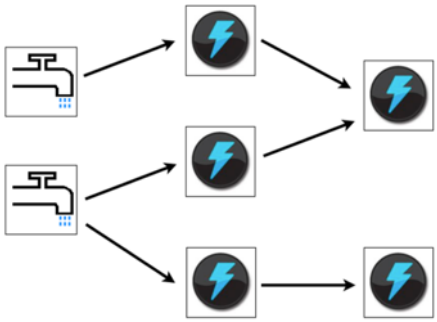

Figure 4.4: *Architecture of Apache Storm. Spouts are the sources while Bolts are processing nodes. [60]*

Data are fetched with a pull strategy so the nodes ask for new data and start to process them autonomously. This approach limits to one the number of potential points of failure because data can be lost due to a plenty buffer only on first stage [1]. Bolt nodes are processing elements which receive raw tuples from Spouts or pre-processed data from other Bolts. The Storm's architecture describes a stage-like topology of Bolts nodes designated to execute a list of tasks which can comprises aggregations, filtering, joins or custom operations. Like Hadoop, Storm uses a master-slave paradigm: slaves

are also called workers and on top of them there are logical supervisors which periodically exchange messages with the master. It is interesting to note that also if the master node goes down, all workers can continue computation and produce outputs; same observation clearly is still valid if a worker fails.

4.2.5 Apache Flink

Apache Flink is an open source platform for distributed streaming and batch processing deployed in 2010 in Berlin and initially named *Stratosphere Project*. Later, in 2014 its name changed in Flink for commercial reason and also because the term "Flink" - i.e. speed/agile in German - better fits the main property of the framework, specialized in effective real-time stream computing. Today, Flink is an open source Apache top project and it is a strong competitor of Apache Storm and Apache Spark while it is largely used by some of the most important world companies like Zalando, ResearchGate and Alibaba Group. The reasons behind the Flink success are the high real-time throughput and the very low latency during processing; they make Flink one of the fastest stream processor on the market. Moreover, Flink provides also a batch processing engine to work on static data.

As reported by the official page, Flink is stateful and fault tolerant with zero data loss; it can recover from failures while maintaining exactly once application state; furthermore it performs large scale processing with thousands of nodes without affecting throughput and latency. Another great Flink feature is the support to event time semantic which allows to compute accurate results over streams also when messages arrive out-of-order. Moreover, Flink supports flexible windowing on time, count and sessions and even data-driven one: in this way it can model the reality of the data environment. Like other distributed systems, Apache Flink can be executed on clusters or on a single

machine, in standalone way or on top of YARN or MESOS. In the Flink Forward Conference in 2015 a research group owned by Bouygues [4] noted that the framework can handle 500,000 events per second with an end-to-end latency less than 200 milliseconds: this result was achieved on a small cluster of 10 nodes with 1GB of memory each.

4.2.6 Comparison

Finding an absolute winner in Big Data processing platform panorama is impossible. Each one has weakness and strengths respect to the others and a choice between them can be taken only considering the particular application to develop. Actually, we can distinguish Hadoop from Spark, Storm and Flink because the Hadoop goal is to perform batch processing without any type of streaming analytics: if the application do not present data stream to analyze, Hadoop maybe is the right choice for its large compatibility with other software. It has the longest history in IT panorama so it has also the biggest community and a great stability given by its larger use in the world.

Spark, Storm and Flink can be all together used to perform stream computing but they fit different use cases. Storm has some predefined structures for Bolts and Spouts so if the requirements match perfectly with them opting for Storm is a good idea to simplify the development. However, others aspects have to be considered to choose the right framework, for instance the latency: Storm and Flink operate in order of sub-seconds, lesser than Spark because this one does not perform pure stream processing.

Generally, an evaluation between Storm and Spark is difficult because they provide solutions to slightly different problems: Spark is a general purpose framework for distributed computing and it has both batch and stream capabilities while Storm is exactly a pure stream computing system. In the

same stream context Storm and Spark offer solutions in very distinctive ways and with different performances. For instance, Spark (with Spark Streaming extension) does not treat data as a continuous unbounded flow, like Storm, but it adopts a micro-batching technique to simulate it. On the other hand, if a potential data loss is not acceptable probably Spark is a better choice than Storm because the former presents more delivery guarantees.

With only streaming requirements we can evaluate a comparison between Flink and Storm. Indeed, they are both pure stream processors and they both have similar pipelined engines to handle stream: anyway, accordingly to [6] Flink presents better throughput performance. Actually, in [31] a comparison exposes similar performances for Storm and Flink: the latter achieves a better throughput while Storm reaches lesser latencies Figure 4.5. Really, Storm presents very low latency with a - sometimes unacceptable - trade off on assured level of correctness: it does not provide exactly-once guarantee and even those which are provided came at a high overhead [4]. On the other hand, Flink provides exactly-once guarantee. Actually, using *Trident* extension Storm can provide exactly-once guarantee: unfortunately, in this case it becomes a micro-batching processor like Spark, affecting its performances. Finally, Flink has higher-level API compared to Storm, so application's development with the latter could be more complicated because all functionalities need to be manually implemented.

Due to the dual analytic possibilities offered by Flink - batch and stream processing - a comparison with Spark it is very interesting. Actually, Flink and Spark are similar just in use cases but they are very different in internals. Indeed, for streaming processing Spark uses micro-batching while Flink offers pure streaming analysis. It should be noted that Spark needs a file system under itself so if the system is based on a pre-existent Hadoop instance,

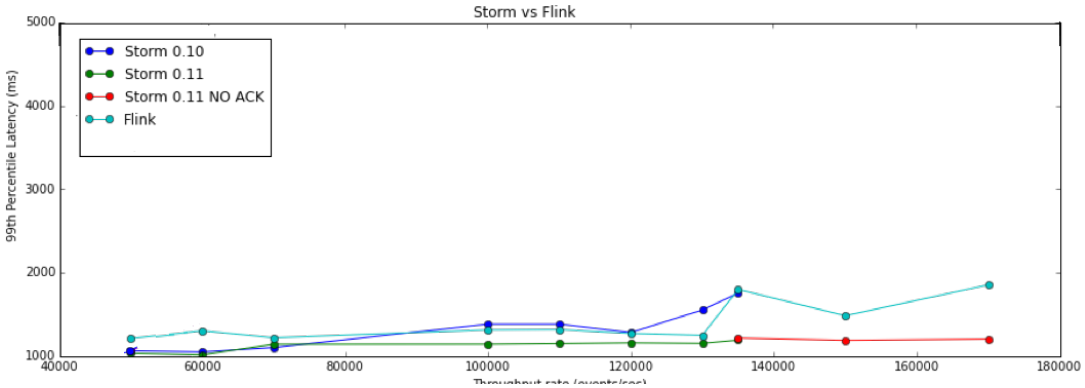

Figure 4.5: *Performance comparison between Flink and Storm. The former shows an higher throughput while Storm reaches better latency. [31]*

opting for Spark on top of HDFS is a better choice than building a Flink instance from scratch.

The choice between these frameworks depends on particular application requirements and even on context: if we want to extend a Spark instance with a stream processing infrastructure there are no reason to choose Flink. However, if a new application requires fast and real-time streaming analysis and batch computing maybe Flink is the best choice.

	HADOOP	SPARK	STORM	FLINK
Open Source	✓	✓	✓	✓
Batch processing	✓	✓		✓
Stream processing		Micro batching	✓	✓
Exactly once guarantes			Trident	✓
Latency	High	Medium	Very low	Low
Throughput	High	High	Low	Medium
Fault tolerant	✓	✓	✓	✓
Kafka supporting	✓	✓	✓	✓
First release	2011	2014	2011	2015
Community	Large	Large	Medium	Medium

Table 4.1: *Brief comparison of main distributed computing frameworks*

4.3 Storage layer

The storage layer has a primarily importance in a system designed for Big Data. It has to provide fast and safe access to data from several agents simultaneously. Unfortunately, the standard RDBMS are not able to face with Big Data properties of scalability, high performances requirements and very high volume of data. In fact, Big Data applications has to deal with petabytes and with large variety of data; moreover, RDBMS usually cannot scale because they have limited capabilities in adding new nodes and in redistributing the load automatically when data amount grows or needs a better management. Furthermore, having good performances with a single node which has to manage dozen of simultaneous reads and writes is unrealistic, mostly because splitting a relational structure in smaller pieces and spreading them in different nodes is a demanding affair.

On the other hand, in recent years another category of databases started

to spread: NoSQL databases, indeed, fit perfectly this scenario and they answer to Big Data challenges with great performances in high volume data management and scalability. Therefore, in this context several NoSQL databases were proposed and many of them have strong integration with the distributed computing technologies presented in above sections. In the remainder, some of the most important storage solutions in Big Data context are presented.

4.3.1 Apache Cassandra

Apache Cassandra is a top level open source project designed in 2008 to provide a high performance persistence layer without using a standard relational database. Originally Cassandra was developed by Facebook to solve some problems about an old weird use of MySQL, successively it was published on Google Code and therefore included in Apache Incubator program; today Cassandra is used in many Big Data frameworks thanks to its great performance. It belongs to column-oriented database's family hence it handles objects made of three values: a key column reference, a timestamp assigned to the value and the value itself. Cassandra is a distributed NoSQL database management system useful to work with a huge amount of data and offering fault tolerance and scalability. It is usually used in clusters of commodity hardware where data are spread to optimize performances. Every node of the cluster is identical to others so there are no master units and data are replicated many times on different nodes to guarantee consistency and to improve read velocity. A Cassandra database is very tunable: there are many parameters to configure the desired grade of consistency but everyone of these obviously affects general system's performances so the administrator has to found an adequate trade off between velocity and reliability.

Cassandra follows the model of Google's *BigTable* so a table could be

considered essentially as a multi-dimensional map indexed with keys and populated by highly structured values of unlimited length. Like *BigTable*, columns can be grouped in families similar to the tables of RDBMS model (indeed in CQL3 families are called tables): anyway they are more flexible and dynamic than the latter because they has not to be declared at schema definition time. Finally, Cassandra has linear scale performance so the response time increase linearly adding new data; furthermore, it supports many programming languages and has the own query language: CQL.

4.3.2 Apache HBase

HBase was developed by Google in 2006 and then it was absorbed in Apache Incubator in order to extend persistence layer for Hadoop. Indeed, HBase has a strong compatibility with the cited processor and runs on top of HDFS. Now it is largely adopted in IT industry by companies like Netflix, Adobe and Google itself because it provides a great fault tolerant way to manage Big Data. It derives from Google's *BigTable* and it is a member of the large family of column oriented database: the documentation describes it as an

"*Open source BigTable implementation*" [10]

It can be used in distribute or standalone ways so it is very flexible. Note that in standalone mode it can be used also without HDFS; instead, in distributed and pseudo-distributed architectures HDFS is required since there are more machines to handle. Unlike Cassandra, Apache HBase follows a master-slave paradigm where the master guarantees the consistence of the cluster, balances the load and handles node failures while slaves manage I/O requests in the cluster and towards the distributed file system.

In HBase data are modeled in tables, i.e. maps of rows, where each row represents a key-value pair: similar keys are stored close to each other in order to improve performances. A column is another model entity: it represents a type of row's key; actually a value for a column key can be viewed as a key-value entry where the key is a timestamp. Like Cassandra, columns can be organized in families, but in HBase they have to be declared at creation time.

4.3.3 HDFS

The use of a NoSQL database is not mandatory to deal with Big Data and HDFS is another efficient way to store data with fault tolerance and consistency guarantees. HDFS is the acronym of *Hadoop Distributed File System* so it offers a full integration with Hadoop infrastructure and can be used also as foundation for other solutions (NoSQL for example). Main characteristics of HDFS are the reliability, the failover recovery mechanism, distributed data replication, extreme scalability, low cost infrastructure and portability. Moreover, it moves the computation where data are placed so reaching high performances.

The HDFS architecture considers a single main node, called NameNode, containing file's metadata and many DataNodes where data are stored and replicated in fixed dimension blocks. Data are organized into files and directories and they are retrieved from clients directly in DataNodes without consider NameNodes.

4.3.4 Comparison

The choice of a system is not absolute but it is strictly dependent from the application we want to develop. The most important thing to consider is the

compatibility between all parts of the system: if Hadoop is utilized as batch processor maybe the most natural choice at storage layer can be HBase or HDFS because they are fully integrated in its ecosystem. In a comparison between them we have to consider the performance of both systems: HBase for example (like others NoSQL databases) allows random reads over data while HDFS provides only sequential reads and, in a random context, the latter requires a complexity of *O(n)*, worse than HBase.

Both Cassandra and HBase are NoSQL databases and they share many characteristics: basically, both databases cannot be manipulated with SQL instruments, however it is possible to make a comparison relying on performances and on various facilities offered to developers. Apache Cassandra implements CQL, a query language very similar to SQL, which could be very helpful for developers are migrating from RDBMS. Moreover, Cassandra has a larger documentation than HBase. Both are distributed databases and both adopt column oriented paradigm, furthermore they share access data methodologies. On the other hand Cassandra, accordingly to [8], allows a better consistency tuning respect to HBase so it offers the greatest control for developers. About performances, as reported in [29] and in Figure 4.6, Cassandra beats HBase for number of operations executed per second in load process context. This effect is highlighted in a scenario with a balanced number of writes and reads: Cassandra overtakes HBase hundreds times [28]. Moreover, it should be noted that the resulted gap remains also when the number of nodes involved in the test grows from 1 to 32. Finally, in [1] HBase is indicated as a database management system optimized for read operations, while Cassandra is better in a context with many writes.

NoSQL databases are designed to deal with the need of fast data accesses: indeed, they provide a parallel way to execute reads and writes so they are

more indicated than HDFS for Big Data stream and real-time processing, which is usually used in batch - then slower - computing. To better clarify the comparison of NoSQL databases, Table 4.2 is provided.

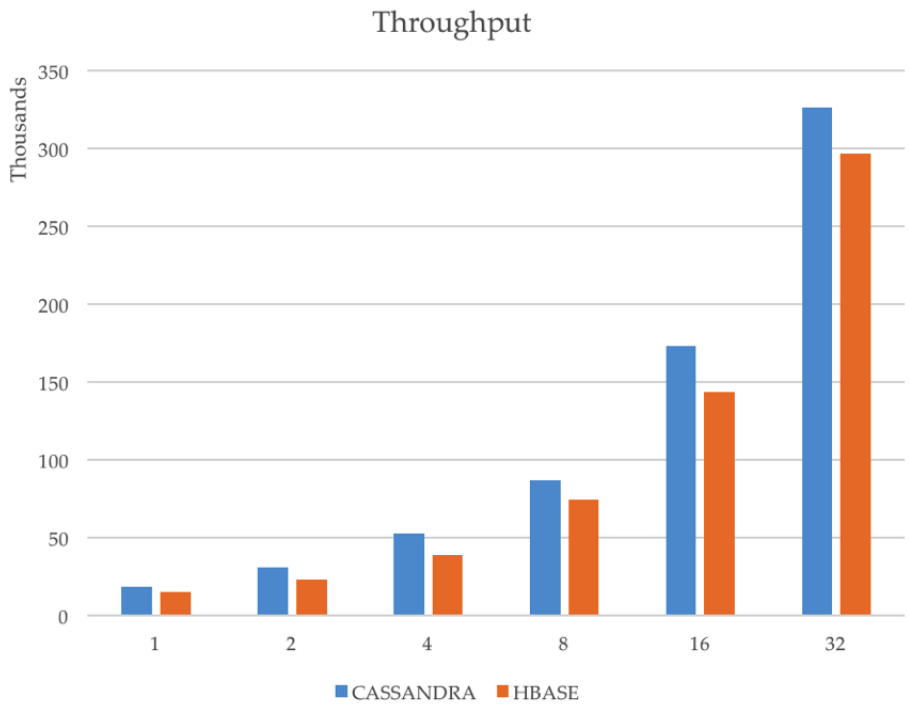

Figure 4.6: *Comparison between Cassandra and HBase about operations-per second when cluster's node number varying. Data from [29]*

	CASSANDRA	HBASE
Open Source	✓	✓
Hadoop Supporting	✓	✓
No single points-of-failure	✓	
Supported languages	13	8
Optimized operation	Write	Read
Typing	3	
Api	Proprietary	Java
Concurrency	✓	✓
Durability	✓	✓
First release	2008	2008

Table 4.2: *General comparison between Cassandra and HBase*

4.4 Real-time anomaly detection

Some of the most interesting information that may emerge from real-time monitoring are the detection of anomaly patterns of data, abnormal values and sensor faults. Especially in healthcare, discovery of anomalous vital signs has an extreme importance to prevent sudden disease and to assure an immediate medical intervention in order to solve issues as soon as possible with advantages in term of costs for hospitals and benefit for the patient's health. Moreover, it has also a great importance to know how to distinguish between real anomalous value and false alarms, in order to avoid useless anxieties in patients and to reduce load on analyzing systems.

An anomaly is defined as a point in time where the behaviour of the system is unusual and significantly different from the past [22]. This definition implies a problem despite an anomaly can be considered in general like an unusual value in a continuous data flow. we can distinguish between two

kinds of anomalies: they can be spatial when a value overtakes a threshold or temporal when a value correctly fit in a threshold but it occurs in an unusual sequence. In following sections are briefly described some techniques used in IT industry to perform anomalies detection. More details about them can be found at [7].

4.4.1 Numenta HTM

Hierarchical Temporal Memory is a foundational technology for the future of machine intelligence based upon the biology of the neocortex. The project borns in 2004 and it is still fully supported by Numenta's community while all HTM related project are committed as open source. Info about Numenta and HTM's theory can be found at [18] while details and use cases of HTM algorithms are described in [22].

HTM can be used to achieve multiple prediction goals with online and unsupervised learning properties, providing also a high order representation of data and supporting multiple simultaneous prediction. NuPIC and HTM.java are some of the most relevant implementation of HTM theory but the community is still at work to improve them and to develop other applications.

In this section it is briefly described how HTM theory can be used to perform powerful anomaly detection. A peculiarity of the HTM algorithm is that it continuously learn and model the input. Note that HTM does not evaluate directly if there is an anomaly in data flow, but starting from HTM output it is possible to decide if the interested value is a anomaly or not. In the picture below (Figure 4.7) is fully represented the role of HTM in a typical anomaly detection algorithm:

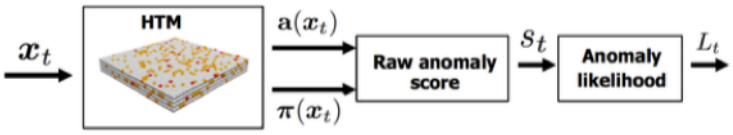

Figure 4.7: *The role of HTM in a real-time anomaly detection algorithm [22].*

Considering *Xt* the current input of the system, HTM will compute two values: *a(Xt)* and π*(Xt)*. The former is a sparse binary code representation of the current value while the latter is a vector which represents a prediction of the "a function" for the future input. Using *a(Xt)* and π*(Xt)* the algorithm evaluates a first raw anomaly score with the following equation:

$$s_t = 1 - \frac{\pi(x_{t-1}) \cdot a(x_t)}{|a(x_t)|}$$

St represents a 0 to 1 constrained value which conveys how much the current input is predicted, in particular 0 means fully predicted and 1 is unpredicted. Raw anomaly scores and involved functions are computed every time a new value arrives as input of the system. In order to detect anomalies another step is required: a raw anomaly score is just a predictive parameter which does not represent a reliable way to describe anomalies. Sometimes having a spike or out-of-bound values in data flow is absolutely normal so to obtain a useful information we have to apply a threshold method to the raw anomaly score. Therefore, a real anomaly likelihood can be evaluated considering a window of the last n-calculated raw score and computing a normal distribution with the following average and variance:

$$\mu_t = \frac{\sum_{i=0}^{i=W-1} s_{t-1}}{k}$$

$$\sigma_t^2 = \frac{\sum_{i=0}^{i=W-1}(s_{t-i} - \mu_t)^2}{k-1}$$

A threshold is applied to the Gaussian tail probability in order to decide if it is necessary to raise or not an alarm. So, the final anomaly likelihood is defined as the complement of the tail probability Lt:

$$L_t = 1 - Q(\frac{\widetilde{\mu}_t - \mu_t}{\sigma_t})$$

It is interesting that in a noisy scenario variance will be large and a spike in values flow has no great impact on anomaly likelihood score: accordingly to the noisy nature of the case, instead a series of abnormal value influences Lt score and highlights an anomaly in the observed system's behaviour. Finally anomalies can be detected thresholding the Lt score, triggering a particular event or alarm depending on the application.

The power of HTM algorithm lies also on the opportunity of spreading these concepts over multiple source of data streams: many industrial applications present a large number of sensors and continuous data flows to analyze simultaneously. In Figure 4.8 an extension of HTM anomaly detection algorithm with different sources is illustrated :

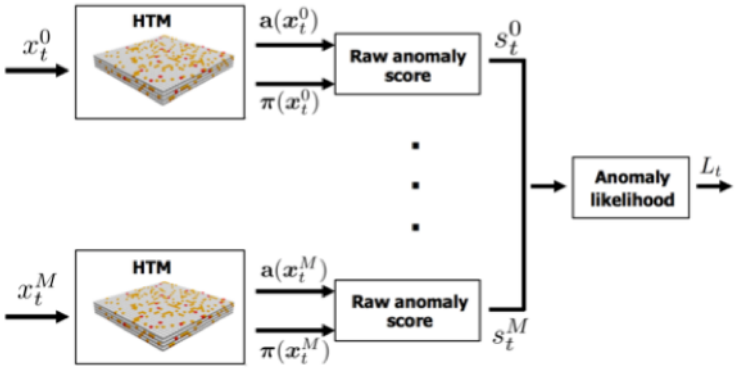

Figure 4.8: *The previously presented algorithm can be adapted in a multiple stream scenario [22].*

A solution for this problem is to consider joint probability of raw anomaly scores and apply a threshold to the tail probability. Nevertheless, this computation can be very difficult in a streaming context so, without big loss of generality, it could be assumed sources' models as independent each other and so simplifying the computation considering to following estimation:

$$P(s_t^0, ..., s_t^{M-1}) = \prod_{i=0}^{i=M-1} P(s_t^i)$$

Then anomaly likelihood score can be represented as:

$$1 - \prod_{i=0}^{i=M-1} Q(\frac{\widetilde{\mu_t}^i - \mu_t^i}{\sigma_t^i})$$

In [7] are also described some practical considerations and experimentation results on NAB real-world benchmark which shows as HTM owns other famous anomaly detector.

ALGORITHM	NAB SCORE
Perfect	100
HTM	65.3
Twitter ADVec	47.1
ETSY Skyline	35.7
Bayes Change Pt.	17.7
Sliding threshold	15.0
Random	11.0

Table 4.3: *Performance comparison for most famous real-time anomaly detection. The "perfect" line represents the ideal detector [22].*

4.5 Semantic web

In this subsection we describe briefly semantic web basis in order to provide a panoramic vision of the context in which this project is developed. Recently, semantic technologies have had an important role in the growth of Internet of Things and Big Data. Tim Berners Lee in a famous speech in 2001 defined the semantic web as

> "*a web of data that can be processed directly and indirectly by machines*"

while W3C presents it as follows:

> "*The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries.*"

Effectively, having a huge amount of raw data is quite useless, or more correctly, data are not enough exploited if they do not have any significance.

Semantic Web technologies get raw data and enrich them performing reasoning and producing new knowledge potentially using pre-existing information. Generally, traditional web has some problems due to a lack of semantic: the contents have no structure or they are structured to allow only human comprehension, so they do not allow an effective automation of information extraction from machines. Moreover, it is important to note that semantic web will not to be a replacement for traditional web but an extension and lots of efforts are employed to assure full compatibility with standards and existing contents of the current web. Probably we can imagine semantic web like a new powerful representation of the web.

With *Linked Data* term we describe data which are using semantic web technologies, published on the web and linked with each others using various relationships and properties: in this way new knowledge can be inferred efficaciously, rapidly and autonomously. Today semantic web technologies are largely used in static contexts or quasi-persistent data while reasoning over transient data is still an emerging field of study due to its complexity. Generally, monitoring a scenario in real-time presents some requirements about time, knowledge, and locality. Semantic data analysis can occur in short or long term that depends on particular application: for instance, in case of detection of sudden changes or abnormal events a real-time analysis is required, otherwise other type of analysis may be performed. Same reasoning can be done about locality requirements: sometimes it is essential understand if we have to correlate different information each others or if a local monitoring is sufficient.

Occasionally, the context is very complex and in order to do a good analysis it is essential to have a full comprehension of the scenario: on the other hand, this type of reasoning can require a lot of time to consider all

parameters and clearly this collides with real-time computing requirements. Finally, both real-time monitoring and long term analysis could be performed simultaneously.

4.5.1 Resource Description Framework

Resource Description Framework is a standard used to define web resources, their properties and relationships with other web entities. It is designed to represent knowledge in a distributed way. It is important to note that RDF is directed to characterize meaning and knowledge and sometimes it can refer to abstract concepts or to non-physical characteristics. Actually, an RDF statement is based on a triple formed of three elements: *Subject*, *Predicate* and *Object*. The former is exactly the subject of an assertion and it can be a person, a thing, a topic but even an interest or any abstract concepts. Predicate represents a relationship between the Subject and the Object while the Object is the concept modified by the predicate associated.

Each resource has an *Internationalized Resource Identifier* (IRI) which can be a URL, frequently, or another unique identifier: with this one every people and every machines can identify uniquely a particular resource in the world, eventually constructing and extending its definition and relationships or potentially adding other resource names. Also a property is identified with a IRI but truly it is a special kind of resources describing relations between other resources, for instance the age of a person.

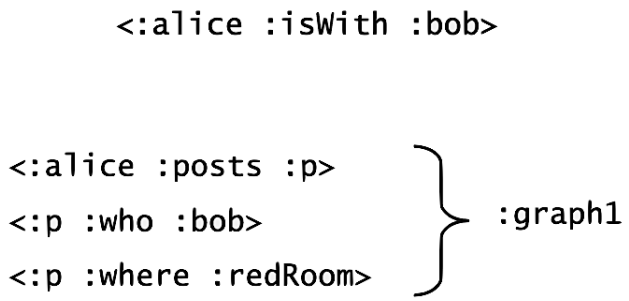

Figure 4.9: *On top is represented a typical RDF triple. Below there is an RDF semantic graph.*

Note that an RDF document can be viewed as a set of triples or like a semantic graph (Figure 4.9). However, int the RDF convention described here time is not accounted; effectively until few years ago semantic enrichment was performed thinking only to static data.

Reasoning over static data is a challenge solved with a lot of frameworks and middleware but achieving the same task in real-time is a very different question. This topic is focused in next section.

4.5.2 Resource Description Framework Stream

Today the IT world can be represented also an unbounded flow of time-varying data originated from billions of sensors geographically spread in the planet: smartphones, social media, urban and medical sensors are just few of them. Consequently, reasoning over static data is not enough anymore: there are data that have a very short lifecycle which have to be analyzed in real-time to provide useful knowledge. In [15] is considered the problem to suit RDF to these new forms of data.

RDF standard is not adequate anymore because it does not consider time in its triples, so it cannot be used to represent transient data. To overtake

these challenges RDFStreams was designed from RDF to maintain full compatibility with standard. RDFStreams represents an extension of RDF where time is accounted and which allows reasoning over streams instead of persistent data. We talk about continuous semantic and a completely different query paradigm: indeed, to execute an automatic extraction of knowledge queries have to be registered. In RDFStreams time is accounted using a timestamp: usually it is used the data generation time but there are other solutions as the use of two timestamps to highlights that exists a time interval in which data are legal. Note that an RDFStream without timestamp is just an ordered sequence of data items and it is not very useful because it does not provide any information about what happened earlier than another event.

There are two different paradigm to process RDFStreams: DSMS and CEP. The former is based on DBMS concept so it supports common SQL operators like join and aggregation while CEP (*Complex Event Processor*) considers data like discrete events and effectively sees data as a flow of occurred events, like in real world. DSMS and CEP follow two different paradigms: the former produces query results which are continuously updated to adapt to the changes of input data while CEP offers detection and notification of complex data patterns involving sequences and ordered relationships [27]. It should be noted that currently main stream processing systems support features of both approaches.

Generally, *Stream Reasoning* term refers to a computation performed in real-time, with multiple, gigantic and heterogeneous data streams. In order to deal with these streams a lot of RDF stream processor were developed in recent years: *C-SPARQL*, *SPARQLStream*, *CQELS* and *Streaming Linked Data*.

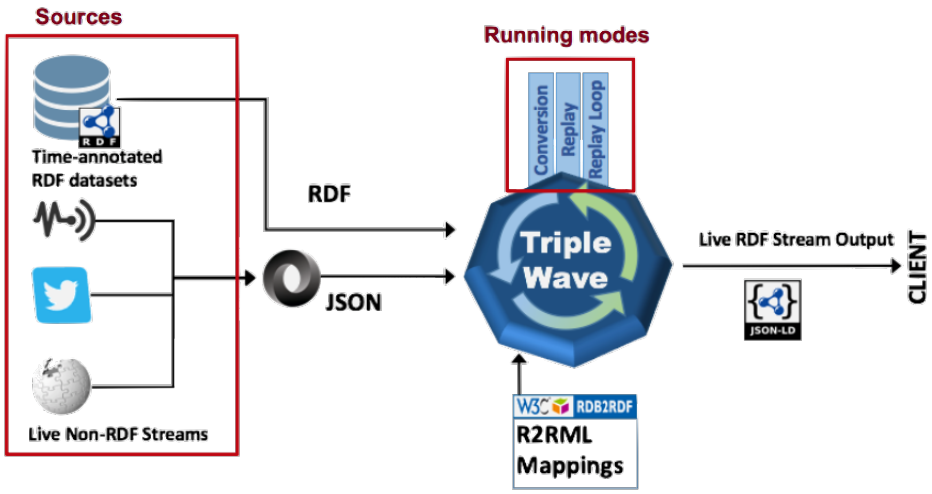

Figure 4.10: *TripleWave framework [15].*

4.5.3 TripleWave

TripleWave is a framework presented at ISWC2016 and described in [15]. It is released as an open source project and it is designed to create and publish RDFStream over the web. Due to lack of technologies which dealing with annotated data strefams currently there are no standards to handle RDFStreams then *TripleWave* represents an answer to this need.

TripleWave provides many operative modes: It can get data from different sources: from existing RDF dataset in order to refactor standard RDF data in RDFStreams or from live raw streams like social network flows in order to produce a live RDFStream output. *TripleWave* is written in Javascript and its architecture provides three running modes: conversion, replay and replay loop. The former is the standard way to convert live raw stream in annotated RDFStreams while the others are used to refactor existing time-annotated data. To express a custom mapping from RDBMS data *TripleWave* uses a R2RML language to map data field in RDF triples.

Chapter 5

Architecture

This chapter introduces the architecture of the designed system and their components. However, first of all in section 5.1 the faced problem is recalled and better described in term of functional and non-functional requirements. Thus, in section 5.2 the experimental infrastructure thought to realize the architecture will be illustrated and then a deeper attention will be granted to analyze every single components. In each section, the architectural components are firstly described at a conceptual level and then more technical aspects are examined with an overview, when it is possible, on used algorithms and chosen technical methodology approached. The *Overview* section represents a summary of the system as a whole and describes briefly the responsibility of all high-level component. Next, in *Sensing* section are described the input data and how these are pushed in the subsequent *Data Preprocessing* part; this introduces a Raspberry Pi 3 which collect input data and preprocess them before to submit the streams to a commodity hardware cluster with running instances of Apache Kafka and Apache Flink. Hence, a distributed solution to memorize processed data is described while this chapter is followed by another one where some real test outcomes are analyzed.

5.1 Problem definition and requirements

This thesis aims to investigate and face the issues related to a Big Data dealing system. Actually, the particular healthcare context does not present important additional difficulties to the problem because it fits perfectly the Big Data scenario. As already listed in chapter 2, Big Data are characterized by four properties: Volume, Velocity, Variety and Veracity. The first one concerns the relevant data amount generated by sensors and others device while Velocity and Variety affect respectively how quickly data must be treated and how their large variety type must be handled. The Veracity instead raises the most ambiguous problem: how to evaluate the credibility of data.

All these properties represent hard challenges to deal with. In particular, a single machine would find many difficulties to handle a big amount of messages even sent dozen times each second. Performing analysis directly on sensors and wearables could be impossible: to be low cost, these devices have just the strictly required capability to connect to a network and send messages. Actually the issues are not related to how much fast a single machine runs: we can imagine to have a very powerful hardware but still it could not be the best solution to handle Big Data. Handling a single powerful machine has an advantage related to the logical simplicity of a well-tested paradigm; despite it, the most important IT companies do not use single servers to face with Big Data. Indeed, using a single server presents also many disadvantages like high costs and the possibility that the machine goes down for a period leading to a dramatic loss of data: in this case we would have hole data flows which would be useless to perform knowledge extraction. Moreover, a single powerful server is a special machine which requires specific hardware, software and specialized staff to deal with it; other requirements

are a considerable space to host such a voluminous computing node and air conditioning to avoid overheating: all these factors contribute to raise the cost of a solution based on a single server so much that the expenses could overcome the advantages coming from the Big Data analysis.

Instead, employing a cluster of commodity hardware represents a better solution which can deal more easily with the previously presented V's. The difficulties of handling many different messages incoming with high frequency can be shared among dozens, hundreds or thousands of nodes. In a cluster, each node has a hardware configuration which is relatively low: that will be enough to bear the assigned task but at the same time it will be not so expensive in case of node replacement. Moreover, a node fail does not lead to a job failure: the task assigned to the node will be moved to an unoccupied node or put in an idle state until the interested node is recovered or replaced.

Since that, this system need an architecture based on a cluster of commodity hardware.

In the remainder of this section functional and non-functional requirements of the application will be listed. Generally, the problem requirements specify the system behaviour from an external point of view, without going into details with hardware and software employed. Functional requirements describe the services and functionalities offered by a system or the effects of some operations while non-functional requirements depict system constraints and its properties: sometimes constraints on development process can be specified too. The non-functional requirements are more critical than the others: often, if they cannot be complied the system cannot be realized. It should be noted that some requirements could be both functional and non-functional.

5.1.1 Functional requirements

The application to develop within this thesis is characterized by a number of functional requirements. The scenario is composed of many individuals which wear sensors operating at 50 Hz: in particular the involved persons wear 8 sensors each one so as minimal requirement the system has to be able to retrieve messages sent from sensors. The number of people involved in the system could be quite big and it could increase over time. The medium dimension of the messages ingested by sensors is around 113 bytes, then the total throughput generated by a single person is around 45 KB/s. During the processing the messages could be enlarged in order to add more information and their size can raise until 1 KB around. Data coming from sensors and enriched by the system must be safely memorized in a persistence storage to allow further elaborations. Moreover, the core of the thesis is the search for anomalous values within the data stream originated by sensors: these anomalies have to be detected, annotated and memorized in a persistent storage. Finally, all the data memorized may be queried by third-party systems also during the processing.

Given that and in order to sum up them, the functional requirements are listed:

- *The system must be able to retrieve data streams from sensors.*
- *The system has to bear a big number of sensors which asynchronously send data concurrently.*
- *The input throughput the system has to handle is amounting to some MB/s.*
- *The throughput the system has to handle is amounting to many MB/s.*

- *The system has to have the ability to edit coming data in order to add more information.*
- *In order to assure availability the system has to replicate data through the cluster.*
- *The system must memorize in a persistent storage all data coming from sensors.*
- *The system must have the ability to perform anomaly detection on data streams.*
- *The system must memorize in a persistent storage all detected anomalies.*
- *The system has to make memorized data available for external systems also during the processing.*

5.1.2 Non-Functional requirements

Some of the non-functional requirements described here are typical of any Big Data dealing system. However, some of them are not mandatory in general but they are very important for the purpose of this thesis. First of all, a fundamental property is the resiliency: data cannot be lost for a network lack or if a node fails; moreover if the latter case occurs, it must not affect the other nodes and the job has to continue regularly. Another fundamental property to achieve is the real-time processing for each coming stream. A delay within one second could be accepted. The data must be always available for consuming by third-party system. The system must be composed only of open source frameworks and has to expose properties of

simplicity and reusability. A very important requirement concerns the integration with semantic web technologies in order to take part to the forefront of Big Data technologies. Potential future improvements of the system must not revolutionize its original architecture.

Given that and in order to sum up them, the non-functional requirements are listed:

- *The system must be able to buffer and temporarily store data coming from sensors in order to avoid data loss if the computation is delayed or in case of network lack.*
- *The system must be able to continue regularly the job also if one or more nodes fail.*
- *The system has to process coming data streams at least in Near-Real-Time.*
- *The system has to exposes data to external systems at any time so it must be always available.*
- *The system must employ only open source frameworks.*
- *The system must be easily expandable, modular and has to allow integration with other systems.*
- *The system has to allow future improvements and framework replacement without revolutionize itself.*
- *The system has to integrate semantic web technologies.*

5.2 Overview

The proposed architecture supports a system which can be used to perform real-time anomaly detection on a stream of data originated from a bunch of wearable sensors. The main purpose of the system is to provide a useful instrument to early detect abnormalities and irregularities in data patterns coming from individuals which wear sensors in different parts of their bodies. In the previously described scenario, performing real-time detection is essential because it provides the ability to react immediately to critical events capturing symptoms or signs which are invisible to humans due to its high-frequency sampling. Moreover, it allows to discover and get details about every single abnormal samples: on the other hand, sensors are fallible hardware objects so they can fail reading a value due to signal noise or for damaged devices; consequently it is also essential handle potential false-alarm events. Hence, in order to implement this kind of analysis the following architecture was designed:

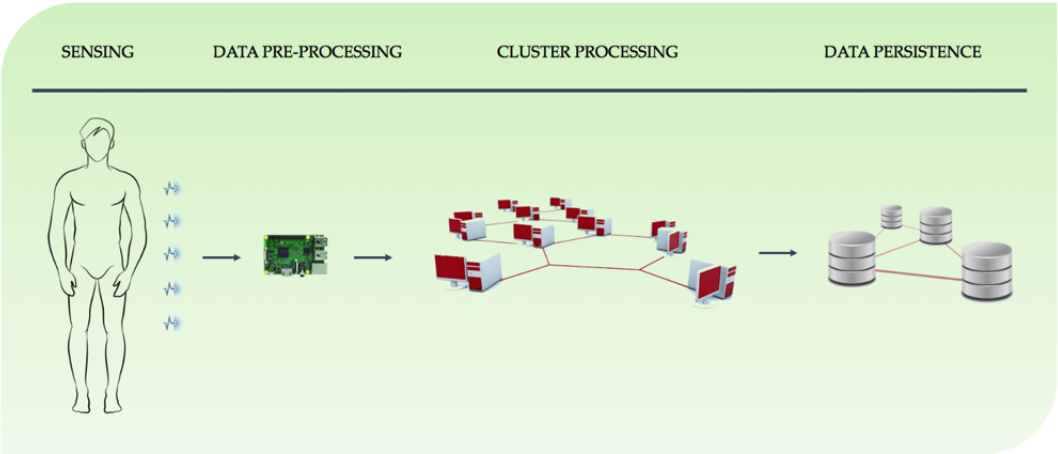

Figure 5.1: *High level architecture of designed system.*

In the next sections each block of the figure, briefly presented here, is discussed extensively:

- The **Sensing subsystem** feeds the entire system and it represents the first functional block of the architectural schema. It ingests data to analyze and it is composed of 8 sensors which generate information about acceleration and gyroscope on left calf, right calf, left thigh and right thigh of individuals which wear them.

- The **Data preprocessing** block is implemented with a Raspberry Pi 3. The goal of this block is to collect inputs deriving from sensing subsystem as an unbounded data flow and to execute a conversion from raw sensor data in an RDFStream representation. Finally, the converted stream is sent to the cluster.

- The **Cluster processing** box is composed of instances of Apache Kafka and Apache Flink. The purpose of the former is to buffer streams and to offer a reliable and fault-tolerant access point for a cluster of commodity hardware. Kafka might be seen as the messaging hub between the *Data preprocessing* block and the rest of the cluster, where Apache Flink is installed and real-time anomaly detection is performed.

- The **Data persistence** block is deputy to store data obtained as outcome from the processing in order to allow further elaborations in future. In *Future works* section potential extensions of the system which use this layer as starting point are presented.

5.3 Sensing subsystem

The sensing subsystem represents the source of the entire framework and it is composed of wearable sensors. For the specific implementations, instead of real sensors, a preexisting dataset named *REALDISP* was used as source to generate a data stream of sensors collected data. *REALDISP* was presented in [32] and [33] and fully described in [34].

5.3.1 REALDISP Dataset

The cited dataset was filled in order to create a fitness dataset for activity recognition; it contains several log files where wearable sensor values sampled at 50 Hz are recorded. Note that the values come from different sensors on 17 involved individuals and the whole dataset contains about 7 GB of data. Each record contains information about seconds and microseconds registered when data was collected and acceleration and orientation sensor values on three axes *(x,y,z)*. Sensors were placed on the following position:

CODENAME	POSITION	OBSERVED VALUE
LC-ACC	Left calf	Acceleration
LC-GYR	Left calf	Gyroscope
LT-ACC	Left thigh	Acceleration
LT-GYR	Left thigh	Gyroscope
RC-ACC	Right calf	Acceleration
RC-GYR	Right calf	Gyroscope
RT-ACC	Right thigh	Acceleration
RT-GYR	Right thigh	Gyroscope

Table 5.1: *Types of sensors. "Codename" column shows the codename employed in script to refer the specific values.*

In order to simulate sensor readings some scripts have been developed, using Python 2.7 as programming languages, to parse *REALDISP* log files and to fire out recorded data with a specific rate according to seconds and microseconds info annotated in every records. In particular, for each record a timestamp (in standard date format: *yyyy-mm-dd hh:mm:ss.mmm*) was created to provide a more human-readable temporal information. Note that for each sensor a different script was developed and each one runs independently from others in order to simulate a realistic scenario. Finally, the output of scripts was structured as a serialized JSON object containing sensor information as described in the figure below:


```
{
    "timestamp": "1970-01-01 01:00:01.460",
    "LC-acc-x": "-1.5553",
    "LC-acc-y": "-0.078098",
    "LC-acc-z": "0.0013644"
}
```


Figure 5.2: *An example from Left calf accelerometer which measures the acceleration value on axes x-y-z.*

In order to represent data, the JSON standard was chosen for its wide diffusion and for the presence of a vast number of plugins to handle it with every programming language: in addition, the choice is due to a reason related to RDFStreams which will be investigated later in *Data Preprocessing* section. Hence, the considered serialized JSON object was sent, using the MQTT protocol, towards a broker installed on an embedded system described in *Data Preprocessing* block. It should be noted that in a realistic scenario, sensors could be linked with the cited embedded system in several ways and technologies, wired or not, like GPIO pins or *HSDPA-LTE*, *Bluetooth*, *Wi-Fi*

and other wireless standard; in order to maintain the most general approach, in this particular implementation MQTT was used to perform this link to take advantage of its platform-independent features and also because fits perfectly the simulated situation where the sensors push out data towards their consumer as data are generated. Moreover, MQTT stores automatically data coming from sensors, even for days, allowing to reach easily a good level of resiliency for the subsystem in case of sensors failure.

5.3.2 MQTT

MQTT is the acronym for *Message Queue Telemetry Transport* and denotes a standard (*ISO/IEC PRF 20922*) designed to describe a publish-subscribe messaging protocol. The official website [35] describes it also as a machine-to-machine connectivity protocol because it fits perfectly IoT requirements, since it is very lightweight and requires limited network bandwidth to transmit data. MQTT is particularly adequate when there are lots of (perhaps different) publishers and consumers because it provides an independent communication hub and it can be implemented in many software, developed with different languages, just with few lines of code. The central nodes in a MQTT infrastructure are called *brokers*: they are responsible for distributing messages to the interested clients based on message's *topic*. Figuratively, a topic can be viewed as a post office box where publishers send messages and subscribed clients are authorized to retrieve them: topics are used from publishers and subscribers to establish a common access point on the broker.

5.4 Data preprocessing

Data preprocessing block is physically located on an embedded system and it is deputy to perform an initial elaboration of raw data coming from the sensors. Getting into details, the pre-elaboration phase was added in order to reduce load amount on the cluster and especially to convert raw streams originated from sensors in RDFStream following the standard showed in [36]. So, an independent subsystem was created providing a modular architecture where each block can be joined with subsystems developed separately, for instance by other designers to fit additional requirements. The only one constraint is related to the use of a standard and largely used messaging hub like Apache Kafka. The choice around the embedded system to implement the preprocessing phase has fallen on Raspberry Pi 3 since its features completely fit the scenario requirements and because it is one of the most widely used board on the market and in IoT context. Those features are described in the next section.

5.4.1 Raspberry Pi 3

Raspberry Pi is a credit-card-sized computer which can be used in electronic projects and for small scale computations in a similar way as a standard personal computer does [37]. Today RPI3 is known in the world as one of most useful and flexible embedded system. In the following, its specs are described:

- *ARM Cortex-A53 64 bit 1.2 GHz quad core*
- *1 GB RAM (shared with GPU)*
- *4 USB ports*

- *HDMI rev1.3 port*
- *MicroSDHC slot*
- *Bluetooth 4.1, Wi-Fi, 100 Mbit Ethernet and 17 GPIO pins*

Hence, the choice to adopt a Raspberry Pi 3 (from here simply Raspberry) as embedded system to perform data preprocessing is mainly due to its low cost (35$), low consumption, powerful architecture, limited size and wide spread in the world. For these reasons Raspberry is supported by a very large community and many open source projects and several Linux distributions were developed to fit every needs; moreover, due to its success many general purpose softwares run over its board.

Speaking of the described scenario, RPI3 is used as a processing bridge between *Sensing subsystem* and *Cluster Processing* block (Figure:5.1).

On it *Mosquitto* [38], a broker which implements the MQTT protocol, and a *Node-Red* server were installed: the latter is a software tool developed by IBM to easily write code for wiring together devices and online services as part of an IoT application. Essentially, *Node-Red* provides a browser-based flow editor; it offers some pre-developed nodes in order to implement most used online services. Then, in Figure 5.3 the *Node-Red* environment installed on RPI3 is illustrated. The *Mosquitto* nodes represent the broker consumers: their responsibility is fetch data from sensors in *Sensing Subsystem* and to push out them towards the function nodes. These last nodes implement a *TripleWave* approach to convert raw live streams in RDFStreams. *Node-Red* offers a set of built-in nodes (MQTT is just one of those) but it allows also the definition and employment of custom nodes. *node-red-contrib-kafka-node* [59] is a set of custom nodes which offers the functionality of a Kafka client: the one described in the figure is a producer node used to deliver data to the next

block. However, the concerned node offers only the opportunity to specify the reserved topic for messages and it does not present all the functionalities offered by a Kafka producer; hence for this project a custom version of it was written with JavaScript and installed on the local *Node-Red* environment in order to fit project's requirements. Furthermore, the opportunity to send also keyed messages to the broker and to set a partitioning strategy based on hash values were implemented.

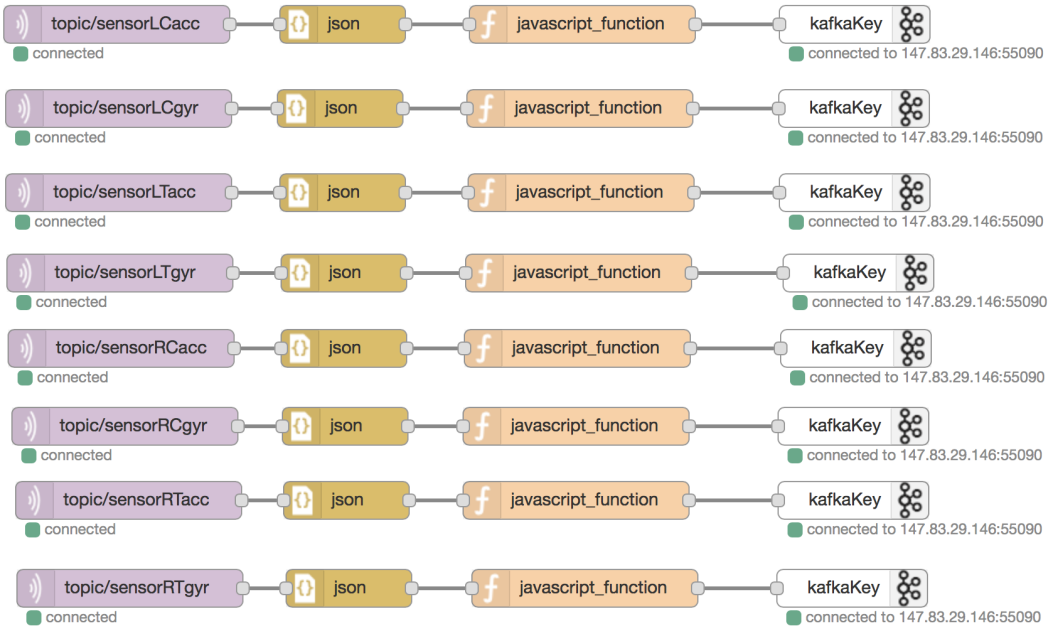

Figure 5.3: *Node-Red environment designed on RPI3.*

5.4.2 TripleWave approach

TripleWave is released as executable software available on GitHub [39]; despite this, it is still in an early stage and today some functionalities are not yet implemented: for instance, the project-required MQTT connector is planned but currently under developing. Since that a custom solution, inspired to the *TripleWave* approach, to convert raw streams in RDFStreams

was preferred and the `javascript_function` displayed in Figure 5.3 were used. Python was considered as an alternative language to write the scripts above due to its power, simplicity and raising spread in IT panorama, but the python nodes implementable in *Node-Red* currently are unstable with an high rate of incoming data so the Javascript version was preferred.

Technically, *TripleWave* creates a JSON-LD starting from a non-refined live stream where *LD* means *Linked Data*. JSON-LD is a specific method to indicate linked data in JSON: now the previously described choice to represent raw sensor data with JSON makes sense because the conversion from JSON to JSON-LD requires just a little effort for developers so simplifying this processing step. Moreover, a JSON-LD can be serialized and handled as JSON, being itself just a particular version of a JSON. JSON-LD is a standard drafted by RDF Working Group and it is designed around the concept of "context" to provide additional mappings from JSON to RDF: the "context" is employed to link object properties of JSON to concept of ontologies. In the following, the ontologies employed to characterize data are listed:

- **IoTDB** [40]: to describe sensor values as acceleration and orientation.
- **MiMuWear** [41]: to describe anatomical body parts.
- **SSN Ontology** [42]: to describe sensor features.

Hence, each `javascript_function` receives raw data from a *Mosquitto* consumer, parses them, creates new JSON with JSON-LD syntax and then provides an output coherent with an RDFStream. This one perfectly fits semantic stream conventions and then it can be sent to an instance of Apache Kafka installed on the cluster in order to be employed in further semantic analysis. In the following figure a sample of the produced JSON-LD is showed:

```
[
    {
        "@context": {
            "generatedAt": {
                "@id": "http://www.w3.org/ns/prov#generatedAtTime",
                "@type": "http://www.w3.org/2001/XMLSchema#date"
            },
            "Sensor": "http://purl.oclc.org/NET/ssnx/ssn#Sensor",
            "SensorName": "http://purl.oclc.org/NET/ssnx/ssn#SensingDevice",
            "ObservedParam": "http://purl.oclc.org/NET/ssnx/ssn#MeasurementProperty",
            "Observes": "http://purl.oclc.org/NET/ssnx/ssn#observes",
            "hasPosition": "http://www.loa-cnr.it/ontologies/DUL.owl#hasLocation",
            "hasValue": "http://purl.oclc.org/NET/ssnx/ssn#hasValue"
        },
        "@id": "3601460.0GYR_SensorXYZ-LT",
        "generatedAt": "1970-01-01 01:00:01.460",
        "@graph": [
            {
                "@id": "http://www.sensor-market-example/sensor/GYR_SensorXYZ",
                "@type": "Sensor",
                "SensorName": "Gyroscope Sensor x-y-z",
                "Observes": "https://iotdb.org/pub/iot-facet#sensor.spatial.orientation"
            },
            {
                "@id": "https://iotdb.org/pub/iot-facet#sensor.spatial.orientation",
                "@type": "ObservedParam",
                "hasValue": "0.005579#-0.0090044#0.78651"
            },
            {
                "@id": "http://www.sensor-market-example/sensor/GYR_SensorXYZ",
                "@type": "Sensor",
                "SensorName": "Gyroscope Sensor x-y-z",
                "hasPosition": "http://www.semanticweb.org/mimu-wear/wear-body#RightCalf"
            }
        ]
    }
]
```


Figure 5.4: *Sample of produced JSON-LD.*

5.5 Cluster processing

The *Cluster Processing* block has a central role in the project. As already explained in the *Introduction*, in chapter 2 and in section 5.1, there are many reasons which carry this system to adopt a cluster processing logic instead of one based on a single machine: in particular, the main one is the enormous potential data amount generated from a Big Data application which cannot be supported effectively by a single operating node. The recent technology

growth helps us to define a powerful but fundamentally intuitive and cheap solution to deal with Big Data issues. A cluster of commodity hardware today can solve problems which few years ago seemed intractable to computers, at least if we do not talk about very complex hi-end calculators which presents other inconvenience like the costs and the need for specialized workforce to deal with. In order to manage the application described in this thesis, the chosen architecture considers the use of a cluster of several nodes which execute three different tasks:

- *Communication with sensing system*
- *Evaluation and processing*
- *Data storage*

The first point is covered with a small cluster of three Apache Kafka brokers. The second one is executed using ten nodes with a running instance of Apache Flink which perform anomaly detection using a distributed implementation of HTM library. At last, a group of three nodes forms an Apache Cassandra cluster which represents the persistent storage for potential further processing.

5.5.1 Kafka cluster

As already seen in chapter 4, Kafka represents one of the most widely spread messaging system in cluster processing field. In this thesis it is used as an hub to collect data coming from sensors and to provide a reliable access point for the Flink application. The Kafka cluster is composed of 3 nodes: the broker's number was chosen considering the desired replication factor for the hub. This one represents how many times data have to be replicated in

order to offer a reliable access point: if a node fails, data are preserved in other brokers and Kafka can expose always an access point to consumers. In the described architecture, a replication factor of 3 was chosen accordingly to Kafka documentation because it represents, in most realistic scenario, an appropriate backup. On the other hand, Kafka allows to set up a cluster with a number of brokers greater than the replication factor in order to offer different ways to balance loads. Anyway, a single Kafka broker is usually able to handle many terabytes of data so in this project, where data loads do not reach such a huge amount, a number of brokers equals to the replication factor was considered widely sufficient.

It is important to note that Kafka server autonomously choose a cluster leader and automatically moves the cluster control to other brokers if the leader fails. Due to these reasons, each broker is chosen identical to others and each one manages exactly 8 topics. A Kafka topic is an abstract area where data are sent by producers and retrieved by consumers: we can see that as a postal box where everyone can deliver messages but just subscribed members can consume from it. In this architecture, there is a topic number equals to the sensors number in order to reserve a topic for sensor. In this way, the sensors have a preferred zone to send own readings and consumers can subscribe to particular topics being sure to retrieve just data they care about.

Kafka allows to split topics in partitions: the number of these represents the parallelism degree of a Kafka broker because producers can execute write operations on different partitions in full parallel way. However, on consumer's side each thread always get data from one partition at time. Partitioning topic has several benefits and one of these is a general improved scalability: when a topic presents just one partition every messages sent to it are stored in

a single partition-related log file which completely resides in a single machine. Given that, the maximum size of the log file is constrained by the physical size of disks equipped by the particular node. Partitioning allows to spread data in different log files which can be host on different machines without worrying about physical disk space of specific nodes. Finally, other benefits concern server and client load balancing allowing parallel operations on brokers, where each one can be the leader of a particular single partition.

Besides, Kafka is able to send keyed messages which are strictly related with partition's concept. For each keyed message Kafka calculates an hash value for the key: the messages which exhibit the same hash are deterministically mapped and sent to the same partition This behaviour is useful to distribute the load on the broker or through the cluster and it can be fundamental for certain applications: messages within a partition are always delivered in-order to the consumer. It is important to note that if we want to modify partition number when Kafka server is running the in-order guarantee cannot be hold: in this case, messages with same hash could be spread in different partitions so a good rule is to set partition's number at first.

Unfortunately, an exaggerated number of partitions carries some disadvantages. As already seen, every partitions are replicated on every brokers and each partition has the own log file. Every time a message is sent to a partition an I/O operation is performed and a number of writes, that depends on replication factor, is executed. These operations are fundamental because they permit to Kafka to guarantee its feature but at same time they affect system's performance. However, Kafka is able to manage thousands partitions at same time without problems.

In the described system a fair partition number for topic was established; accordingly to [43], it should depends of the target and achieved throughput

of the system. Note that the architecture described here contemplates an estimated throughput of 400 KB/s for individual so the partition number should depends of how many people are going to use the system simultaneously. The proposed work is just a concept of a cluster processing system so a flat number of 10 people is considered a good target in order to obtain a consistent data throughput of 4.0 MB/s: hence, each topic will be divided in 10 partitions. Note that in this way data related to a single individual are always posted in same partition exploiting Kafka hashing function.

5.5.2 Flink cluster

The purpose of the Flink cluster is to fetch sensor data stored in Kafka and process them. In particular, an instance of the HTM algorithm to perform anomaly detection with cluster processing was employed exploiting a java library (details in section 6.3.2). Originally, HTM was not designed to run on a cluster, anyway it supports some of the principal programming languages like C++, Java and Scala so it can assist applications implemented on a Linux cluster. Some researchers have already developed an HTM library supported by Apache Flink [49]: in this work the described library was retrieved and adapted to the project's characteristics and requirements.

Developing an efficient version of the Flink-HTM algorithm needs a cluster of multiple nodes because the high arrival rate requires a spread elaboration to satisfy the real-time demands. Actually, the resources requested to perform the anomaly detection are not so high: a bunch of single-core machines could be sufficient to handle effectively the throughput for a single subject so the architecture must be designed taking into account the number of persons involved in the system. It should be noted that the cluster can be extended progressively simply adding nodes as the individual number grows.

In this section, the basis of Flink architecture will be illustrated to better clarify the choices taken in the cluster design process. Figure 5.5 describes the Flink architectural stack: on the lower level three development possibilities are showed. In the presented case, the *Cluster* was the only one considered and in particular the *Standalone* mode was chosen to deploy it. The other possibilities implement Flink on a preexisting installation of YARN or MESOS. These ones are two celebrated cluster managers extremely useful in large scale clusters due to their strong capabilities to manage failures and resources. However, in this project the cluster is composed of few nodes, so a standalone set was preferred due to its greater simplicity and to maintain platform's technology independence. In the third level of the stack Flink allows programming against two API sets: *DataStream* and *DataSet*. The former was widely used in the designed system in order to deal with streams coming from Kafka's topics while DataSet are dedicated to batch elaborations, which are basically absents in the described work.

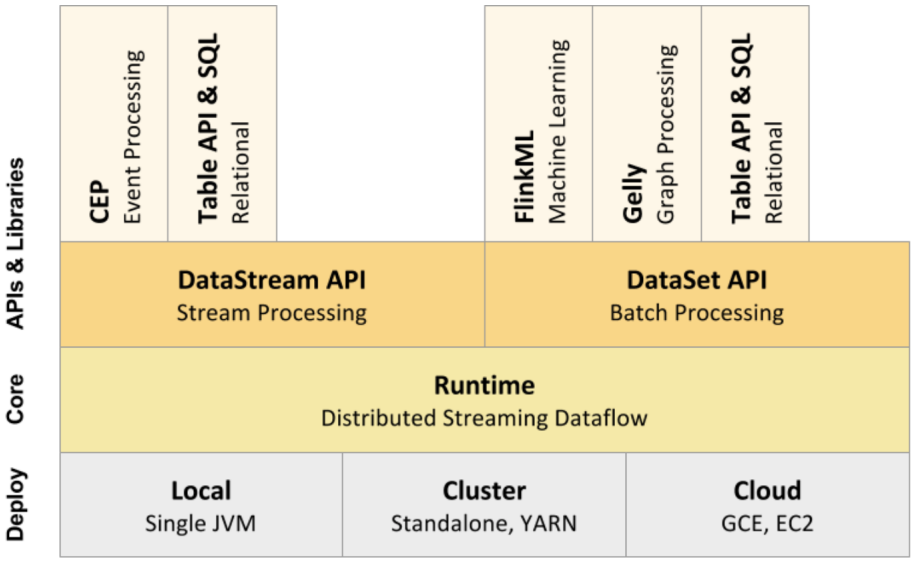

Figure 5.5: *Basic schema of Apache Flink architecture [6].*

In Flink, there are two types of nodes: *Job Managers* and *Task Managers*. The formers are also called *Masters* and coordinate the distributed execution, schedule tasks, plan checkpoints, provide recovery in case of failures. There is always at least a Job Manager but in big clusters or when high-availability is a fundamental property there could be multiple masters: in any case there is just one leader and the others are backup masters. When a job is running, the Job Manager keeps track of distributed tasks, decides when to schedule the next one and reacts to finished tasks. About Task Managers, they are also called *Workers*. They executes tasks and subtasks about data flows and buffers data streams. The Task Managers are connected to the Job Managers to notify their availability or to announcing they are computing a task. It should be noted that there is always at least one Task Manager but usually there are many of them. In the described project we have to deal with a small sized cluster so the presence of a single Job Manager is widely sufficient to handle it.

A task in Flink is the basic execution unit and the place where each parallel operator instance is executed. It should be noted that a Task Manager node hosts a JVM process so it can execute more than a task using multithreading architecture and separating a single task in many subtasks. Initially, each worker has to specify its number of task slots. A slot represents a fixed subset of Task Manager's resources: on the other hand, just the memory is split and reserved to the specified slot while the CPU capability is shared by the slots. By default, Flink allows subtasks to share slots even if they belong to different tasks. Slot sharing provides some advantages about getting better resource utilization: without it lightest subtasks will block resources as heaviest operations do. Instead, using slot sharing there will be a fairly distribution of resources. The choice about number of slots per node

and the use of slot sharing depends by the particular application: generally, the documentation reports that a good rule is to assign a number of task slots equals to the number of CPU cores of the node. From this perspective, the latter choice was taken in this project so assigning a number of task slots equal to CPU cores.

In the project a particular type of stream named *KeyedStream* is used. It partitions data on a specified key: essentially Flink creates a set of sub-streams with same key and autonomously distributes those to different slots. For these reasons, no constraints were applied to Flink nodes about tasks assignments, which is delegated to Flink's engine. On an higher level we can distinguish 3 essential tasks to execute: data retrieval from the Kafka broker, HTM data elaboration and data storing into Cassandra database. About Kafka and Cassandra, Flink's default connectors have been employed whereas for HTM it the java library developed in [49] was integrated.

Flink considers three different notions of time in streaming programs:

- **Processing time**: It refers to the time as the system time of the machine when the operation on data is executed.
- **Ingestion time**: It refers to the time as the system time of the machine when data get into the system.
- **Event time**: It refers to the time incapsulated in the data, for instance a timestamp tied to the specific value.

5.5.3 Data stream output consistency

It should be noted that the processing and ingestion time are very influenced by delays and latencies due to causes external to the system, so it would be possible that some values are received out-of-order and the uncorrelated

timing influences the computation. Sometimes maintaining the order of data respect to their generation time is essential so the event time mechanism is strictly necessary. In the project, data are medical values and their generation moment is crucial so the event time characterization is used to establish an order between data values: the time field is generated and included like a timestamp by the sensor and it is made explicit in the message sent to Flink. In order to exploit event time characteristic a custom timestamp extractor was developed.

About data reordering, a window of fixed length was used to check the data order in the streams: Flink applies a tumbling window (with duration of 800 ms in the implemented case) which collects data coming from the streams. Then, within the window Flink sorts messages on event time basis. Unfortunately, Flink does not implement this functionality with own API so it is not possible to perform incremental aggregations of data, i.e. messages are not ordered as they arrive: a routine will be launched when the windows ends leading to a greater resource consumption and even lesser performance due the unoptimized ordering function. It should be noted that the choice of the window duration could be very important: a too short one affects the performance because could trigger the sorter too often, while if it is too long the function will be called on a consistent amount of messages so requiring a lot of times to complete the sorting.

The event time it is also used to emit watermarks, which are used to measure time progress. Essentially, a watermark declares that the event time has reached the specific instant t and it means that no messages with event time lesser than t should arrive in future. Clearly, it cannot be assured so the mechanism marks out-of-order messages comparing its event time with the last computed watermark: if they arrive out-of-order with a lateness

greater than a fixed value they will be dropped because there are no chance of re-insert them in the stream. Effectively, if the element's sorting is very important dropping an element is better than adding it, unordered, within the stream.

Finally, Flink has a limited knowledge about data types hence it handles serialization just for java primitive types: given that and since Flink does not support natively deserialization of Kafka messages even a custom deserializer was developed.

5.6 Data persistence

The *Data persistence* block is designed to offer a reliable storage for data generated from sensors and computed in the Flink cluster. In the project it is composed of a NoSQL cluster with an installed instance of Apache Cassandra. The designed cluster is formed of three nodes. In the remainder of section the reasons which have led to the carried out choices about Cassandra's nodes configuration are described.

5.6.1 Cassandra cluster

Apache Cassandra is a NoSQL database belonging to the column-oriented family. It is a system designed to run on cheap commodity hardware and a platform to handle high write throughput without sacrificing a good read efficiency. NoSQL paradigm is completely different than the RDBMS's one; here its features are not explored completely because it falls outside the purposes of this thesis. Further details about column-oriented paradigm can be found at [45] and [46].

Cassandra, as other NoSQL databases, is optimized to work in distribute

ways; the reasons are encapsulated in Big Data concepts. In fact, the need for managing a huge amount of data, making them constantly available and handling high throughput lead to the use of multiple nodes in order to satisfy efficiently every requests. Cassandra, in particular, offers a P2P architecture which avoid single points of failures and allows to reach high availability property more easily respect to other storage systems. In a Cassandra cluster there are no master nodes and, through a gossip protocol, each node is informed about the status of other machines in the cluster. Intuitively we can imagine Cassandra cluster's topology as the ring showed in Figure 5.6 where each node is directly linked with the adjacent ones. Each node is the first responsible for a portion of data indexed with a key named *partitioning key*. Nevertheless, data are replicated in many nodes in order to guarantee their availability also if the responsible machine is down.

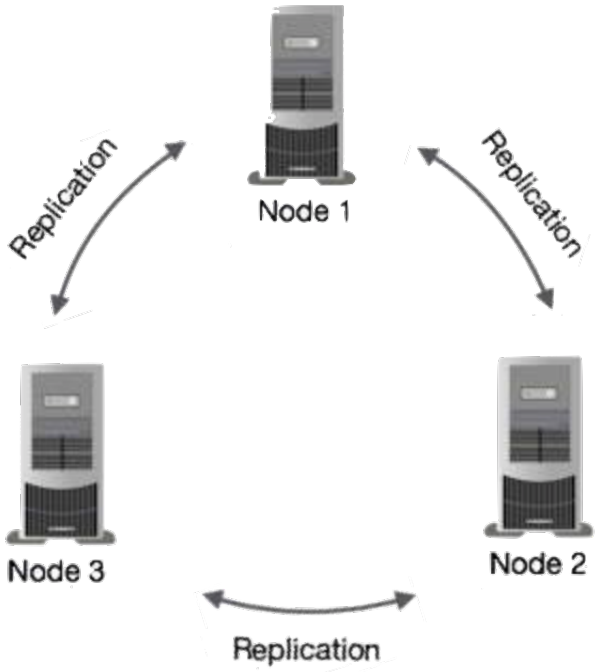

Figure 5.6: *Topology of a Cassandra cluster. In the figure the ring designed in this project is displayed*

Cassandra distinguishes between 2 different replication strategies:

- ***Simple Strategy***: The default strategy is used with clusters formed of a single rack. Data are replicated a number of times on adjacent nodes with natural clockwise order.
- ***Network Topology Strategy***: It is used with multiple data centers, eventually distributed in different geographical locations.

The data distribution criteria is based on the hash value of the *partitioning keys* so Cassandra assigns to each cluster node a random value which represents its position within the ring. Each partition key is mapped on a specific node which becomes the coordinator for the particular keyed data. Accordingly to the requirements, data are replicated on a number of nodes in clockwise-order starting from the first host. In [45] some arrangements to avoid an unbalanced cluster are described.

Another important aspect to consider during the configuration of a Cassandra cluster is the desired *consistency*. It can be *immediate*, which assure that when a client reads a value the system returns the updated one, or *eventually* which returns "eventually" the last updated value.

In Cassandra a write operation returns a "success" response with three different level of correctness, tunable by the developer:

- **ONE**: The operation returns a success if at least one replica has acknowledgment.
- **QUORUM**: The operation returns a success if the majority of replicas has acknowledgment.
- **ALL**: The operation returns a success if all the replicas have acknowledgment.

Choosing a level of write correctness affects system performance depending of the size of the cluster. Clearly, in a single node cluster just the "ALL" option is available.

It should be noted that the most of the optimizations and configurations exposed in the remainder of the section are focused on write operations because in this project Cassandra has to handle a big amount of writes compared to reads. Moreover, the great performance in executing writes is the main reason which have lead to the choice of Cassandra for the persistence layer. The design of the cluster's configuration was drawn up accordingly to the previous considerations and taking into account recommendations of official Cassandra documentation. The replication factor chosen for the cluster is 3: then data are stored and copied on 3 different nodes. It allows to obtain the best trade off between performance, consistency and availability using QUORUM criteria to define a successful write operation.

It should be noted that a replication factor less than 3, although it offers better performance with less latency, does not provide much flexibility to tune consistency and availability while a greater RF influences performance raising latency to perform writes on more nodes and exposing the system to a greater number of failed writes. About replication strategy, due to the presence of a cluster composed of a single rack, a Simple Strategy was chosen.

About the cluster design, an important aspect is the hardware sizing. A configuration composed of multi core processors was preferred respect than a single core due to the high rate of write operations to perform. In particular the documentation advises to employ CPU with 16 cores for production and 2 for tests. Examining the simplicity of the environment presented in this document, a common quad core CPU was considered sufficient. About memory, some considerations must be taken into account: Cassandra's write

operations are performed on different types of memory. Initially write operations are marked on a *commit log* which lies on HDD. Then, data are written primarily on a *mem table* on volatile memory and finally, when mem table is full, it is flushed on mass storage in a structure called *SSTable*. Generally, having a configurations with a lot of RAM is better because it reduces the number of memory dumps; anyway the documentation advises to use at least 8 GB in production and 4 GB in tests. The first choice is widely adequate for the studied project. About mass storage, the doc [47] advises to use SSD instead of HDD due to their less latency but a hi-end disk could b a better compromise between costs and performance. Cassandra stores in mass storage both commit logs and SSTables but the space occupied by the formers is negligible respect to SSTable. Cassandra periodically executes a space compaction of SSTable in order to reduce space waste and get more space for further data: there are many strategies to perform it but everyone utilizes disk space so a good practice is to provide always an additional amount of space (between 10% and 50% of the total occupied by data). Accordingly to the project's data size, a 2 TB of disk space per node is required in order to store safely data computed and enriched by the cluster. For the cluster sizing evaluations also the following paper [48] was considered.

5.6.2 Cassandra data modeling

Data modeling in Cassandra is completely different respect to a RDBMS so a typical relational approach should be very inefficient. Usually, and this is the case, NoSQL databases are modeled thinking about queries which will be performed on it. Normalization and relations between tables, always taken into account when we design a relational database, have to be avoided in Cassandra because they lead to a catastrophic use of resources. Cassandra

is strongly optimized to execute write operations so they have a negligible cost compared to reads: given that, the primary goal of data modeling is to organize data in order to provide reads as efficient as possible. Generally, data modeling in Cassandra has to consider two purposes:

- Data must be spread nearly the cluster as much as possible.
- Queries have to scan the minimum number of partitions to retrieve data of interest.

Often this objectives are in contradiction so the developer is in charge to find a good trade off. Choosing an adequate primary key is a good way to satisfy the first goal. The key and its hash value spreads data in partitions which are hosted within the cluster. Reading a partition is an expensive operation because each one can be placed on a different node, so developers must avoid it as much as possible. On the other hand, also reading partitions which reside on same node is more expensive than scanning a single one: possibly, each query has to scan the minimum number of partitions so correlated and requested data have to reside on same partition. Since that, some queries were formulated in order to provide a correct implementation of Cassandra:

A) *Retrieve every values belonging to a specified sensor of a specified user*

B) *Retrieve every abnormal values belonging to a specified sensor of a specified user*

C) *Retrieve every values belonging to a specified sensor of a specified user in a fixed time interval*

In order to execute efficiently the query A and B the tables represented in Figure 5.7 and Figure 5.8 were implemented. The table in Figure 5.7 is employed also by the query C.

```
CREATE TABLE values_by_sensors_users (
  user text,
  sensor text,
  observed_value text,
  value double,
  timestamp timestamp,
  rdf_stream text,
  PRIMARY KEY ((user,sensor), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC)
```

Figure 5.7: *Table used to provide efficient reads for queries A and C*

```
CREATE TABLE anomalies_by_sensors_users(
   user text,
   sensor text,
   observed_value text,
   value double,
   anomaly_index double,
   timestamp timestamp,
   rdf_stream text,
   PRIMARY KEY ((user,sensor),timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC)
```

Figure 5.8: *Table used to provide efficient reads for query B*

It should be noted that query A and C use the same table: the only difference is in the query formulation which includes a temporal criteria. The primary key definition of the table creates a *compound partition key* composed of *sensor* and *user* fields: so there will be a separate partition for

every pairs sensor-user and the query allows to scan just a single partition to retrieve interested data. The second field of the primary key is called *clustering key* and it is used to establish a sorting among records: in the particular case records are ordered on time basis. The last query clause is used to optimize the query execution: it reduces latency because the data sorting is performed at insertion instead to do it when the query is called up. It should be noted that a new query like "*Retrieve all sensor data for a specified user*" would be very inefficient with this model, because it would require a scan of a number of partitions equals to the number of sensors. In this last case a new table must be designed.

Another table was implemented to perform efficient data fetching for query *B*. In a relational database this table would be a violation because it produces data redundancy: in a NoSQL database instead it represents a good example of data modeling based on query requirements.

Chapter 6

Experimental study

6.1 Computational infrastructure

In the chapter 5 has been described an ideal infrastructure which suits perfectly the real requirements of a typical application representing the document's purposes. Then, it is summarized in Figure 6.1 in order to expose the architecture needed to efficiently support the application.

6.2 Adopted infrastructure

The application designed and exposed in this thesis was elaborated during a study period at *Universitat Politècnica de Catalunya*. The cluster's hardware used for experimentations and tests was provided by *Research and Development Laboratory* (RDLab) of [44] Computer Science Department at UPC. Due to some limitations tied to hardware availability, the development of the application followed a simplified architecture. In particular, the adopted infrastructure is composed of the node listed in Table 6.1 :

This infrastructure is sufficient to handle the amount of data and the

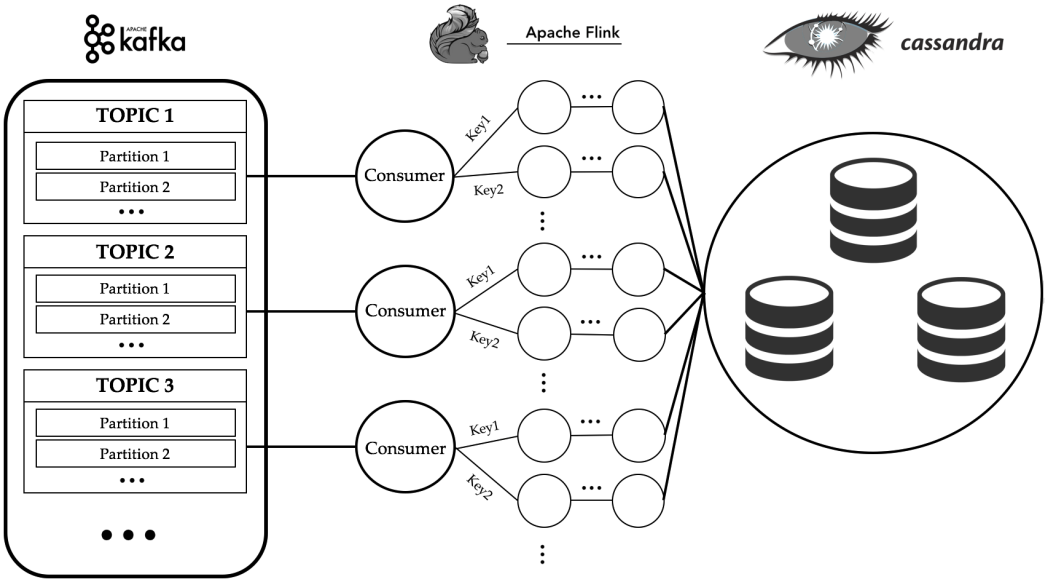

Figure 6.1: *In the picture the designed architecture is depicted. Starting from the left: structure of kafka brokers, Flink's cluster and Cassandra system*

N. NODES	CPU	RAM	HDD	RUNNING INSTANCE
1	Single core	4 GB	200 GB	Kafka
1	Dual core	4 GB	1 TB	Cassandra
3	Dual core	2 GB	100 GB	Flink

Table 6.1: *Details about nodes composing the adopted infrastructure.*

throughput required by the developed application considering a single individual involved. The Sensing subsystem produces around 45 KB/s each person; then the Raspberry collect, enrich and convert the stream in a JSON-LD stream and finally send it producing in output a throughput of 34 GB/day each person. The available disk space for the Kafka node is 200 GB so it can offer data caching for a single individual (i.e. 8 sensors) for 5 days.

Flink's input throughput is equal to 1.2 MB/s each person: anyway these data are transient and are stored in mass storage only if the memory is full. Considering the throughput and the fact that Flink consumes data in real-time, the possibility that a cache of 2 GB (the RAM size for Flink nodes) would filled due to a bottleneck during the processing are really low so the 200 GB of available disk space are largely sufficient.

Due to the computation and data enlargement operated by Flink, the throughput towards Cassandra is increased until 100 GB/day each person. In Cassandra, half of the available disk space has to be reserved for SSTable compaction to keep high performances (details in section 5.6.1): given that, a disk space of 1 TB is enough to handle 5 days of continuous work while 4 GB of RAM memory assures a sustainable dump rate between mem table and SSTable.

Unfortunately, the absence of redundant nodes for Kafka and Cassandra makes impossible to evaluate performances in terms of fail recovery, data replication and leader-role switching that are typical situations in a real cluster. Moreover, it should be noted that the three nodes running Flink are not installed on separate physical machines, but they are simulated using three virtual machines (with reserved resources) on the same hardware, so any analysis about linkage performance within Flink cluster cannot be performed.

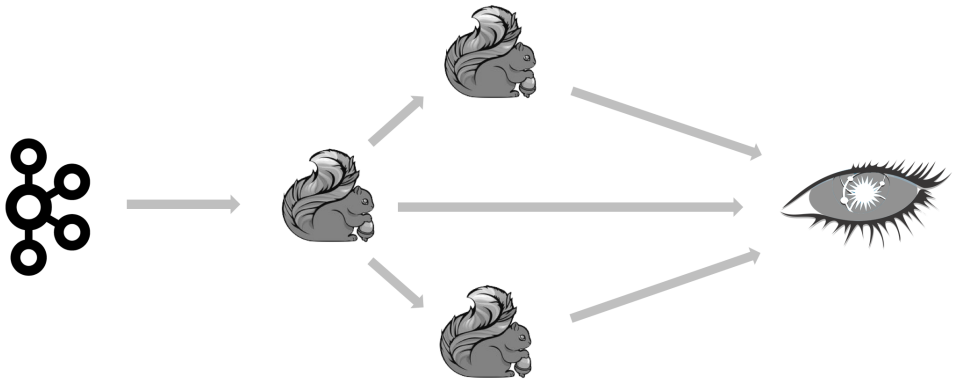

Figure 6.2: *Schema of the adopted infrastructure. The leftmost Flink node is the Job Manager which fetches data from the Kafka broker and distributes the load through the Task Managers (even to itself since it is a Task Manager also) which are responsible of publishing data to the Cassandra's database.*

6.3 Testing and evaluation

The infrastructure described previously was employed to evaluate the results obtained with some tests and experiments performed in a case of use. Taking into account the limitations explained in the above paragraphs, the system's performances were evaluated in order to provide a yardstick to understand how many nodes and what kind of resources are needed to handle efficiently a real production system.

The infrastructure is located at the *Research and Development Laboratory* (RDLab) [44], property of the Computer Science Department at UPC University. Each node presents an instance of *Ubuntu 12.04.2 LTS* whereas a recent version of *Java* (1.8.0.131) has been installed to exploit completely the features of the employed software. The nodes are accessible via *Secure Shell* protocol then each operation on nodes was executed using the functionalities offered by the Linux console. In Table 6.2 the software installed on specific nodes are showed:

NODE NAME	INSTALLED SOFTWARE
Giordano-1-4-200	Apache Kafka 2.11_0.11.0.0
Giordano-2-4-1000	Apache Cassandra 3.11.0.0
Giordano-2-2-100-1	Apache Flink 1.3.2
Giordano-2-2-100-2	Apache Flink 1.3.2
Giordano-2-2-100-3	Apache Flink 1.3.2

Table 6.2: *List of the software installed on specific nodes.*

It should be noted that on `Giordano-2-2-100-1` resides both the Job Manager and the first of the Task Managers of the Flink cluster while on the other nodes run only Task Managers. Since the machines host a Linux OS, the Linux console tools were used to evaluate resources performances. `sar` command was employed to asses CPU utilization: it shows all running processes on the node and the associated CPU consumption percentage for each core or the cumulated one which is more useful in this case; moreover, `sar` shows also statistics about memory usage. The outcomes were obtained parsing an annotated output file with an interval of 1 second. These considerations are listed in section 6.3.1 while in section 6.3.2 the outcomes of the HTM analysis and the evidence of the anomalies found within the ingested streams are reported. The data employed in the test come from an abstract of *REALDISP* dataset: it contains measurements of sensors like accelerometer or gyroscope: a full description of the dataset is available in [34].

6.3.1 Nodes performance

As already seen in section 6.2 the described system is formed of a cluster divided in four parts: a Raspberry Pi, a Kafka broker, a Flink mini-cluster and a Cassandra database. In this section node's performances are examined

especially in term of CPU and memory usage during tasks execution: in some case (e.g. Apache Kafka) also other parameters are examined. Many words will be spent on Apache Flink because it represents the core and the most complex part of the system guiding even the analysis concerning the other parts of the system as a whole.

A task which provides the full functionalities of the system was fired up employing a set of 8 sensors simultaneously in a fixed ingestion frequency: on these basis the performance on the adopted infrastructure were evaluated. In the remainder, a comparison of nodes' performances obtained changing the number of sensors involved in the system is illustrated to study how the system reacts and how much the entity of load increase hits the node's resources.

Ingestion frequency: 50 Hz

The first experiment set an ingestion frequency of 50 Hz which is generally considered an high value for medical sensors that usually have a transmission rate of 20-25 Hz: due to the high rate the duration of this experiment is quite short however it is useful to do some considerations about nodes and the software involved. In table Table:6.3 the throughputs produced for each software are illustrated.

Firstly, in Figure 6.3 the performance of the Raspberry Pi is analyzed: the graph displays the percentage of utilization of the quad-core CPU and memory amount required to handle the process. When the system starts the Raspberry Pi runs the *Mosquitto* broker with 8 active topics and the related consumers, 8 running javascript independent functions and 8 Kafka producers which send keyed messages to 8 different Kafka topics. In order to get most reliable results and to avoid to affect statistics all non-essential interfaces like

	INPUT	OUTPUT
Raspberry	45 KB/s	400 KB/s
Kafka	400 KB/s	1.2 MB/s
Flink	1.2 MB/s	$>$ 1.2 MB/s
Cassandra	$>$ 1.2 MB/s	n.d

Table 6.3: *Input and output produced throughputs with an ingestion frequency of 50 Hz.*

bluetooth, GPIO, Serial and others were deactivated. Data originated from the scripts (which simulate the sensors) consist of 8 sequences of messages of 113 bytes sent with a frequency of 50 Hz each one.

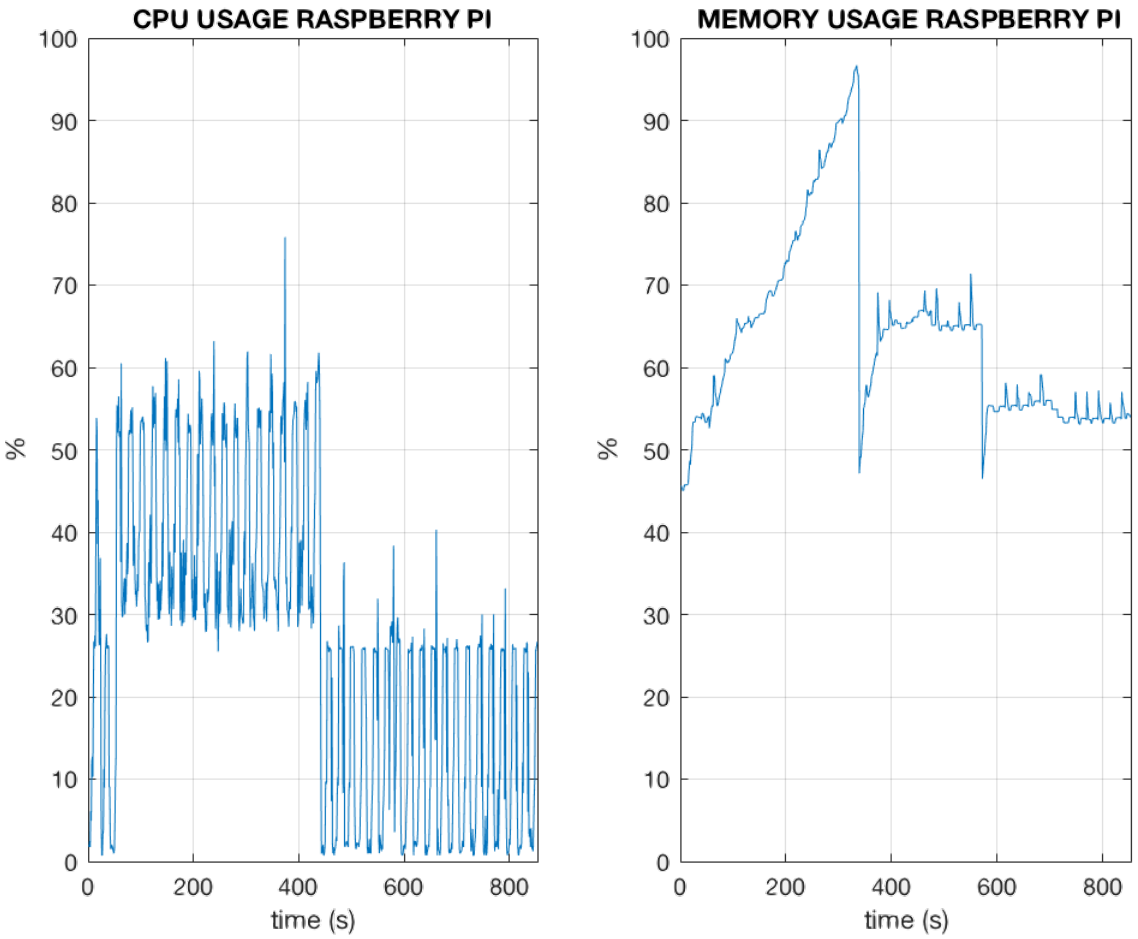

Figure 6.3: *The left side shows the CPU utilization while on the right we can see the memory consumption. On the y-axes there is the usage percentage while on the x-axes the time is expressed in seconds.*

Initially the graph describes an idle situation, i.e. when the Raspberry Pi has no application running, so CPU and memory usages are due only to OS tasks. At the 10th second the *Mosquitto* broker and the *Node-Red* server are fired up and the initial peak shows the efforts required to start them: however, the situation quickly returns to a normal percentage. At second 53 the data stream is launched and it begins to be elaborated by the Raspberry. It should be noted that at this moment the CPU usage has a strong rise until around 60%. It is interesting discovering that Raspberry's CPU is essentially

unaffected by the efforts required to run the MQTT broker and the *Node-Red* server. An important increment in CPU's usage is clear only when the data stream is pushed into the MQTT broker and it ends at second 441 when the task ends: in any case the load appears to be largely supported by the Raspberry Pi.

About memory a premise should be considered: memory management is primarily an OS responsibility and it is true for all nodes of the system so the following results are partially affected by that. Generally we can note as the usage in idle state lies between 40% and 50%; later when *Mosquitto* and *Node-Red* started it reaches the 55% while at second 53 the utilization begins to raise because data streams arrive to the Raspberry: the increment is gradual as the data arrives and at second 337 the OS is obliged to free RAM memory to handle forthcoming data, reasonably. After that, the utilization remains under 70% until the stream ends and later drops towards an idle state. Anyway, these results do not indicate the Raspberry's RAM is not able to handle much more data than just performed: the OS generally uses more memory in comparison to how much is strictly required in order to spare time and resources needed to execute a dump on mass storage. Based on the experimental results we observed that the system bears safely the proposed load.

The Flink's cluster is composed of more than one node. The 3 interested nodes have the same configuration with a dual-core CPU, 2 GB of RAM and a disk space of 200 GB. Generally, Flink nodes can cover two different roles within the cluster so in this case we have one Job Manager, which manages and distributes the job, and 3 Task Managers because one of the machines works both as Job Manager and Task Manager. Having a node which has a double role is unusual in Flink due to the required resource's sharing:

FLINK CONFIGURATION			
NODE	**ROLE**	**HEAP (MB)**	**N. SLOTS**
Giordano-2-2-100-1	JobManager	256	Not defined
Giordano-2-2-100-1	TaskManager	1512	1
Giordano-2-2-100-2	TaskManager	1512	1
Giordano-2-2-100-3	TaskManager	1512	1

Table 6.4: *The configuration chosen for each Flink node*

here the choice is mandatory to assure a good performance level with of a cluster composed of few nodes. Particular attention was paid to tune the right parameter set about heap size, slot number and parallelism for each TM so many configurations were tested in order to find the best one. Flink runs operators and user-defined functions inside the Task Manager JVM, so the heap amount reserved for each TM should be as large as possible to get more benefits: memory is shared with the OS, hence an analysis about how much memory was occupied by Flink was performed. In order to measure it, some experiments were ran setting an increasing heap size to discover the reachable limit: the `top` linux tool was used for this purpose and finally the chosen values for each node are showed in Table 6.4. Clearly, the node `giordano-2-2-100-1` divides its memory between the Job Manager and the Task Manager: the former, due to the few nodes to handle, does not need a big amount of MB so the majority is left to the TM. In the same figure the established slot numbers are displayed and in this case a single slot for TM was set arguing with documentation advises: the justification for this choice will be clear soon in the following of this section.

The first experiment was initially started using 2 slots each TM, accordingly to the documentation, since each TM is equipped with a dual-core CPU.

All the 8 sensor streams ingested in Kafka were analyzed. Unfortunately, all the TMs was failing continuously due to memory overflow errors while the Job Manager restarted the job many times in vain. The reason of the failure lies on the inference task performed by the detection algorithm: it produces an heavy effort analyzing too many streams on the same node. On the other hand, it should be considered that for each sensor 3 neural networks have to be implemented because each sensor is composed of one stream for each "physical direction" of data (axes x,y,z). Performing real-time anomaly detection on even just one of them is a very expensive operation in term of memory consumption and CPU usage. Essentially, data were ingested faster than the network was able to elaborate them so the long formed data queue led the JVM to a crash.

Rehearsing to solve the issue more memory was reserved to the application reducing the heap portion allocated for Flink's internal operations from 70% to 20%: regrettably it was not enough. Another attempt concerns the distribution of job's tasks within the cluster. Usually, Flink distributes tasks through the nodes trying to maximize efficiency so it tends to allocate in the same slot operators which share data or with similar task: in our case this behaviour could lead to an unbalanced cluster. Flink does not recognize machines properly but it organizes the cluster looking at slots: it could distribute many HTM operators in 2 different slots which reside on the same machine, causing a dramatic load on the specific node. The issue can be mitigated influencing manually the task distribution strategy, setting up just 1 slot each TM and forcing the application to reserve a specific slot for particular operators: Flink calls this functionality *Slot Sharing Group*. Usually, SSG is used to force the application to put in the same slot a group of operators for user needs. Anyway, the developer cannot choose exactly which

slot of which machine has to host the group, because in Flink does not exist the concept of machine itself.

Therefore, to figure out the limits of the configuration, an experiment with just 1 sensor (i.e. 3 simultaneous streams) was planned. Precisely, the sensor analyzed by Flink was the accelerometer located on the left calf of an individual. Anyway, the active sensors in the system were still 8 so the number of streams seen by Raspberry and Kafka is always the same. Differences are in the number of analyzed streams in Flink, the data amount sent to the Cassandra database and the number of Kafka consumers. Thus, each Task Manager was deployed with a single slot and the SSG was adopted to have a balanced cluster. Using SSG Flink automatically creates a so-called *Default Slot* where will be deployed all the operators not assigned to a particular slot. Hence, in order to implement the analysis of 3 streams, 2 custom slots were specified to host respectively two of the network operators while the third one was deployed by Flink in the *Default Slot* with all remaining operators.

As consequence of SSG, we have to sacrifice one of the most interesting Flink's feature: the operator parallelism. It allows to split the execution of operators in many parts, so partitioning the streams and spreading the elaboration within the slots. Each partition is elaborated in a different slot while the results can be unified in a single stream, forwarded towards an operator with same parallelism or spread to another with an higher one. Clearly, the data distribution affects the ordering within the stream and it cannot be applied in every cases: for instance, the *flink-htm* network operator cannot be parallelized to execute correctly its operations. Since two of the tree slots are reserved, the other Flink operators are constrained to reside in the *Default Slot* with no possibilities to use parallelism: this one requires a separate slot for each parallel instance. The parallelism would provides an

important performance boost for applications but it could be implementable just adding more TMs to have more available slots or employing a bunch of more powerful TMs to avoid the use of SSG.

In Figure 6.4 and in Figure 6.5 the CPU usage percentages for the Flink nodes are displayed: we can note that after an initial phase where the CPU utilization is high, due to the communications need for job submission, the percentage hardly exceeds the 50%. The sudden low usage period nearly 330th second is probably due to a network lack which prevented the application to consume data from Kafka; this insight is confirmed by the subsequent peak showed in the graph which corresponds to a relative large number of data to analyze. Anyway this event confirms that together Flink and Kafka face successfully an issue like a network lack. Finally, the elaboration as a whole is performed in real-time as confirmed by the fact that the CPU percentage drops exactly when the raspberry stops to send data.

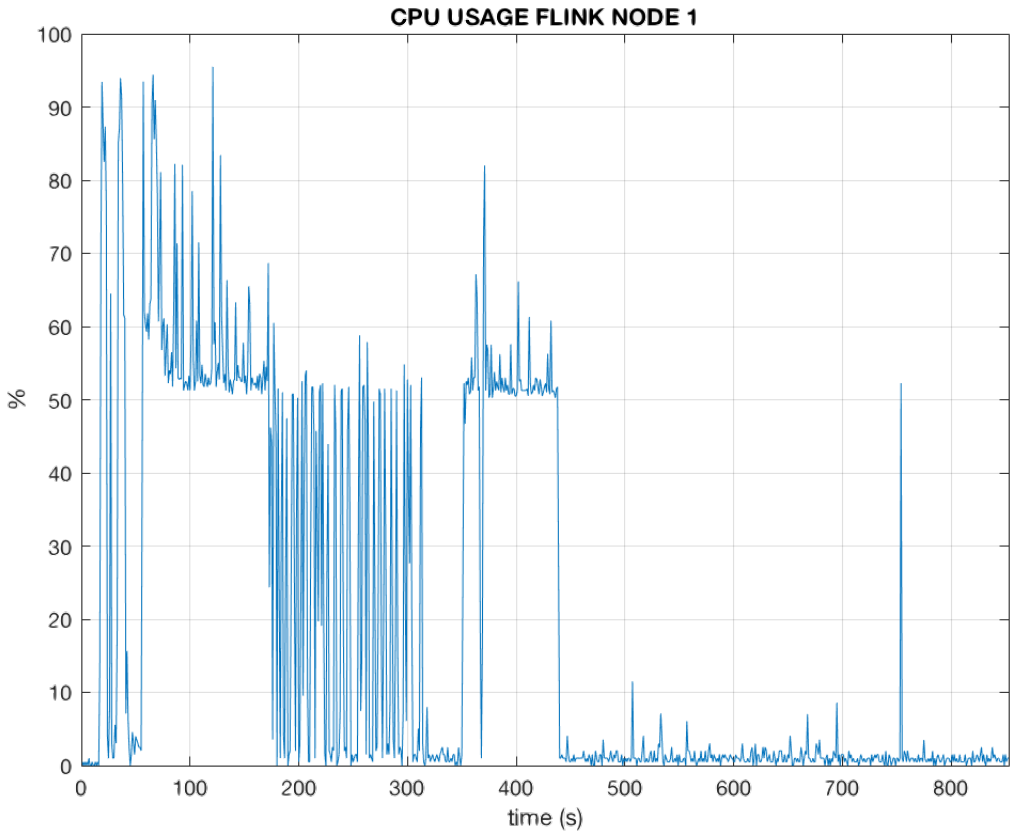

Figure 6.4: *The CPU usage for the node* `giordano-2-2-100-1` *which hosts both Job Manager and Task Manager*

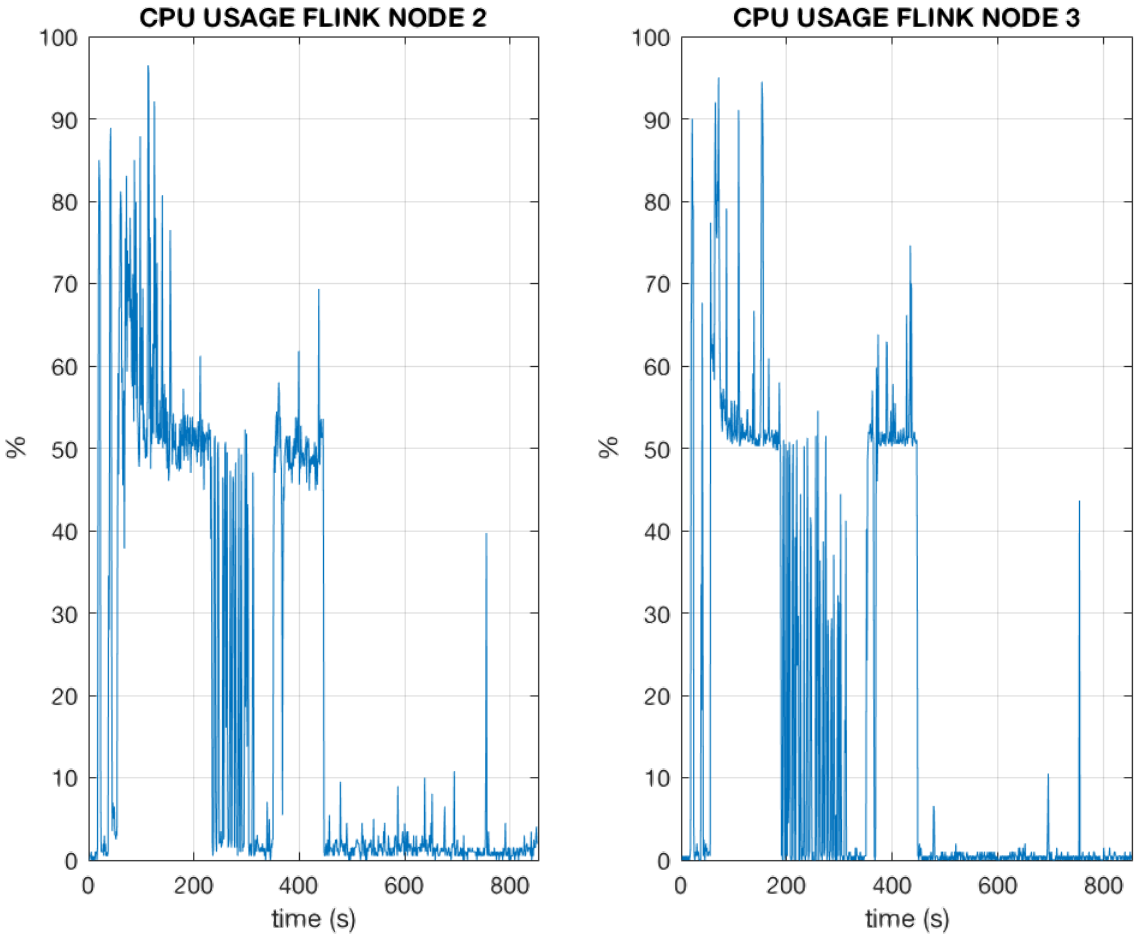

Figure 6.5: *The CPU usage for the other Flink nodes.*

The Apache Kafka broker was deployed on a node equipped with a single core CPU, 4 GB of RAM and a 200 GB of mass storage. The broker provides 8 topics and each one is divided in 10 partitions. Having more partitions represents a performance boost when there are many consumers which read from the same topic; however, in this case it is not relevant since we have just one subject involved in the system (i.e. one Raspberry) and the established partition strategy assigns a single partition for each individual. In Figure 6.6 the CPU percentage usage is represented; moreover in Figure 6.7 a view of RAM usage percentage and disk space depletion is presented.

As already said in the chapter 4 Kafka is one of the most efficient Big Data technology. Since that, the extremely low resources' usage annotated about CPU is unsurprising: a throughput of around 400 KB/s is largely bearable by a broker which, considering the specific machine characteristics, is able to handle a rate of several MB/s and millions of concurrent writes without effort. It should be noted that the throughput of the test depends on the number of Raspberrys involved in the system: we have just one Raspberry so any stress test on the Kafka broker is limited by this constraint. Based on the results we can estimate that even this unusual low Kafka configuration can handle several Raspberrys easily. As we can observe, the CPU usage remains averagely on a low percentage except for the spikes probably due to a network congestion in the instants immediately precedents (this is confirmed by the CPU effort below the 1% at the corresponding seconds) which caused a long queue of messages to handle.

It is interesting to note that the RAM usage remains always on a fixed level and the ingested data stream essentially does not affect the memory employment: it is due to the Kafka engine which stores incoming data directly in the mass storage rather than do it on RAM. Kafka writes always sequentially hence it benefits of the sequential access on disk which are often faster than random access in memory [58]. Moreover, since data on Kafka are almost never deleted, the filesystem is not fragmented and reads are executed sequentially too.

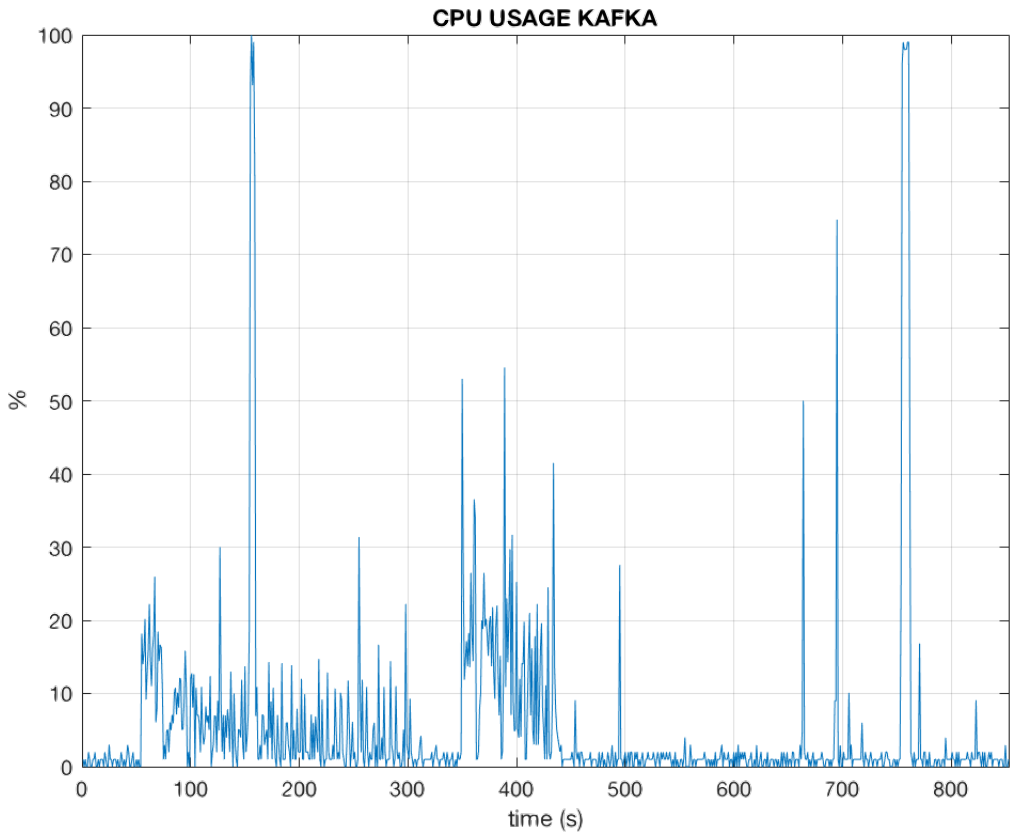

Figure 6.6: *The Kafka CPU usage. On the y-axe the usage percentage while on the x-axe the time is expressed in seconds.*

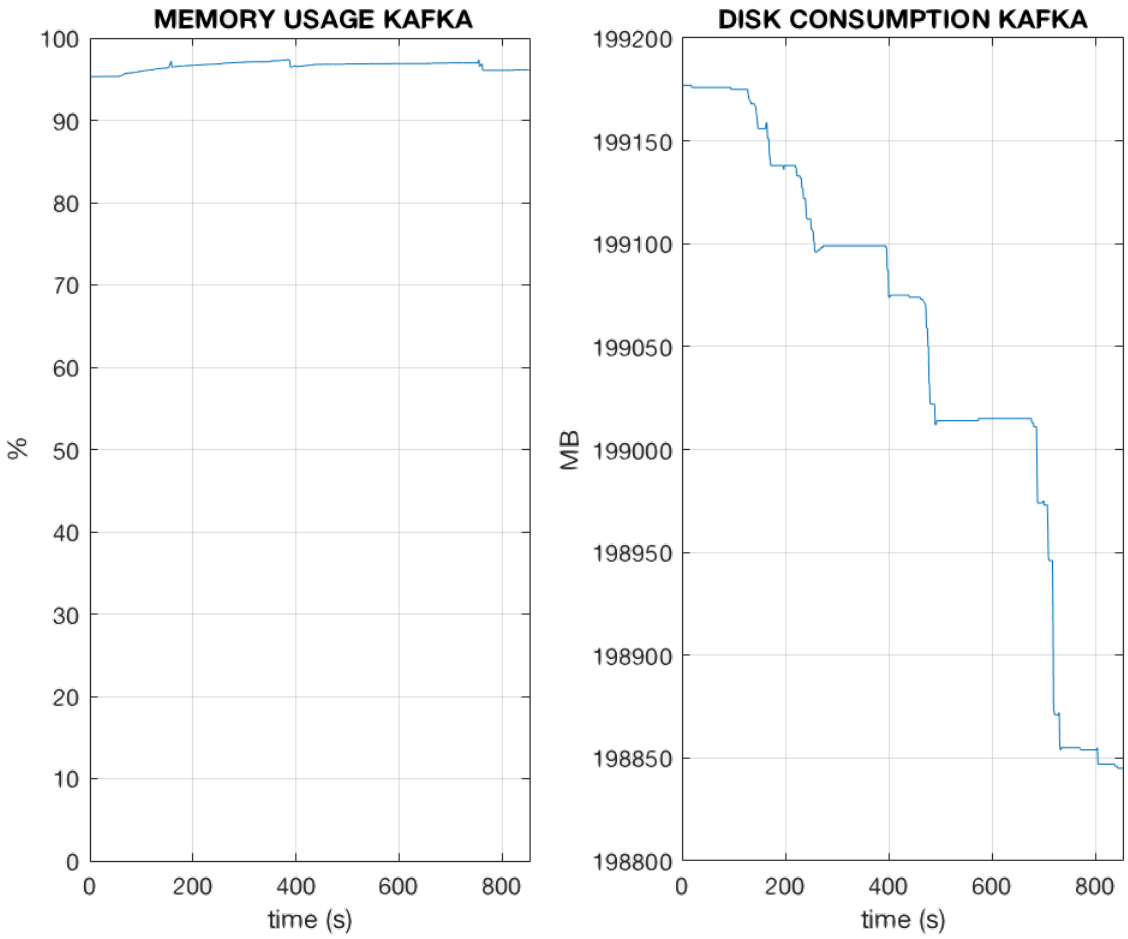

Figure 6.7: *The RAM memory employment (on the left) and the disk space depletion (on the right). On the y-axe of the rightmost figure the disk amount is expressed in MB while on the x-axe the time is expressed in seconds.*

The node's disk space depletion also was obtained using `sar` command. The total disk space is 200 GB and the graph in Figure 6.7 illustrates how the space drops when the data stream arrives to Kafka. The plain regions of the graph are the result of the application of a particular routine of every Linux OS: it intercepts every Kafka writes on the filesystem and builds a page cache which is flushed into the disk after a certain time or if the cache is full. The plain regions are the occurrence in which the OS holds the data in the

cache and the subsequent drops represent the flush operations. Kafka allows to force the flushing obtaining the desired behaviour but the documentation recommend to leave the default settings which seems to be the best trade-off between latency and throughput.

As consequence of this successful experiment, the same one was launched analyzing 2 sensors simultaneously. Recalling the issues related to the slots and the cluster limitations, every Task Managers were equipped with 1 slot and for each one a couple of HTM operators were deployed in order to maintain the cluster as balanced as possible. The following figures shows the performance of Flink nodes receiving 2 sensors, which corresponds to an analysis of 6 streams simultaneously. The heavy usage of CPU immediately stands out.

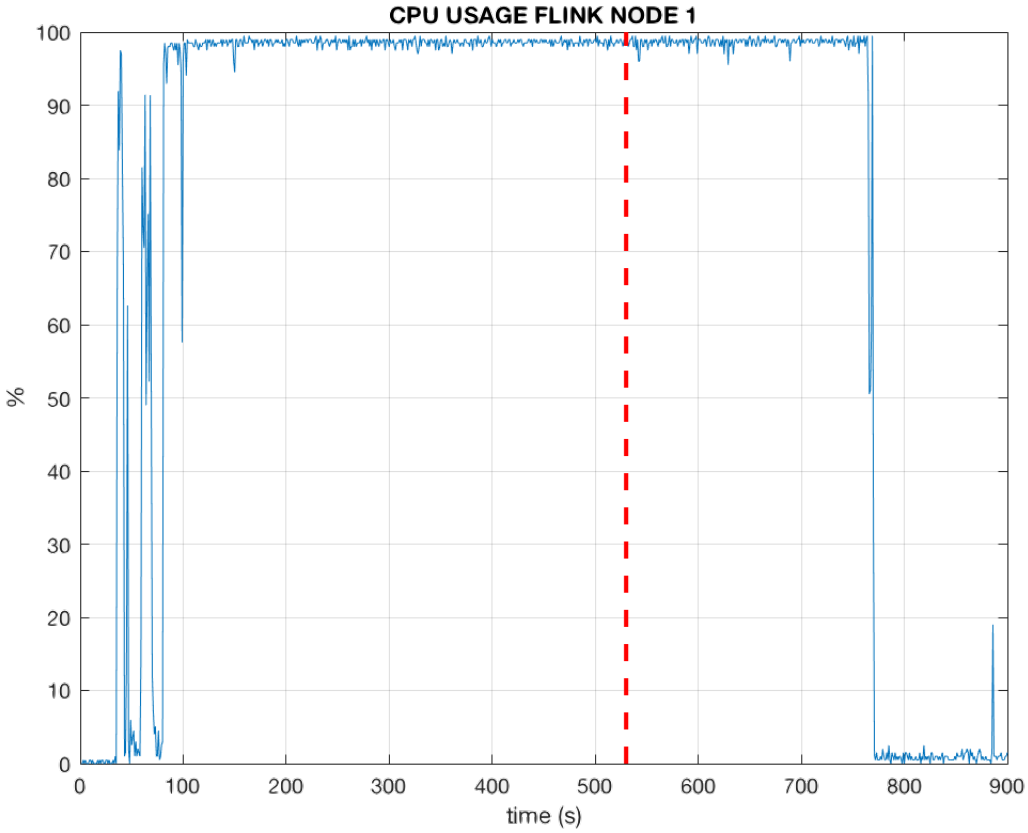

Figure 6.8: *The CPU usage of the Job/Task Manager of the Flink cluster with 6 streams ingested towards it. On x-axe the time is expressed in seconds. The dashed vertical line represents the instant when the streams end.*

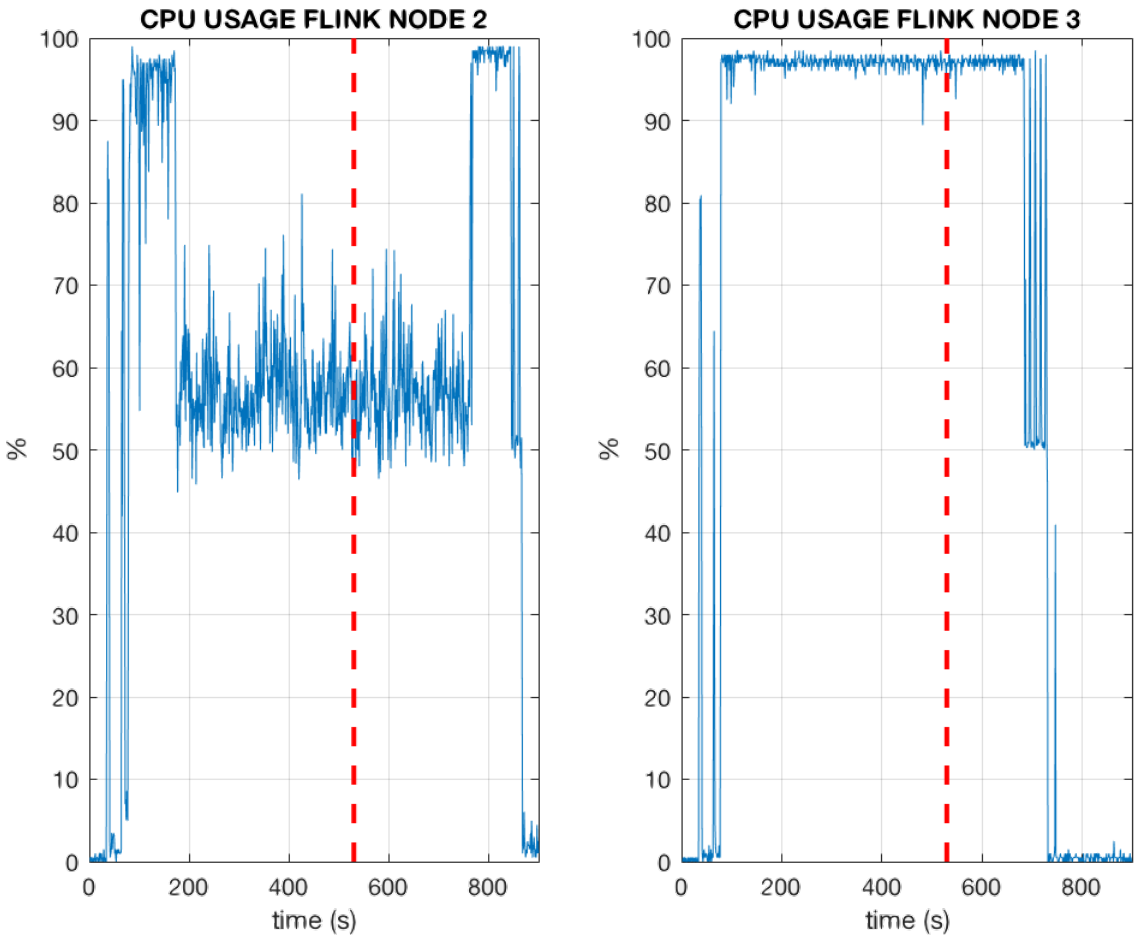

Figure 6.9: *The 2 Flink nodes and their CPU usage expressed in percentage. On x-axe the time is expressed in seconds. The dashed vertical line represents the time when the streams end.*

The figures describe the undesirable situation of a very delayed data elaboration: the vertical dashed line on the graph represents the instant of the end of stream ingestion. More or less, all the nodes were not able to perform a real-time analysis and in the worst case, i.e. the node `giordano-2-2-100-2`, the task is completed with an impressive delay of more than 300 seconds. Essentially, the data are queued in a long buffer on the network operators because they are not able to consume them enough rapidly, causing the delay. Clearly, this is not acceptable in a system where the real-time analysis

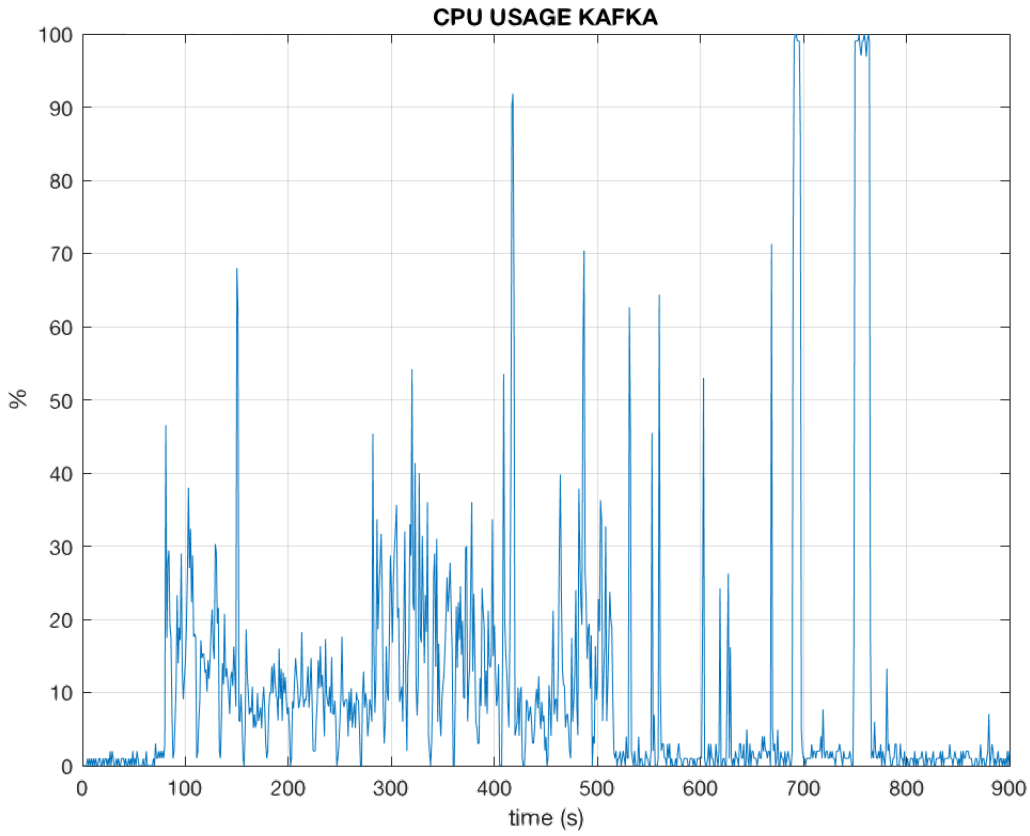

Figure 6.10: *The second experiment with an ingestion frequency of 50 Hz causes an appreciable increment in Kafka CPU usage.*

is essential: in a real medical system a delay of just few minutes could be fatal. The outcome leads to consider the ingestion frequency of 50 Hz too much high to handle more than 1 sensor with the adopted infrastructure.

In the Figure 6.17 is illustrated the Kafka CPU usage when the consumers are doubled. As we can note effectively there is a visible increment in percentage: the greater number of Kafka consumers, raised from 3 to 6, affects the CPU performance although it remains on a low level.

Apache Cassandra, for technology and node configuration (dual-core CPU, 4 GB RAM and a disk of 1 TB) is able to handle an heavier load compared to the amount ingested from Flink. On the other hand, Cassandra is placed

downstream respect to the entire system hence it is obliged to face the limitations imposed by the Flink's cluster. About memory consumption, it is around 90% even in idle state while under the load it raises from 87% to 91% with an increase of only few points in both experiments: this suggest that the effort due to the load is really low with the provided throughput. The CPU usage instead is illustrated in the figure below:

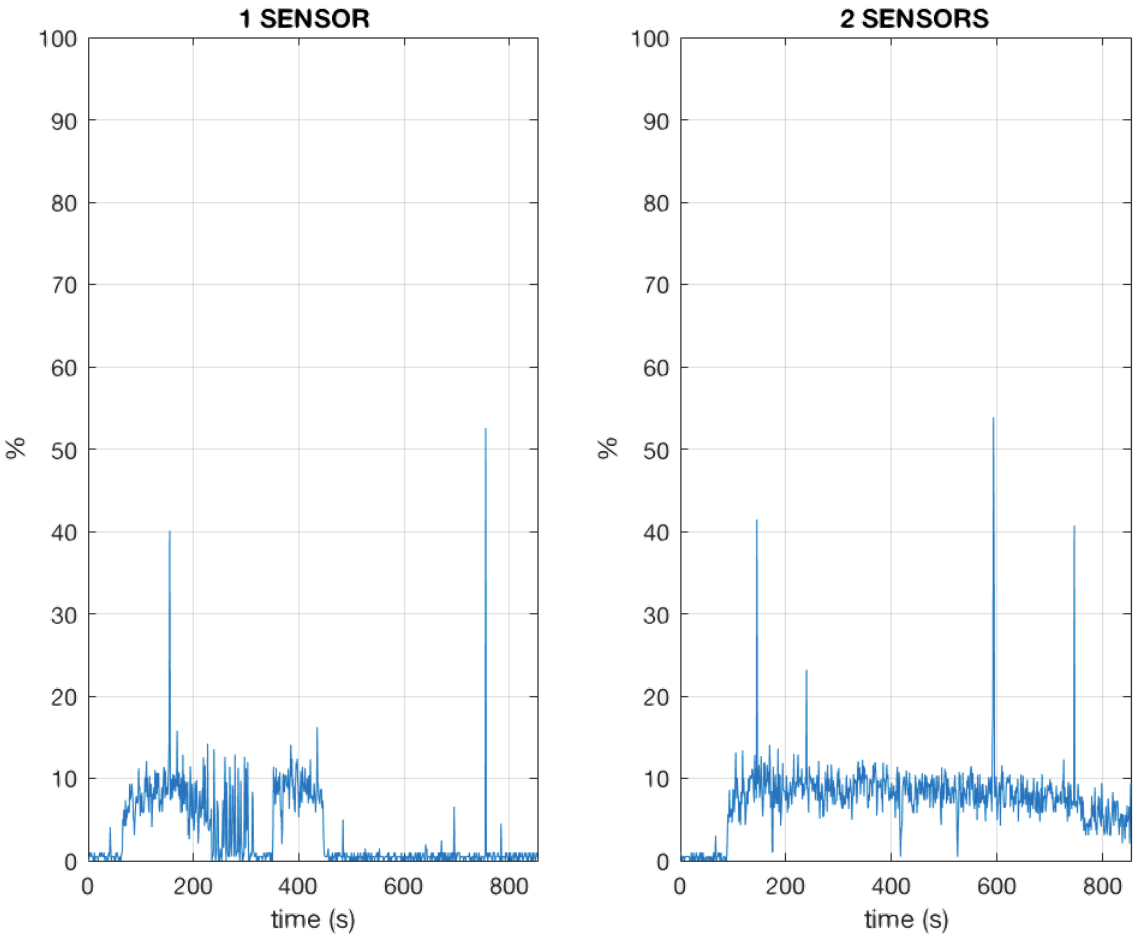

Figure 6.11: *On the left the CPU usage for the Cassandra node in the first experiment. On the other side the second one is showed. On x-axe the time is expressed in seconds.*

The graph put in evidence a low CPU usage, which remains averagely

under the 20% in both cases. This outcomes is due the cut operated by Flink to the throughput. It should be considered that in the second experiment, with the highest load for Cassandra, the throughput is nearly 300 KB/s whereas without Flink's limitations it would be at least 1.2 MB/s: these values represent both a load widely bearable by Cassandra. Someone could be note that the throughput of data destined to Cassandra is greater than the one registered on Raspberry: this is not strange if we think that Kafka, for instance, gets a stream of data where the 3 "directions" values are packed in one (since usually sensors transmit these data jointly in a single message) while Cassandra receives the streams separated, one for each "direction" analyzed by Flink; moreover, if a record appears to be anomalous it will be recorded twice in two different tables producing an higher throughput. Comparing the graphs in Figure 6.11 we can note there are no substantial differences in CPU usage percentage except for the duration of the task, which is longer in the second case.

The table in Table 6.5 highlights an interesting aspect about memory usage about Flink. The memory average percentage stands on very high values in both tests: this outcome is counterintuitive because the latter experiment doubles up the operators and it should be greedier than the first one. This confirms that Apache Flink gets practically all the memory allocated for it without consider if this memory will be employed or not.

Finally, a statistic about performance with an ingestion frequency of 50 Hz and using a raising number of sensors is provided in the following histograms. Except for Flink, all the other systems present great performances with the designed system and it is interesting finding out their limits. In order to do it many simulations were done using a number of sensors between 2 and 32 for the Raspberry and Kafka and between 2 and 64 for Cassandra. The

AVERAGE MEMORY CONSUMPTION (%)		
NODE	**EXPERIMENT #1**	**EXPERIMENT #2**
Giordano-2-2-100-1	94.73	95.90
Giordano-2-2-100-2	93.12	95.00
Giordano-2-2-100-3	93.47	93.56

Table 6.5: *RAM memory consumption expressed in percentage respect to the total amount available (2 GB).*

Raspberry showed signs of elaboration issues with a number of sensors greater than 32: in fact, the coming messages were computed lately and many of them were dropped. Anyway, handling 32 sensors is a good result because it corresponds to an input throughput of 181 KB/s and to an output throughput of 1.6 MB/s. Also the Kafka results are very good: the test with 32 producers can simulate a scenario with 4 concurrent Raspberry. Unfortunately, due to the limit reached by the Raspberry, a more challenging test cannot be executed. About Cassandra, a separate Flink program which simulates just the stream dispatching of the original application was deployed in order to define better the performance of the database. To improve readability some graphs present a re-sized scale on y-axe. The Figure 6.12 describes, from the highest to the lowest, the average CPU and RAM memory employment for the Raspberry Pi, the Kafka broker and the Cassandra database.

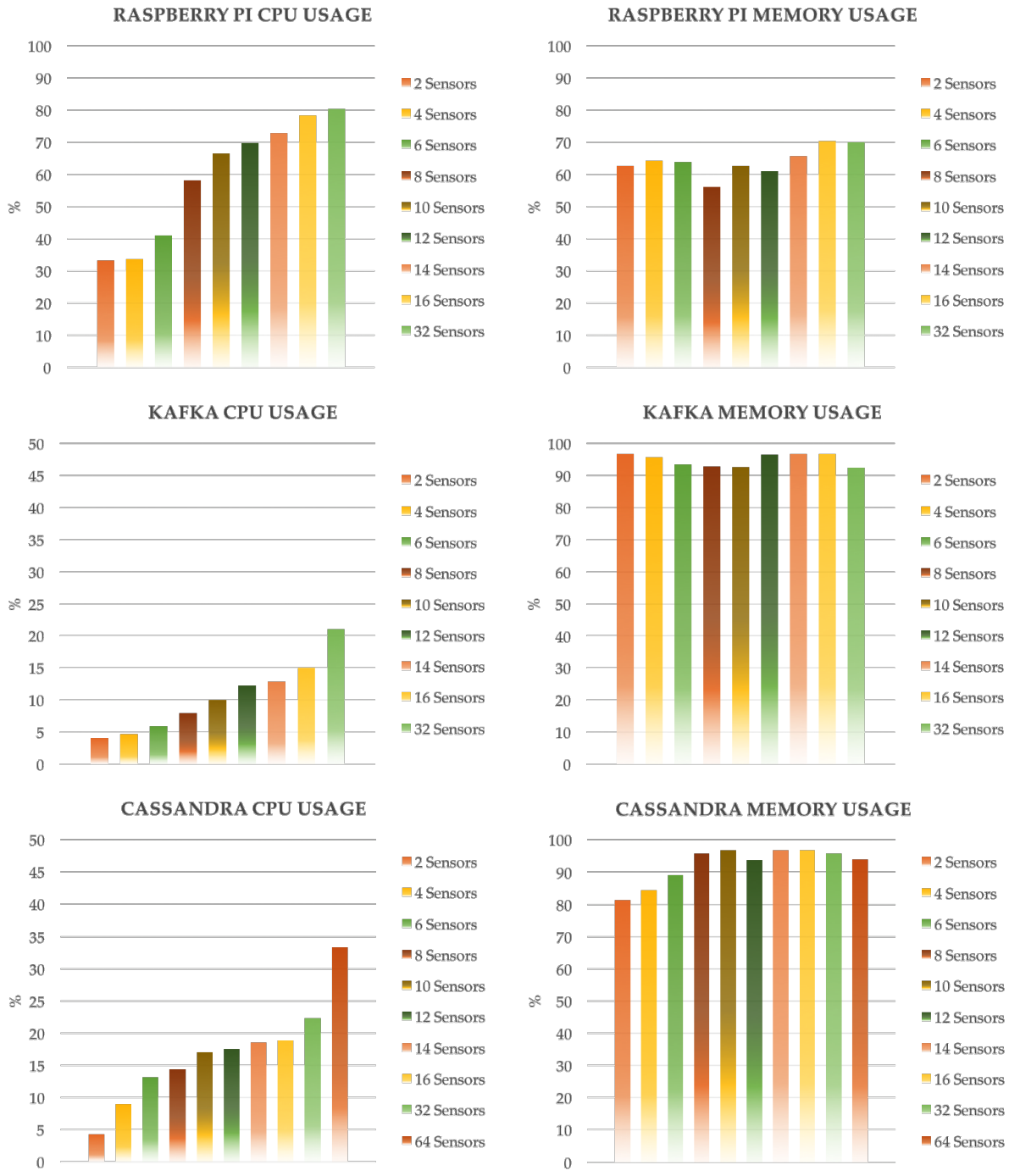

Figure 6.12: *The bars represent the average values registered experimenting with an increasing number of sensors.*

About the Raspberry the results were very good. We can note as the CPU usage presents an slight and gradual increment of few percentage points also among the two higher cases (i.e. 16 and 32 sensors). Today no application uses so many wearables on a single person but the result suggests that in future an embedded system like the Raspberry could perform the task very well: that is confirmed also by the memory graph which highlights as the gap between the tests consists only of 8 percentage points.

Actually, an important issue was found during the test: *Node-Red* is deployed as a single-core process so after a certain limit, discovered using 12 sensors, it shows many difficulties to handle the throughput generated. In order to solve the issue and to continue the test *Node-Red* was deactivated and all its functions were developed in a separate group of Python scripts which perform the same task: they act as MQTT consumers, enrich messages and dispatch data to Kafka with the appropriate producers. This solution represents a multi-core process and it exploits a pool of thread to manage separately reception and message dispatching. Nevertheless, the results obtained with *Node-Red* are still valuable: with a load of 8 sensors the framework is completely able to execute the task efficiently.

In the Kafka CPU graph is clear that for the broker there are no substantial differences using more or less sensors at least when this gap is small. Looking at the histogram in Figure 6.12 we can note that only when the number of sensors explodes Kafka shows a significant increment in CPU effort. The project provides 8 sensors each individual: this test demonstrates that using 4 person (i.e. 32 sensors) Kafka is able to own the scenario also with a poor node configuration. The Kafka memory consumption is always high and its changes are negligible: it depends by the OS and its routines to handle I/O operations. Kafka effectively assigns data to write on disks to

the operating system but the latter is the one that decides when and how to do it accordingly to the policies exposed previously.

Apache Cassandra is located downstream to Flink, so the most of the experiments on it were executed using a simplified version original program: it dispatch data operating a dummy analysis. In this way an outlook of the Cassandra performance was obtained. As we can observe, Cassandra exposes a boost in CPU usage when the number of sensors raises from 32 to 64: this last result corresponds to the case of 8 person analyzed simultaneously. Despite that, the CPU usage percentage remains still under the 35%. Based on the experimental results, the provided configuration is able to handle far more sensors but unfortunately a further test with 128 sensors was not completed because the Flink's Job Manager has not enough heap memory to handle a so large number of operators.

Regrettably, due to the limited resources available it was impossible to do the same comparison just seen with the Flink cluster. The HTM algorithm is a too heavy task: with the preferred network settings, exposed in section 6.3.2, every experiments which have analyzed 3 or more sensor streams led to node's crash. It seems that *flink-htm* operators used to store a lot of intermediate data in order to perform an accurate anomaly detection.

Ingestion frequency: 25 Hz

In order to understand how much the ingestion frequency penalizes the adopted infrastructure a new experiment was performed. We wanted to check if halving the ingestion frequency, so sending 25 messages per second instead of 50, it is possible to employ more than a single sensor. No one of the other parameters were changed. In the Table:6.6 the input and output throughputs generated with an ingestion frequency of 25 Hz are showed. It

	INPUT	OUTPUT
Raspberry	23 KB/s	200 KB/s
Kafka	200 KB/s	150 KB/s
Flink	150 KB/s	$>$ 150 KB/s
Cassandra	$>$ 150 KB/s	n.d

Table 6.6: *Input and output produced throughputs with an ingestion frequency of 25 Hz.*

should be noted that the values related to Flink, Cassandra and Kafka for the output throughput are affected by the limitations provided by the anomaly detection: within them, at most 2 sensors can be evaluated simultaneously.

Despite an experiment with 1 sensor was executed also in this case, no graphs about that will be reported because they do not represent an interesting case of study: the 3 streams were elaborated in real-time with a significant decrease of efforts in term of CPU. About memory, as already seen, Flink gets all the available amount also if it is not necessary. On the other hand there were no reasons to justify a deterioration of performance applying a lighter load to the system. Instead, the core of this second attempt is to verify if, halving the ingestion frequency, the Flink application is able to analyze 2 sensors simultaneously.

The Figure 6.13 shows the CPU and memory performance on Raspberry with an ingestion frequency of 25 Hz:

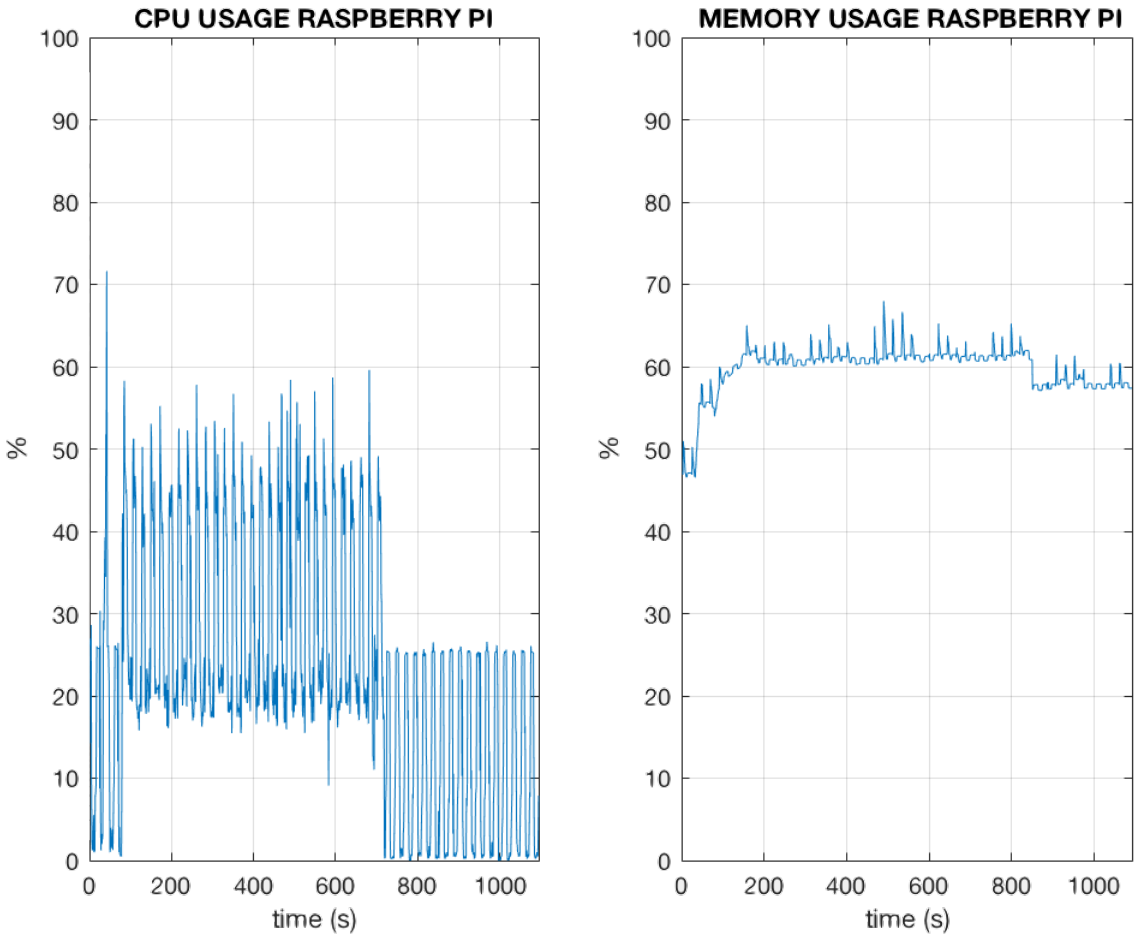

Figure 6.13: *The CPU and memory usage on Raspberry Pi with an ingestion rate of 25 Hz.*

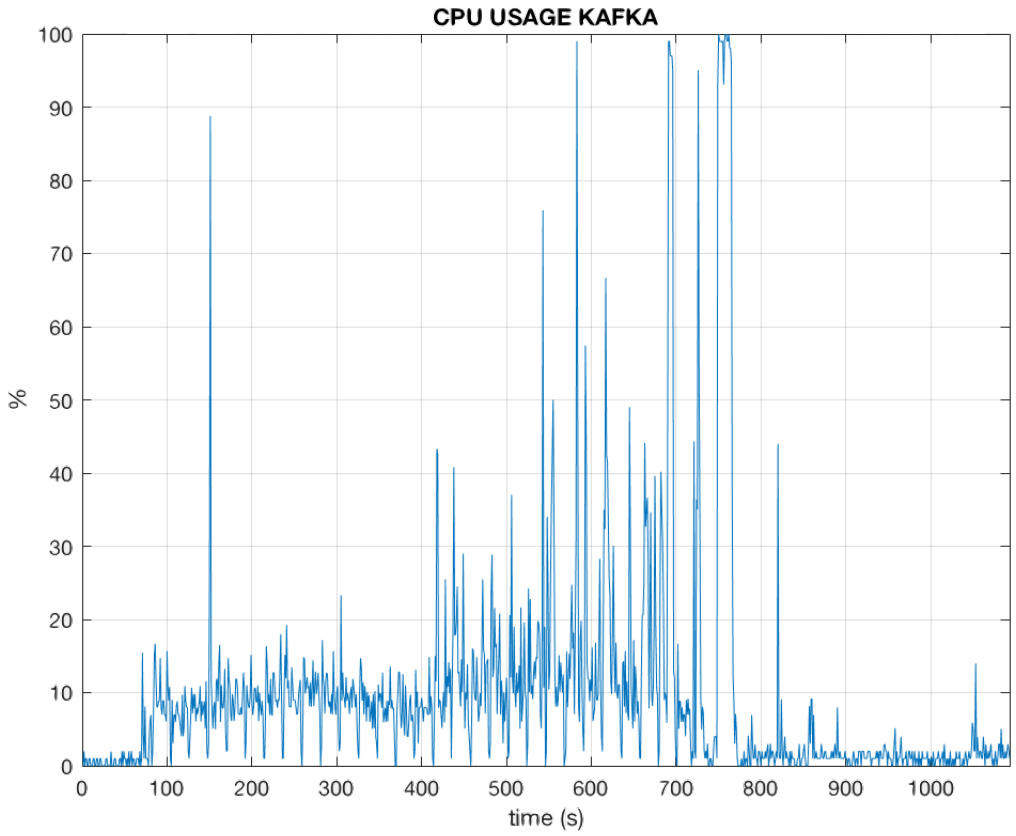

Figure 6.14: *The CPU graph about Apache Kafka in the second experiments.*

The CPU effort shows only negligible differences while the memory consumption is characterized by a decrease of about 6-7 percentage points. The Kafka graph in Figure 6.14 about CPU shows an average effort essentially equal to the previous one but with a more stable trend thanks to the lesser ingestion frequency which avoids sudden back-pressures due to potential network congestions. The central part of the experiment is represented in Figure 6.15 and Figure 6.16

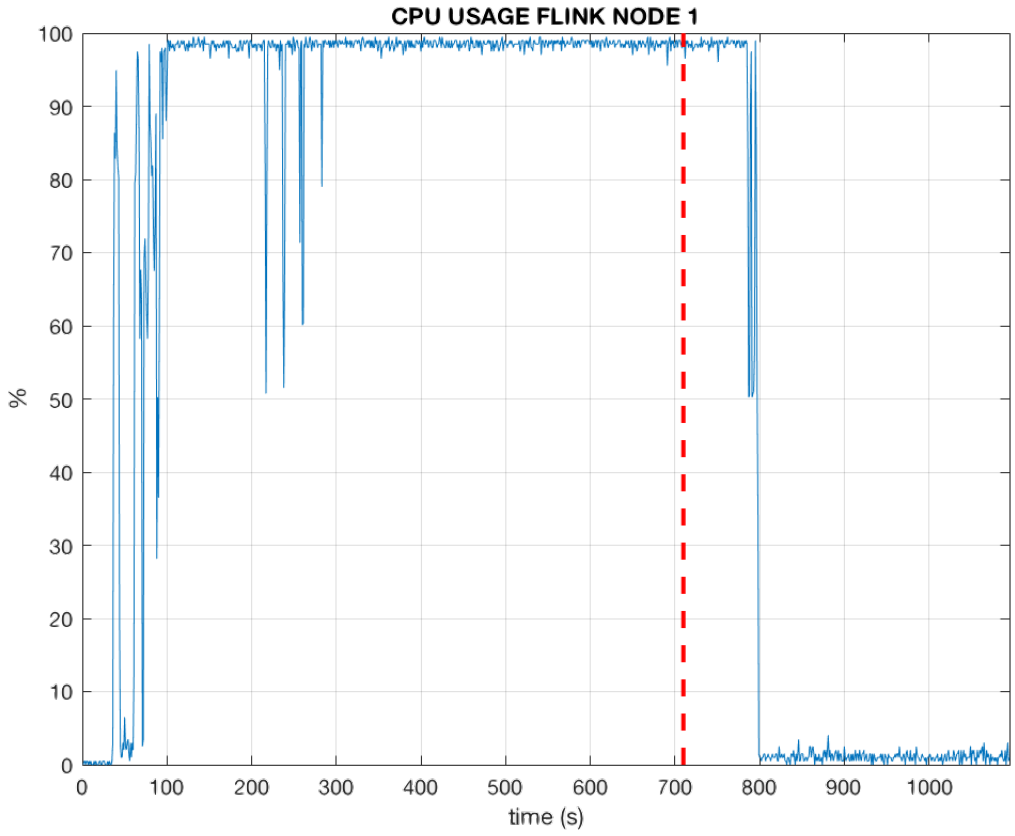

Figure 6.15: *The CPU graph of the first node of the Flink cluster. The dashed line represents the instant when the stream ends.*

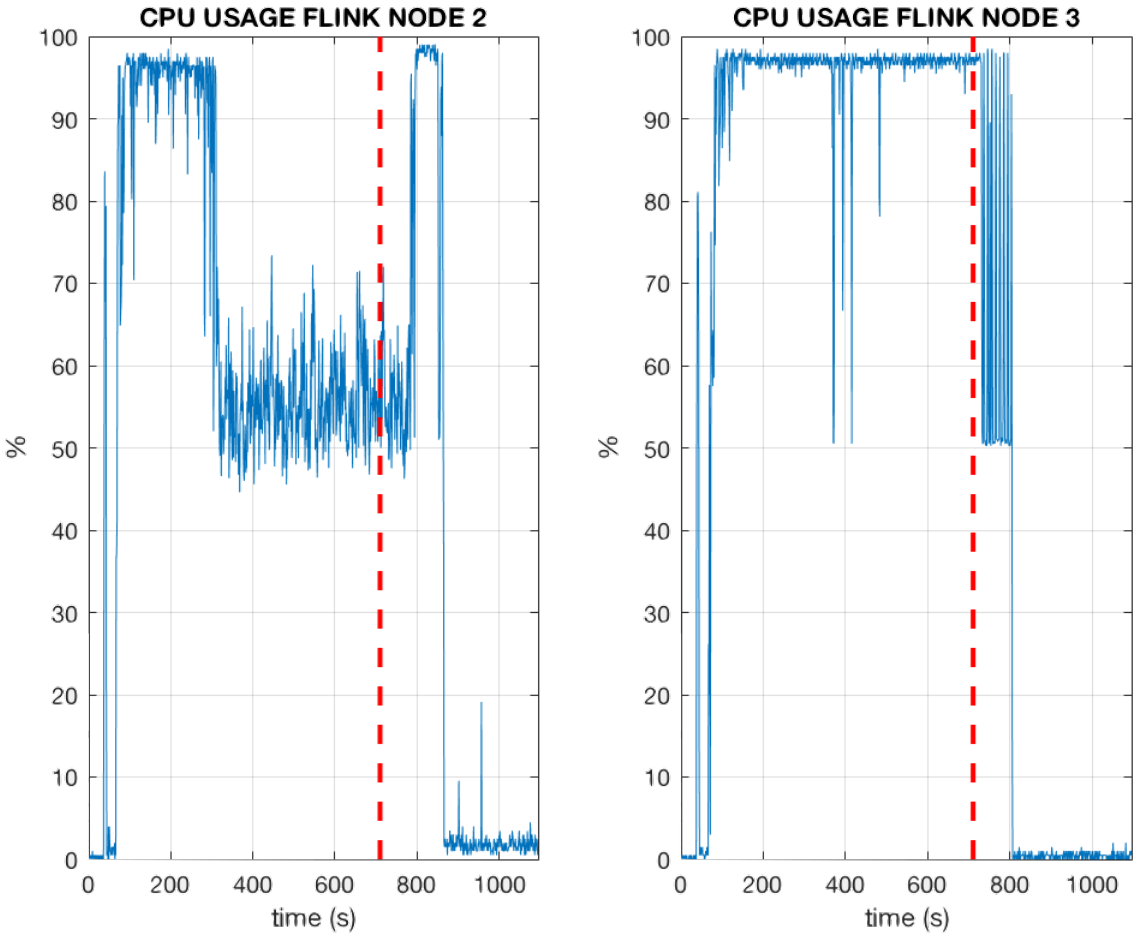

Figure 6.16: *The CPU graph about the other two Task Managers. The dashed line represents the instant when the stream ends.*

Using an ingestion frequency of 50 Hz we observed an heavy CPU effort to complete the task: despite it, the task was completed with an unacceptable delay. Halving the ingestion frequency the CPU percentage of utilize is still very high and touches the 99%. This is symptomatic of an heavy task like the simultaneous elaboration of 6 six streams. Unfortunately even in this case the job is not completed on time and halving the ingestion rate seems not to be enough to analyze more than one sensor: anyway, the delay was very reduced compared to the previous case. It suggests that probably analyzing

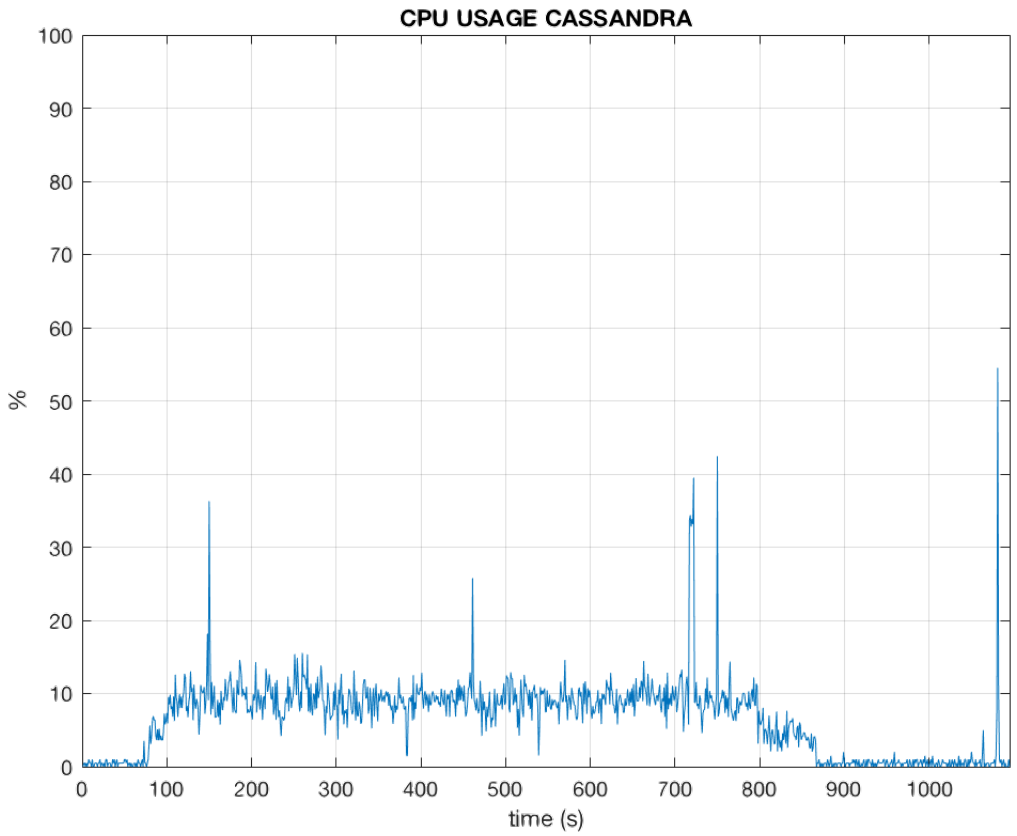

Figure 6.17: *The CPU graph about the Cassandra database within the second experiment.*

more than 1 sensor at time accepting is possible if we accept a trade off with the sensor reading frequency. Finally, a predictable decreasing CPU utilization of Apache Cassandra is displayed in the figure below while its memory consumption is always stable around an interval of 90-95%.

Ingestion frequency: 15 Hz

The last one experiment cuts the original ingestion frequency of 70% in order to verify if it is possible analyze more than 1 sensor with the adopted infrastructure choosing a lower throughput. In the Table:6.6 the input and output throughputs generated with an ingestion frequency of 25 Hz are showed. It

	INPUT	OUTPUT
Raspberry	14 KB/s	120 KB/s
Kafka	120 KB/s	90 KB/s
Flink	90 KB/s	$>$ 90 KB/s
Cassandra	$>$ 90 KB/s	n.d

Table 6.7: *Input and output produced throughputs with an ingestion frequency of 15 Hz.*

should be noted that the values related to Flink, Cassandra and Kafka for the output throughput are affected by the limitations provided by the anomaly detection: within them, at most 2 sensors can be evaluated simultaneously.

The Figure 6.18 - Figure 6.20 depict the behaviour of Raspberry, Kafka and Cassandra with the new ingestion frequency: they decrease slightly the efforts or maintain same performances of the previous test. The most interesting graphs are Figure 6.21 and Figure 6.22 which display the CPU utilization for Flink's nodes: the percentage is still very high but finally the job is elaborated in real-time, the dashed lines now correspond exactly to the end of the computation.

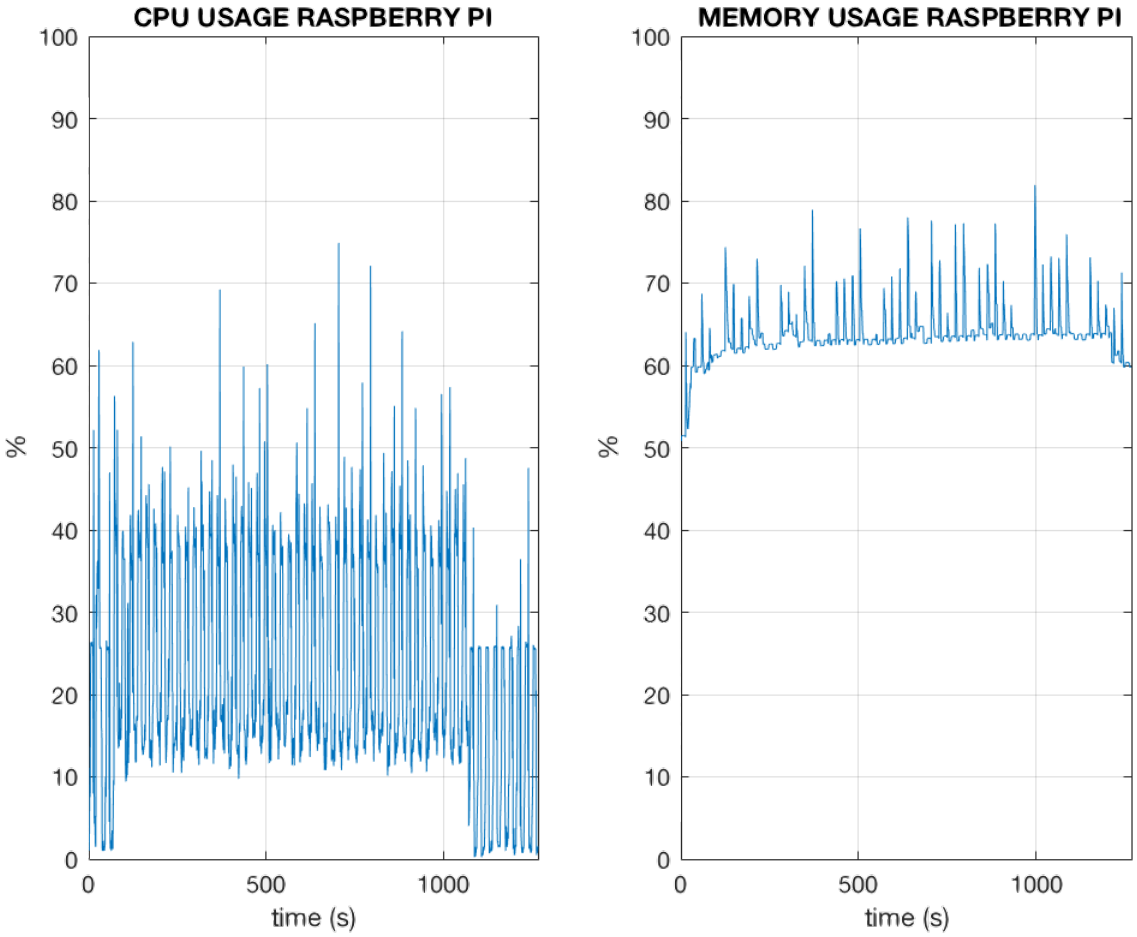

Figure 6.18: *The graph about Raspberry's CPU and memory consumption with an ingestion rate of 15 Hz.*

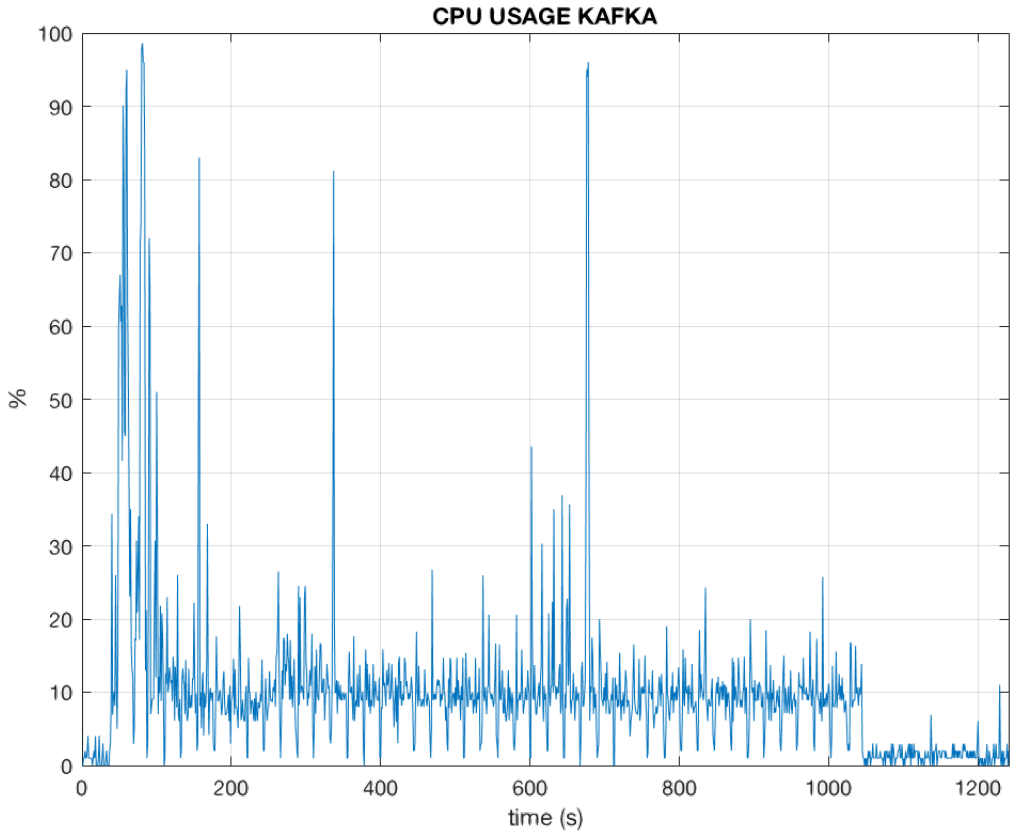

Figure 6.19: *The graph about Kafka CPU usage with an ingestion rate of 15 Hz.*

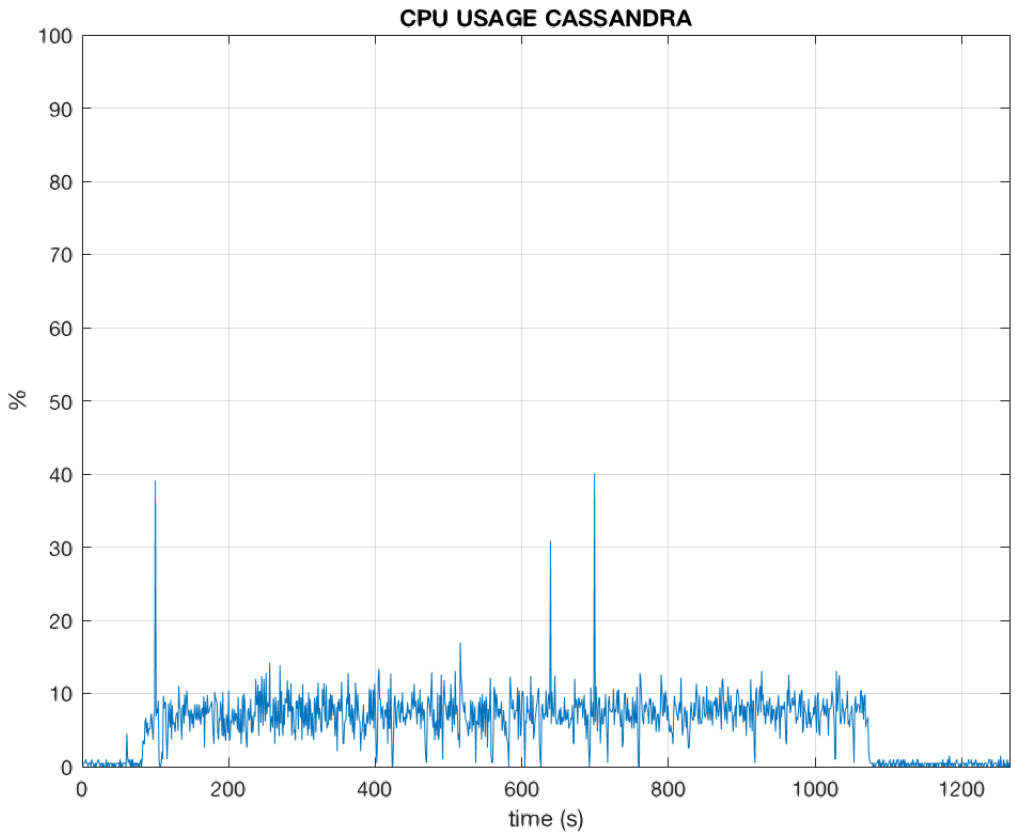

Figure 6.20: *The CPU graph about the Cassandra database within the third experiment.*

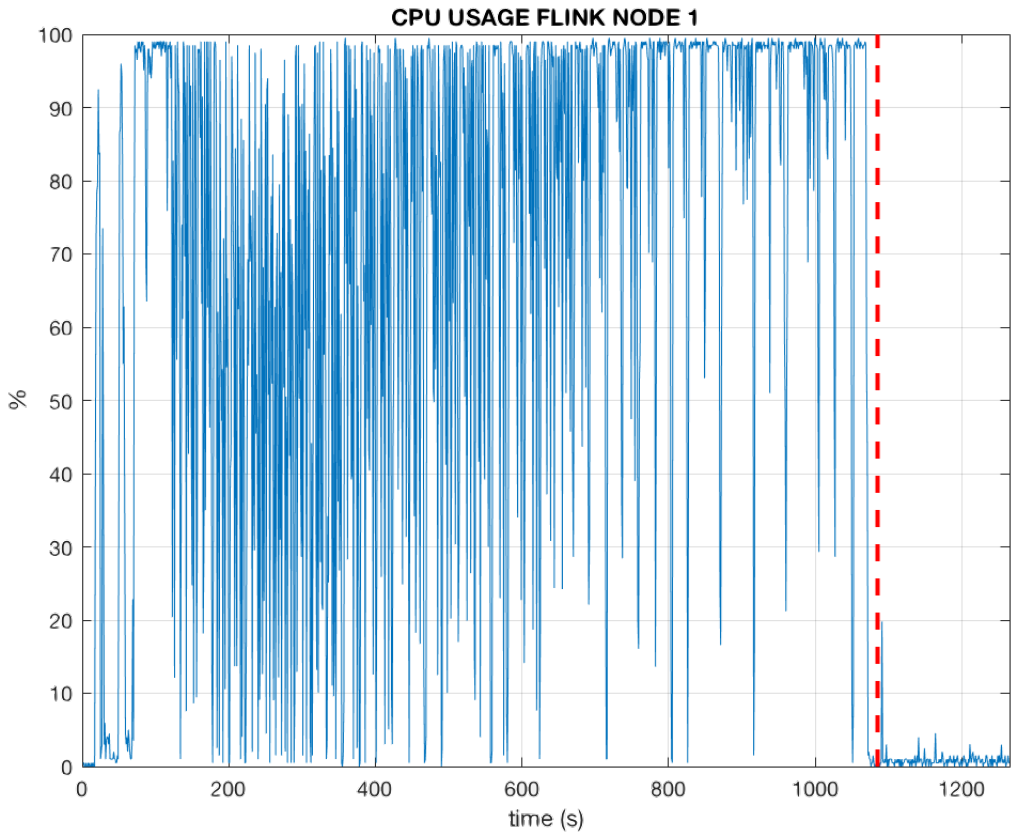

Figure 6.21: *The usage caused by the job on the first node of the Flink cluster, finally computed in real-time.*

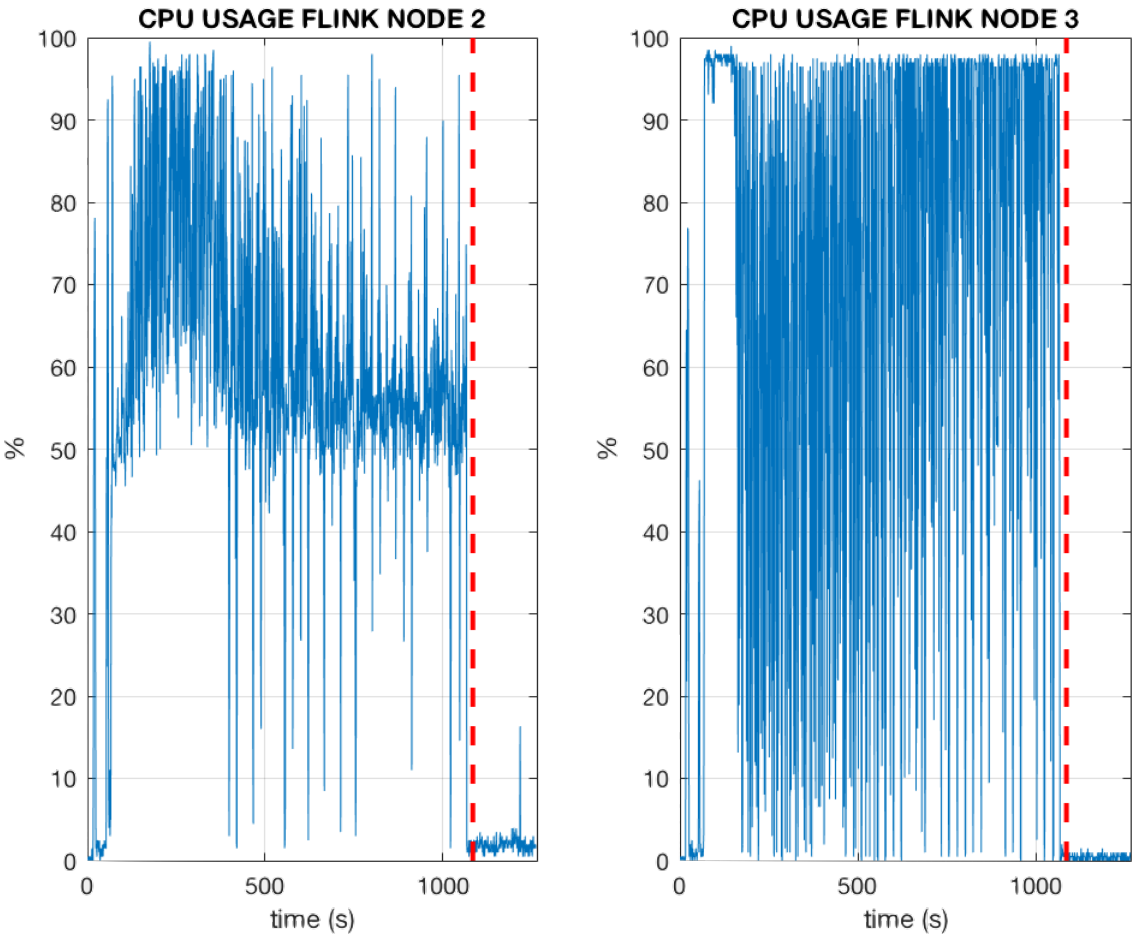

Figure 6.22: *The usage caused by the job on the 2 last node of the Flink cluster, finally computed in real-time.*

The confused trend of the last graphs is due to the low ingestion frequency which reduces the throughput towards Flink from 300 KB/s to 90 KB/s. The lower throughput causes an increment of oscillations in CPU graph respect to the case of the first experiment: thanks to the lower frequency the network operators have enough time to detect anomalies before the next messages arrive that oscillations are caused by periods of intensive computation alternated with periods of quite "silence".

AVERAGE DATA LOSS (%)		
INGESTION FREQ. (Hz)	**3 STREAMS**	**6 STREAMS**
50	0.302 ± 0.02	0.366 ± 0.03
25	0.284 ± 0.01	0.326 ± 0.1
15	0.273 ± 0.01	0.293 ± 0.07

Table 6.8: *The average data loss in the system.*

Data loss

During the processing data sent and partially elaborated by the system can be lost for many reasons. The most probable is a network lack but also a dropping by Flink can occurs. As explained in section 5.5.2 if a message arrives with a lateness greater than a fixed value (800 ms in the implemented case) it is dropped because it cannot be more added to the stream to analyze.

Some statistics about the average data loss are depicted in the following table. The experiments were executed with the 3 different ingestion rate which are employed in the test and for the 2 Flink applications with 3 and 6 data streams simultaneously analyzed. The test was repeated 3 times for each value. Reasonably, the percentage loss decreases with the lower ingestion rates and analyzing less sensors, thank's to the lower effort required to run the application.

6.3.2 HTM results

At the present the Numenta HTM algorithms can be implemented using several object-oriented languages, in particular Python, C++ and Java while Apache Flink employs Java and Scala. As consequence *flink-htm* library was specifically built-up using these last two languages. Due to its simplicity and

spread, in this project Java 1.8 version was preferred.

The first step to execute the anomaly detection algorithm is building an HTM network. Creating it from scratch could be a very complex task due to a lot of parameters to set and because it requires a strong knowledge of neural networks and machine learning topics. The construction of a network is strongly related to the data it has to analyze. Actually the particular network employed in this project was built following a standard template which is considered generally adequate to the 90% of cases by the Numenta engineers. The network template was refined and improved in accordance with HTM documentations and community's tips in order to make it more adherent to the provided data patterns. Indeed, to obtain the best results many attempts were performed changing options and parameters: finally, the set showed in the Figure 6.23 was chosen as the best compromise between accuracy and velocity of execution. The class `Harness.AnomalyNetwork` represents the employed HTM network.

One of the most important tuned parameters is the network resolution: if it is low it produces a more accurate output although it will be paid with a greater delay to perform the analysis. The Table:6.9 sums up the time required to compute anomaly degree for each record testing 4 different networks with a decreasing resolution.

We can intend the resolution as a measurement of how much the values differs each other: if they are very close a more fine-grained resolution will be required to detect anomalies, otherwise just few anomalies will be found. Going down into details, the network creates a set of bins with a length fixed to the resolution value: the bins represent a sort of data quantization. The algorithm computes a prediction value with the equations listed in section 4.4.1 and puts the obtained value in the correspondent bin correspondent.

```
public static class AnomalyNetwork implements org.numenta.nupic.flink.streaming.api.NetworkFactory<Harness.KafkaRecord>{

    private static final long serialVersionUID = -396782890555756963L;


    public Network createNetwork(Object key) {
        Parameters p = getParameters();
        p = p.union(getEncoderParams());

        Map<String, Object> params = new HashMap<>();
        params.put(KEY_MODE, Anomaly.Mode.PURE);

        Network n = Network.create("Network", p)
                .add(Network.createRegion("r1")
                        .add(Network.createLayer("1", p)
                                .alterParameter(Parameters.KEY.AUTO_CLASSIFY, Boolean.TRUE)
                                .alterParameter(Parameters.KEY.INFERRED_FIELDS, getInferredFieldsMaps("value",CLAClassifier.class))
                                .add(Anomaly.create(params))
                                .add(new TemporalMemory())
                                .add(new SpatialPooler())
                                .add(MultiEncoder.builder().name("").build())));
        return n;
    }


    public static Map<String, Map<String, Object>> getSensorFieldEncodingMap() {
        Map<String, Map<String, Object>> fieldEncodings = setupMap(
            null,
            2000,
            141,
            0, 0, 0, 0.3, Boolean.FALSE, Boolean.FALSE, null,
            "value", "float", "RandomDistributedScalarEncoder");

    return fieldEncodings;
}

   public static Parameters getEncoderParams() {
       Map<String, Map<String, Object>> fieldEncodings = getSensorFieldEncodingMap();

       Parameters p = Parameters.getEncoderDefaultParameters();
       p.set(Parameters.KEY.GLOBAL_INHIBITION, true);
       p.set(Parameters.KEY.COLUMN_DIMENSIONS, new int[] { 2048 });
       p.set(Parameters.KEY.CELLS_PER_COLUMN, 32);
       p.set(Parameters.KEY.NUM_ACTIVE_COLUMNS_PER_INH_AREA, 40.0);
       p.set(Parameters.KEY.POTENTIAL_PCT, 0.8);
       p.set(Parameters.KEY.SYN_PERM_CONNECTED,0.1);
       p.set(Parameters.KEY.SYN_PERM_ACTIVE_INC, 0.0001);
       p.set(Parameters.KEY.SYN_PERM_INACTIVE_DEC, 0.0005);
       p.set(Parameters.KEY.MAX_BOOST, 1.0);

       p.set(Parameters.KEY.MAX_NEW_SYNAPSE_COUNT, 20);
       p.set(Parameters.KEY.INITIAL_PERMANENCE, 0.21);
       p.set(Parameters.KEY.PERMANENCE_INCREMENT, 0.1);
       p.set(Parameters.KEY.PERMANENCE_DECREMENT, 0.1);
       p.set(Parameters.KEY.MIN_THRESHOLD, 9);
       p.set(Parameters.KEY.ACTIVATION_THRESHOLD, 12);

       p.set(Parameters.KEY.CLIP_INPUT, true);
       p.set(Parameters.KEY.FIELD_ENCODING_MAP, fieldEncodings);
```

Figure 6.23: *An abstract of the class Harness.AnomalyNetwork which represents the adopted anomaly network.*

RESOLUTION	TIME PER RECORD (ms)
0.1	25.98
0.3	16.35
0.5	15.19
0.7	15.17

Table 6.9: *Comparison of the elaboration time required to compute a single record with several network resolutions.*

At the next step the algorithm compares the "predicted" bin with the destination bin of the real value: if they are equal the prediction was good. If the the resolution was too high, the bins will be very large and many values will fall in bins which do not represent adequately the data; moreover, if there are few bins the wrong predictions will fall in the same bin of the real values leading to a false negative and returning no anomalies. The chosen network has a resolution of 0.3 because it provided the best trade-off between accuracy and computation speed.

The dataset used to represent sensor data belonging to a subject is composed of 180.000 records temporally separated from 20 milliseconds. The values considered here as example match the acceleration along the X-axe. In order to improve readability the Figure 6.25 and Figure 6.27 present only portions of the entire dataset, in particular are showed the first 20.000 and 50.000 records. First of all, we can note that the employed one is a *continuous learning* algorithm: this means the network do not use a learning phase as step to perform anomaly detection. However, it requires a time period to understand and figure out the data pattern during the processing itself: this phase take up a time which depends on how variable are the data. Indeed, in the Figure 6.26 and Figure 6.28, which represents respectively the anomalies found in the first 20.000 and 50.000 records, we can observe the presence of

a initial stage of *1* values representing strong anomalies. This is due to the initial lack of data knowledge of the network. In this test after around 6 seconds the network has a sufficient knowledge about the data pattern and it is ready to evaluate the forthcoming data.

It should be noted that HTM is a memory-based system and has not ability to "understand" data meaning, instead it evaluates repeatability and recurrence of patterns so, for instance, if the data set represents a strict widening array of natural numbers HTM anomaly detection algorithm will present a bad behaviour. HTM won't be able to predict a never seen value because it has not the ability to understand the numbers for what they represent. Otherwise it will give great results in case of recurrent values, just as in the present case, because it knows data pattern and can compute the value which has the greatest probability to appear. The capability of "understand" data usually is a prerogative of the system which use using specific formulae to learn quickly but only on particular types of patterns: HTM is probably slower but it can learn every patterns even if they are difficult to express with mathematical expressions.

Finally, HTM tends to consider more probable the arrive of a recent seen pattern rather than an elder one: the latter is forgot after a long time so HTM describe an ability to adapt itself analyzing the changes which occurr in data patterns.

Comparing respectively the pairs Figure 6.25 - Figure 6.26 and Figure 6.27 - Figure 6.28 we can observe how HTM registers high anomaly degree spike (values closer to 1 respect to 0) exactly where there are abnormal peaks in the dataset graph. It should be noted that actually HTM do not push out anomalies directly, instead it calculates an anomaly degree, constrained from 0 to 1. Selecting which values represent anomalies and which not is a task

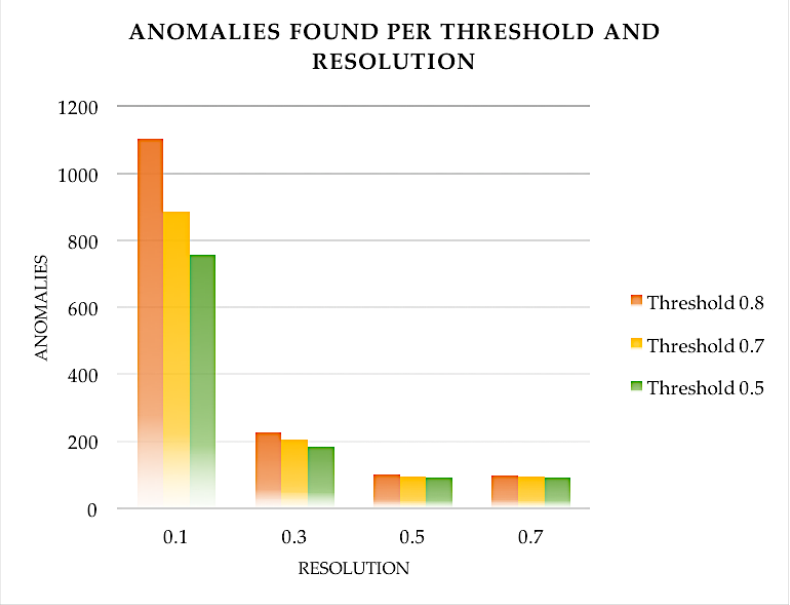

Figure 6.24: *A summary of the consequences of using a network with different resolution value.*

heavily dependent by the application. In this project many attempts were performed to find an adequate threshold to fit the data set and a value of 0.8 was considered as a good limit to distinguish between anomalies ($>=$ 0.8) and standard values ($<$ 0.8). Since that, only values with an anomaly degree greater than 0.8 were memorized in Cassandra as anomalies. Anyway, the anomaly threshold is easily changeable in `StreamingJob` class.

Furthermore, the histogram in Figure 6.24 depicts the number of anomalies found with 4 networks varying thresholds. Generally, resolution has to be chosen on data basis, in fact an high number of anomalies does not correspond necessarily to an higher accuracy because with a fine-grained resolution raises even the risk to get false positives. Given that, in this case was useless testing a resolution of 0.1 or 0.7 because it would lead respectively a too heavy and a too inaccurate task.

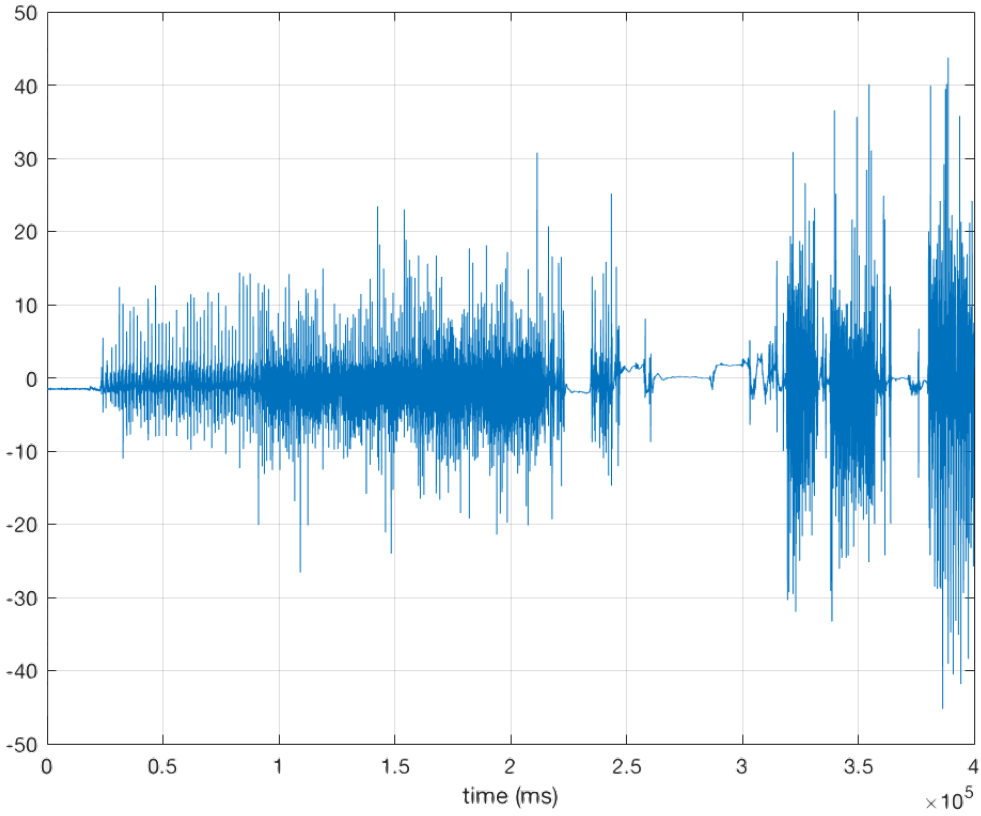

Figure 6.25: *The first 20.000 records of the dataset.*

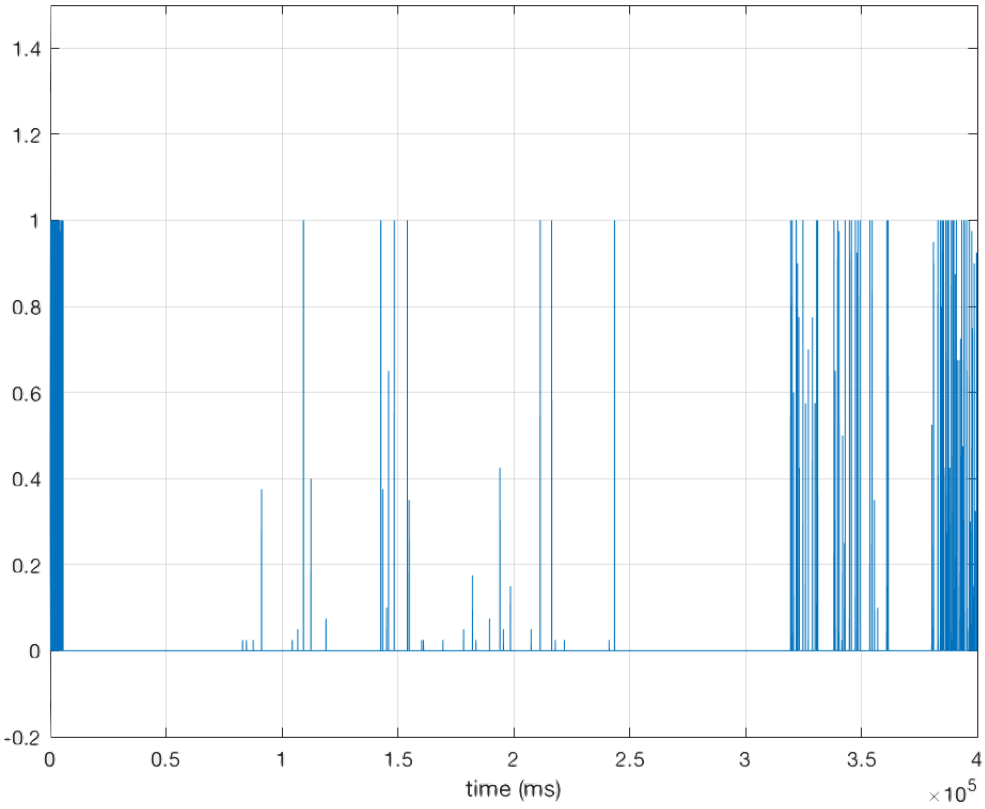

Figure 6.26: *Anomaly peaks found in the first 20.000.*

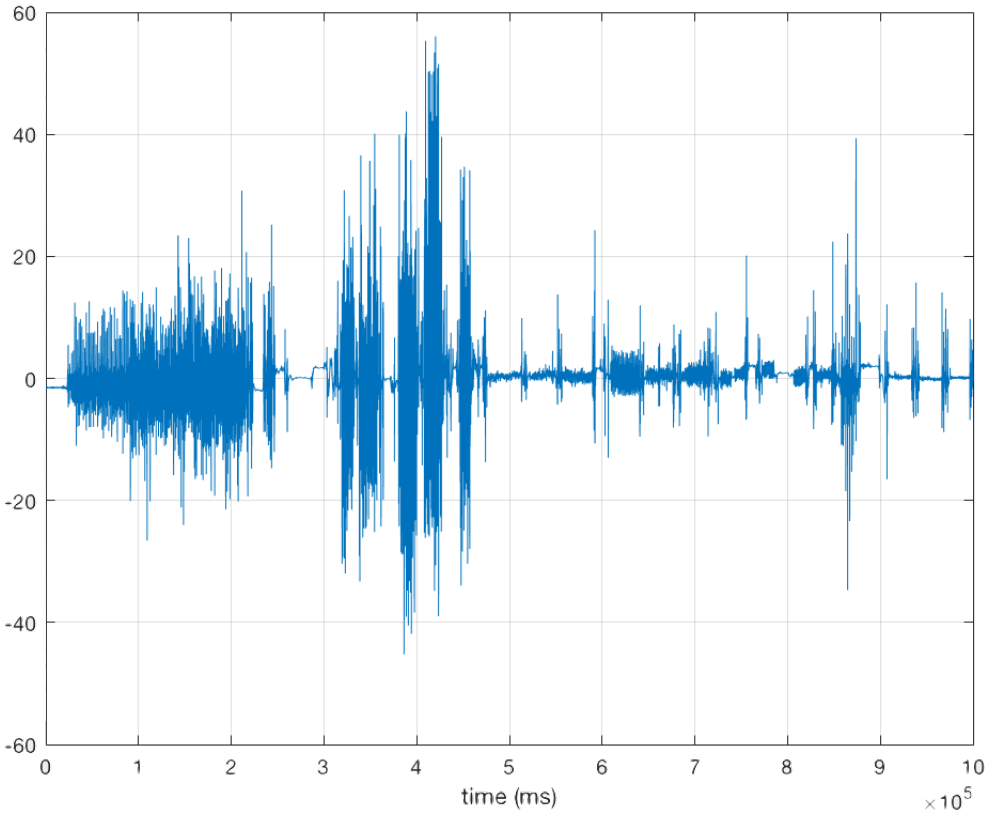

Figure 6.27: *The first 50.000 records of the dataset.*

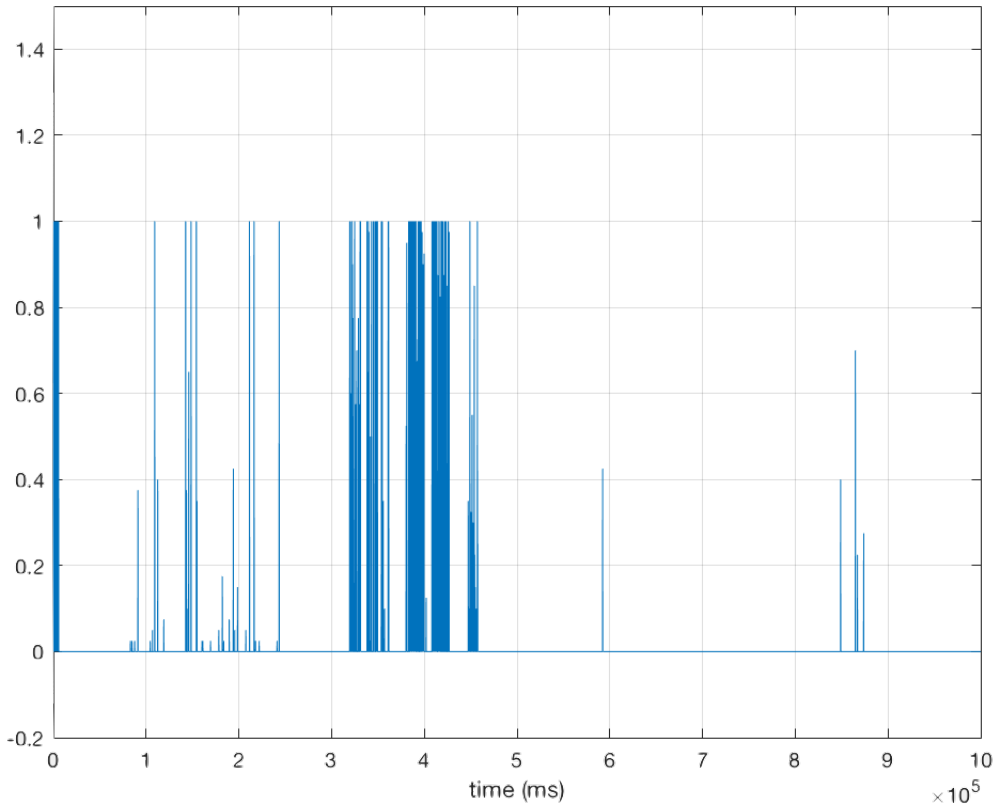

Figure 6.28: *Anomaly peaks found in the first 50.000 records*

Implementing *flink-htm*

In order to implement HTM algorithms on Apache Flink the library *flink-htm* was employed. Unfortunately, using it some fatal bugs were found. Firstly, a mismatch about Flink version was found, it caused an incorrect functioning of the application that was unable to complete the task: the library in fact was deployed when Flink's last version was the 1.2.0 while in this project the most recent version of Flink was employed (1.3.2). In collaboration with the owner of *flink-htm* a bit a work was needed to edit the library in order to match the newer Flink version. Moreover, also the HTM API was changed during the last year hence some hours were spent to

re-define some methods used in the library classes. Currently the fixed *flink-htm* library is available on GitHub [49]. A third corrected bug concerns the implementation of parallelism for an operation related to the computation of the anomaly degree: regrettably, adding it to the most heavy operator is impossible without affect the detection efficacy. However, the most important contribute provided to the library was the implementation of the *Slot Sharing Group* to the HTM network operator, the one which own the computation of anomaly values. It allowed to distribute the task over the cluster and to execute the task: without it no one sensors could be analyzed by the application.

6.4 Summative evaluation

The testing infrastructure looks unusual, for size and equipment, compared to the most common used in production: despite this, almost every frameworks proved to be able to complete the assigned task without problems. The only critical challenge was represented by the execution of the anomaly detection routine provided by HTM on Apache Flink. Actually, we cannot state that Flink was unable to achieve the assigned task: the functionalities offered by the *flink-htm* library have proved to be heavier than we assumed. Moreover, the Cassandra's test with 64 sensors proved that the Flink node was able to handle an elevated throughput (3.2 MB/s): the responsibilities of the failure within the main test (8 sensors at 50 Hz) have to be attributed only to the external library. Unfortunately, the adopted infrastructure was not able to perform real-time detection on more than 2 sensors with an ingestion rate greater than 15 Hz. Actually an ingestion frequency of 15 Hz could be an adequate value for the most of medical sensors: indeed, sending a message

each 66 ms does not represent a low rate and values collected in healthcare context usually does not require an extremely high sampling. Healthcare taking more advantages from the use of a real-time system which could raise alarms or trigger autonomous reactions within few seconds of an emergency. In tests the aim was to achieve real-time analysis with 8 sensors but the objective was respected only with single sensor: the results obtained with 2 sensors are quite poor considering the requirements listed in chapter 2 and should be considered a serious problem. Definitely, in order to obtain a system to analyze many sensors in real-time a more powerful cluster is required. Another unfulfilled requirement concerns the data loss: anyway a loss limited to 0.3% of the entire dataset is acceptable, especially in a context where the whole stream is much more important than the single value.

On the other hand, results exposed in section 6.3.2 illustrates that *flink-htm* is a great instrument to implement an anomaly detector. It can work on almost every dataset, if enough time is dedicated to the build of the network. Moreover, the accuracy depicted comparing Figure 6.25 - Figure 6.28 is impressive and, at least in term of anomalies detected, HTM fulfills completely its scope.

About the other blocks of the system, Raspberry reached unexpected results. It seems to be able to handle easily a number of sensors surely greater than the necessary. The outcomes of Kafka and Cassandra are unsurprising: they represent the State of the Art of Big Data technologies and they are used all around the world in most important companies. Their impressive abilities emerged also with the little infrastructure employed here and the results confirm their great performances also with many sensors which send data simultaneously. Furthermore, the result achieved by these framework have much more significance because were obtained with an infrastructure

which is very distant from a real production cluster so higher performance could be obtained with a bunch of more powerful machines or with a larger set of commodity hardware.

Providing some improvements around the number of involved nodes or using more powerful machines, the designed architecture can be used to realize a real system to face the healthcare scenario presented in this thesis. Today wearables sensors are widely spread in the world and their cost is decreasing quickly: one of the main strength of the designed system is the modularity; it can be sized depending on the specific scenario and the number of involved patients reducing the cost. The main challenge addressed here is represented by real-time anomaly detection of multiple sensors but it can be easily evaded choosing an appropriate processing cluster: the cost to face to realize the infrastructure would be compensated by the improvements achieved with a better and advanced service.

Chapter 7

Conclusions

The dissertation has presented a summarize of the most used technologies in Big Data field and it wanted to describe a working architecture to face with a Healthcare problem. The depicted scenario represented an hard testing ground to evaluate the listed software also in order to understand which kinds of applications are effectively runnable on a cluster.

The technical outcomes were largely commented in chapter 6.3 but viewing on practical applications, we can easily imagine how to use data fetched from sensors. For example, accelerometers and gyroscopes can be used to reconstruct patient's behaviour and avoid false positives: jointly with other wearables (e.g. an heart rate belt) the real-time analysis can distinguishes a real emergency or the symptom of a disease from a natural change of physiological values. As described in section 6.3.2 the HTM algoritm is able to gradually adapt the prediction to the specific current situation: for instance, if a patient is moving after a standstill period, HTM will detect the data pattern change but quickly will understand that is not an emergency.

In this regard, semantic technologies can assist the algorithm and improve data comprehensibility. They have a fundamental role to assist both person

and machine to interpret data and to provide better services to people: further considerations on improvements of medical analysis starting from these statement are described in the chapter 8.

Chapter 8

Future works

The designed project could be considered as a completed system: despite this, some upgrades and extensions are imaginable. Potential supplementary features on the system concern mainly a batch elaborations on data stored on Cassandra cluster. These data are particularly important because they allow to execute deep analysis on patient's data, predicting potential disease and obtaining statistical results after the use of a specific drug. Anymore, trend analysis can describe patient's progress in therapy or points out deficiencies. The usefulness of these data is impressive and they can incredibly increase clinic efficiency while patients gain an higher quality of treatments.

Further developments regard the final part of the system: the addition a new evaluating cluster downstream Cassandra could offers additional data elaborations. In this regard two types of elaborations have to be considered: the streaming and the batch ones. After the choice a processing framework can be selected between those already described in chapter 4: other streaming analysis should be integrated in the pre-existent data processing block. Querying over RDF data streams is quite challenging because it requires a very fast inference process and at the moment the existing semantic pro-

cessors represent a performance bottleneck. About a batch analysis, unless there are specific requirements and in order to avoid unnecessary complications a good option could be use another instance of Apache Flink; other possibilities are Hadoop or Spark.

Furthermore, the semantic engine to query the storage and to infer new knowledge has to be chosen. There are many possibilities released as open-source or commercial systems to perform reasoning over streaming or static dataset. They can be divided in two big families:

- **Centralized engines**: They usually run on single machines. The category includes C-SPARQL [54], CQELS [50], ETALIS [57], SPARQLStream, INSTANS, Streaming Linked Data and SparkWave.
- **Distributed engines**: Mainly deployed on cloud infrastructure. Examples are CQELS-Cloud [51] and Katts [52].

No centralized engines are currently able to face a very massive data stream or to run on a cluster while there are many other engines successfully used against static dataset like SPARQL. On the other hand, CQELS-Cloud is released as a commercial product and it is a bit inflexible about data feeding modalities and query customization while Katts is more a prototype respect to the other engines. About stream reasoning, today many efforts are striving to implements systems to perform it effectively: some examples are showed in [53] and [56] although the former considers the addition of a pre-processing stage before the query execution which affects the real-time requirements, while the latter is proposed in *Strider*, an hybrid adaptive distributed RDF Stream Processing engine based on an implementation of Apache Spark and SPARQL. Besides, it should be noted that Big Data analysis always requires a distributed approach and there is a significant hole to perform semantic

reasoning over distributed clusters: the research of solutions in this sense will be surely an interesting field in semantic web studies.

The system as whole is meant as a modular structure. This configuration promotes and simplify future restructuring of the architecture or extensions needed by the growth of involved people's number in the system. For instance, about Kafka it should be noted that a potential increase of sensors each person can be translated in the addition of new topics, without affecting the preexisting ones. If the number of individuals or sensors grows, in Flink it is sufficient adding more nodes to handle a greater data throughput. Same reasoning can be done about Cassandra: in this regard the number of nodes could be increased also in case of new needs about requested availability.

Acronyms

CEP Complex Event Processing

CQL Cassandra Query Language

DBMS DataBase Management System

DSMS Data Stream Management System

ECG Electrocardiography

IRI Internationalized Resource Identifier

IT Innovation Technology

JSON JavaScript Object Notation

JSON-LD JSON-Linked Data

MQTT Message Queue Telemetry Transport

P2P Peer to Peer

R2RML RDB to RDF Mapping Language

RDBMS Relational DataBase Management System

RDD Resilient Distributed Dataset

RDF Resource Description Framework

RF Replication Factor

RPI3 Raspberry Pi 3

RT Real Time

SSG Slot Sharing Group

TM Task Manager

UK United Kingdom

URL Uniform Resource Locator

US United States

Glossary of Terms

Big Data

Big data is an evolving term that describes any voluminous amount of structured, semistructured and unstructured data that has the potential to be mined for information.

Cluster computing

Cluster computing is a type of computing where a group of several computers are linked together, allowing the entire group of computers to behave as if it were a single entity.

Grid computing

Grid computing refers to a group of computer resources from multiple locations to reach a common goal. It is distinguished from high-performance computing systems such as cluster computing in that grid computers have each node set to perform a different task. Grid computers also tend to be more geographically dispersed than cluster computers.

Near-Real-Time

Pertaining to the timeliness of data or information which has been delayed by the time required for electronic communication and automatic data processing. This implies that there are no significant delays.

Semantic web

An extension of the current Web that provides an easier way to find, share, reuse and combine information. It is based on machine-readable information and builds

on XML technology's capability to define customized tagging schemes and RDF's flexible approach to representing data.

References

[1] Adriana Maria, Bogza, *Performance evaluation of Apache Mahout for mining large datasets*, Master Thesis, FIB, UPC 2016. Under the supervision of Prof. Fatos Xhafa

[2] Çetintemel, Stonebraker, Zdonik, *The 8 Requirements of Real-Time Stream Processing*, 2005

[3] Ballou, Kenny, Apache Storm vs Apache Spark, *https://zdatainc.com/2014/09/apache-storm-apache-spark/*

[4] Friedman, Tzoumas, *Introduction to Apache Flink*, 2016, OReilly Media

[5] Xhafa, Caballè, Naranjo, *Processing and Analytics of Big Data Streams with Yahoo! S4*, 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, 2015, IEEE

[6] Apache Flink official documentation, Introduction to Apache Flink, 2017 *https://flink.apache.org* (accessed as November 2017)

[7] Girik Pachauri, Sandeep Sharma, *Anomaly detection in medical wireless sensor networks using machine learning algorithms*, 4thInternational Conference on Eco-friendly Computing and Communication Systems, 2015, Elsevier

[8] Grehan, Big data showdown: Cassandra vs. HBase, 2014, InfoWorld *https://www.infoworld.com/article/2610656/database/big-data-showdown–cassandra-vs–hbase.html* (accessed as of November 2017)

[9] Haadi Banaee, Mobyen Uddin Ahmed, Ami Loutfi, *Data Mining for Wearable Sensors in Health Monitoring Systems: A Review of Recent Trends and Challenges*, Sensors 2013, volume 13, issue 12, 17472-17500, 2013, MDPI AG

[10] HBase official documentation, 2016 *https://hbase.apache.org* (accessed as of November 2017)

[11] Cassandra official documentation, 2017, Introduction to Apache Cassandra *http://cassandra.apache.org/doc/latest/* (accessed as of November 2017)

[12] Spark official docuementation, 2017, Introduction to Apache Spark *http://spark.apache.org/doc/latest/* (accessed as of November 2017)

[13] Institute for Health Technology Transformation, *Transforming Health Care through Big Data Strategies for leveraging Big Data in the health care industry*, 2013

[14] Jimmy Lin, Erick Josson, *Introduction to MapReduce/Hadoop*

[15] Ali, Calbimonte, Dell'Aglio, Della Valle, Mauri, *RDF Streams processing. ISWC 2016*, The 15th International Semantic Web Conference, 2016

[16] Manyika, Chui, Brown, Buhin, Dobbs, Roxburgh, *Big Data: The Next Frontier for innovation, competition and productivity*, 2011, McKinsey and Company

[17] Noack, *Real-Time Monitoring and Long-Term Analysis by Means of Embedded Systems*, TU Cottbus, 2011

[18] Numenta Community, 2017, Introduction to HTM, *https://numenta.org* (accessed as of November 2017)

[19] Raghupathi W., Raghupathi V., *Big data analytics in healthcare: promise and potential*, 2014, BioMed Central

[20] Cortes, Bonnair, Marin, Sens, The 4th International Workshop on Body Area Sensor Networks, *Stream processing of healthcare sensor data: studying user traces to identify challenges from a Big Data perspective*, 2015, Elsevier

[21] Hussain, Kang, Lee, *A wearable device base personalized Big Data analysis Model*, Lecture Notes in Computer Science, volume 8867, 2014, Springer International Publishing Switzerland

[22] Ahmadi, Purdy, *Real-Time Anomaly Detection for Streaming Analytics*, 2016, arXiv

[23] Van-Dai-Ta, Chuan-Ming Liu, Goodwill Wandile Nkabinde, *Big Data Stream Computing in Healthcare Real-Time Analytics*, International Conference on Cloud Computing and Big Data Analysis, 2016, IEEE

[24] Weihiua He, Yongcai Guo, Chao Gao, Xinke Li, *Recognition of human activities with wearable sensor*, EURASIP Journal on Advances in Signal Processing, 2012, Springer International Publishing

[25] Walker, Every day Big Data statistics, 2015 *http://www.vcloudnews.com/every-day-big-data-statistics-2-5-*

quintillion-bytes-of-data-created-daily/ (accessed as of November 2017)

[26] Andrew Meola, Business Insider, *Internet of Things in healthcare: Information technology in health*, 2016

[27] Goebel, Plagemann, Søberg, Universitetet i Oslo, (2010), *Complex event processing*, Sixth International Conference on Intelligent Sensors, Sensor Networks and Information Processing, 2010, IEEE

[28] Sahu, A real comparison of NoSQL databases, 2015 *https://www.linkedin.com/pulse/real-comparison-nosql-databases-hbase-cassandra-mongodb-sahu/* (accessed as of November 2017)

[29] Datastax, *Benchmarking top NoSQL Databases*, 2015, End Point

[30] Chen, Agrawal, Cochinwala, Rosenblut, *Stream query processing for healthcare bio-sensor applications*,20th International Conference on Data Engineering, 2004, IEEE

[31] Farivar, Knusbaum, *Performance Comparison of Streaming Big Data Platforms*, DataWorks Summit/Hadoop Summit, 2016

[32] Banos, Toth, Damas, Pomares, Rojas, *Dealing with the Effects of Sensor Displacement in Wearable Activity Recognition*, Sensors, Volume 14, Issue 6, 2014, MDPI AG

[33] Banos, Damas, Pomares, Rojas, Toth, Amft, *A benchmark dataset to evaluate sensor displacement in activity recognition*, ACM Conference on Ubiquitous Computing, 2012, ACM

[34] Toth, Banos, *Realistic sensor displacement benchmark dataset*, 2014

[35] MQTT Official documentation, 2017, The MQTT Protocol *http://mqtt.org* (accessed as of November 2017)

[36] ISWC2016 Manifesto, *15th International Semantic Web Conference*, 2016

[37] Raspberry Foundation, Raspberry Pi Official site, 2017 *https://www.raspberrypi.org* (accessed as of November 2017)

[38] Light, *Mosquitto: server and client implementation of the MQTT protocol*, Journal of Open Source Software, 2017, Nottingham ePrints

[39] TripleWave GitHub page, *https://github.com/streamreasoning/TripleWave*, 2016

[40] IoTDB, 2013, *https://iotdb.org* (accessed as of November 2017)

[41] Villalonga, Pomares, Rojas, Banos, *MIMU-Wear: Ontology-based sensor selection for real-world wearable activity recognition*, Neurocomputing, Volume 250, 2016, Elsevier

[42] W3C, SSN Ontology, 2011 *https://www.w3.org/2005/Incubator/ssn/ssnx/ssn* (accessed as of November 2017)

[43] Jun Rao, Connecting to Apache Kafka *Connecting to Apache Kafka*

[44] RDLab - UPC - FIB, *https://rdlab.cs.upc.edu/* (accessed as of November 2017)

[45] Lakshman, Malik, *Cassandra - A decentralized structured storage system*, ACM SIGOPS Operating Systems Review, Volume 44, Issue 2, 2010, ACM

[46] DataStax, Apache Cassandra 3.0 Datastax documentation, 2017, (accessed as of November 2017)

[47] DataStax, Selecting hardware for Apache Cassandra, 2017 *https://docs.datastax.com/en/dse-planning/doc/planning/planningHardware.html* (accessed as of November 2017)

[48] Joe Chu, How to size up a Cassandra cluster, 2014 *https://www.slideshare.net/planetcassandra/201404-cluster-sizing* (accessed as of November 2017)

[49] Wright, flink-htm GitHub page, 2016 *https://github.com/htm-community/flink-htm* (accessed as of November 2017)

[50] Le Phuoc, Danh, *A Native and Adaptive Approach for Linked Stream Data Processing*, NUI Galway Theses, 2013

[51] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Chan Le Van, Manfred Hauswirth *Elastic and scalable processing of linked stream data in the cloud*, International Semantic Web Conference, 2013

[52] Fisher, Charrenbach, Bernstein *Scalable linked data stream processing via network-aware workload scheduling*, SSWS'13 Proceedings of the 9th International Conference on Scalable Semantic Web Knowledge Base Systems, Volume 1046, 2013, CEUR-WS

[53] Schatzle, Przyjaciel-Zablocki, Skilevic, Lausen, *S2rdf: Rdf querying with SPARQL on Spark*, 2015, arXiv

[54] Barbieri, Braga, Ceri, Della Valle, Grossniklaus, *C-SPARQL: A continuous query language for RDF data streams*, International Journal of Semantic Computing, Volume 04, Issue 01, 2010, World Scientific

[55] Kreps, Putting Apache Kafka to use, 2015, Confluent *https://www.confluent.io/blog/stream-data-platform-1/* (accessed as of November 2017)

[56] Ren, Cure, *Strider: A Hybrid Adaptive Distributed RDF Stream Processing Engine*, 2016, arXiv

[57] Anicic, Rudolph, Fodor, Stojanovic, *Stream Reasoning and Complex Event Processing in ETALIS*, Semantic Web, 2009, IOS Press

[58] Kafka official documentation, Persistence section, 2017 *https://kafka.apache.org/documentation/* (accessed as of November 2017)

[59] Wang, 2016 *https://www.npmjs.com/package/node-red-contrib-kafka-node* (accessed as of November 2017)

[60] Nioche, *Low latency scalable web crawling on Apache Storm*, Berlin Buzzwords, 2015