ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

DEPARTMENT OF INFORMATION ENGINEERING
AND COMPUTER SCIENCE

THESIS IN
DATA MINING

Big Data for the Real-Time Analysis of the Cherenkov Telescope Array Observatory

Author:
Giancarlo ZOLLINO

Supervisors:
Prof. Claudio SARTORI
Eng. Andrea BULGARELLI

March 14, 2019
Session III

Abstract [ITA]

Lo scopo di questo lavoro di tesi è quello di progettare e sviluppare un framework che supporti l'analisi in tempo reale nel contesto del Cherenkov Telescope Array (CTA). CTA è un consorzio internazionale che comprende 1420 membri provenienti da oltre 200 istituti da 31 Nazioni. CTA punta ad essere il più grande e più sensibile osservatorio ground-based di raggi gamma di prossima generazione in grado di gestire un'elevata quantità di dati e un'alta velocità di trasmissione, compresa tra i 0,5 e i 10 GB/s, con una rate di acquisizione nominale di 6 kHz. A tale riguardo, è stata sviluppata la RTAlib in grado di fornire un'API semplice e ad alte prestazioni per archiviare o fare caching dei dati generati durante la fase di ricostruzione e analisi. Per far fronte alle elevate velocità di trasmissione di CTA, la RTAlib sfrutta il multiprocesso, il multi-threading, le transazioni ed un accesso trasparente a MySQL o Redis per far fronte a diversi casi d’uso. Tutte queste funzionalità sono state testate ottenendo risultati entro i requisiti richiesti. In particolare, con la libreria sviluppata si riesce a fare caching di dati con Redis, con processi scrittori e lettori che lavorano in parallelo, ad una rate di 8 kHz in scrittura e 30 kHz in lettura.

Il team in cui ho lavorato ha basato sui principi dell'approccio Scrum e DevOps il proprio processo di sviluppo del software, in particolare dalle unit test fino alla continuous integration, utilizzando tools ad accesso pubblico su GitHub oppure tramite Jenkins. Grazie a questo approccio si è puntato ad avere una elevata qualità del codice fin dall’inizio del progetto, e questo è risultato uno degli approcci più importanti per ottenere i risultati raggiunti.

Abstract [ENG]

The aim of this work is to design and develop a framework that supports the Real-Time Analysis of the Cherenkov Telescope Array (CTA). CTA is an international consortium that includes 1420 members from more than 200 institutes in 31 countries. CTA aims to be the biggest next-generation and most sensitive ground-based gamma-ray observatory capable to deal a high amount of data and high data rate, between 0,5 and 10 GB/s, with a nominal acquisition rate of 6 kHz. In this regard, the RTAlib has been developed to provide a simple but powerful high-performance API to store or to caching the data that is generated during the reconstruction and analysis phase. In order to cope with the high data rates of the CTA reconstruction and analysis chain, the RTAlib exploits multiprocess, multithreading, database transactions and a transparent access to MySQL or Redis. All these functionalities have been tested providing results that satisfy the project requirements. In particular, with the developed library it is possible to make data caching with Redis, with writers and readers processes working in parallel achieving a writing rate of 8 kHz and a reading rate of 30 kHz.

The team in which I worked for this project, based its strategy and software development process on the principles of the Scrum and DevOps, especially from unit tests to continuous integration, using public access tools interacting with GitHub or through Jenkins.

Thanks to this approach, the aim was to have a high quality of the code from the beginning of the project, and this was one of the most important approaches to obtain the results achieved.

Contents

1. Introduction

1.1. Technological introduction

In the last ten years three topics took our attention: Artificial Intelligence, IoT or Internet of Things, and Big Data. These are topics closely linked to each other. We could identify in these three keywords the production chain of a new consumer product: Information.

Let's start with the word Big Data, which in its extreme simplification indicates the massive amount of data available on the Internet today. It is sufficient to take a look at the report issued by CISCO in November 2018 to realise how much the data circulating on the internet have experienced massive growth in the last twenty years. In fact, in 1992 global Internet networks carried approximately 100 GB of traffic per day. Ten years later, in 2002, global Internet traffic increased to 100 Gigabytes per second (GB/second). In 2017, global Internet traffic reached more than 45,000 GB/second. Cisco Visual Networking Index expected that annual global IP traffic would reach 4.8 ZettaByte (10^12 GB) per year by 2022.[1]

This dizzying increase in the number of devices connected to the Internet is partially due to a greater possibility of Internet access worldwide, partially because more and more objects will connect to the Internet.

We talk in this context of the Internet of Things, which consist of various devices and sensors that have an Internet interface able to connect to the network and send data. Therefore there is always a higher number of devices that generate an enormous amount of data requiring ad hoc techniques that allow them to be analysed and managed.

One of these techniques is the artificial intelligence, a discipline able to develop algorithms to acquire experience and knowledge that can consolidate and select data automatically, within this vast amount of data.

Big Data expression has now become a value proposition within every productive sector. As for Italy, according to the Big Data Analytics & Business Intelligence Observatory of the Management School of the Polytechnic of Milan, the turnover reached a total value of 1.393 billion euros, up 26% compared to 2017. Again according to Milan Polytechnic University

Report, in 2018 there has been a significant increase in large organisations that have adopted a mature Data Science governance model, from 17% to 31%, inserting Data Scientists, Data Engineers and Data Analysts to lead the company towards optimal Big Data and Real-Time analysis management.[2]
However, Big Data is not just a question of volume. The first definition of Big Data was provided in February 2001 in an article written by Douglas Laney, of Meta Group,[3] which highlighted the need to manage the growing data produced by e-commerce platforms following a three-dimensional approach: Volume, Velocity and Variety.[4] Over time the Big Data definition has changed to add other features until 5 V or even 8V describe it.
Below are reported the 5 V considered most qualifying:

1. Volume: data quantity transmitted on Internet is continuously growing. Some examples are uninterrupted social networks data flow, the ever-increasing purchases number on online market, in addition to traditional data released and managed today in a digital way.
2. Velocity: this is a property required to Big Data analysis algorithms that want to extract information within data as quickly as possible.
3. Variety: Today various kinds of data are available on the Internet, in addition to the textual and structured data of traditional websites. Network is mainly flooded with various non-structured contents typical of social networks like Facebook, which boasts over 4,5 billion posts published every day.
4. Veracity: given the variability of the sources and given the importance that these data is playing, is essential that these data are cleaned up to be considered truthful and accurate.
5. Value: it is undoubtedly the characteristic that is attracting more investment in this field. The value of Big Data lies in the information that allows large companies to react quickly to changes in the market if not even anticipate them and to the world of research to open new horizons.

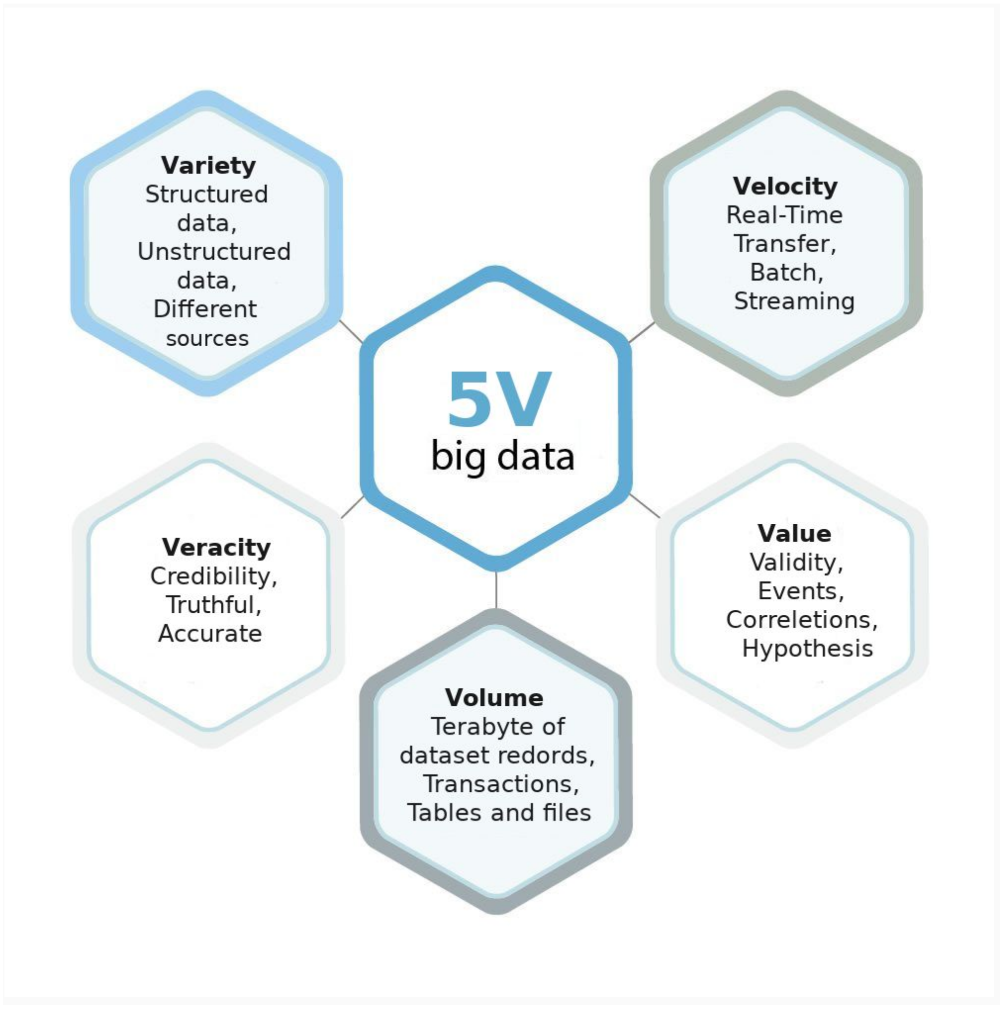

Variety
Structured data, Unstructured data, Different sources
Velocity
Real-Time Transfer, Batch, Streaming
5V
big data
Veracity
Credibility, Truthful, Accurate
Value
Validity, Events, Correletions, Hypothesis
Volume
Terabyte of dataset redords, Transactions, Tables and files

Figure 1.1: Big Data 5V

Big Data is used in very numerous fields, from economic, elaborating accurate and precise market analysis for business strategies, to the medical for genomic medicine to scientific field more generally.[5]

In the scientific field Big Data finds a propulsive drive, to create a virtuous circle that on the one side pushes to the existing technologies optimisation and on the other one pushes towards what Jim Gray, father of the Google Earth project, called "The Four Paradigms of Science".

Inside Four Paradigms, data represent the starter point from which algorithms mine enormous databases looking for relationships and correlations. It requires a new set of skills that help scientists to discover new laws.[6]

In most fields the Value of the information is strictly connected with the concept of Velocity, i.e. the information has value if is processed immediately during the data acquisition. For this reason, Real-Time Analysis is now becoming a reality more widespread. All big companies are equipping themselves with systems for this type of analysis; some directly produce the programs they will use (IBM, Microsoft and CISCO to mention some of the most important), others use open source platforms that are free and available to everyone (e.g. Spark, Storm and Flink).

Real-time data is often not kept or stored but is passed along to the end user as quickly as it gathered. It is important to note that real-time data does not mean that the data gets to the end user instantly. There may be any number of bottlenecks related to the data collection infrastructure, the bandwidth between various parties, or for someone else reason. Real-time data does not promise data within a certain number of microseconds. It just means that the data is not designed to be kept back from its eventual use after it collected. In general, this type of data is described as immediately available data: just generated or just collected.[7]

The analysis in real time is not entirely new, think about the data analysis for weather forecasting or to data analysis for the financial trading, fraud detection at points of sale, traffic or infrastructure monitoring systems. In recent years, also the research world has identified the context of real-time analysis as one of the challenges that has to be faced in the following years.[8]

Some of the aspects identified in this first part of the introduction concerning Big Data and Real-Time analysis represent some of the crucial challenges within my thesis work in collaboration with the Cherenkov Telescope Array Observatory (CTAO).

1.2. Science introduction

This thesis was carried out in collaboration with INAF-OAS researchers and in collaboration with the CTA Consortium, an international partnership that is dealing with the design and construction of the CTAO.

The Astrophysics and Space Science Observatory (OAS) of Bologna is part of the Italian National Institute for Astrophysics (INAF) , and was born on 1 January 2018 from the merger of two INAF institutes in Bologna: Bologna Astronomical Observatory and the Institute of Space Astrophysics and Cosmic Physics.

The National Institute of Astrophysics (INAF) is the leading Italian Research Institute for the study of the Universe. It promotes, carries out and coordinates, also within the framework of European Union programs and international organisations, research activities in the fields of astronomy and astrophysics, both in collaboration with universities and international public and private institutes. It designs and develops innovative technologies and cutting-edge instrumentation for the study and exploration of the Cosmos. Promotes scientific and Astronomy culture spread thanks to projects of teaching in the School and in the Society.

The CTA Consortium includes 1420 members from more than 200 institutes in 31 countries. Italy actively participates in the CTA through INAF, INFN (Istituto Nazionale di Fisica Nucleare) and some universities.

Since 2016 Bologna hosts the CTA headquarters.

CTA will address a wide range of significant questions in and beyond astrophysics, which can be grouped into three broad themes:

1. Understanding the Origin and Role of Relativistic Cosmic Particles;
2. Probing Extreme Environments (environments of extreme energy density);
3. Exploring Frontiers in Physics (dark matter, quantum gravitational effect, axion-like particles).

CTA aims to answer to these questions using Imaging Atmospheric Cherenkov Telescopes (IACTs)[9]. When gamma rays reach the earth's atmosphere, they interact with it, producing cascades of subatomic particles. These cascades are also known as air or particle showers. Nothing can travel faster than the speed of light in a vacuum, but light travels 0.03 % slower in the air. Thus, these ultra-high energy particles can move faster than light in the sky, creating a blue flash of “Cherenkov light” (discovered by Russian physicist Pavel Cherenkov in 1934). These blue flashes are observed with IACTs.

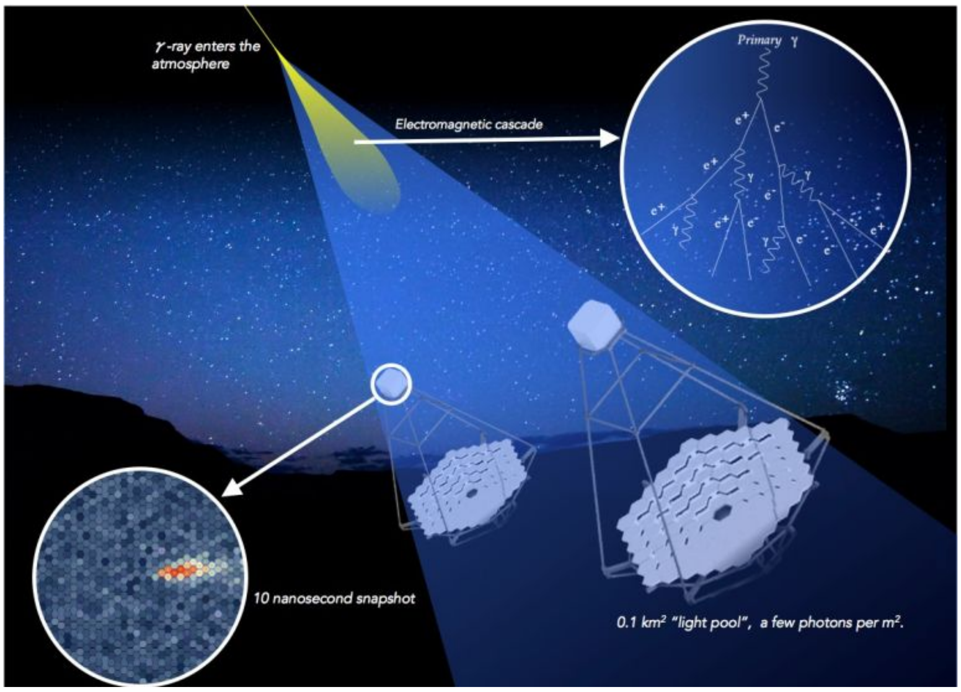

γ-ray enters the atmosphere
Electromagnetic cascade
Primary γ
10 nanosecond snapshot
0.1 km² "light pool", a few photons per m².

Figure 1.2: IACT, Imaging Atmospheric Cherenkov Technique

The CTA observatory will consist of more than 100 IACT telescopes located at two sites.
In the Northern hemisphere, the CTA will be hosted in La Palma in the ‘Roque de Los Muchachos Observatory’, while in the southern hemisphere the CTA will be hosted in Paranal, Chile.

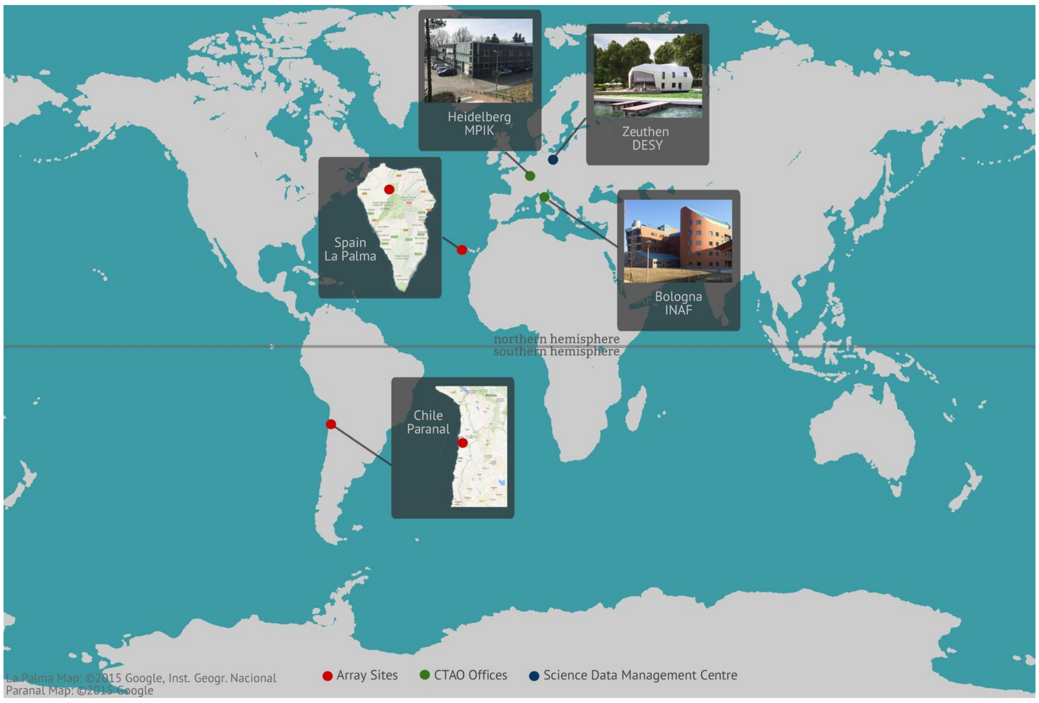


Figure 1.3: CTA Offices and locations

Three classes of telescope types are required to cover the full CTA energy range (20 GeV to 300 TeV):

1. Small-Size Telescope (SST): telescopes with an energetic range from few TeV to 330 TeV, there will be installed 70 elements of this telescopes;
2. Medium-Sized Telescopes (MST): telescopes with an energetic range from 100 GeV to 10 TeV, there will be installed 40 elements of this telescopes;
3. Large-Sized Telescopes (LST): telescopes with an energetic range from 20 GeV to 200 GeV, there will be installed 8 elements of this telescopes.

CTA will have also a **Real-Time Analysis system**, that must be capable of generating science alert during observations or provide real-time feedbacks to external triggered ToO events[1] with a latency of 30 s starting from the last acquired event that contributes to the alert. The Real-Time Analysis differential sensitivity should be as close as possible to the one of the final offline analysis pipeline but not worse than a factor of 3: despite those caveats, an unprecedented sensitivity for short term exposures is achieved. The search for transient phenomena must be performed on multiple timescales (i.e. using different integration time

[1] To Observe events

windows) from seconds to hours, both within an a-priori defined source region, and elsewhere in the FoV[2]. A Real-Time Analysis pipeline specific to each sub-array configuration is foreseen to run in parallel. The availability of the Real-Time analysis during observations must be greater than 98%. The Real-Time Analysis enables CTA to provide real-time feedback to external received alerts, and to maximize the science return on time-variable and transient phenomena by issuing alerts of unexpected events from astrophysical sources.[10]

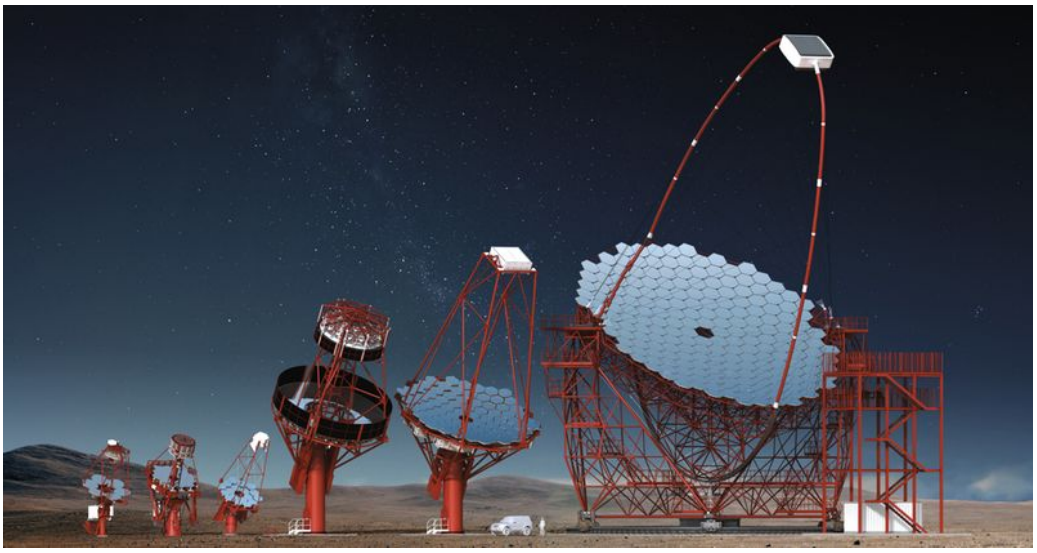


Figure 1.4: From Left: Three different SST prototype designs, the two MST prototype designs and the LST prototype design.

To understand how the gamma-ray will be acquired by CTA and analysed by the Real-Time Analysis system, it's important to know the electromagnetic spectrum structure to understand what we are talking about. Light is made up of waves of alternating electric and magnetic fields that can be measured by frequency (number of waves that pass by a point in one second) or wavelength (the distance from the peak of one wave to the next). The larger the frequency, the smaller the wavelength. The spectrum ranges from the very lowest frequencies of radio waves and microwaves; to the mid-range frequencies found in infrared, optical (visible) and ultraviolet light; to the very highest frequencies of X-rays and gamma rays. The frequency range of gamma rays is so vast that it doesn't even have a definite upper limit. The gamma rays CTA will detect are about 10 trillion times more energetic than visible light.[11]

[2] Field of View

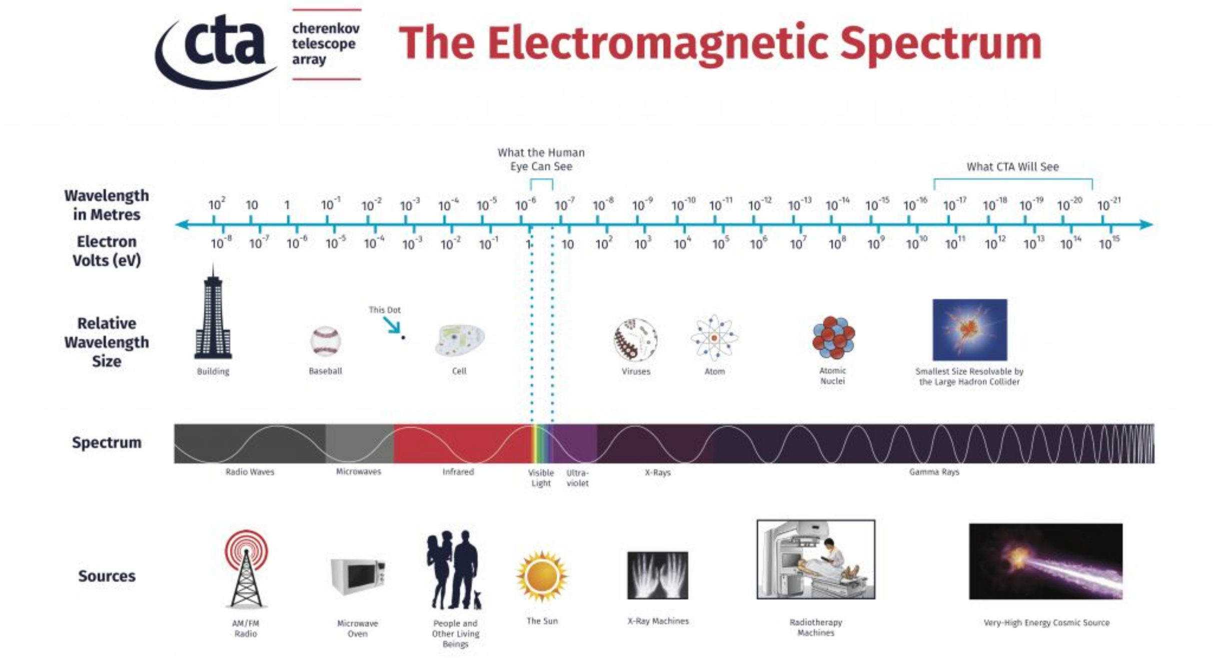


Figure 1.5: The Electromagnetic Spectrum

Gamma-ray are produced by the most violent and energetic phenomena of the Universe. Between them, just to provide an example, we can cite the Gamma-Ray Bursts (GRB), the most high energy astrophysics phenomena observed so far. For decades, almost nothing was known about gamma-ray bursts: its origins and distribution, spatial and statistical, were unknown. The gamma-ray bursts themselves showed a great variety: they could last from a fraction of a second to several minutes, they present a great range of spectra, which did not resemble anything that was then known. Now the big picture is much more clear, 14 January 2019 it's been observed the first Cherenkov light produced by a gamma-ray burst. GRB 190114C, this is the name of the observed event, it has been seen from MAGIC, the biggest gamma-ray telescope (until CTA will start to work), located in La Palma in the 'Roque de Los Muchachos Observatory'. This discovery opens new frontiers in the exploration of the gamma-ray sky, and makes the **Real-time Analysis systems** even more important, because we can follow the evolution of transient phenomena in real time only using these systems.

This thesis work, as better explained in Chapter 4, is placed in the "On Site" area of CTA and it deal with the “Data Acquisition & Reduction” and “On-Site Storage” phases.

2. Technology

In this chapter languages and tools used during software development are introduced.

2.1. Languages

The central core of the library developed during this project is written in C and C++ language. It also has been used a Python version of the same library. To manage structured data the SQL language has been used.

2.1.1. C and C++

C is a compiled general-purpose high-level programming language. It was developed in 1972 in Bell laboratories by Dennis M. Ritchie to develop UNIX operating system. In 1978 Brian Kernighan and Dennis Ritchie published a text that has become a legend: "The C Programming Language". After the K & R version of the C language there followed a large number of variations and dialects able to program in C or pseudo C any microprocessor system. In 1988 become a standardised language by ANSI.
Despite C is a high-level language, it provides methods and artefact to handle low-level activity and this makes C a very fast programming language.[12]
Today C is used to write the most popular Linux Operating System features, different RDBMS (MySQL is written by C and C++) and different language interpreter too. (see Python).
C++ was born in 1979 as a merger of two languages, Simula67 and C used by Bjarne Stroustrup during his PhD thesis. Simula67 was the first simulation language that includes some object-oriented paradigm.
Shortly after that, it was born the first “C with Classes” version that in 1983 became “C++”.
C++ as C language is a compiled general-purpose high-level language, and today it’s used to write very different kind of software, from operating system to computer vision software, from gaming to high-performance data management system. From 1983 to today different C++ versions have been released, each of them added some features to differentiate C++ from

C. The first step introduced inheritance, virtual function and function overloading and much more.
During this thesis C++11 standard has been used.

2.1.2. Python

Python is an interpreted general-purpose high-level language. Python has been designed by Guido Van Rossum in the late 1980s, its first release was in February 1991 and supported different programming paradigm, functional, imperative, procedural and object-oriented. It’s used for very different kind of purpose, from web development to simple gaming, from mathematical software to data management.
Python provides worst performance compared with C++ or other optimised compiled language, nevertheless thanks to its simple syntax it represents an excellent choice for creating a prototype. Furthermore, in Python it's possible to simply add compiled source code as C++ to use more performant code where necessary.
In this thesis work, it has been used 3.6.8 Python version.

2.1.3. SQL

SQL (Structured Query Language) is a standard language used to create, store and manage a relational database.
It has been created in IBM laboratory in 1974 by Donald Chamberlin, and become ANSI standard in 1986.
SQL is a declarative language that can be divided into four language subsection:

- Data Definition Language (DDL): to create, delete and modify the database object (e.g CREATE, DROP, ALTER etc);
- Data Manipulation Language (DML): to insert, modify and delete data into the database (INSERT, UPDATE, TRUNCATE etc);
- Data Control Language (DCL) to manage users and access control (GRANT, REVOKE etc);
- Query language (QL): to read tables and extract information from them (SELECT, FROM, WHERE etc);

2.2. Database

The term Database indicates a set of data collected in a digital form, that is frequently written, read and updated, organised so that it can be easily accessed. All operations executed on a database are managed from a "*software system that enables users to define, create, maintain and control access to the database*".[13]

This software is called DataBase Management System (DBMS).

From their inception, Databases, have evolved following different model design but today two are the main model widely used: relational and post-relational or NoSql.

The relational model uses mainly SQL language, a declarative language that improves programmer productivity and database performances.

This model was designed by Edgar F. Codd, an IBM programmer, in 1970. He wrote a paper, "A Relational Model of Data for Large Shared Data Banks", in which proposed a new way to organise and store data. In this model, data are grouped in tables. Each table represents a generic object as a collection of attributes, each of them is stored in a column. Each row within the tables, also called record or tuple, represent an object instance.

Every instance has identified by a primary key, an object attribute that can be used to refer to a particular row. Two or more tables can be linked one to each other using the respective primary key, forming the relationship.

The NoSql database becomes increasingly popular in the first 2000s when cloud computing and ever more performance web services appeared on the Internet.

NoSQL, which stands for "not only SQL," is an alternative to traditional relational databases when we are working with a very large amount of data managed on a distributed system.

NoSql database differs from the relational database for any difference:

- Database structure;
- Scalability;
- Data consistency;

The relational database requires to design the internal structure, made up of tables and relationships. This initial conceptual work conditions the execution of every operation on the DB, from the insertion of data to the query. The rigid structure of the contents is absent in the NoSQL databases; the informations are stored in a completely different way and not necessarily structured objects, such as documents stored in collections.
Centralised nature of Relational Database make it vertical scalability, that means to increase its performance is necessary to increase the amount of RAM, SSD or CPU. In contrast, NoSQL databases are horizontally scalable, which means that they can handle more traffic by merely adding more servers to the database.[14]
The very significant difference between the two models regards Data consistency. In fact, in the relational database, DBMS require atomically this features every time data change, while in NoSQL model this property is left to application logic.
In this thesis work, it has been used both models to respond to the project requirements.

2.2.1. MySQL

MySQL is the world's most popular open source Relational DataBase Management System.
Originally this RDBMS was developed from MySQL AB, a Swedish company, that then has been purchased from Oracle company.
MySQL provides besides Server-side software Client API for the main programming language. In this project work it has been used Version 14.14 Distribution 5.7.24 for the server-side software and C++ X DevAPI 8.0 for the software client-side.
MySQL offers different table engines, which have some distinctive features. Following we can see all tables engine:

- MyISAM
- InnoDB (transactional engine)
- Memory (before Heap)
- Merge
- NDB, o ClusterDB (> 5.0)
- CSV (> 5.1)
- Federated (> 5.0)

- Archive (> 5.0)
- BLACKHOLE (> 5.0)

During this project, I have used three engines: MyISAM, InnoDB and Memory.

- InnoDB: represent the default storage engine in MySQL 5.7. InnoDB is a transaction-safe (ACID compliant) storage engine for MySQL that has commit, rollback, and crash-recovery capabilities to protect user data. InnoDB row-level locking and Oracle-style consistent nonlocking reads increase multi-user concurrency and performance. InnoDB stores user data in clustered indexes to reduce I/O for common queries based on primary keys. To maintain data integrity, InnoDB also supports FOREIGN KEY referential-integrity constraints.
- MyISAM: These tables have a small footprint. Table-level locking limits the performance in read/write workloads, so it is often used in read-only or read-mostly workloads in Web and data warehousing configurations.
- Memory: Stores all data in RAM, for fast access in environments that require quick lookups of non-critical data. This engine was formerly known as the HEAP engine. Its use cases are decreasing; InnoDB with its buffer pool memory area provides a general-purpose and durable way to keep most or all data in memory, and NDBCLUSTER provides fast key-value lookups for huge distributed data sets.[15]

In the following table, we can see the main engine features:

Feature	**MyISAM**	**Memory**	**InnoDB**
B-tree indexes	Yes	Yes	Yes
Backup/point-in-tim	Yes	Yes	Yes

e recovery			
Cluster database support	No	No	No
Clustered indexes	No	No	Yes
Compressed data	Yes	No	Yes
Data caches	No	N/A	Yes
Encrypted data	Yes	Yes	Yes
Foreign key support	No	No	Yes
Full-text search indexes	Yes	No	Yes
Geospatial data type support	Yes	No	Yes
Geospatial indexing support	Yes	No	Yes
Hash indexes	No	Yes	No
Index caches	Yes	N/A	Yes
Locking granularity	Table	Table	Row
MVCC	No	No	Yes
Replication support	Yes	Limited	Yes
Storage limits	256TB	RAM	64TB
T-tree indexes	No	No	No
Transactions	No	No	Yes
Update statistics for data dictionary	Yes	Yes	Yes

Table 2.1: MySQL engine features

2.2.2. Redis

Redis is an open source key-value DBMS, developed by programmer Salvatore Sanfilippo programmer, and released for the first time in 2009.

It works with in-memory adaptive data structure used as databases, cache and message broker. Redis data type include:

- String: textual or binary safe blobs with a maximum length of 512 MB ;
- List: linked list of string;
- Set: unordered list of string ;
- Sorted Set: list of string ordered by a value;
- Hash: a data structure to store a key-value list;
- Bitmap: is not an actual data type, but a set of bit-oriented operations defined on the String type;
- HyperLogLog: a probabilistic data structure used in order to count unique things (technically this is referred to estimating the cardinality of a set).

Thanks to its high write and read performance due to massive RAM using, Redis becomes a popular choice for caching, storing, messaging, streaming and real-time analysis system.

Redis works in central memory, however it provide two persistence mechanism:

- RDB: execute point-in-time snapshot of the dataset in a fixed interval;
- AOF: logs every write operation received by the server;

Besides, Redis supports Replication configuration that allows creating a dataset copy instance. It used a master-slave architecture, where master offers the dataset and slave asynchronously create a copy of them. It is possible to use this configuration also in a cluster configuration. This strategy makes Redis fault tolerant and allows better workload balance. To improve write and read performances it possible to create a master Redis instance for write operations and one or more Redis slave to perform reading.

Redis implements the Publish/Subscribe messaging paradigm where senders (publishers) are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterised into channels, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more channels, and only receive messages that are of interest, without knowledge of what (if any) publishers there are. This decoupling of publishers and subscribers allows for greater scalability and more dynamic network topology.[16]

Redis provides a fast memory datastore ideal for real-time streaming; it is ideal to work with Apache Kafka and Amazon Kinesis for real-time analysis, social media e customising IoT.[17]

Redis provides server and client API software too. In this project it has been used Redis server v. 3.2.12 and C hiredis Client API.

2.3. Profiling

To analyse code quality and performance may be necessary to execute profiling software.

In this thesis work has been utilised Valgrind software.

2.3.1. Valgrind

The Valgrind tool suite provides some debugging and profiling tools that help to make a program faster and more correct. The most popular of these tools is called Memcheck. It can detect many memory-related errors that are common in C and C++ programs, and that can lead to crashes and unpredictable behaviour.[18]

To use Valgrind tool has been necessary to rebuild the software and all its dependencies in debug mode.

3. RTA System Software Engineering

In this chapter, the software development and DevOps processes adopted during the thesis work are presented.
In the following paragraphs, the technologies behind the software development process that has been used to implement the software of this thesis work (the RTAlib), are listed, along with their description.
Furthermore, the theory of Continuous Integration and Continuous delivery is explained. Finally, the Continuous Integration processes adopted for the RTAlib are described: the first approach uses the online CI service CircleCI, the second is implemented with a local Jenkins CI server and the usage of a Singularity container to wrap the RTAlib software dependencies.

3.1. Agile&Scrum

The software development process implemented for this thesis work is based on the Agile approach. The Agile development process began to spread in the early 2000s, based on the "Manifesto for Agile Software Development". In 2001, seventeen leading thinkers of the software engineering and development research field, wrote this document in which they proposed a lightweight set of values and principles against heavyweight software development process such as the waterfall one.
The goal of the Agile approach is to reduce software development costs and time by increasing code quality and customers satisfaction. The Agile development process aims to achieve these goals pointing up four key aspects:

- Individuals and Interactions over processes and tools;
- Working Software over comprehensive documentation;
- Customer Collaboration over contract negotiation;
- Responding to Change over following a plan;

The Agile methodologies allow to continuously review the software specifications, adjusting them during the development process, through an iterative and incremental framework. They

also plan a strong exchange of information and opinions between the developers and the client. On the basis of these principles, every organization can develop its own implementation and habitual practices to improve outcomes.
Among the several Agile models available the Scrum model was used.
Scrum is a framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.

Scrum is:

- Lightweight
- Simple to understand
- Difficult to master

Scrum is a framework within which every team can employ various processes and techniques.
The Scrum framework consists of Scrum Teams and their associated roles, events, artifacts, and rules. Rules bind together the team roles, events, and artifacts, governing the relationships and interaction between them.
Scrum is founded on empirical process control theory or empiricism. Empiricism asserts that knowledge comes from experience and making decisions based on what is known. Scrum employs an iterative, incremental approach to optimise predictability and control risk. Three pillars uphold every implementation of empirical process control: transparency, inspection, and adaptation.[19]

- **Transparency:** requires that all significant project aspect be defined by a common standard easily understandable by everyone.
- **Inspection:** Scrum users must frequently inspect Scrum artifacts and progress toward a Sprint Goal to detect undesirable variances.
- **Adaption:** if an inspection notices one or more aspects of a process deviates outside acceptable limits an adjustment must be made as soon as possible to minimize further deviation.

The Scrum team is composed of the Product Owner, Development Team, and a Scrum Master.
The Product Owner is responsible for maximizing the value of the product resulting from

work of the Development Team. The Product Owner's decisions are visible in the content and ordering of the Product Backlog.
The development team is composed of a limited number of developers, possibly between 4 and 9 members, and they are responsible for the incremental software development.
The Scrum Master helps everyone to understand which interactions are helpful to maximize the knowledge share and value created by the Scrum Team.
The Scrum artifacts are something that provide to the team the key informations about the software development status: among the artifacts there are the Product Backlog, the Sprint Backlog and the Increment.
The Product Backlog is the single source of product requirements and for any changes to be made to the product. The Product Owner is responsible for the Product Backlog, and it must to keep it updated to the latest version. The two others type of artifacts will be discussed later in this paragraph.
The heart of the Scrum process is the Sprint, a time-box during which a working, useable, and potentially releasable product Increment is created. The Sprint Backlog is the set of Product Backlog items selected for the Sprint and it is a forecast by the Development Team about what functionality will be in the next Increment. The Increment is the sum of all the Product Backlog items completed during a Sprint.[19] The process of selecting the Product Backlog items for the Sprint Backlog is called Sprint planning and it is performed by the team.
Another key element of the Scrum approach is the Daily Scrum meeting, it is a meeting of the team members, during which the Development Team plans work for the next 24 hours. This optimises team collaboration and performance by inspecting the work since the last Daily Scrum and forecasting upcoming Sprint work.

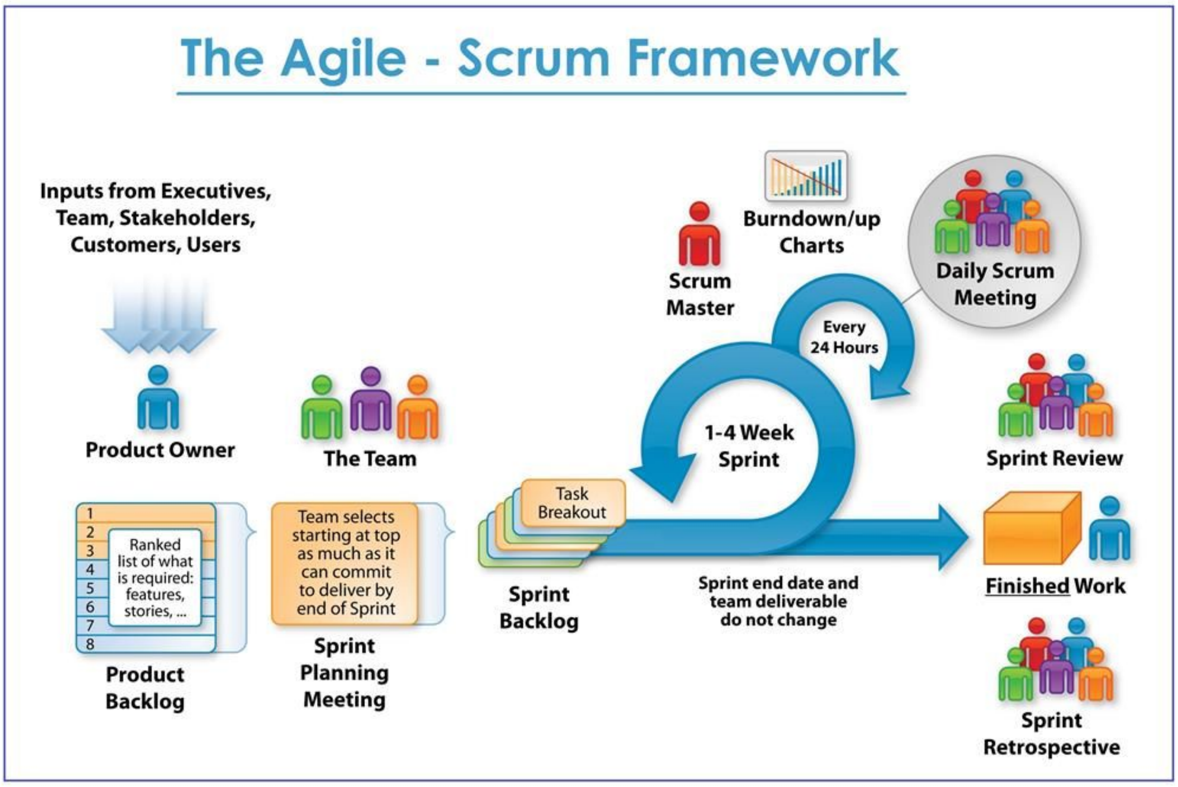

The Agile - Scrum Framework
Inputs from Executives, Team, Stakeholders, Customers, Users
Product Owner
The Team
1
2
3
4
5
6
7
8
Ranked list of what is required: features, stories, ...
Product Backlog
Team selects starting at top as much as it can commit to deliver by end of Sprint
Sprint Planning Meeting
Task Breakout
Sprint Backlog
Scrum Master
Burndown/up Charts
Daily Scrum Meeting
Every 24 Hours
1-4 Week Sprint
Sprint end date and team deliverable do not change
Sprint Review
Finished Work
Sprint Retrospective

Figure 3.1: Agile Scrum representation

3.2. GitHub

Before starting to talk about GitHub is necessary to explain what a Version Control System (VCS) is.

VCSs are software tools used principally from the software development team to manage their project and keep track of all changes made on them.

In the beginning, the code developed by the members of the development team had been stored and executed locally by each developer. The moment when multiple team members started to collaborate to the same software project, brought to a Centralised Version Control System (CVCSs).

Thanks to it, it was possible to store the work of each developer on a centralised server. This allow a developer that wants to collaborate to the project, to clone the source code locally.

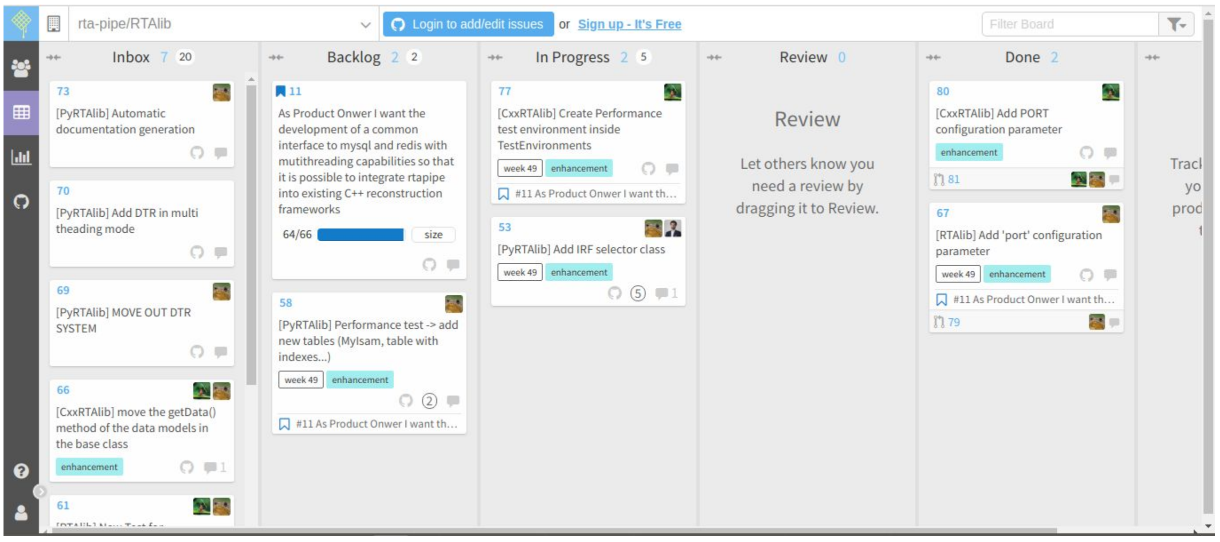
This system allows everyone to know who does what but it presents a severe downside: the centralised server represents a single point of failure. If the central hard disk was corrupted or destroyed all work may be lost forever.
To deal this issue, it has been developed a Decentralised Control Version System (DCVS).
In DCVS when a user clones the project in a local machine pulls up the code with its full history so that if any server dies any user can restore the project on the server system.
GitHub is a web-based DCVS, founded on Git project, born in 2008.
On 3 April 2005, Linus Torvalds and the Linux community began to develop this free and open source system to collaborate in a decentralised way to develop the Operating System kernel.
The new system provides:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)[20]

On 16 June 2005, it has been published the Linux kernel version 2.6.12, the first one managed with Git.

3.3. Waffle

Waffle is a online Scrum board: it visually depicts the iterations progress using cards to represent task items and columns to represent each stage of the process.
Waffle is built on top of the GitHub API and it allows to plan, organises and tracks in a graphical and interactive fashion the work of team.


rta-pipe/RTAlib

Login to add/edit issues or Sign up - It's Free

Filter Board

Inbox 7 20

73
[PyRTAlib] Automatic documentation generation

70
[PyRTAlib] Add DTR in multi theading mode

69
[PyRTAlib] MOVE OUT DTR SYSTEM

66
[CxxRTAlib] move the getData() method of the data models in the base class
enhancement 1

61

Backlog 2 2

11
As Product Onwer I want the development of a common interface to mysql and redis with mutithreading capabilities so that it is possible to integrate rtapipe into existing C++ reconstruction frameworks
64/66 size

58
[PyRTAlib] Performance test -> add new tables (MyIsam, table with indexes...)
week 49 enhancement 2
#11 As Product Onwer I want th...

In Progress 2 5

77
[CxxRTAlib] Create Performance test environment inside TestEnvironments
week 49 enhancement
#11 As Product Onwer I want th...

53
[PyRTAlib] Add IRF selector class
week 49 enhancement 5 1

Review 0

Review

Let others know you need a review by dragging it to Review.

Done 2

80
[CxxRTAlib] Add PORT configuration parameter
enhancement
81

67
[RTAlib] Add 'port' configuration parameter
week 49 enhancement
#11 As Product Onwer I want th...
79

Figure 3.2: Waffle Scrum Board

This software is mainly characterized by three features:

- **Real-time Sync with GitHub**: Waffle is synced with GitHub repositories, so the changes made to branches and commits is reflected to the position of the cards in the boards. As it is possible to view in Fig. 3.2 Waffle provide a more friendly visual interface. In Waffle board is possible to view which are task currently in progress, which are completed and which are the next tasks to be developed.
- **Automatic Tracking and Powerful Metrics**: during the design stage in Waffle board, each programmer plans the next task to work on, defining priority and estimated time. Based on these parameters Waffle can automatically update the milestone burndown graph and throughput graph. These graphs help teams to know its productivity and team capabilities in general.
- **On-premise deployment, integrations and more**: It also provided an on-premise version called Waffle Takeout. Other features include connecting multiple repositories in a single view, customizable workflows, and filtering via labels, issue text, milestone, or owner.

Figure 3.3 shows an extracted Burndown of a Scrum Sprint.

Waffle was launched in 2013 by a team led by Andrew Homeyer, a software engineer and intrapreneur at Rally Software in Boulder, CO, USA. Rally was acquired by CA Technologies in May 2015. At present, Andrew is continuing to lead the enterprise lean startup company.[21]

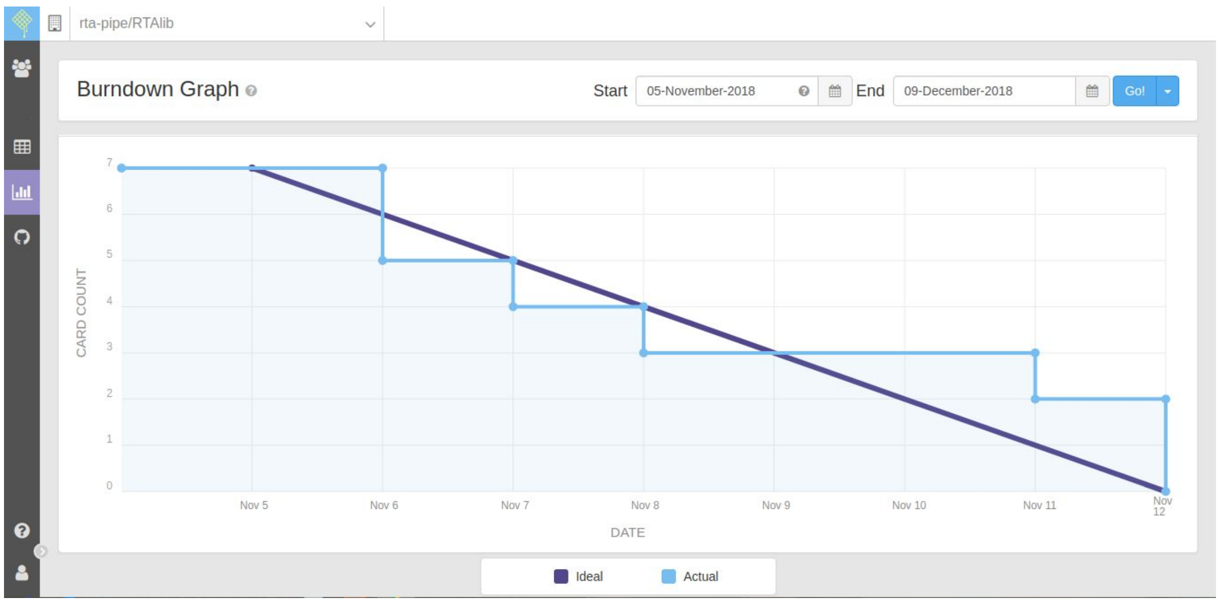


Figure 3.3: RTAlib Burndown graph of a Sprint

3.4. Containers

A container is a software package that contains applications and all the needed dependencies. Furthermore, it does not require any installation and so make its environment isolated from the guest operating system.

These software units are particularly suitable in software development because its easy portability allows to create, develop and testing software always in the same environment.

Containers share the guesting operating system kernel, in contrast to virtualisation technique which executes different operating system on the same hardware shared by a hypervisor mediator.[22]

A virtual machine has to virtualise all operating system, application and hardware too and this makes it heavy and slow. Furthermore, containers have a small size because add only necessary project source and for this containers require fewer resources and make them faster than a virtual machine.

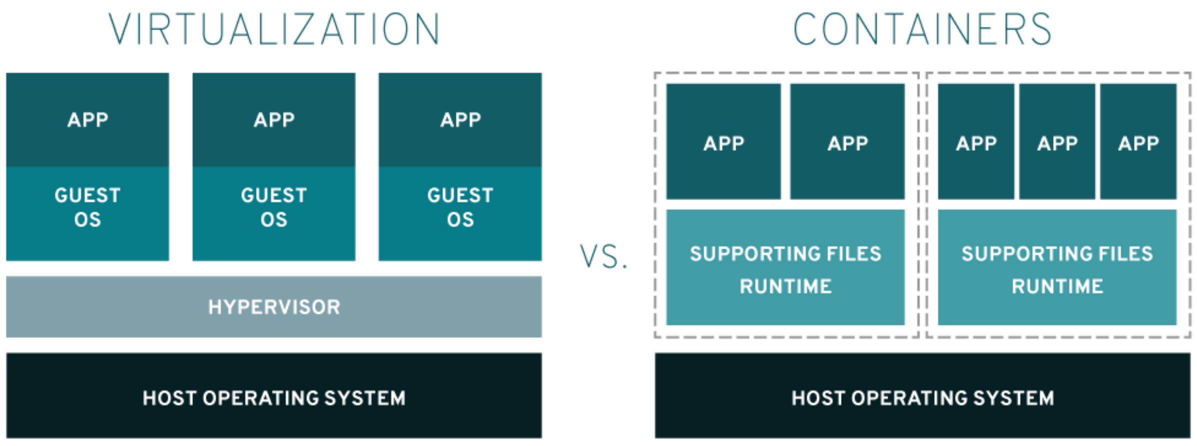


Figure 3.4: Virtual Machine architecture vs Container architecture

3.4.1. Docker

Docker was born in 2008 as an open source technology that allows creating a modular and incremental lightweight virtual machine.

Docker is mainly used in DevOps context to provide to developers an environment that can be used in production phase allowing them to focus on writing code without worrying about the system that it will ultimately be running on.

Each Docker image file consists of multiple layers, which together make up a single image. Docker reuses these layers to speed up the process of creating containers. The changes are shared between the images, further improving speed, size and efficiency. Version control is part of the stratification process. This multilayer structure allows to extract and modify a single component, and besides, it is so possible to execute the rollback if the new container version is not satisfied as the previous one.[23]

Docker image is launched and managed from a user software execute in the user space: Docker Container. This approach has much less overhead between the host kernel and the application, improving performance respect than virtualisation environment.

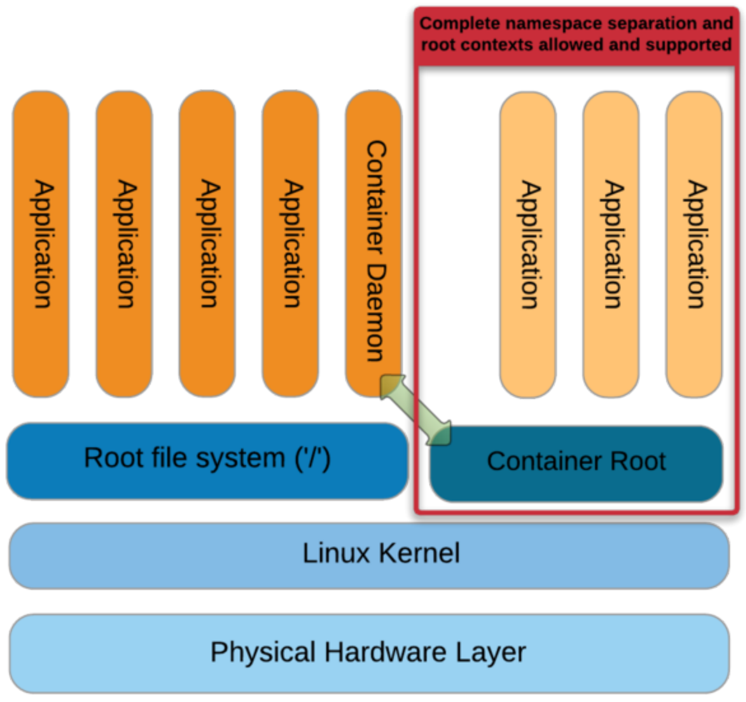


Figure 3.5: Docker architecture

3.4.2. Singularity

Singularity developed by Gregory M. Kurtzer in April 2016 as an open source High-Performance Computing solution to satisfy researcher need for “mobility of compute”. Singularity differentiates from Docker for two main reasons:

- namespace usage;
- root privilege;

In contrast to Docker, Singularity uses less microservice as possible to use the smallest number of namespaces necessary to achieve its primary design goals. This gives Singularity a lighter footprint, greater performances potential and easier integration than container platforms that are designed for full isolation.

The Singularity security paradigm is different from the Docker’s one. Singularity it was developed for HPC type shared-infrastructures where every user is considered untrusted. In order to let untrusted users running untrusted containers the users can run the containers as

themselves: the root privileges are only required for the container building process that could be done in another environment (and not in the shared one).
In this thesis work, it has been used the version 2.6 of Singularity.

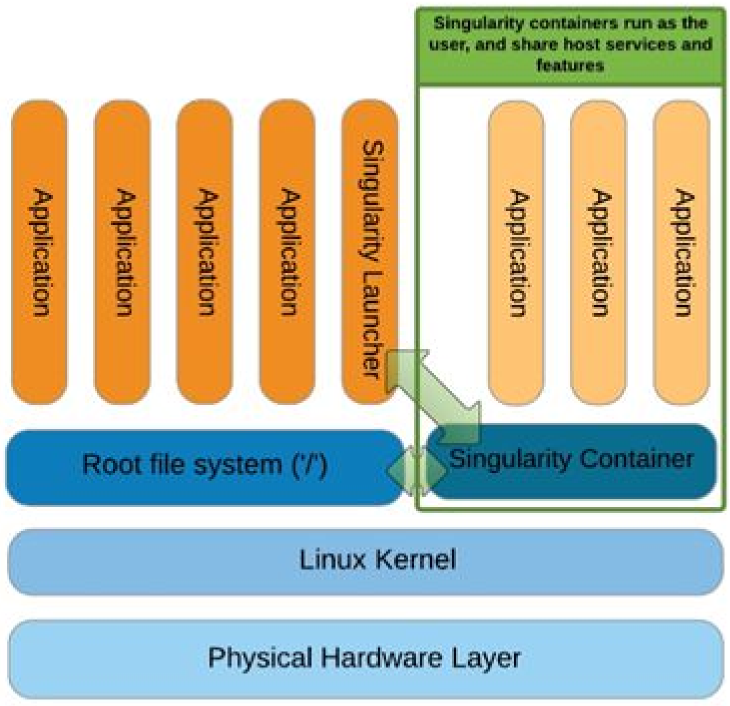


Figure 3.6: Singularity Container

3.5. Continuous Testing and Integration

Continuous Integration (CI) is a widely established development practice in software development process, in which members of a team integrate and merge development work frequently, for example multiple times per day. CI enables software companies to have shorter and frequent release cycle, improve software quality, and increase their teams‘ productivity. This practice includes automated software building and testing.[24]
It is essential in this context to have an automatic framework for the testing and the quality check of the code that is updated frequently and shared by several collaborators.
Continuous Testing (CT) is a software testing phase in which the individual software units are tested and if the tests are successfully passed, software units are combined together with the

others and tested as a group. If the software fails the test phase, a notification is sent to the developers who are will have to solve the problem. In addition to the testing phase, the integration of tools for the evaluation of the code quality is often included in the continuous integration.
For this thesis work, two state-of-the-art software tools for continuous integration have been used: CircleCI and Jenkins.
CircleCI is a web-based continuous integration platform that can be integrated with GitHub repository. If a developer interacts with the GitHub API (for example to open a pull request), the CircleCI system is automatically triggered and a .yml recipe is executed. The recipe uses a docker container to run the code and test it: if the software pass the test then it can be released.
Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software. It's executed inside a "local" web server and in a similar way to CircleCI executes a Jenkins file to perform software testing. It requires a more complex previous configuration than CircleCI. It has a very large community of developers and a huge set of plugins that can be added to the default installation to increase the functionalities of Jenkins or to improve the existing ones. The plugin used for this thesis work are: the Blue Ocean plugin and the GitHub Integration plugin. The first allows a user to create a testing pipeline in a visual fashion, automatically creating the Jenkins file from the graphic scheme the user made. The second plugin allows Jenkins to trigger build processes after interactions as pull requests are made with the GitHub servers.
To perform CT and the overall CI containers are needed. Issues are tracked by an Issue Tracking system. Graphical User Interface tools can be used to check metris of code quality and suggestion to improve them. Unit testing metrics are also provided (e.g. code coverage).

3.5.1. Unit testing

Unit testing is a software development process aimed to test the "units": the smallest piece of code that can be logically isolated in a system.

To perform the unit testing of C++ code it has been used a framework developed by the Google Testing Technology team. Among the googletest features, it is possible to found a rich set of assertions, user-defined assertions, death tests, fatal and non-fatal failures, value-parameterized tests, type-parameterized tests and XML test report generation.

3.5.2. Code quality metrics

It is difficult to give a general definition of code quality. Most developers agree that high quality code uses coding conventions, is readable and well documented, avoids duplication, provides an error management system, includes a tests environment, and provides security best practices. In this context particular attention is given to following aspect:

- **Maintainability**: represent a qualitative measurement of how easy it is to make changes in the code.
- **Duplication code:** indicates the part of code repeated in the software.
- **Testing suite:** Unit and Performance test provides an indicator of software functionality and requirement satisfaction.

3.5.3. Issue tracking system

Issue Tracking Software (ITS) is a software that manages and maintains lists of issues, as needed by an organization or developer team. Issue tracking systems are used in development work tracking to create, update, and resolve reported software issues. An issue tracking system is similar to a "bugtracker", and is considered one of the "hallmarks of a good software team".

3.5.4. Testing metrics

There are several test metrics that allow a software development team to evaluate its test quality strategy. During my thesis work particular attention is given to code coverage. Code coverage represent a percentage indicator that measures the amount of code evaluated during the tests.

3.6. From Continuous Integration to Continuous Delivery

Continuous Integration (CI) aims to improve the Scrum development methodology providing a frequent integration of all developer working copies into a shared repository, where also code quality is checked. In this context, Continuous Testing is just a step added to the CI. The CI has success if code quality and tests runned during the continuous test have success. Another step is the Continuous Delivery.

Continuous Delivery aims to ensure that the code is always in a deployed state, and that all code checked in to main branch of shared repository can be safely deployed into production.

The points presented in this paragraph represent some of the crucial points of the DevOps process. DevOps (Development and Operations) is a software development process that aims on collaboration and integration between developers and information technology operations. Figure 3.7 show a brief scheme of DevOps process.

3.6.1. Infrastructure as code

Infrastructure as code (IaC) is the process of managing and provisioning data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. The infrastructure could comprise both physical equipment, virtual machine or containers in definition files. The definition files may be in a version control system. It can use either scripts or declarative definitions, rather than manual processes, but the term is more often used to promote declarative approaches.

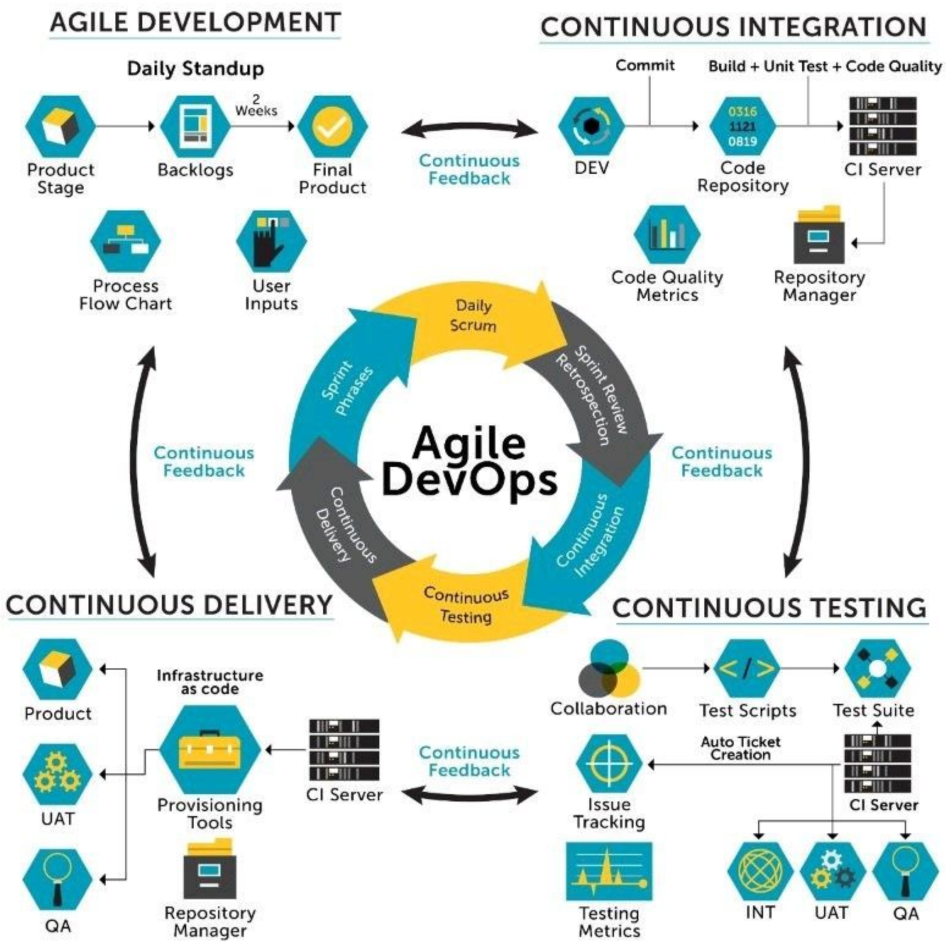


Figure 3.7: Agile DevOps representation

3.7. RTA software development implementation

The team in which I worked for this thesis, based its strategy and software development process on the principles of the Scrum and DevOps approaches previously described.

3.7.1. Working environment

A Linux CentOs user account has been given to each team developer allowing the developers to remotely log in to the INAF servers using the ssh protocol. The Git version-control system has been used. The developed projects have been stored in remote repositories on the GitHub web platform.

3.7.2. Development workflow

The RTA Scrum team is composed by a Scrum Master/Product Owner and four developers. The client is represented by INAF and the CTA Consortium.

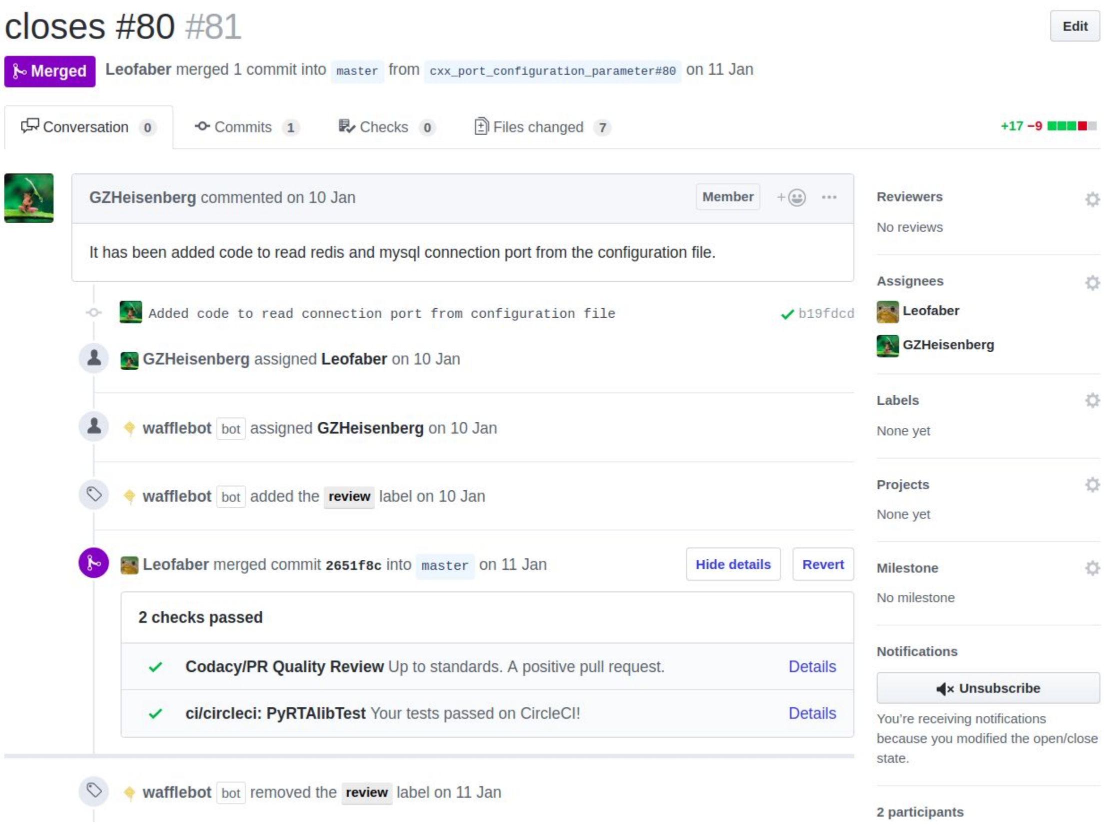
The Scrum planning phase has been managed through daily meetings of 30 minutes or 1 hour and weekly sprints. During the daily meeting held on Skype, each team member described the work done until that time, the problems he encountered and what task he would carry out next. If the daily meeting was the first meeting of the week, new tasks to be performed in the next weekly sprint, were identified by the Scrum Master.

A Scrum board has been used to organize and assign the tasks to the team members: a Scrum boards visually depict the iterations progress using cards to represent task items and columns to represent each stage of the process. Cards are moved from left to right to show Sprint progress and to help coordinate team performing the work.

The Waffle platform has been used as Scrum board implementation. The "RTAlib" board has been created composed by the following stages: "Inbox", "In Progress", "Review", "Done".

The RTAlib Waffle board has been linked with the RTAlib GitHub repository: this has allowed the automatic generation of a GitHub issue upon the creation of a new card. The card is therefore linked to the issue and they share the status: they are assigned to teams members and labeled with tags, milestones and manpower value expressed in story points (i.e. hours). The connection between Waffle and GitHub is reflected also into the process of starting the development of new cards/tasks: the beginning of a new task involves the generation of a new card that is hung on the "Inbox" column of the Scrum board and an unique identifier is automatically created and assigned to the card; then, a development branch must be opened and its name must contains an hashtag followed by the card ID. This connection allows the GitHub issue to be tracked down by the Waffle platform: the card is automatically moved into the "In Progress" column by Waffle itself. Once the development iteration is complete, the code is pushed into the remote repository and a pull request must be opened: if the name of the pull request contains the string “closes #<card-id>”, the card is automatically moved into the “Review” column.

Pull requests let team members to tell others about changes that will be merged into the master branch. Furthermore, the pull request can be used to trigger automatic code reviews: the code review highlights possible problems in terms of code quality and performance leaks. Once the problems have been solved, the code can be merged with the mainline of development by the Scrum master. In this step, conflicts between the new and old code can be arise and they must be resolved in order to merge successfully the pull request code into the master branch.


closes #80 #81

Merged Leofaber merged 1 commit into `master` from `cxx_port_configuration_parameter#80` on 11 Jan

GZHeisenberg commented on 10 Jan

It has been added code to read redis and mysql connection port from the configuration file.

`Added code to read connection port from configuration file` ✓ `b19fdcd`

GZHeisenberg assigned Leofaber on 10 Jan

wafflebot bot assigned GZHeisenberg on 10 Jan

wafflebot bot added the review label on 10 Jan

Leofaber merged commit `2651f8c` into `master` on 11 Jan

2 checks passed

✓ Codacy/PR Quality Review Up to standards. A positive pull request. Details

✓ ci/circleci: PyRTAlibTest Your tests passed on CircleCI! Details

wafflebot bot removed the review label on 11 Jan

Reviewers: No reviews

Assignees: Leofaber, GZHeisenberg

Labels: None yet

Projects: None yet

Milestone: No milestone

Notifications: You're receiving notifications because you modified the open/close state.

2 participants

Figure 3.8: GitHub pipeline

3.7.3. RTA Continuous Integration

Two technologies for continuous integration have been explored and implemented. In both solutions an integration with the GitHub servers has been setted up and therefore the automatic triggering of the unit testing process and the estimation of the corresponding code coverage.

The first solution uses the CircleCI platform and also exploits the code quality features provided by the Code Climate and Codacy tools.

The second solution implements a custom local continuous integration server with the Jenkins platform.

In both solutions the same unit test suite that was developed for this thesis work has been used. In particular the *googletest* framework to test C++ project and the *unittest* framework to test the Python version of the project. In Chapter 7 the unit testing for the C++ project is described.

3.7.3.1. Remote CI with CircleCI

In order to set up a continuous integration process with CircleCI, the first thing to do is to create a YAML file that describes how the testing environment must be setted up and which tests must be performed. The YAML file must be placed in the root folder of the project into an hidden folder called .circleci. The YAML file is a recipe that instructs the CircleCI server telling what steps must be performed in order to setting up a test environment. The CircleCI platform uses Docker containers to guarantee reproducibility and portability. In particular, in the first section of the YAML file, the name of a Docker image must be specified. For convenience, CircleCI maintains several Docker images. These images are typically extensions of official Docker images and include tools especially useful for CI/CD. For this purpose the circleci/python:3.6.6 Docker image has been used. Two other Docker containers have been applied as further layers on top of the first layer. Again, two images maintained by the CircleCI community have been used: circleci/mysql:5.7.23-ram and circleci/redis:4.0.10-stretch. The resulting environment contains the Python 3.6.6 interpreter and a MySQL and Redis database servers.

The next step in order to successfully deploy a CircleCI continuous integration process, is to define one or more workflows: a workflow is composed by Jobs that, in turn, are collections of Steps. All of the steps in the job are executed in a single unit which is executed within the Docker container. The jobs belonging to the same workflow can run in sequence or in parallel.
The "circleci_rtalib_test" workflow has been created running two jobs in parallel: the "CxxRTAlibTest"and "PyRTAlibTest" jobs.
Within the job, the first thing to do is to define the working directory, in which all commands, later defined, will be executed.

```
working_directory: ~/rtalib-circleci
```

Subsequently, it is necessary to define the Job implementation in terms of Steps: a collection of command to execute delimited by a key-value pair. The keywords are provided by CircleCI to personalize the workflow execution:

- **checkout:** checkout project source code from the working directory;
- **run:** it allows to specify a bash command to execute;
- **save_cache:** it allows to generates and stores a cache of files such as dependencies or source code to reuse later; it requires a key to identify those files;
- **restore_cache:** it restores a previously saved cache based on a key;
- **deploy:** a special step for deploying artifacts;
- **store_artifacts:** step to store artifacts as logs or binaries files;

The first commands of the Step are used to create some directories in which the software dependencies will be installed.

```
mkdir ~/rtalib-circleci/External_Libraries_Building_Area
mkdir ~/rtalib-circleci/CxxRTAlib_External_Dependencies_Installed
```

The next Job's Steps create the directories and files to store tests results.

```
mkdir ~/rtalib-circleci/test_artifacts;
touch ~/rtalib-circleci/test_artifacts/python_unittest_results.txt;
touch ~/rtalib-circleci/test_artifacts/cxx_unittest_results.txt;
touch ~/rtalib-circleci/test_artifacts/
      /python_codecoverage_results.txt;
```

Subsequently, the Steps to install the external dependencies are executed. Installing the numerous external library dependencies during every building process is very expensive. For this reason, the caching feature offered by CircleCI is used.
The “run” keyword is used to execute the bash commands to compile and install the packages inside the CircleCI server. Once the installation phase is completed, the "save_cache" keyword is used to associate a key to the installed packages.
This key will be used in subsequent builds by the "restore_cache" which will check if the key corresponds to a previously installed and working process in the cache. If it does not correspond, the installation commands are executed again.
Following the installed packages are listed:

- MySQL client
- MySQL C++ connector
- MySQL Python connector
- pipenv package
- hiredis client connector
- cmake software;
- Boost library;
- cfitsio NASA C library;
- googletest framework;

In Appendix 7.2 the commands to compile and install the packages are reported.
Now it’s possible to compile the C++ version of library developed for this thesis work, and to execute the unit test.

The unit tests generate a xml log report that can be sent to the CodeClimate service using the CodeClimate test-report tool: the informations about the code quality, the results of the unit tests and the unit test coverage percentage of the developed code will be reported on the CodeClimate custom web page of the project. The test logs can be also stored in CircleCi through the artifact_store keyword.

3.7.3.2. Local CI with Jenkins

Jenkins can be installed through native system packages, Docker, or even run standalone by any machine with a Java Runtime Environment (JRE) installed. The latter solution has been chosen. The Jenkins source code has been downloaded and installed through yum, with the commands reported in Appendix 7.1. A new Jenkins project can be setted up and configured through the Jenkins GUI. The project creation process and its configuration are showed in Appendix 7.2 .

3.7.3.2.1. The pipeline

Jenkins is, fundamentally, an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive CD pipelines. The definition of a Jenkins Pipeline is written into a text file (called a Jenkinsfile) which in turn can be committed to a project’s source control repository. This is the foundation of "Pipeline-as-code"; treating the CI/CD pipeline a part of the application to be versioned and reviewed like any other code.

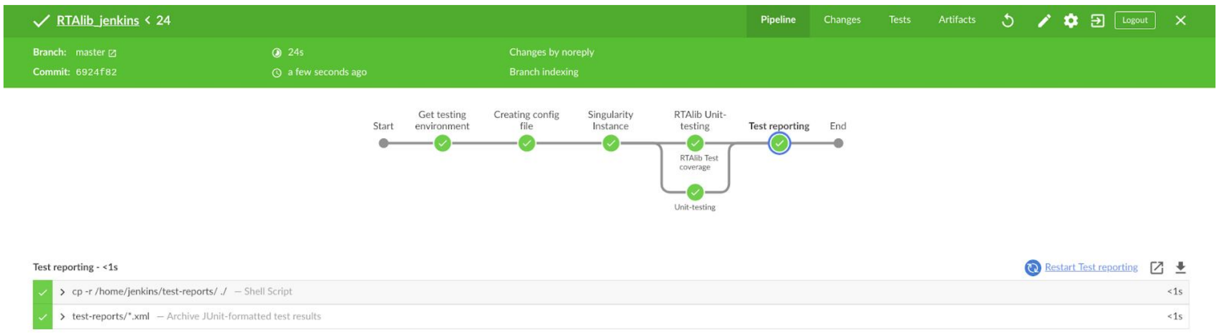
Jenkinsfile example:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        //
      }
```

```
        }
        stage('Test') {
            steps {
                //
            }
        }
        stage('Deploy') {
            steps {
                //
            }
        }
    }
}
```

The example shows the structure of a CD Jenkins pipeline: stages are used to define processes that are made by multiple steps.

The Blue Ocean plugin has been installed. Blue Ocean is a project that rethinks the user experience of Jenkins, modelling and presenting the process of software integration and delivery with a new UI that aims to improve clarity, reduce clutter and navigational depth to make the user experience very concise.[25] It also gives the possibility to create Jenkins pipelines in a graphical and interactive way. The Blue Ocean UI has been used to create the continuous integration pipeline for the software developed for this thesis work. In the Appendix 7.3 the Jenkinsfile and the build process configuration is described.



```
Branch: master          24s                   Changes by noreply
Commit: 6924f82         a few seconds ago     Branch indexing

Test reporting - <1s
> cp -r /home/jenkins/test-reports/ ./  — Shell Script                        <1s
> test-reports/*.xml  — Archive JUnit-formatted test results                  <1s
```

Figure 6.9: Jenkins pipeline

The Jenkins pipeline developed for this thesis work, uses a Singularity image. In the next paragraph, the purpose of the Singularity image will be described.

3.7.3.2.2. The Singularity image

There are three main reasons that led to the development of a Singularity container: the first reason is to have a reproducible environment in which the software developed for this thesis work could be tested. The second is that the container environment it can be the same environment for the production phase of the software. The third reason, is about Continuous Delivery: if the unit testing (that are executed automatically thanks to the Continuous Integration pipeline) executes successfully inside the container, the container can be delivered and used in production. In addition, the container keeps all the required dependencies of the software, decreasing the time requirement of a Jenkins build process (no need to reinstall the dependencies every time).

The process to develop a Singularity container is composed by several steps: the first step is to create a Singularity Recipe. Then, the recipe must be built in order to create a Singularity Image. Finally the Singularity image can be used as a container in four different ways: executing a simple command inside the container, executing a shell inside the container, running the container or executing the container as a stand-alone container instance.

In the Appendix 7.3.1 the process with which the Singularity image has been obtained is described.

4. RTAlib

The aim of this chapter is to describe the problematics behind the CTA data handling and how the RTAlib software framework developed for this thesis can enable the Real-Time Data Analysis in the "On-Site" area of CTA.

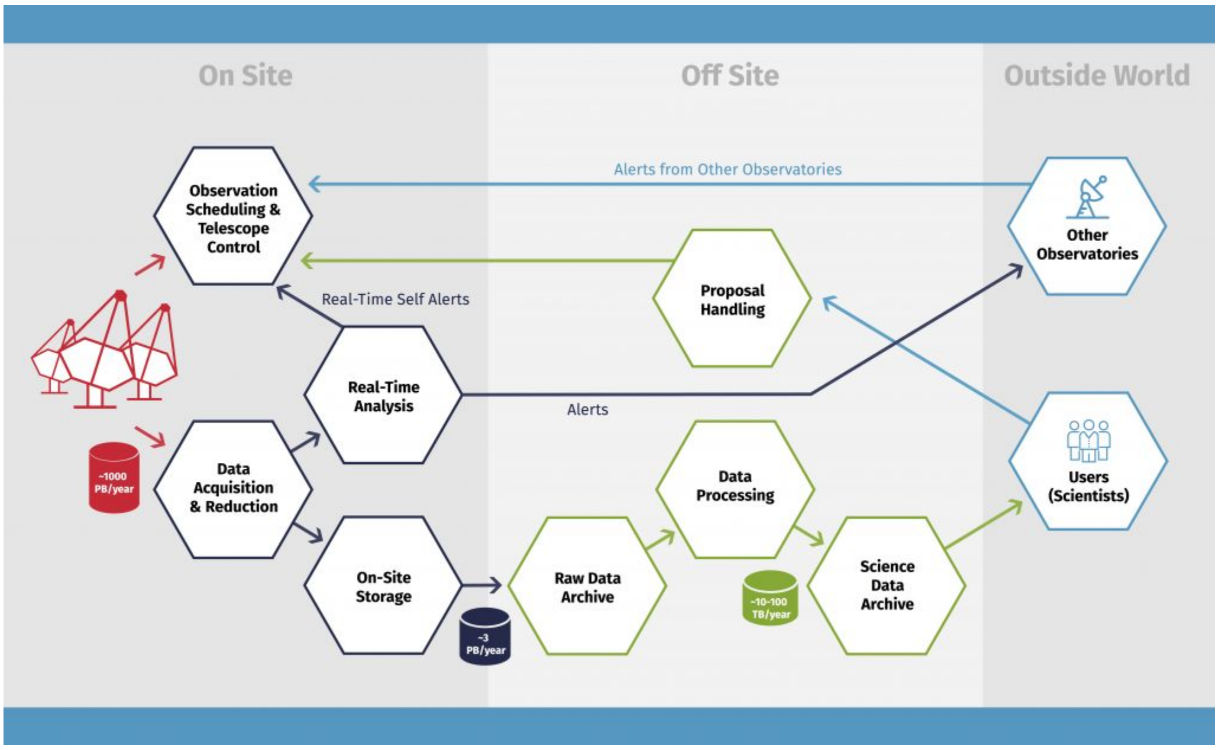


Figure 4.1: CTA Data Flow

4.1. CTA On-Site Analysis

The CTA On-Site Analysis is an hardware and software system for the analysis of scientific data of the Cherenkov events and for the performance/data quality monitoring of the telescopes cameras that will be performed on the same site of the telescopes.

CTA is expected to generate a large data rate, due to the large number of telescopes, the amount of pixels in each telescope camera and the fact that several time samples are recorded in each pixel of the triggered telescopes. Depending on (i) the array layout, (ii) the data reduction scheme adopted, and (iii) the trigger criteria, the data rate estimate vary from 0.5 to 8 GiB/s.

The On-Site Analysis is composed of two pipelines:

1. Real-Time Analysis pipeline;
2. Off-line Analysis pipeline.

Both the pipelines have the dual purpose of delivering science feedback from the CTA data and monitoring the instruments performance and the data quality.

Both the pipelines read raw data, apply calibration algorithms, reconstruct Cherenkov events and produce event list. The focus of the Real-Time Analysis is to maximize the speed of the analysis while the focus of Off-line Analysis is to maximize the sensitivity.

The Real-Time Analysis enables CTA to provide real-time feedback to externally received alerts and to maximize the science return on time-variable and transient phenomena by issuing an alert of unexpected events from astrophysical sources.

When the telescopes and the Real-Time Analysis switch off, the Off-line pipelines should run and provides science feedback with greater sensitivity.[10]

4.2. Real-Time Analysis

The Real-Time Analysis pipeline has the following components:

1. the Real-Time (RT) Reconstruction components, that perform a fast reconstruction of the acquired events;
2. the RT Science Alert Monitoring components, that detect unexpected astrophysical events and generate Science Alerts;
3. the RT Data Quality (DQ) Monitoring components, that perform a basic data quality check to evaluate the correct execution of the observations in real-time. This system generates RT DQ Alerts and Reports (statistics, graphs, and summaries of the data generated by the real-time analysis).[26]

Moreover, the Real-Time Analysis pipeline must be able to receive external alerts from the telescopes all around the world and in space, in order to evaluate the events that triggered the external alert.

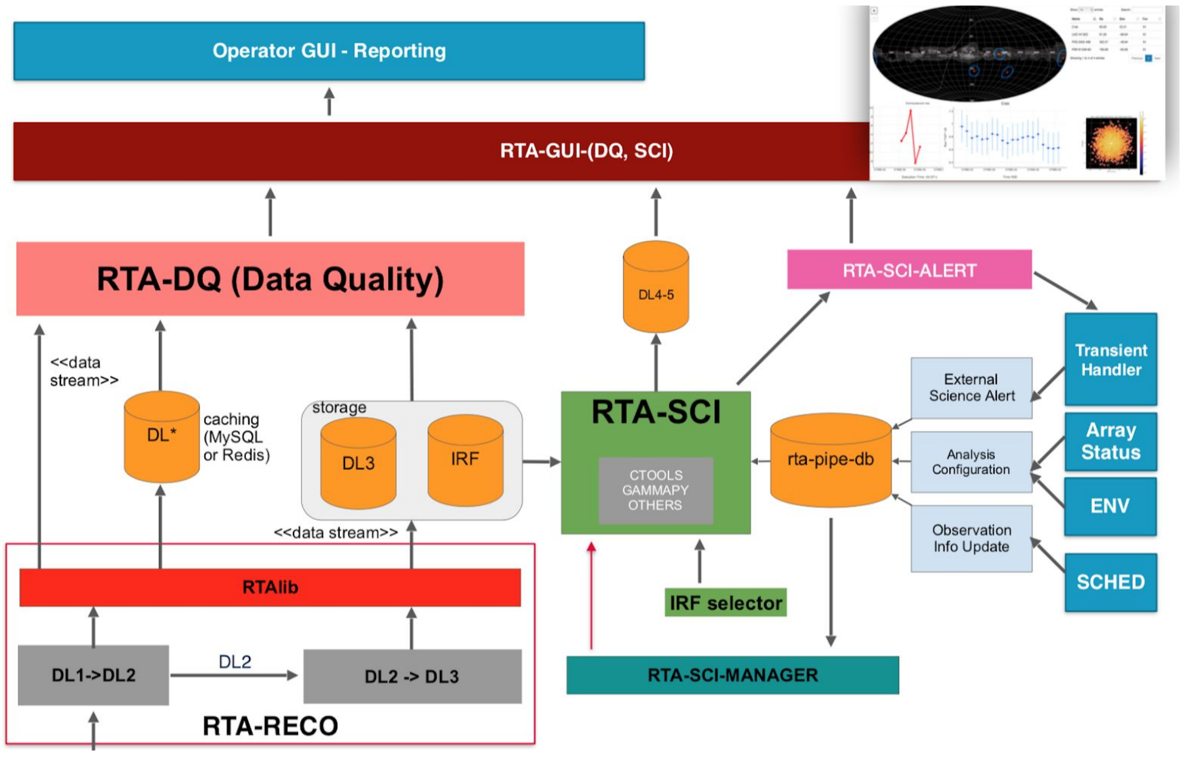


Figure 4.2: Functional logical view of the CTA On-Site Real-Time Analysis (courtesy of INAF RTA Team)

4.3. Data Flow And Data Definition

CTA uses the Imaging Atmospheric Cherenkov Telescopes (IACTs) technique to observe gamma rays that cause particle showers in the Earth's upper atmosphere. Large telescopes with ultraviolet-optical reflecting mirrors focus flashes of Cherenkov light onto cameras.[9] Starting from the camera data acquisition, CTA pipeline handles different data type obtained as the result of different elaboration step. Generally, the size of data at a particular data level tends to decrease compared to the previous one.

In the following list, each data level is presented along with a short description for each one:

- **R0 (raw low-level):** camera data transmitted from the telescope to central servers. R0 content and format is internal to each camera and is specified and coordinated between individual camera teams.
- **R1 (raw common):** data transmitted on-site by a camera or device in a common format, not intended for archival storage. This is the first level of data seen by the

Observation Execution System (OES) and is therefore as common as possible between all cameras/hardware/devices. Exceptionally, some R1 data may be stored for engineering purposes.

- **DL0 (raw archived):** all archival data from the data acquisition hardware/software. This is the first level of data that are stored in the bulk archive. This includes both camera event data and technical data from other subsystems, such as non-camera devices or software.
- **DL1 (processed):** processed DL0 data that may still include some telescope data and parameters derived from them. For example, this includes a calibrated image charge, Hillas parameters, and a usable telescope pattern. This is only optionally stored in the archive. Expected data reduction factor about 1 - 0,2.
- **DL2 (reconstructed):** reconstructed shower parameters such as energy, direction, particle ID, and related signal discrimination parameters. At this point, no telescope information is stored. For each event, this information may be repeated for multiple reconstruction and discrimination methods. This is only optionally stored in the archive. At this point, telescope-wise info is generally dropped. Expected data reduction factor about 0,5.
- **DL3 (reduced):** Sets of selected (e.g. gamma-ray candidates, electron candidates, selected hadron candidates, etc.) events with a single final set of reconstruction and discrimination parameters, along with associated instrumental response characterizations and any technical data needed for scientific analysis. Expected data reduction factor about $10^{\wedge-2}$.
- **DL4 (science):** binned data products like spectra, sky maps, or light curves, along with associated data (source models, fit results, etc). Expected data reduction factor about $10^{\wedge-3}$.
- **DL5 (high-level):** high-level or "legacy" observatory data, such as CTA survey sky maps or the CTA source catalogue. Expected data reduction factor about $10^{\wedge-5}$ - $10^{\wedge-3}$.[27]

The Reconstruction pipeline (RECO) is scientific analysis software, devoted to reconstruct the physical characteristics of the astrophysical γ-ray and background cosmic rays. In the first phase, the calibration task processes the DL0 data to evaluate its temporal evolution and performs a first data separation according to the events arrival time. After this process, the camera image is reconstructed and is available for further analysis. The Hillas parameters[3] are also computed from the camera image.

The reconstruction step takes the result data (DL1) of the previous one and use machine learning algorithms to provide a parametrization of each event (DL2). This data, fully reconstructed, are further processed in order to obtain the fully reduced data (DL3). In this last analysis step, gamma events are separated from hadron events, also possible ones responsible for the Cherenkov flashes.

The DL3 data are subsequently analysed with science tools to provide the final scientific product as the light curve, sky-maps, detection plots and more (DL4). These products, if needed, can be further processed to get the high-level observatory data (DL5).

4.4. RTAlib

The RTAlib, developed for this Thesis work, is a software framework that is part of the Real Real-Time analysis system prototype that has been developed by the AGILE team of the INAF/OAS of Bologna. The main purpose of the RTAlib is to provide a simple but powerful high-performance API to store the data that is generated during the reconstruction and analysis phase into the Real-Time Analysis database. In order to cope with the high data rates of the CTA reconstruction and analysis chain, the RTAlib exploits multi-threading and database transactions. All these functionalities can be configured and tuned through a configuration file.

4.4.1. Software Requirements

The RTAlib must provide cache and storage features for DL1, DL2 and DL3 data layers.

[3] Parameters used to model images of gamma showers

The size of a DL1 event is about 30 kilobytes, the expected DL1 data event rate is about 6 kHz (expected) until 10 kHz (worst case) and the corresponding data rate is about 300 MB/s in the worst case.

The DL2 data is devoid of the telescope information and the image payload, so its weight decrease to about 300 bytes and with the same event rate its data rate is about of 3MB/s.

The only interesting DL3 data is the only one that represents the gamma events, so its weight is about of 300 bytes while the rate decrease about to 30 kB/s .

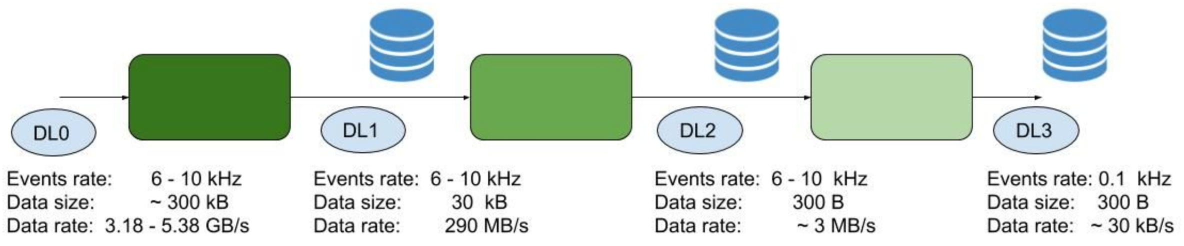


Figure 4.3: RTAlib Event and Data Rate

4.4.2. Implementation

Two versions of the RTAlib have been developed, the PyRTAlib and CxxRTAlib, the first one written with Python 3.6 programming language while the second one has been written with the C++11 version programming language.

The Python version was developed to obtain as fast as possible the first working prototype within the highly competitive CTA context. However, the project requires a high data rate handling capability that Python can't satisfy, and for this reason, and also because some RECO codes provided by CTA Team are written in C++, it has been developed the C++ version. In any case a lot of CTA code is written in Python and a future wrapper of the C++ version of this library will be provided.

The software was designed using the object-oriented modelling paradigm and the software's functionalities were grouped in feature areas. Figure 4.4 shows RTAlib software architecture. The next sub-paragraph will describe each feature area.

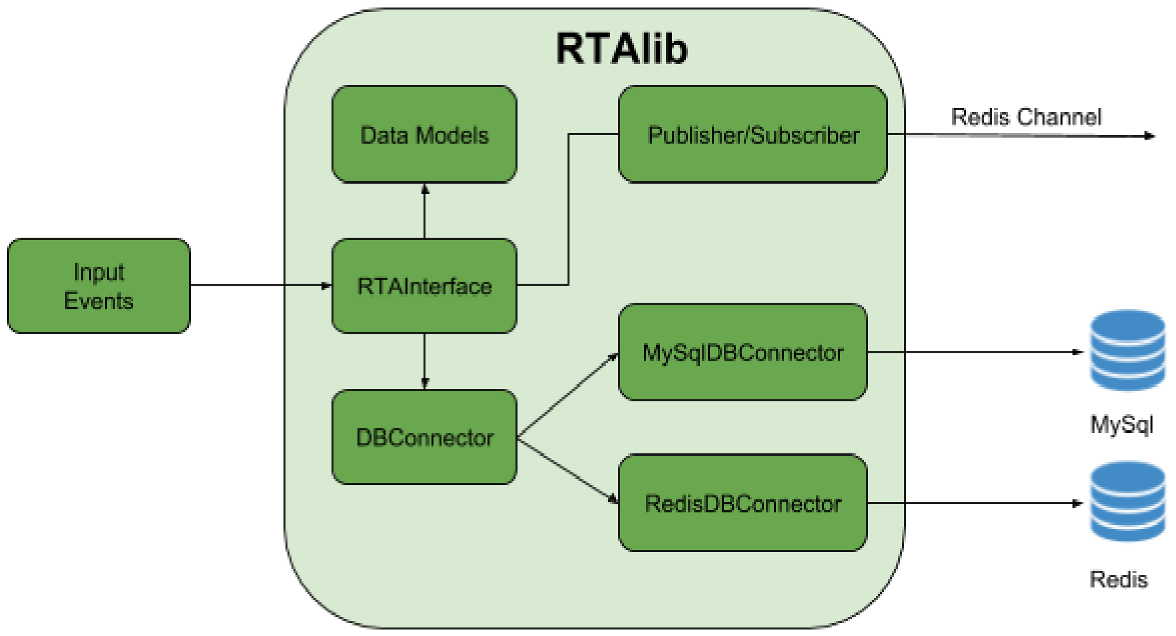


Figure 4.4: RTAlib software architecture

4.4.3. Physical and logical data models

From the High-Level CTA data definitions (see Data Definition) it has been extracted the data representation.

Within the MySQL environment, every data layer is represented as a MySQL table and every table has its own structure defined in accordance with the High-Level Definition. On the other hand, within Redis environment, every data layer is represented as a sorted set of key-value. The generic class EVTbase represent a generic event. Then, for each data layer, a specialized class has been developed, inheriting from the EVTbase class.

Figure 4.5 shows the UML model representing the "EVT" classes: different data layers have a different number of parameters and so a different size. The DL1 data is represented by 39 single values and 2 matrices for the images. Each parameter is encoded with a variable of 8 bytes and each image consists of a matrix made of 1849 8-byte variables for a total weight of about 30 kB. The DL2 according to the High-data definition and software requirements is represented by the same number of DL1 parameters without the two matrices. The DL3 is is composed of the same number of parameters of DL2, but in this case the difference is in the event rate. Within the data layer classes the events are represented by map<string, string> template. The first string variable contains the parameter name, the second one the parameter value.

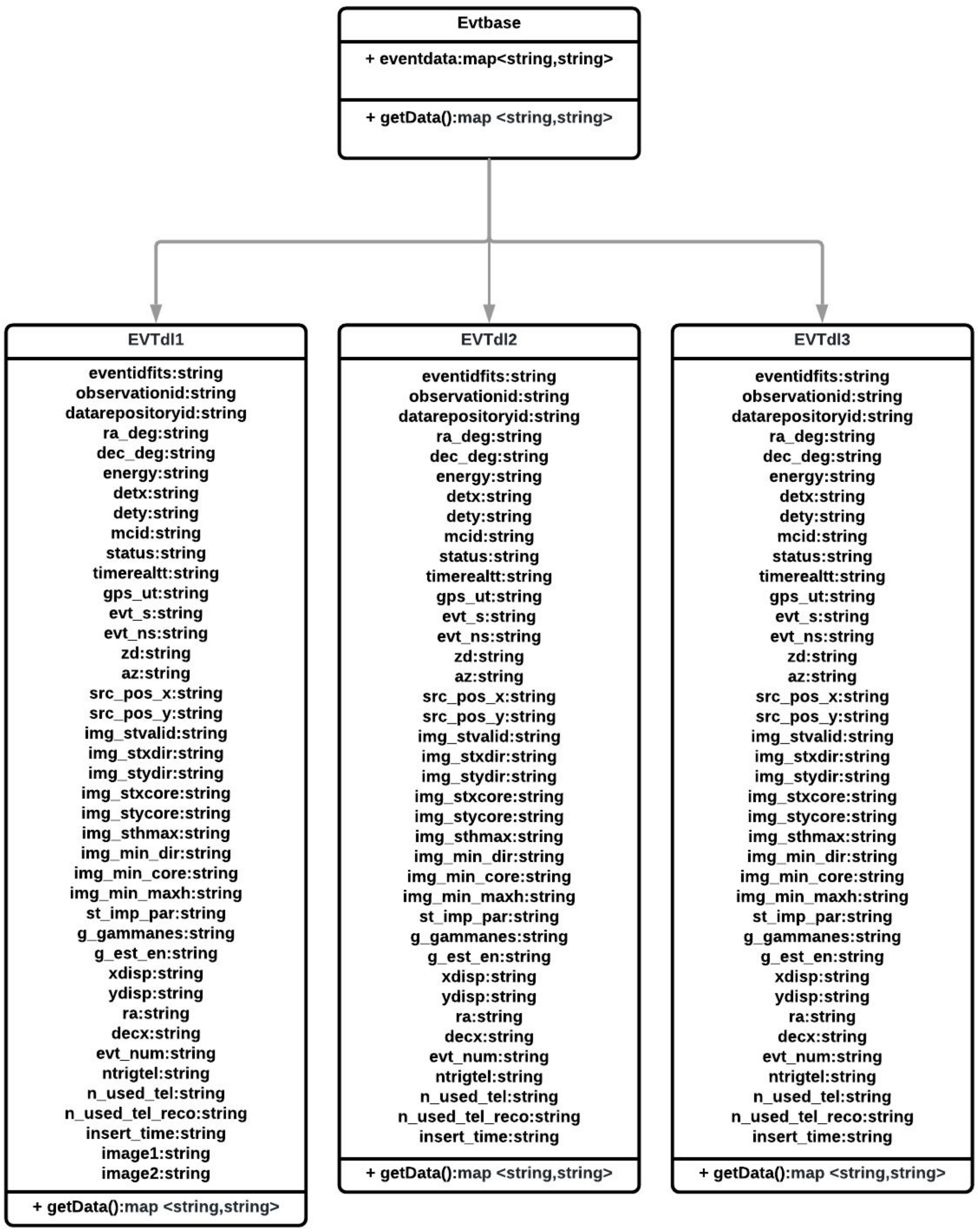


Figure 4.5: EVT class UML diagram

4.4.4. The database connectors

A generic DBConnector class has been created to interface with the databases. Subsequently, two specific classes have been inherited from this class, one to interface with MySQL and one to interface with Redis. Figure 4.7 shows the UML model representing the classes for the database interaction.

4.4.5. The core class

The RTAlib core logic is encapsulated into the RTA_DL_DB abstract class which handles the storage and the streaming functionalities in according to the use cases. The class, during its initialization, reads a configuration file in which are contained several parameters. Some of those are connection parameters while the other are used to decide in which way the data is stored in the database: as said before, the class can use several threads and store the data with a streaming or batch fashion. In the "Configuration" paragraph will be described those parameters.

In order to use the RTAlib, for each data layer a specific RTA_DLx_DB class must be created, inheriting from the RTA_DL_DB class. The child class must contain an insertEvent method that accepts the DLx-specific parameters as input. It must build also the corresponding EVTx event object and pass it to the super class method _insertEvent(EVTbase).

In order to store event data in the database, RTA_DL_DB interacts with the DBConnector, a generic class that provides the interface for MySQL or Redis databases. The MysqlDBConnector and RedisDBConnector inherited classes exposes functions to open and close the database connection, create the queries and execute them with or without transactions.

On the other hand, in order to stream the events data through Redis channels, RTA_DL_DB interact with RedisPublisher class that uses Redis Publisher/Subscriber messaging paradigm.

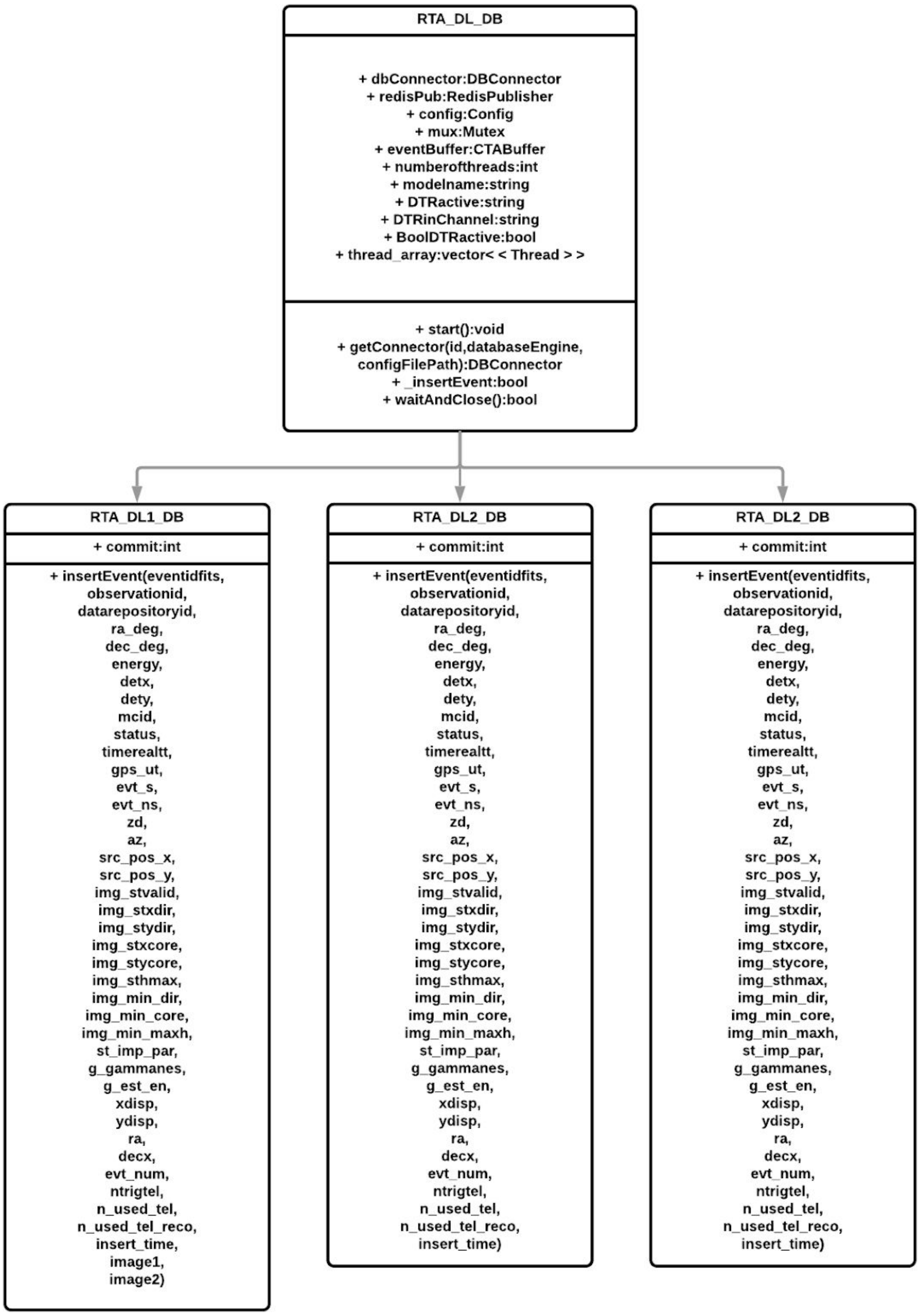

RTA_DL_DB
+ dbConnector:DBConnector
+ redisPub:RedisPublisher
+ config:Config
+ mux:Mutex
+ eventBuffer:CTABuffer
+ numberofthreads:int
+ modelname:string
+ DTRactive:string
+ DTRinChannel:string
+ BoolDTRactive:bool
+ thread_array:vector< < Thread > >
+ start():void
+ getConnector(id,databaseEngine, configFilePath):DBConnector
+ _insertEvent:bool
+ waitAndClose():bool
RTA_DL1_DB
+ commit:int
+ insertEvent(eventidfits, observationid, datarepositoryid, ra_deg, dec_deg, energy, detx, dety, mcid, status, timerealtt, gps_ut, evt_s, evt_ns, zd, az, src_pos_x, src_pos_y, img_stvalid, img_stxdir, img_stydir, img_stxcore, img_stycore, img_sthmax, img_min_dir, img_min_core, img_min_maxh, st_imp_par, g_gammanes, g_est_en, xdisp, ydisp, ra, decx, evt_num, ntrigtel, n_used_tel, n_used_tel_reco, insert_time, image1, image2)
RTA_DL2_DB
+ commit:int
+ insertEvent(eventidfits, observationid, datarepositoryid, ra_deg, dec_deg, energy, detx, dety, mcid, status, timerealtt, gps_ut, evt_s, evt_ns, zd, az, src_pos_x, src_pos_y, img_stvalid, img_stxdir, img_stydir, img_stxcore, img_stycore, img_sthmax, img_min_dir, img_min_core, img_min_maxh, st_imp_par, g_gammanes, g_est_en, xdisp, ydisp, ra, decx, evt_num, ntrigtel, n_used_tel, n_used_tel_reco, insert_time)
RTA_DL2_DB
+ commit:int
+ insertEvent(eventidfits, observationid, datarepositoryid, ra_deg, dec_deg, energy, detx, dety, mcid, status, timerealtt, gps_ut, evt_s, evt_ns, zd, az, src_pos_x, src_pos_y, img_stvalid, img_stxdir, img_stydir, img_stxcore, img_stycore, img_sthmax, img_min_dir, img_min_core, img_min_maxh, st_imp_par, g_gammanes, g_est_en, xdisp, ydisp, ra, decx, evt_num, ntrigtel, n_used_tel, n_used_tel_reco, insert_time)

Figure 4.6: RTA_DL_DB class UML diagram

4.4.5.1. MySQL connector class

The MySqlDBConnector class use the MySQL Connector/C++ 8.0 that provides interface and code to communicate with the MySQL server. This latest version of the connector is highly recommended by the MySQL community and use X devAPI that provides a high-level interface easy to use and supports NoSQL document storage.

The Connector/C++ 8.0 consist of a complete re-implementation of the connector creating a good foundation for future innovation and improvements. This new implementation is based entirely on the new X Protocol of MySQL Server 8.0 that allows rapid and highly modular development. On previously server version then 8.0, X Protocol has to explicitly enable.[28]

4.4.5.2. Redis connector class

The RedisDBConnetor class represent a C/C++ wrapper to allow the use of the hiredis C library. The RedisDBConnector class inherits the DBConnector class methods and structure and it wraps the Hiredis library.

Hiredis is a minimalistic C library written by the Redis developer, Salvatore Sanfilippo, that provides all features to operate with Redis Server.

4.4.6. Redis publisher class

The purpose of the RedisPublisher class is to provide a simple method to allow the use of the Redis publisher mechanism. Using this function, the RTAlib can send event data asynchronously to other RTA software module in a streaming fashion. The publisher mechanism requires to be enabled setting to “yes” the corresponding field in the configuration file, where it are also contained the database connection parameters and the output channel name.

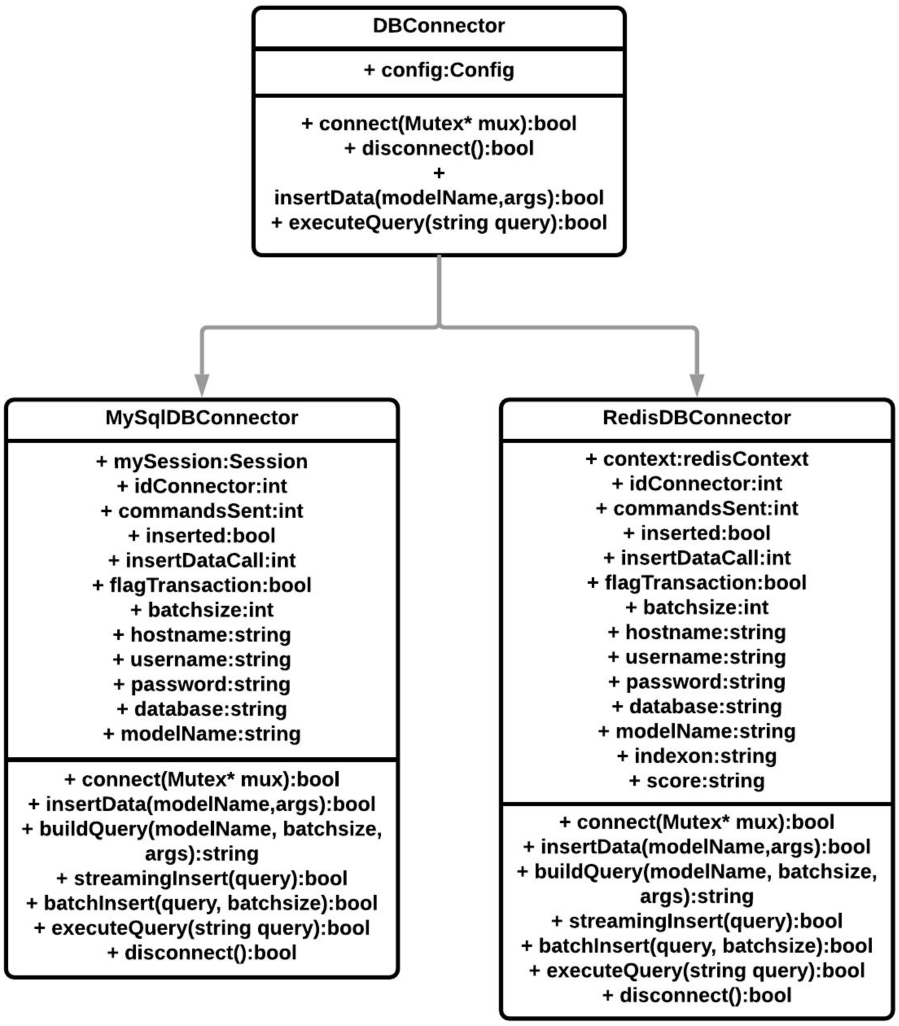


Figure 4.7: DBConnectors UML class diagram

4.4.7. Configurations

The .ini file format has been chosen for the configuration file. The basic element contained in a .ini file is a property. Each property is represented by a “key = value” pair. This pair can be grouped in a “Section”, every section begins with a section header containing the section name in the square brackets “[SectionName]”.[29] In the following is reported the configuration file skeleton.

```
[General]
modelname=
mjdref=
debug=
batchsize=
numberofthreads=

[DTR]
active=
debug=
inputchannel=

[MySql]
host=
port=
username=
password=
dbname=

[Redis]
host=
port=
password=
dbname=
indexon=
```

In the following paragraphs are described the strategies the RTAlib can use to efficiently store the event data in the database.

4.4.7.1. Streaming mode

The batch size parameter of the configuration file specifies how many events are stored in a single query. When this value is equal to one, the RTAlib performs the streaming strategy, executing an "insert" query for each single event.

4.4.7.2. Batch mode

When the batch size is different from one the RTAlib performs the batch strategy: RTAlib waits a number of events equal to the batch size value before starting the storing process. In this configuration, RTAlib uses transaction operations for the InnoDB and the Redis engine.

4.4.7.3. Single thread

It is possible to set the number of threads that RTAlib can use. When the number of thread is equal to one the RTAlib executes in “synchronous” mode: when an event is captured, RTA_DL_DB interacts directly with the database connectors API and only when the storing process is ended the RTAlib can handle a new event. This is the simplest execution flow.

4.4.7.4. Multithreading

When the number of threads is greater than one, RTAlib is executed in “asynchronous” mode. In this case, RTA_DL_DB class creates as many threads as specified in the configuration file and each thread instantiate its DBConnector to open a connection with the database. When an event is captured, it is inserted in a circular buffer and all threads concurrently access to the buffer, taking the events out and inserting them into the database.

4.4.8. Unit testing

Googletest allows to define several test case as a collection of tests that check the correct functioning of a code unit. At present the following test cases have been defined within CxxRTAlib :

- MYSQL TEST CASE
- REDIS TEST CASE
- RTA_DL_DB TEST CASE

Several TESTs have been defined within each Test case. Within each test, ASSERTION are used to verify that a particular code condition is satisfied or not.

4.4.9. DTR

The DTR class is a stand-alone Python daemon that implements the publisher/subscriber mechanism. Its purpose is to act like a router: the DTR subscribes to one or more channels, it receives all messages sent on those channels, it caches the messages in a circular buffer and it publishes them to output channels. The data payload contained in those messages can be read and transformed before it is forwarded into the output channels: the DTR can use particular classes, each one inheriting from the DTR_Transformer class, that contains custom logic for data transformation.

Internally the DTR class is composed by a main thread which purpose is to listen the Redis input channel for messages and to put each arrived messages into a circular buffer. Another thread pops out the messages from the buffer, it transforms the message payload and finally it sends the transformed messages into output channels. This producer-consumer approach has been chosen for one main reason: the need to separate the subscriber logic from the data transformation logic; the main thread is exclusively dedicated to the messages retrieving with a very high reading rate and to the buffer message caching: this allow not to lose any incoming message. In the Figure 4.8 shows the DTR architecture.

During this thesis work, the DTR system has been used to perform some performance measure of the RTAlib software. In particular, DTR_Transformers classes have been used as a “database readers” to allow the “Write&Read” scheme to be implemented and measured in terms of performances (see Cap 5).

The DTR class require a dedicated configuration file in which are specified the database connection parameters and the names of the input and output channels.

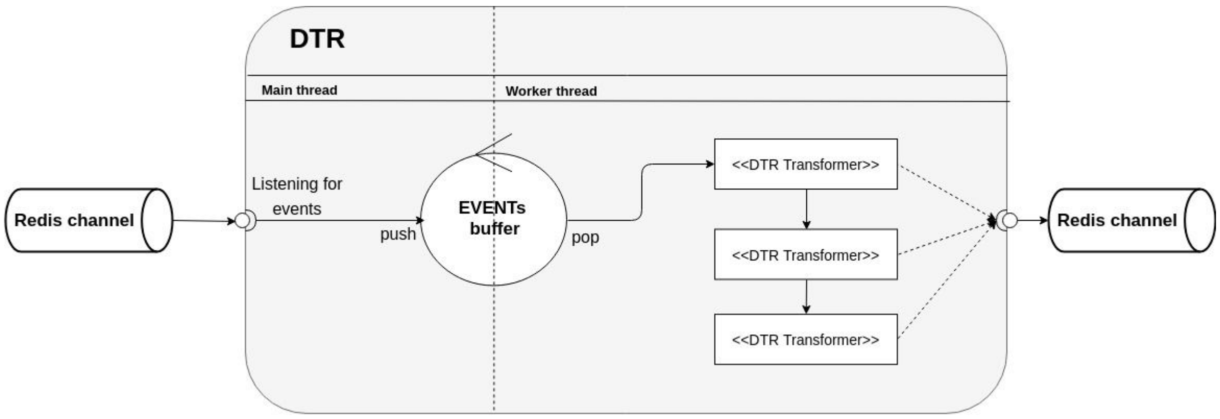


Figure 4.8: DTR architecture

5. Performance

In this chapter will be presented the performance test environment, different test schemes and the performance achieved by RTAlib.

5.1. Test Environments

The performance tests were executed on a INAF/OAS server. In the next lines, the description of the hardware characteristics of the server is reported:

- **2 x Xeon 18-Core E5-2697v4 2,3Ghz:** 18 Cores, 36 Threads. FCLGA2011 Socket. 45M Cache. 2,3 GHz Frequency. 9,6 GT/s QPI Speed. 145W max.
- **4 x DDR4-2400 Reg. ECC 32 GB module.**
- **1 x 13734 LSI 9361-8i RAID 12Gb/s SAS/SATA 8Port PCI-EX:** MegaRAID SAS 9341-8i. x8 PCI Express 3.0 host interface. RAID 0, 1, 5, 10, 50 and JBOD mode. Low-profile form factor.
- **4 x HGST 4TB SATA III 7.200 RPM 128MB 512E:** HGST Ultrastar 7K6000 hard disk drive. Form factor: 3,5''.
    - Capacity: 4TB. Interface: 512e SATA 6Gb/s.
    - Buffer: 128MB. Rotational Speed: 7200RPM.
    - Transfer rate: 600MB/s (max). 2Million-hour MTBF.

On this hardware platform the following softwares have been installed:

- **CentOS:** CentOS Linux is a community-supported distribution derived from sources freely provided to the public by Red Hat for Red Hat Enterprise Linux (RHEL). As such, CentOS Linux aims to be functionally compatible with RHEL. A new CentOS version is released approximately every 2 years and each CentOS version is periodically updated (roughly every 6 months) to support newer hardware. This results in a secure, low-maintenance, reliable, predictable and reproducible Linux environment.[30]
- **MySql Server:** Version 14.14 Distribution 5.7.24.

- **Redis Server:** Version 3.2.12.
- **GCC 4.8.5:** C and C++ GNU compiler.
- **anaconda-3.6:** Anaconda is a package manager, an environment manager, a Python/R data science distribution, and a collection of over 1,500+ open source packages.[31] Within Anaconda environment has been installed Python 3.6.7.

The MySQL and the Redis database services have been installed within a Singularity 2.6.1 container.

5.2. Task and criteria

The aim of the tests performed with the RTAlib is to compute the performances obtained by the developed code.
The ultimate goal is to verify if the RTAlib can satisfy the software requirements described in Chapter 3. In particular, the main observable is the Event-Rate (i.e. the number of events per second) of the database storing process.
The results obtained from these preliminary tests, highlights the critical issues of the RTAlib data flow and its strengths. The future developments of the RTAlib will be based on those hints.
It has been identified a parameters space in order to explore the greatest number of possible configurations. In the following lines are listed all the possible elements that compose the parameters space:

- RTAlib version = {C++, Python}
- HW virtualization = {host2, container}
- DB = {mysql, redis}
- MEMORY = {RAM, SSD}
- Number of writers = {1, 4}
- Number of threads = {0, 2, 9, 18, 36}

In a first phase, the performance tests have been performed only in the host environment. Furthermore, two types of tests have been implemented:

- one-shot;
- long-run;

The "one-shot" tests are short-lived tests, consisting of the storage of 50,000 or 100,000 events that correspond roughly to 5 or 10 seconds of real observation. This kind of test provides a means of evaluating the scalability of the two RTAlib versions (C++ and Python).
The "long-run" tests simulate about one hour of real observation, performing the storage of 43.200.000 events. This kind of test can evaluate any performance degradation due to “continuous stress" over time.

5.3. Features to be tested

According to the CTA data definition and data flow specifications, two main test cases have been identified:

1. Caching test case;
2. Permanent storage test case;

The caching use case is particularly useful to handle the DL1 data. Due to its size (30 kB) the RTAlib and CTA in general points to non-permanent storage of this data. In addition, other data layers different from DL1, can require caching in some stages of reconstruction or data analysis.
The permanent data storage is closely required by the data definition and project requirement provided from CTAO. Not all data layers need to be permanently stored but in our context both DL2 and DL3 data layers require this functionality. As previously mentioned in Chapter 4.3, Data Flow and Data Definition, the DL2 and DL3 data layers have the same size (312 bytes) but a different event rate. In particular, the DL2 has an higher event rate (10 kHz of the DL2 against 0.1 kHz of the DL3). Satisfying the project requirement for the DL2 data layer implies to automatically satisfy the DL3 requirement too. For this reason, the DL3 performance tests were not executed and, in the next paragraphs, the DL2 tests can be taken as a reference for the DL3 performance requirements.

The “caching” and “permanent storage” test cases have been first executed in “WriteOnly” mode and subsequently in “Write&Read” mode. The latter, represents the real use case. In order to run the caching test case, the MySQL Memory engine and Redis (in memory by default) have been used. On the other hand, to test the permanent storage test case the MySQL MyISAM and InnoDB engines have been used.

5.4. Implementation of the performance tests suite

5.4.1. WriteOnly Test

In order to execute the “WriteOnly” performance test, an event simulator has been implemented. The simulator can generate random events that are stored into a c++ vector. Then, the vector is iterated and the RTAlib API is called in order to store the events in the database. The performance test code accepts the following input parameters:

- Database;
- TableName;
- NumberOfEvents;
- NumberOfIterationPerTest;
- ConfigurationFilePath;
- CleanerTrigger;

The first input parameter, “Database”, specifies which DBMS will be used. The “Tablename” parameter allows to choose the name of the table and therefore the logical model of the data to be used. The “NumberOfEvent” parameter indicates the number of events to be simulated and then to be inserted into the database. The “NumberOfIteration” parameter indicates how many times to run the same test. Performing multiple test iterations is mandatory to calculate average values and obtain an estimation of the error, calculated as standard deviation. The last parameter establishes whether or not performing the test in the ideal condition of an empty table.

The performance test code generates a report in which the following performance results are collected:

- Number of inserted events
- Number of used threads
- Batch size
- Event rate
- Event rate error
- Execution time
- Execution time error
- Data rate

The "WriteOnly" tests were the ones carried out first. They were performed using a single writing process by varying the number of threads used by the RTAlib to perform the events storage. Each thread instantiates its own DBConnector object. From now on, this configuration will be referred to "scheme 1".

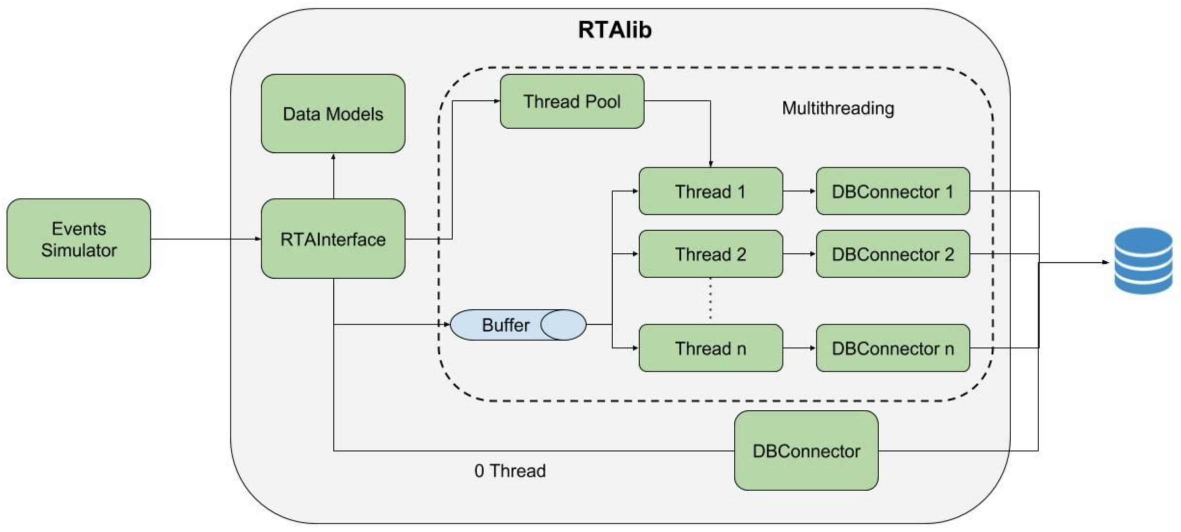


Figure 5.1: scheme 1 "WriteOnly" with a single process

Another "WriteOnly" scheme has been implemented. It was inspired by the LST-1 telescope research team. During a meeting, the LST-1 team showed us how the data captured by the telescope will be transported from the camera server to the "On-site" server: four fibre-optic lines will be used in order to guarantee the high data rates requisites of the project. From the sharing of this knowledge, scheme 2 is born. Figure 5.3 shows the schema 2: four writing processes execute in a parallel way. Furthermore, each process can adopt a multithread strategy.

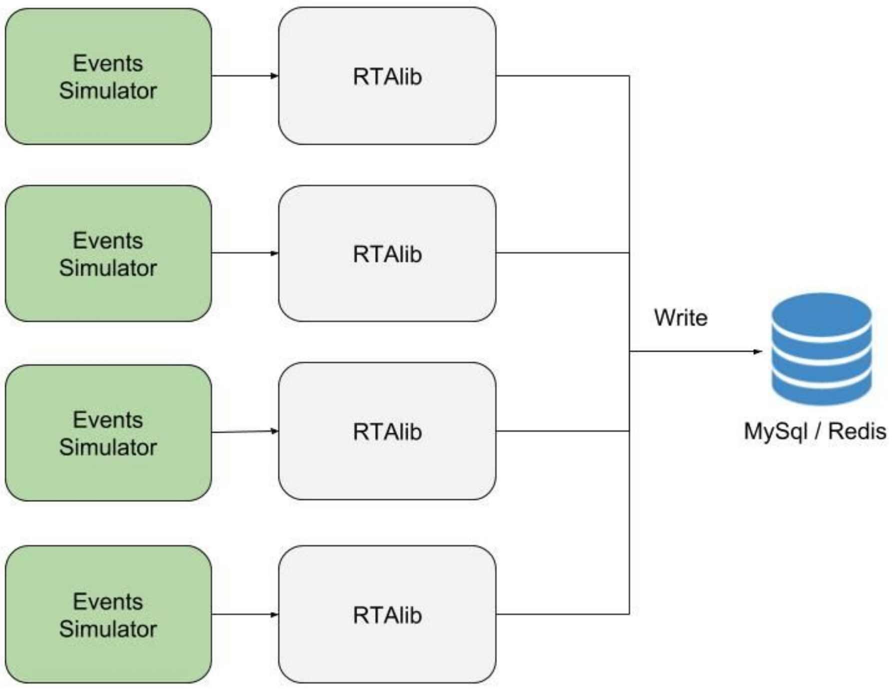


Figure 5.3: scheme 2 “WriteOnly” with four writing process

5.4.2. Write&Read Test

As the requirement of the CTAO the “Write&Read” schema has been implemented. For this reason, in order to obtain a working scheme, in addition of the usage of the performance code described in the previous paragraph, it has been necessary to configure the DTR component (described in Chapter RTAlib) in a way they can simulate readers processes.

Chapter (RTALIB) has already pointed out that the DTR component is a daemon that exploits the Redis Publisher/Subscribe protocol. If enabled, it will subscribe to a Redis channel which name can be specified in the DTR configuration file. This channel has been used to notify the demon after each event is stored. The DTR uses a master-slave architecture and, based on the type of data sent on the Redis channel, it can assign work to a specifically configured slave, the DTR_Tranformer. In the context of the CTA real-time analysis pipeline, these components have been designed to perform data transformations, for example, to compute data quality metrics. Instead, in this context, they have been configured to perform reading queries from the database periodically, after every storage of a fixed number of events.
In addition to the queries execution, the DTR_Tranformer writes on file a report containing the following informations:

- Number of the last event read;
- Execution time;
- The event rate of the reading process;

Finally, in order to execute the "Write&Read" test, a bash script has been implemented. The script uses the *nohup* command to execute the writing and reading processes at the same time. Furthermore, the nohup command can redirect the output of the DTR_tranformer to a log file.
Some tuning tests have been previously carried out in order to identify a good DTR system configuration that satisfies the project requirement. This led to the elaboration of scheme 3.

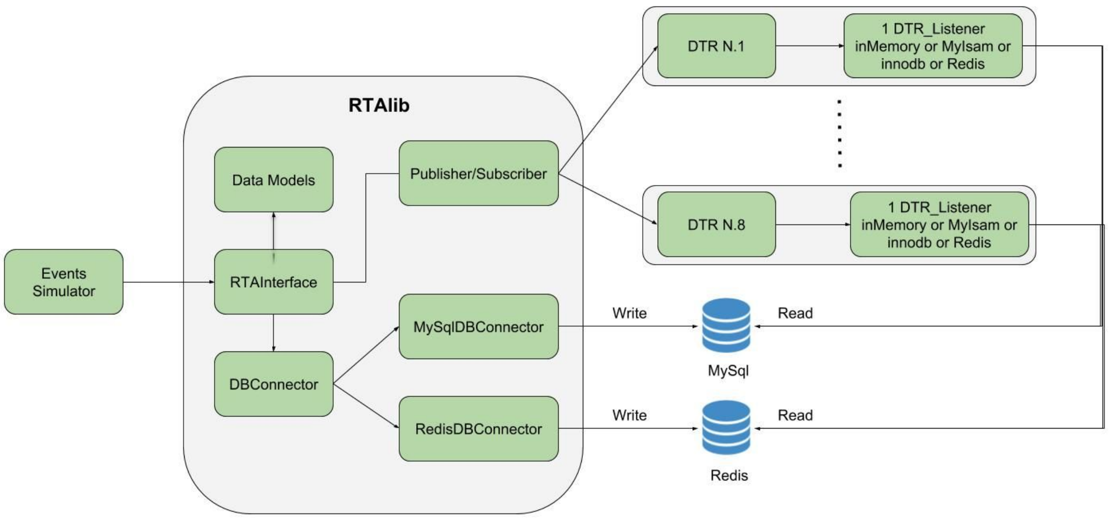


Figure 5.2: scheme 3 “Write&Read” single writing process and N reader process

Scheme 3 aims to identify any performance degradation when the writing and reading processes are performed at the same time, on the same table.
In order to evaluate the degradation of the performances obtained when more than one writing process and several reading processes occur simultaneously, another “Write&Read” scheme has been identified. This scheme will be mentioned as scheme 4.

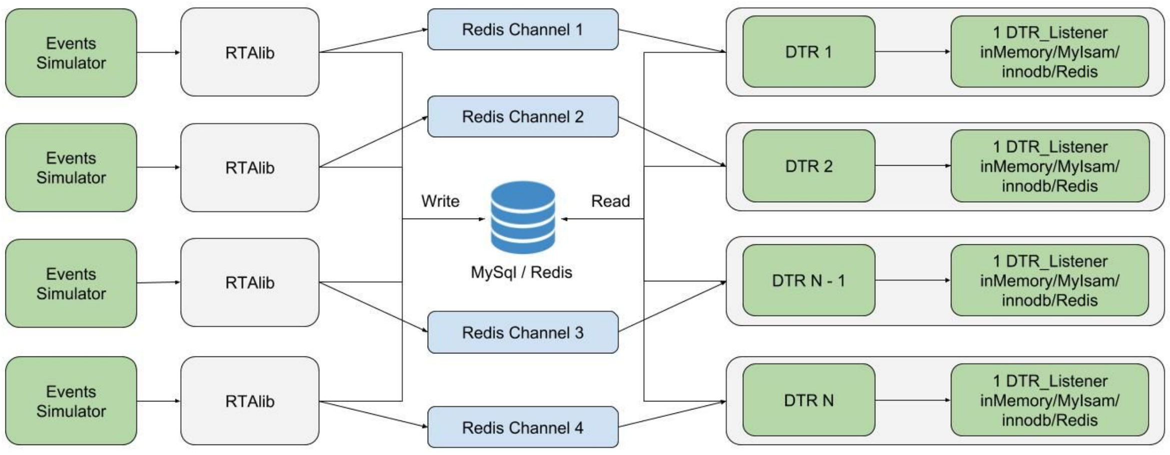


Figure 5.4: scheme 4 “Write&Read” four writing process n reader process

5.5. Scalability

Within the High-Performance Computing context, it is essential to measure the scaling efficiency when the software has to able to respond to an even larger problem. One way to try to enhance the software performance is to apply parallelism with a multithreading pattern.
There are two ways to measure the parallel performance of a given application: the strong and the weak scaling.[32]

- **Strong scaling:** the total problem size stays fixed as more processors are added. The goal is to solve the problem in less time.

- **Weak scaling:** the problem size per processor stays fixed as more processors are added in order to resolve a larger problem. The goal is to keep a constant run time while the workload increase in direct proportion to the number of processors.

During this performance test only strong scalability it has been evaluated.

5.6. Performance tests results

The first tests performed on the INAF server have been the DL2 performance tests. This kind of data, as reported in Chapter 4, is composed by 39 fields encoded with a double variable of 8 bytes, for a total size of 312 bytes. Furthermore, the "insert_time" field of the DL2 MySQL table was used as table index.

5.6.1. DL2 performances WriteOnly one-shot with one process

In the following table are reported the performance results obtained for each engine, executing the tests in the "one-shot" mode using the "scheme 1". The following results have been evaluated simulating the arrival and the storage of 50.000 events. Since InnoDB and Redis provide transactions operation, the tests have been performed with different batch sizes: 100 and 1000. Furthermore, also the "streaming" mode (with batchsize = 1) has been evaluated.

5.6.1.1. MySQL Memory

Number of Threads	Event rate (kHz)		Time (s)		Data rate (MB/s)	
	C++	Python	C++	Python	C++	Python
0	3,9 ± 0,2	3,5 ± 0,2	12,70±0,55	14,29±0,45	1,14±0,06	1,03±0,06
9	11,2 ± 0,4	3,2 ± 0,2	4,49±0,17	15,63±0,50	3,28±0,12	0,94±0,06
18	10 ±0,2	1,5 ± 0,3	5,00±0,12	33,33±0,95	3±0,06	0,44±0,09

36	10 ±0,1	0,5 ± 0,02	4,98±0,07	100±2,55	3±0,03	0,15±0,06

Table 5.1: MySQL in Memory multithreading "WriteOnly" "one-shot" performances

5.6.1.2. Redis

5.6.1.2.1. Batch size: 1

Number of Threads	**Event rate (kHz)**		**Time (s)**		**Data rate (MB/s)**	
	C++	**Python**	**C++**	**Python**	**C++**	**Python**
0	7,4 ± 0,3	7,4 ± 0,2	6,76±0,25	6,75±0,15	2,17±0,09	2,17±0,6
9	13,2 ± 0,2	2,5 ± 0,5	3,79±0,29	20,13±0,21	3,87±0,6	0,73±0,15
18	12,4 ± 0,4	1,2 ± 0,1	4,03±0,14	41,67±0,38	3,63±0,12	0,35±0,03
36	12,6 ± 0,5	0,7 ± 0,07	3,97±0,21	71,42±0,36	3,69±0,15	0,21±0,02

Table 5.2: Redis multithreading "WriteOnly" "one-shot" performances batchsize 1

5.6.1.2.2. Batch size: 100

Number of Threads	**Event rate (kHz)**		**Time (s)**		**Data rate (MB/s)**	
	C++	**Python**	**C++**	**Python**	**C++**	**Python**
0	7,8 ± 0,4	13,3±0,15	6,76±0,25	3,75±0,1	2,17±0,09	3,91±0,04
9	12,8 ± 0,6	3,3±0,55	3,9±0,17	15,1±0,5	3,67±0,2	0,98±0,16
18	11,8 ± 0,1	1,2 ± 0,1	4,2±0,05	41,67±0,4	3,39±0,3	0,35±0,03
36	11,8 ± 0,1	0,8 ± 0,05	4,23±0,47	62,5±0,4	3,38±0,3	0,23±0,02

Table 5.3: Redis multithreading "WriteOnly" "one-shot" performances batchsize 100

5.6.1.2.3. Batch size: 1000

Number of Threads	Event rate (kHz)		Time (s)		Data rate (MB/s)	
	C++	Python	C++	Python	C++	Python
0	7,5 ± 0,24	12,4 ± 0,9	9,03±0,38	4,03±0,2	1,59±0,1	3,63±0,7
9	13,1 ± 0,5	3,26 ± 0,2	3,8±0,13	15,33±0,7	3,76±0,2	0,96±0,06
18	11,7 ± 0,1	1,4 ± 0,1	4,3±0,04	35,7±0,5	3,35±0,3	0,41±0,03
36	11,8 ± 0,1	0,8 ± 0,05	4,24±0,02	62,5±0,4	3,37±0,3	0,23±0,02

Table 5.4: Redis multithreading "WriteOnly" "one-shot" performances batchsize 1000

5.6.1.3. MySQL MyISAM Permanent Storage

Number of Threads	Event rate (kHz)		Time (s)		Data rate (MB/s)	
	C++	Python	C++	Python	C++	Python
0	3,6 ± 0,2	3,9±0,1	14,1±0,74	12,8±0,27	1,05±0,05	1,14±0,03
9	12,4 ± 0,5	2,7±0,5	4,05±0,15	18,5±0,78	3,63±0,13	0,79±0,15
18	10,7 ± 0,2	1,1 ±0,1	4,66±0,11	50,6±0,81	3,13±0,07	0,29±0,03
36	10,5±0,32	0,5 ±0,06	4,76±0,15	101±1,24	3,08±0,09	0,15±0,02

Table 5.5: MySQL MyISAM multithreading "WriteOnly" "one-shot" performances

5.6.1.4. MySQL InnoDB Permanent Storage

5.6.1.4.1. Batch size: 1

Number of Threads	Event rate (kHz)		Time (s)		Data rate (MB/s)	
	C++	Python	C++	Python	C++	Python
0	1 ±0,1	1,4 ±0,1	48,47±5,6	35,71±2,32	0,29±0,04	0,41±0,04

9	11,9 ± 0,5	3,3 ±0,1	4,25±0,19	15,15±0,21	3,49±0,16	0,97±0,04
18	10,7 ±0,4	2,1 ±0,2	4,67±0,18	23,81±1,16	3,13±0,13	0,62±0,07
36	10,7 ±0,2	1,1 ±0,1	4,68±0,72	45,45±1,58	3,13±0,05	0,32±0,04

Table 5.6: MySQL InnoDB multithreading "WriteOnly" "one-shot" performances batchsize 1

5.6.1.4.2. Batch size: 100

Number of Threads	Event rate (kHz)		Time (s)		Data rate (MB/s)	
	C++	Python	C++	Python	C++	Python
0	1,6 ±0,25	1,8 ±0,21	31,9±4,1	27,78±1,68	0,47±0,07	0,52±0,06
9	11,4 ±0,27	3,2 ±0,14	4,40±0,10	15,63±0,24	3,34±0,79	0,93±0,04
18	10,9 ±0,32	1,52 ±0,14	4,57±0,13	32,9±0,96	3,19±0,09	0,44±0,04
36	10,7 ±0,29	0,66 ±0,17	4,68±0,13	75,76±1,88	3,13±0,08	0,19±0,05

Table 5.7: MySQL InnoDB multithreading "WriteOnly" "one-shot" performances batchsize 100

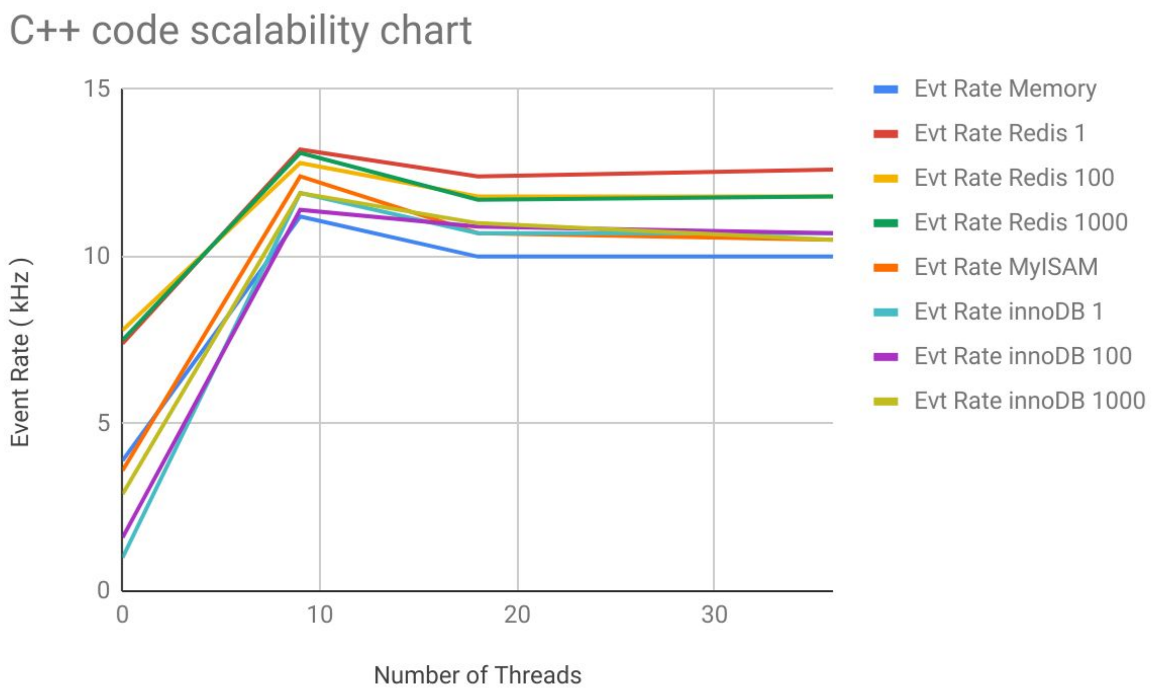
5.6.1.4.3. Batch size: 1000

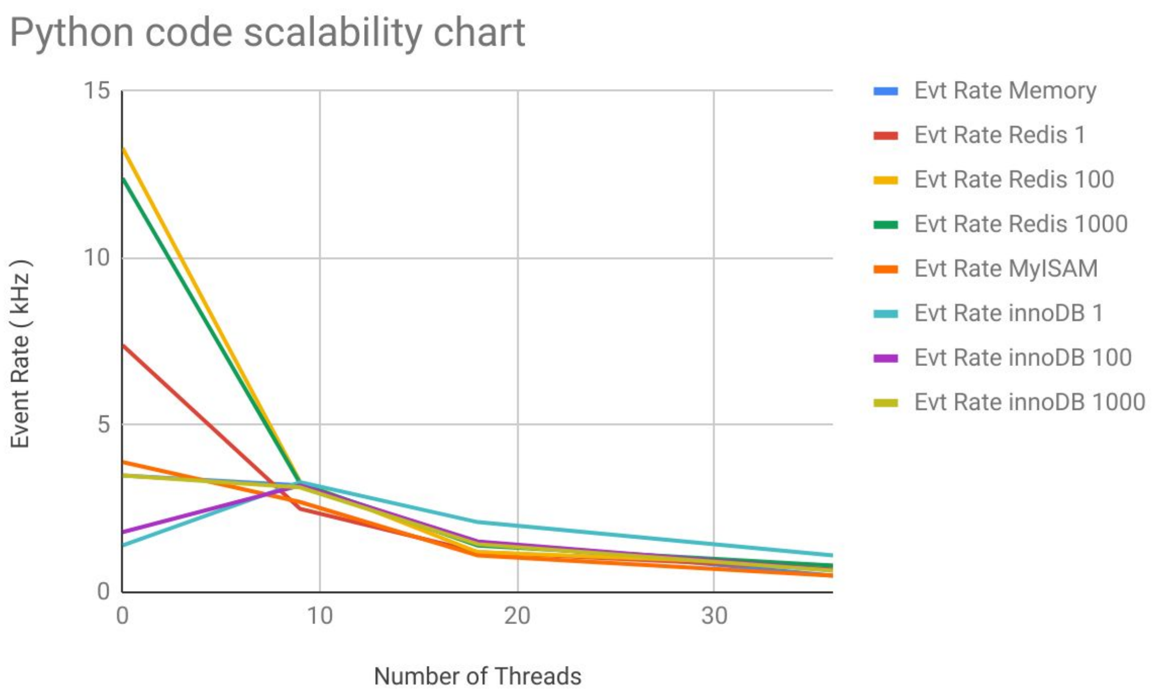
Number of Threads	Event rate (kHz)		Time (s)		Data rate (MB/s)	
	C++	Python	C++	Python	C++	Python
0	2,9 ±0,22	3,5±0,05	16,9±1,4	14,28±0,67	0,47±0,07	1,03±0,01
9	11,9 ±0,42	3,14±0,33	4,22±0,15	15,92±0,34	3,4±0,12	0,92±0,1
18	11 ±0,17	1,44±0,21	4,53±0,07	34,7±0,32	3,16±0,05	0,42±0,06

36	10,5±0,38	0,65±0,08	4,77±0,17	76,92±0,78	3±0,11	0,19±0,02

Table 5.8: MySQL InnoDB multithreading "WriteOnly" "one-shot" performances batchsize 1

In the following charts the multithreading scaling is reported:





The tables and the performance charts above have substantially highlighted two aspects, one for each version of the RTAlib. In particular, observing the results obtained, it is clear that:

1. the Python version of the RTAlib is not scalable in terms of threads;
2. the C ++ version of the RTAlib would seem to have a bottleneck.

The Python version of RTAlib provides its best performance in synchronous execution, when the main thread only is used (following indicated as 0 Thread case[4]). PyRTAlib uses the threading package to perform multithreading, but due to the Global Interface Locking (GIL), only one thread can execute Python code at once, preventing the concurrent execution. This greatly limits the performance of PyRTAlib and its scalability.

The C++ version of the RTAlib reaches, in the synchronous case, performances comparable to the Python version with all engines. Unlike the PyRTAlib, the CxxRTAlib improves its performance when it moves from a synchronous approach (0 thread) to an asynchronous one (multithreading). This is due to the multithreading implementation of the “pthread” library that realizes a real competition pattern. However, with 9 threads, it has been achieved the

[4] The terminology "0 Thread" has been used because in this case no other thread is used except the main one. The pthread library is not used from RTAlib.

same performances with all engines. This could highlight how the C ++ code can present a bottleneck that limits the achievable performance.

5.6.2. DL2 performances WriteOnly long-run with one process

Given the results obtained and noted the poor scalability of the Python version of the RTAlib, it has been decided to continue the tests only for the C++ code in order to understand its current limits. “Long-run” tests have been performed with the best performing CxxRTAlib configuration (inserted 43.200.000 events). In the following table, is presented a summary of the best results obtained:

Engine	**Number of Processes**	**Number of Threads**	**Event rate SSD one shot (kHz)**	**Event rate SSD long-run (kHz)**
Redis	1	9	13,2 ± 0,2	9,9 ± 0,2
MySQL in Memory	1	9	11,2 ± 0,3	9,7 ± 0,3
MySQL MyISAM	1	9	12,4 ± 0,5	8,6 ± 0,5
MySQL InnoDB	1	9	11,9 ± 0,5	9,1 ± 0,5

Table 5.9: All engines "WriteOnly" "long-run", CxxRTAlib single process 9 Threads performance

The long-run have led to a performance decrease. This could be due to both internal code bottlenecks or the use of the "insert_time" field as the table index. In fact, the use of indexes within the databases, on the one hand, speeds up the reading on the other penalises writing. In fact, all the engines have a tree structure for storing indexes that requires a new update for each new entry. This implies that as the database grows, the writing performance may be less and less efficient.

5.6.3. DL2 performance WriteOnly with multiprocess and multithreading test

After a first performance analysis and once identified the first critical issues, other performance tests has been performed using a different RTAlib configuration: the "WriteOnly" test using scheme 4 was performed with 4 writer processes with 2 threads per process. . The following table shows the obtained results:

Engine	**Number of Processes**	**Number of Threads**	**Number of events**	**Event rate (kHz) SSD one shot**
Redis	4	2	100.000	43.2 ± 0,3
MySQL in Memory	4	2	100.000	27.2 ± 0,4
MySQL MyISAM	4	2	100.000	23.3 ± 0,5
MySQL InnoDB	4	2	100.000	13.1 ± 0,4

Table 5.10: All engines "WriteOnly" "one-shot" performances, scheme 4, CxxRTAlib 4 processes 2 Threads per process

With this scheme, better overall performances have been obtained. This may confirm that the limits previously identified are due to bottlenecks within the code: in this configuration, each writer process handles only 2 connections with respect to the 9 connections of the single process scheme (scheme 1). In the multithreading configuration, the RTAlib uses a buffer in which the simulated events are temporary stored. Each worker thread accesses the buffer to extract the event in a mutually exclusive way. This implies that the use of two threads instead of 9 reduces the traffic congestion around the buffer, so this configuration makes a better workload balance hence providing better performance.

In order to identify further bottlenecks in the code, the “write-only” “long-run” tests have been carried out. The following table shows the obtained results:

Engine	Number of Processes	Number of Threads	Event rate SSD long-run (kHz)
Redis	4	2	42,7 ± 0,3
MySQL in Memory	4	2	26,6 ± 0,4
MySQL MyISAM	4	2	23 ± 0,5
MySQL InnoDB	4	2	12,6 ± 04

Table 5.11: All engines "WriteOnly" "long-run" performance, CxxRTAlib 4 writer processes with 2 Threads, 9 reader process in single thread

Once again, the "write-only" "long-runs" tests provided similar performances with respect to the "one-shot" tests. This indicates that the tree structure in charge of managing the indices does not represent a limit for the performance obtained. Therefore, that confirms once again that the reason of the performance limit must be searched in the source code.

5.6.4. DL2 performance Write&Read tests

In order to close the RTAlib data flow, the "Write&Read" test using the scheme 3 has been performed. In this context, the reading processes are performed by nine DTR components. In the following table the results achieved are reported:

Engine	Writer		Event rate (kHz) one shot	Reader		Event rate (kHz) one shot
	Number of Process	Number of Threads		Number of Process	Number of Threads	
Redis	1	9	13 ± 0,2	9	1	26,1 ± 0,3
MySQL Memory	1	9	10,8 ± 0,3	9	1	34,2 ± 0,4

MySQL MyISAM	1	9	12,6 ± 0,4	9	1	31,5 ± 0,5
MySQL InnoDB	1	9	11,9 ± 0,3	9	1	34,2 ± 0,2

Table 5.12: Alle engines "Write&Read" "one-shot", CxxRTAlib single writer process with 9 Threads performance, 9 reader process in single thread

The results obtained in the "one-shot" writing and reading scheme do not show any significant deterioration in write performance.

A new scheme 4 has been built merging scheme 2 and reading tasks of scheme 3. Following are the results obtained:

Engine	**Writer**		**Event rate (kHz) one shot**	**Reader**		**Event rate (kHz) one shot**
	Number of Process	**Number of Threads**		**Number of Process**	**Number of Threads**	
Redis	4	2	18.9 ± 0,4	9	1	24 ± 0,6
MySQL Memory	4	2	19.5 ± 0,4	9	1	25 ± 0,6
MySQL MyISAM	4	2	19 ± 0,4	9	1	3,6 ± 0,5
MySQL InnoDB	4	2	14.8 ± 0,3	9	1	65.8 ± 0,2

Table 5.13: All engines "Write&Read" "one-shot" performance, CxxRTAlib 4 writer processes with 2 Threads, 9 reader process in single thread

The performances obtained with the "one-shot" and "Write&Read" tests showed a dramatic performance drop only using Redis as a database.

Finally, in order to finish the test cycle with the DL2 data, the "Write&Read" "long-run" tests with scheme 4 have been performed. In a first phase, these tests have been interrupted due to a RAM saturation issue. Analysing the memory occupation during the source code runtime, it has been possible to identify the problem. Inside the DTR class, the main thread listens to a

Redis channel and inserts the events into a buffer as soon they are available. Another thread consumes the buffer and after every 1000 events extracted, executes the read query on the database. The RAM occupation increases when the DTR thread that consumes the buffer do it in a slower manner than the main thread that fills the buffer. Due to GIL, these two threads work one at once. This problem becomes evident when multiple processes write on the same channel. A Redis channel for each writer process has been used to balance the workload and to fix partially the issue. The following are the performances obtained:

Engine	**Writer**		**Event rate long-run (kHz)**	**Reader**		**Event rate long-run (kHz)**
	Number of Process	**Number of Threads**		**Number of Process**	**Number of Threads**	
Redis	4	2	full RAM	9	1	full RAM
MySQL Memory	4	2	full RAM	9	1	full RAM
MySQL MyISAM	4	2	full RAM	9	1	full RAM
MySQL InnoDB	4	2	14.8	9	1	65.8

Table 5.14: All engines "Write&Read" "long-run" performance, CxxRTAlib 4 writer processes with 2 Threads, 9 reader process in single thread

As evident from the 5.14 table the “long-run” “Write&Read” tests in the MySQL context only the InnoDB engine provides a good feedback . On the other hand, in Redis context the “long-run” "Write&Read" test highlighted a big limit in Sorted-set. This data structure is a combination of two other ones: hash data structures and skip lists. These two structures allow fast writing and reading operations as well as avoid data duplication within the same Sorted Set. However, the presence of these two structures generates a very large footprint for the Sorted Sets that cause the memory saturation. In order to try to solve this problem, it has been tried to delete the data once read. However, the temporal complexity of adding new data

(ZADD command) is in the order of O(log(N)) for each added element while the temporal complexity of the reading command (ZRANGEBYSCORE) and the delete command (ZREMRANGEBYRANK) is for both in the order of O(log(N)+M) (where N is the number of elements within the sorted set and M is the number to read or to remove).
So from here we understand that with this approach it is difficult if not impossible to avoid saturation of RAM. We will address this topic in the final Chapter.
Regarding the in-Memory and MyISAM engines, the saturation of the memory is due to the internal table management. In fact, both these two engines have a granularity locking on the table and therefore the various processes involved, whether writing or reading, keep the whole table locked for too long. This led to the various DTRs buffers to grow and saturate the central memory. This behaviour represents a big limit in the case of insertions, updates and simultaneous readings executed on this type of tables. For this reason, it is possible to conclude that these engines are not suitable to satisfy the use cases and the project requirements of this thesis work.
InnoDB instead has a row locking granularity, this avoids the problem of saturation of the RAM allows obtaining on "long-run" tests performance of about 15 kHz. This result is in according with the same test performed in “one-shot” mode. This indicates that the tables indices are not responsible for any deterioration of the performance and once again, the performance limits should be found in the code.

5.6.5. DL1 Performances

The real challenge for requirements achieving is in the DL1 storing. The DL1 as reported in Chapter 2 provides in addition to the event data, also two images, represented by two matrices. For this data type, permanent storing is not required, but only caching support. Hence, the in-Memory engine or Redis must be used. However, the in-Memory engine does not support the BLOB type nor the TEXT type and for this reason, the choice is forced on Redis. The image is still encoded as a string. “One-shot” “Write-only” tests provide the following result:

Engine	Number of Processes	Number of Threads	Number of events	Event rate (kHz) one shot	
				Host	Singularity
Redis	4	2	100000	~10	~10

Table 5.15: Redis "WriteOnly" "one-shot" performance, CxxRTAlib 4 writer processes with 2 Threads, 9 reader process in single thread

In this case, the data size impacts on performance, marking a slight decline in performance. In the “one-shot” version, the requirements are still achieved. The writing and reading tests have been carried out again, below are the performances obtained:

Engine	Writer		Event rate one shot (kHz)		Reader		Event rate one shot (kHz)	
	Number of Process	Number of Threads	Host	Singularity	Number of Process	Number of Threads	Host	Singularity
Redis	4	2	**7.2**	**7.2**	9	1	**29.5**	**29.4**

Table 5.16: Redis "Write&Read" "one-shot" performance, CxxRTAlib 4 writer processes with 2 Threads, 9 reader process in single thread

However, because of the problems highlighted previously in the current state this solution requires further work: in fact, delete the data from Redis once read, does not avoid the saturation of RAM on “long-run” tests as well as not being a technique to implement a caching system. At the current state of the software, the results obtained for Redis represent a preliminary feasibility study for a caching system that satisfies the project requirements.

6. Conclusion

In this chapter a brief summary of the work done during my thesis work will be presented. More space will be given to the analysis of the performances obtained. Based on that, the possible future developments of this work will be presented.

6.1. Brief

The aim of this thesis work was to design and develop a framework that would support the Real Time Analysis of the CTAO scientific data. In this regard, the RTAlib has been developed to provide a simple but powerful high-performance API to store the data, generated during the reconstruction and analysis phase of the raw telescopes data, into the Real Time Analysis database. In order to cope with the high data rates of the CTA reconstruction and analysis chain, the RTAlib exploits multiprocess, multithreading and databases. All these functionalities have been tested.
The team in which I worked for this project, based its strategy and software development process on the principles of the Scrum and DevOps approach.
In this context, daily meetings have been carried out on the Skype platform and the Waffle platform has been used as Scrum board to visually depict the Scrum iterations and progress. The project Waffle board has been linked with the RTAlib GitHub repository and this allows the real time alignment between the Scrum board and the DCVS used.
Last but not least, two technologies for continuous integration have been explored and implemented. In both solutions has been setted up an integration with the GitHub servers and therefore the automatic triggering of the unit testing process and the estimation of the corresponding code coverage.
The first solution uses the remote CircleCI platform and utilises a Docker container while the second one implements a custom local continuous integration server with the Jenkins platform that uses a Singularity container.

6.2. Results

As first tests a single writing process have been carried out, focusing heavily on multithreading. These tests highlighted the poor scalability of the PyRTAlib and a bottleneck in C++ version of the RTAlib. Using Python multithreading to improve the code performance is not a good idea, in fact, due to the Global Interface Locking (GIL) only one thread can execute Python code at once, preventing real concurrent execution. This greatly limits the performance of PyRTAlib and its scalability. For this reason, the subsequent efforts have been concentrated on the C ++ version.

Within the performance tests of CxxRTAlib version, the same results have been achieved regardless of the used engine. In order to explore the limits of the architecture, it was decided to increase the number of writer processes decreasing the number of threads for each process. This configuration reduced data congestion on internal RTAlib buffer and provided a better workload balance. This approach has proved to be more performing.

The “long-run” tests allowed verifying that the use of tables indexes doesn't affect the performances. In fact, these tests provided the same performance as the short tests, indicating how the performance limits obtained are not due to server-side limitations.

While the “Write&Read” short tests did not show either criticality or deterioration in performance, the same thing does not apply to the “long-runs”. The “long-run” "Write&Read" test highlighted a big limit in Redis Sorted-set. This data structure is a combination of two other ones: hash data structures and skip lists. These two structures allow fast writing operations as well as avoid data duplication within the same Sorted Set. However, the presence of these two structures generates a very large footprint for the Sorted Sets that cause memory saturation. To try to solve this problem, it has been tried to delete the data once read. However, temporal complexity of the read and delete commands does not allow this solution approach and for this reason, only the “one-shot” tests have been performed for Redis. As for the MySQL environment, the engines in Memory and MyISAM provide bad performance due to table locking granularity. The latter strongly limits the performance and scalability of the problem by making the engines not suitable to satisfy to CTA requirements. After analyzing the performances obtained, it is possible to conclude that the optimisation efforts have to be

carried out on the C++ version of RTAlib, focusing on the use of the InnoDB engine for permanent storing and on optimising the Redis environment for buffering.
In Table 6.1, the best performance obtained for DL2 data storage in “Write&Read” mode is reported.

Engine	**Writer**		**Event rate long-run (kHz)**	**Reader**		**Event rate long-run (kHz)**
	Number of Process	**Number of Threads**		**Number of Process**	**Number of Threads**	
MySQL InnoDB	4	2	14.8	9	1	65.8

Table 6.1: The best "Write&Read" performance obtained for DL2 data layer

In Table 6.2 and 6.3 the preliminary results of feasibility study for DL1 caching system data are reported.

Engine	**Number of Processes**	**Number of Threads**	**Number of events**	**Event rate (kHz) one shot**	
				Host	**Singularity**
Redis	4	2	100000	~10	~10

Table 6.2: "WriteOnly" "one-shot" performance obtained for DL1 data layer

Engine	**Writer**		**Event rate one shot (kHz)**		**Reader**		**Event rate one shot (kHz)**	
	Number of Process	**Number of Threads**	**Host**	**Singula rity**	**Number of Process**	**Number of Threads**	**Host**	**Singula rity**
Redis	4	2	**7.2**	**7.2**	9	1	**29.5**	**29.4**

Table 6.3: "Write&Read" "one-shot" performance obtained for DL1 data layer

6.3. Future development

Future project developments involve several aspects. It however possible identify four branches:

- Improvement of the local Continuous integration pipeline ;
- Writing new features;
- Optimization of the server side environment;
- Optimization of the developed code;

Some performance tests will be included in the testing phase within the Jenkins pipeline. In this way, will be evaluated the impact on the performances besides the right functionality of the individual code units developed.

Some abroad teams, in collaboration with the CTAO, are developing reconstruction and analysis pipelines using the Python language. Therefore, it may be useful to develop a Python wrapper for the CxxRTAlib interface class.

The performances obtained on the "long-run" writing and reading tests have highlighted some critical issues, in particular as regards the use of Redis. To address this issue it is possible to implement different strategies in order to obtain a result in line with the project requirements.

A first attempt includes writing new C++ code.

In order to speed up the management of the reading processes, it is appropriate to proceed with the development of the DTR and DTR_Transformer components in C++ language.

In this way, within the DTR component, it will be possible to implement multithreading, with one thread that inserts the events in the internal buffer and the other one that picks them up.

But it is mainly in the C++ DTR_Transformer implementation that the significant improvements would be possible.

As reported in the performance chapter within the Sorted Sets, the write operation has a temporal complexity in the order of log (N) while the operations of reading and deleting are in the order of log(N) + M. In Python implementation delete events once read did not give the expected results. Read and write operations are sequentially and blocking executed within the

PyRTAlib. By implementing this component in C++ it possible to perform these tasks in a parallel and asynchronous way, guaranteeing less RAM congestion.
However, this could not be enough and it is, therefore, necessary to take some measures on the server side.
Within the Real-Time Analysis of CTA Redis works as a caching system. When Redis is used as a cache, often it is handy to let it automatically evict old data when adding a new one.
LRU is the only one supported eviction methods provided by Redis. It is necessary to set the configuration maxmemory directive within the redis.conf file in order to decide the limit memory usage. When the specified amount of memory is reached, it is possible to select among different policies. The possible policies are:

- **noeviction**: return errors when the memory limit was reached;
- **allkeys-lru**: evict keys by trying to remove the less recently used (LRU) keys first, in order to make space for the new data added.
- **volatile-lru**: evict keys by trying to remove the less recently used (LRU) keys first, but only among keys that have an **expire set**, in order to make space for the new data added.
- **allkeys-random**: evict keys randomly in order to make space for the new data added.
- **volatile-random**: evict keys randomly in order to make space for the new data added, but only evict keys with an **expire set**.
- **volatile-ttl**: evict keys with an **expire set**, and try to evict keys with a shorter time to live (TTL) first, in order to make space for the new data added.[33]

In this context, allkeys-lru or volatile-lru policies could be adopted.
On MySQL, InnoDB is the engine that best satisfy the project requirements for permanent storage. Then is essential to focus efforts on possible optimizations to operate on this engine.
At last but not least, in order to achieve better performance, it is necessary to focus on the code optimisation. In fact, the performance tests have shown how the results obtained limits are to be found in the code; for this reason further code profiling must be carried out.

7. Appendix

7.1. Jenkins installation commands

In the following lines, the Jenkins installation commands are reported:

```
sudo wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat/jenkins.repo
sudo rpm --import https://jenkins-ci.org/redhat/jenkins-ci.org.key
sudo yum install jenkins
sudo service jenkins start/stop/restart
sudo chkconfig jenkins on
```

7.2. Setting up a new project in Jenkins

In order to connect to the Jenkins graphical user interface from a local machine, a ssh tunnel must be setted up linking the local machine with the INAF server. The ssh tunnel can be activated with the following command:

```
ssh -N -L 8500:localhost:8080 **@*****.***.inaf.it
```

In order to create a Jenkins project there is a "New item" button in the home page of the Jenkins GUI. A name and a type must be given to the project. The chosen type is the "multibranch pipeline" that allows to create a set of pipelines according to each detected branches in one source code management repository.
After the creation process, a configuration page is presented: the "Branch Sources" section is needed to specify which are the branches that will trigger the Jenkins build process: first of all, the GitHub account credentials must be added to the Jenkins account. Then, the triggering behaviour must be configured. Jenkins can discover branches, pull requests and so on. The next section is the "Build configuration": it has been chosen the "Jenkinsfile" mode. The preliminary configuration of the Jenkins project is terminated.

The next step is to set up Jenkins in order to decide when start the build process. This configuration is located under the "View configuration" button located in to the project page. The adopted strategy is to poll periodically a GitHub server to check project changes.
This feature is not ideal for compiling software projects on a regular basis because the continuous integration key is to start a compilation and a test phase as soon as possible after a change is pushed to the remote CVS server, to provide a rapid feedback. To accomplish this, GitHub Integration plugin for Jenkins provides this kind of functionality, triggering jenkins when, for example, a pull request is opened.

7.3. Jenkinsfile and the build process

Every pipeline defined in a Jenkinsfile starts with the agent directive to ensures that the source repository is checked out and made available for steps in the subsequent stages. Within RTA Jenkinsfile, 7 main stages are defined:

- "Creating configuration file";
- "Singularity Instance";
- "CxxRTAlib compilation";
- "PyRTAlib Unit-testing";
- "CxxRTAlib Unit-testing";
- "PyRTAlib Test coverage";
- "Test reporting";

The "Creating configuration file" stage creates the configuration file for the unit test.
At this point, the external libraries needed to run the unit test of the RTAlib software are needed. Unlike the CircleCI, building process, in which all the external dependencies are compiled and installed at runtime, a different approach is chosen for the Jenkins pipeline. All the external dependencies (libraries and MySQL/Redis database server instances) are already installed in a Singularity image. The Singularity image can be started as a standalone container instance as will be described in paragraph 7.2. Therefore, within the second stage,

the Singularity image, previously created for the RTAlib, is executed as a Singularity container instance.
The third stage simply compiles the C++ version of the RTAlib invoking the make command.
In the fourth and fifth stages, the unit tests of the C++ and the Python version of the RTAlib are performed in parallel.
In the last stage, tests results are exported in the XML format to allow graphical representation of the performed tests. If tests fail, the Pipeline is marked "unstable", as denoted by a yellow ball in the web UI. Based on the recorded test reports, Jenkins can also provide historical trend analysis and visualization.

7.3.1. The Singularity recipe

A Singularity Recipe is the starting point for designing any custom container. It includes specifics about installation software, environment variables, files to add, and container metadata. A Singularity Recipe file is divided into two main parts: the header that tells Singularity the base Operating System that it should use to build the container and several (optional) sections. Each section is defined by a % character followed by the name of the particular section. The header is composed by a “Bootstrap” field that references what kind of “base” it must used. The “From” field instead, is the name of the container or docker layer that will be used as the base OS.
For this thesis work, it has been chosen the “Docker” base and the centos:7.3.16.11 Linux operating system.
Different sections add different content or execute commands at different times during the build process: commands in the *%setup* section are executed on the host system outside of the container after the base OS has been installed. The *%files* section is used when there is the need of copying files from the host system into the container. The %labels section is used to store metadata with the container. In the %environment section environment variables can be defined but they will be sourced at runtime and not at build time. The commands in the *%post* section are executed within the container after the base OS has been installed at build time.

This is the right place to make directories, install external software and libraries. The *%runscript* section defines actions for the container to take when it is executed (i.e. when the container is triggered with the run command, or simply by called as though it were an executable). Last but not least, the *%startscript* section is executed when the container is started as a stand alone container instance.
Having previously indicated the "base" and the container layer used for the Singularity recipe, let's see which commands in the %post section are executed at build time. In a first time the following dependencies has been installed:

```
yum -y install openssh-clients php
yum -y install gcc gcc-c++ make python-devel openssl-devel
yum -y install bzip2 curl git less nmap-ncat vim wget
yum -y install epel-release
yum -y install redis
yum -y install mysql-server
```

Subsequently, three folders have been created for the RTAlib C ++ version. One folder for containing the source codes of the software packages, another one for containing the compiled code and the last one for containing the installed package.

```
mkdir -p $ RTA_DEP_DIR / libraries_sources
mkdir -p $ RTA_DEP_DIR / libraries_building_area
mkdir -p $ RTA_DEP_DIR / libraries_installed
```

Then, the following packages are cloned, built and installed:

- MySQL cpp connector;
- hiredis library;
- cmake software;
- Boost library;
- cfitsio NASA C library;
- googletest framework;

- anaconda distribution

As the last component, the anaconda distribution is installed. Anaconda ships with Conda, an open source package management system and environment management system that is a necessary dependency for the Python version of the RTAlib. A Python 3.6 virtual environment is then created using conda and some Python dependencies (such as the Python MySQL connector library) are installed within the virtual environment with the pip package manager.
At this point, all the necessary work for the building phase has been done. The last step is to indicate the commands to execute when the Singularity container is started. These commands are collected under the %startscript label. The %startscript section of the RTAlib container developed for this thesis work, does the following:

- activates a MySQL database server;
- activates a Redis database server;
- activates the anaconda virtual environment in order to load the Python dependencies of the PyRTAlib;
- exports several environment variables that are necessary to run the C++ version of the RTAlib.

All the required field for the creation of Singularity are filled. The container can now used within Jenkins pipeline.

7.4. Commands to clone and to compile the code

In the following lines the commands to clone and compile the CxxRTAlib are reported:

```
git clone https://github.com/rta-pipe/RTAlib.git
export HOME=~/
```

Create folders for dependencies source codes

```
mkdir -p ~/rtalib_dep_libs/libraries_sources
mkdir -p ~/rtalib_dep_libs/libraries_building_area
mkdir -p ~/rtalib_dep_libs/libraries_installed
```

To install cmake library:

```
mkdir -p ~/rtalib_dep_libs/libraries_building_area/cmake-3.12.3
mkdir -p ~/rtalib_dep_libs/libraries_installed/cmake-3.12.3
cd ~/rtalib_dep_libs/libraries_sources
wget https://cmake.org/files/v3.12/cmake-3.12.3.tar.gz
cd ~/rtalib_dep_libs/libraries_building_area
tar -zxvf ~/rtalib_dep_libs/libraries_sources/cmake-3.12.3.tar.gz
cd cmake-3.12.3
./bootstrap --prefix=~/rtalib_dep_libs/libraries_installed/
/cmake-3.12.3
make
make install
export CMAKE=~/rtalib_dep_libs/libraries_installed/cmake-3.12.3/bin/
cd ~/
```

To install Boost

```
mkdir -p ~/rtalib_dep_libs/libraries_building_area/boost_1_67_0
mkdir -p ~/rtalib_dep_libs/libraries_installed/boost_1_67_0
cd ~/rtalib_dep_libs/libraries_sources
wget
https://dl.bintray.com/boostorg/release/1.67.0/source/boost_1_67_0.ta
r.bz2
cd ~/rtalib_dep_libs/libraries_building_area
tar --bzip2 -xf
~/rtalib_dep_libs/libraries_sources/boost_1_67_0.tar.bz2
cd ~/rtalib_dep_libs/libraries_building_area/boost_1_67_0/
./bootstrap.sh
--prefix=$HOME/rtalib_dep_libs/libraries_installed/boost_1_67_0
./b2 install
cd ~/
```

To install MySql C++ Connector:

```
mkdir -p
~/rtalib_dep_libs/libraries_building_area/mysql-connector-cpp-build
mkdir -p
~/rtalib_dep_libs/libraries_installed/mysql-connector-cpp-install
cd ~/rtalib_dep_libs/libraries_sources
git clone https://github.com/mysql/mysql-connector-cpp.git
cd mysql-connector-cpp
git checkout 8.0
cd
~/rtalib_dep_libs/libraries_building_area/mysql-connector-cpp-build
cmake -DCMAKE_INSTALL_PREFIX=~/rtalib_dep_libs/libraries_installed/
mysql-connector-cpp-install/ -DBOOST_ROOT=~/rtalib_dep_libs/
libraries_installed/boost_1_67_0 -DCMAKE_BUILD_TYPE=Release
~/rtalib_dep_libs/libraries_sources/mysql-connector-cpp/
cmake --build . --target install --config Release
cd ~/
```

To install hiredis:

```
cd ~/rtalib_dep_libs/libraries_installed
git clone https://github.com/redis/hiredis.git
cd hiredis
make
cp libhiredis.so libhiredis.so.0.14
cd ~/
```

To install cfitsio library:

```
mkdir -p ~/rtalib_dep_libs/libraries_installed/cfitsio_install
cd ~/rtalib_dep_libs/libraries_sources
wget
http://heasarc.gsfc.nasa.gov/FTP/software/fitsio/c/cfitsio3450.tar.gz
tar -zxvf cfitsio3450.tar.gz
cd cfitsio
```

```
./configure
--prefix=$HOME/rtalib_dep_libs/libraries_installed/cfitsio_install
make
make shared
make install
cd ~/
```

To install GoogleTest:

```
mkdir -p ~/rtalib_dep_libs/libraries_building_area/google-test-1.8.1
mkdir -p ~/rtalib_dep_libs/libraries_installed/
/google-test-1.8.1-install
cd ~/rtalib_dep_libs/libraries_sources
git clone https://github.com/google/googletest.git
cd googletest/
git checkout release-1.8.1
cd ~/rtalib_dep_libs/libraries_building_area/google-test-1.8.1
cmake -DCMAKE_INSTALL_PREFIX=~/rtalib_dep_libs/libraries_installed/
/google-test-1.8.1-install ~/rtalib_dep_libs/libraries_sources/
/googletest
make
make install
```

How to compile:

```
cd ~/RTAlib/CxxRTAlib
export REDIS=~/rtalib_dep_libs/libraries_installed/hiredis
export MYSQL_CXX_CNT=~/rtalib_dep_libs/libraries_installed/
       mysql-connector-cpp-install
export BOOST_PATH=~/rtalib_dep_libs/libraries_installed/boost_1_67_0
export GTEST_DIR=~/rtalib_dep_libs/libraries_installed/
/google-test-1.8.1-install
make
```

7.5. Commands to run the tests

In order to execute the performances and unit tests, three environment variables must be defined:

```
export RTALIBDIR=/path/to/root/RTAlib
export LD_LIBRARY_PATH=~/rtalib_dep_libs/libraries_installed/
           mysql-connector-cpp-install/lib64/:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=~/rtalib_dep_libs/libraries_installed/
           hiredis:$LD_LIBRARY_PATH
```

7.5.1. Unit test

In order to execute the C++ unit tests the following command must be executed:

```
./RTAlib/CxxRTAlib/TestEnvironment/bin/unitTest
```

7.5.2. Performance test

In the next paragraphs, the commands to perform the "WriteOnly" and "Write&Read" performance tests are reported.

7.5.2.1. WriteOnly

In order to execute the "WriteOnly" performance test with only one writer process, the following command must be executed:

```
./RTAlib/CxxRTAlib/TestEnvironment/bin/performanceTestDlx database
 tableName numberOfEvents nOfIteration Configs/file/path cleaner
```

Where:

- *performanceTestDlx = { performanceTestDl1, performanceTestDl2}*
- *database = { mysql, redis-basic }*
- *tableName = {evtx_redis_test_table, evtx_innodb_test_table}*
- *cleaner = {'1' to clean table before test, '0' otherwise }*

The *"numberOfEvents", "nOfIteration"* and *"Configs/file/path"* parameters they must be chosen accordingly to the use cases. The first parameter indicates the number of events to simulate, the second one indicates how many times to run the same test, the last one indicates the configuration file path previously created.

In order to execute the "WriteOnly" tests with multiple writer processes, it is sufficient to use the “nohup” bash command in order to run N single processes in parallel, making sure to create a different configuration file for each process.

7.5.2.2. Write&Read

In order to execute the "Write&Read" performance tests the Python DTR process must be started. The DTR can be started with the following command:

```
python PyRTAlib/TestEnvironment/source/DTRInterface/startDTR.py
PyRTAlib/TestEnvironment/source/DTRInterface/dtr_conf_evtx_engine.ini
```

Where “*dtr_conf_evtx_engine.ini”* parameter represents the configuration file that specifies the data layer and the database engine to be used.
Below a complete example to perform the "Write&Read" performance tests is reported. In the example shown, the InnoDB engine and DL2 data type are used.

```
#!/bin/sh
DATE=`date '+%Y-%m-%d-%H:%M:%S'`

mkdir -p
$RTALIBDIR/CxxRTAlib/TestEnvironment/TEST_ENGINE_WRITEANDREAD_$DATE
TESTDIRECTORY=$RTALIBDIR/CxxRTAlib/TestEnvironment/TEST_ENGINE_WRITEA
NDREAD_$DATE

###   4  LISTENER FOR  INNODB    ###

nohup python PyRTAlib/TestEnvironment/source/DTRInterface/startDTR.py
PyRTAlib/TestEnvironment/source/DTRInterface/dtr_conf_evt2_innodb.ini
>> $TESTDIRECTORY/ReaderPerformanceResults.txt

nohup python PyRTAlib/TestEnvironment/source/DTRInterface/startDTR.py
PyRTAlib/TestEnvironment/source/DTRInterface/dtr_conf_evt2_innodb.ini
>> $TESTDIRECTORY/ReaderPerformanceResults.txt
```

```
nohup python PyRTAlib/TestEnvironment/source/DTRInterface/startDTR.py
PyRTAlib/TestEnvironment/source/DTRInterface/dtr_conf_evt2_innodb_2.i
ni >> $TESTDIRECTORY/ReaderPerformanceResults.txt

nohup python PyRTAlib/TestEnvironment/source/DTRInterface/startDTR.py
PyRTAlib/TestEnvironment/source/DTRInterface/dtr_conf_evt2_innodb_2.i
ni >> $TESTDIRECTORY/ReaderPerformanceResults.txt

###   4  WRITERS  INNODB    ###

nohup sleep 5s;

./CxxRTAlib/TestEnvironment/bin/performanceTestDl2 mysql
evt2_innoDb_test_table 3000 1
Configs/rtalibconfig_evt2_innodb_DTR_Cxx_performance_test 0 >
$TESTDIRECTORY/performanceTest1.txt &

nohup ./CxxRTAlib/TestEnvironment/bin/performanceTestDl2_2 mysql
evt2_innoDb_test_table 3000 1
Configs/rtalibconfig_evt2_innodb_DTR_Cxx_performance_test_2 0 >
$TESTDIRECTORY/performanceTest2.txt &

nohup ./CxxRTAlib/TestEnvironment/bin/performanceTestDl2_3 mysql
evt2_innoDb_test_table 3000 1
Configs/rtalibconfig_evt2_innodb_DTR_Cxx_performance_test_3 0 >
$TESTDIRECTORY/performanceTest3.txt &

nohup ./CxxRTAlib/TestEnvironment/bin/performanceTestDl2_4 mysql
evt2_innoDb_test_table 3000 1
Configs/rtalibconfig_evt2_innodb_DTR_Cxx_performance_test_4 0 >
$TESTDIRECTORY/performanceTest4.txt &
```

8. Bibliografia

[1]https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html

[2]https://www.osservatori.net/it_it/osservatori/comunicati-stampa/big-data-analytics-italia-mercato-2018

[3]META Group Inc. APPLICATION DELIVERY STRATEGIES

[4]https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

[5]https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5343946/

[6]https://hbr.org/2010/11/the-big-idea-the-next-scientific-revolution

[7]https://www.techopedia.com/definition/31256/real-time-data

[8]CERN openlab "White Paper: Future ICT in scientific research", September 2017

[9]Wild, Wolfgang (2018). "Cherenkov Telescope Array (CTA): building the world's largest ground-based gamma-ray observatory". Proc. SPIE 10700, Ground-based and Airborne Telescopes VII, 107000X (6 July 2018); doi:10.1117/12.2313470

[10]A. Bulgarelli et al., "The On-Site Analysis of the Cherenkov Telescope Array", The 34th International Cosmic Ray Conference, 30 July- 6 August, 2015, The Hague, The Netherlands.

[11]https://www.cta-observatory.org/science/gamma-rays-cosmic-sources/

[12]A. Di Stefano, “Linguaggio ANSI C, guida alla programmazione embedded”, 2006

[13]Connolly & Begg, “Database Systems: A Practical Approach to Design, Implementation, and Management, 2014

[14]https://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL

[15]https://dev.mysql.com/doc/refman/5.7/en/storage-engines.html

[16]https://redis.io/topics/pubsub

[17]https://aws.amazon.com/it/redis/

[18]http://valgrind.org/docs/manual/

[19]https://www.scrumguides.org/scrum-guide.html

[20]S. Chacon, B. Straub, “Pro Git”,Version 2.1.107, 2019-01-30

[21]https://project-management.com/waffle-software-review/

[22] https://www.redhat.com/it/topics/containers/whats-a-linux-container

[23]https://www.redhat.com/it/topics/containers/what-is-docker

[24]M. Shahina , M. A. Babara , L. Zhub, Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices, on arXiv.org, Cornell University. URL: https://arxiv.org/abs/1703.07019

[25]https://jenkins.io/blog/2016/05/26/introducing-blue-ocean/

[26]A. Bulgarelli et al., ”The Real-Time Analysis of the Cherenkov Telescope Array Observatory”, The 33rd International Cosmic Ray Conference, July, 2013

[27]K. Kosack et al, “CTA High-Level Data Model Definitions”, Version 0.4, August, 2017

[28]https://insidemysql.com/tag/x-devapi/

[29]https://www.techopedia.com/definition/24302/ini-file

[30]https://wiki.centos.org/FrontPage

[31]https://docs.anaconda.com/anaconda/

[32] https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance

[33]https://redis.io/topics/lru-cache