UNIVERSITÀ DEGLI STUDI DI UDINE

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di laurea specialistica in Fisica Computazionale

TESI DI LAUREA

Using the Hadoop/MapReduce approach for monitoring the CERN storage system and improving the ATLAS computing model

Utilizzo dell'approccio Hadoop/MapReduce per il monitoraggio del sistema di storage del CERN e per il miglioramento del modello di calcolo di ATLAS

Laureando: Stefano Alberto Russo
Relatore: prof. Marina Cobal
Correlatore: dr. Massimo Lamanna

Anno Accademico 2011/2012

Abstract

The processing of huge amounts of data, an already fundamental task for the research in the elementary particle physics field, is becoming more and more important also for companies operating in the Information Technology (IT) industry. In this context, if conventional approaches are adopted several problems arise, starting from the congestion of the communication channels. In the IT sector, one of the approaches designed to minimize this congestion on is to exploit the *data locality*, or in other words, to bring the computation as closer as possible to where the data resides. The most common implementation of this concept is the Hadoop/MapReduce framework.

In this thesis work I evaluate the usage of Hadoop/MapReduce in two areas: a standard one similar to typical IT analyses, and an innovative one related to high energy physics analyses. The first consists in monitoring the "history" of the storage cluster which stores the data generated by the LHC experiments, the second in the physics analysis of the latter, and in particular of the data generated by the ATLAS experiment.

In Chapter 2, I introduce the environment in which I have been working: the CERN, the LHC and the ATLAS experiment, while in Chapter 3 I describe the computing model of LHC experiments, giving particular attention to ATLAS. In Chapter 4, I cover the Hadoop/ MapReduce framework, together with the context in which it has been developed and the factors which has lead to a more and more growing importance of approaches centered on data locality. In Chapter 5, I present the work which I have done in the field of the monitoring of the storage cluster for the data generated by the LHC experiments, both in real time and in respect to its "history", walking through the steps that have lead to adopting Hadoop/MapRedue in this contex. The Chapter 6 is the kernel of this thesis: I explain how a typical high energy physics analysis can be ported to the MapReduce model and how the entire Hadoop/MapReduce framework can be used in this field. Finally, I conclude this thesis work by testing this approach on a real case, the top quark cross section measurement analysis, which I present in Chapter 7 together with the results obtained.

Italiano

L' elaborazione di grandi quantitativi di dati, ormai imperativo per la ricerca nel campo della fisica delle particelle elementari, è un tema sempre più di attualità anche per le industrie che lavorano nell' ambito dell'Information Technology (IT). In questo contesto, nel caso si utilizzino degli approcci convenzionali, sorgono dei problemi, a partire dalla congestione dei canali di comunicazione. Nel settore IT, uno degli approcci ideati per minimizzare questa congestione consiste nello sfruttare la *località dei dati*, ovvero nel portare la computazione il più vicino possibile a dove questi risiedono. Una delle implementazioni più diffuse di questo modello è il framework Hadoop/MapReduce.

In questo lavoro di tesi valuto due applicazioni di Hadoop/MapReduce: una standard legata ad analisi tipiche del settore IT, ed una del tutto innovativa legata all'analisi nel campo della fisica delle alte energie. La prima consiste nel monitoraggio della "storia" del cluster di storage dedicato all'immagazzinamento dei dati generati dagli esperimenti LHC, la seconda nel suo utilizzo per l' analisi di questi ultimi, ed in particolare per i dati generati dall' esperimento ATLAS.

Nel Capitolo 2 introduco l'ambiente in cui ho lavorato: il CERN, l'LHC e l'esperimento ATLAS, mentre nel Capitolo 3 descrivo il modello computazionale degli esperimenti LHC con particolare attenzione alle caratteristiche legate ad ATLAS. Nel Capitolo 4 espongo il framework Hadoop/ MapReduce, assieme al contesto in cui è stato sviluppato ed ai fattori che determinano una sempre più crescente importanza degli approcci basati sulla località dei dati. Nel Capitolo 5 espongo il lavoro da me svolto nell'ambito del monitoraggio del cluster di storage dei dati generati dagli esperimenti LHC, sia in tempo reale che rispetto alla sua "storia", percorrendo le tappe che hanno portato all'adozione di Hadoop/MapReduce in questo contesto. Il Capitolo 6 è il cuore di questa tesi: spiego infatti come si può portare una tipica analisi di fisica delle alte energie al modello MapReduce e come si possa utilizzare l'intero framework Hadoop/MapReduce in questo campo. Concludo infine questo lavoro di tesi testando il metodo da me proposto sul caso reale dell'analisi della sezione d'urto del quark top, che espongo assieme ai risultati ottenuti nel Capitolo 7.

Contents

Chapter 1

Introduction

The processing of huge amounts of data, an already fundamental task for the research in the elementary particle physics field, is becoming more and more important also for companies operating in the Information Technology (IT) industry, such as Google, Facebook, and Yahoo. This is due to a phenomena commonly referred as the *Data Deluge* or *Big Data revolution*, that is caused by the extreme technological innovation of the last decade, which has led to data sources more and more widespread and with a constantly increasing resolution. In this context, processing data sets in the order of several Terabytes (TB) is a common requirement. If conventional approaches are adopted, several problems arise: the use of a relational data base results unsatisfactory when both flexibility and costs (or resources needed to fulfill the requested processing times) are considered, while using a distributed system implies frequent heavy data transfers which can cause congestion on the communication channels.

This class of analyses is similar to the ones which can be found in a particle physics experiment, such as the high energy physics experiments at the Large Hadron Collider (LHC) accelerator in Geneva. In the computational model of LHC, after a first reconstruction phase, data is organized in data sets which are usually in the order of tens or hundreds of TB, and the processing time is a critical factor to allow the refinement and finalization of the physics results. Nevertheless there is a fundamental difference between the two sectors, which lies in the type of the data: in LHC experiments data is highly structured and to access the physics informations complex program are required.

In the IT sector, one of the approaches designed to minimize the congestion on communication channels is to exploit the *data locality*, or in other

words, to bring the computation as closer as possible to where the data resides. This type of approach is usually implemented by a model originally introduced by Google and named *MapReduce*. In this computational model, the analysis is parallelized in a particularly efficient way. Since it is planned and organized depending on how data are distributed in terms of their distance from the processors, the data transfers are minimized and the efficiency of the computing infrastructure is therefore improved. In the MapReduce model the data is hence analyzed in parallel directly on the nodes holding them (*Map* phase), and the final result is obtained by combining the partial ones (*Reduce* phase). The analysis is therefore not driven by the computational resource, but by the storage ones (*data-driven* parallelization). One of the most common implementations of this model is the Hadoop/MapReduce framework, which will be therefore taken as reference.

In this thesis work I evaluate the usage of Hadoop/MapReduce in two areas: a standard one similar to typical IT analyses, and an innovative one related to high energy physics analyses.

The first consists in monitoring the "history" of the storage cluster which stores data generated by the LHC experiments. This type of analysis is based on processing the log messages generated in the past by more than 1500 cluster components. Despite this is a quite simple operation, given the huge amount of generated messages, it was leading to serious problems since it was performed by using a relational database. The work I have done in this field ranges from replacing the log messages transport layer, through the complete redesign of the analysis engine (developed to operate both in real time and on the historical data), to the configuration and deployment of an Hadoop cluster which let to solve the previous problems.

The second area of application of the Hadoop/MapReduce framework I have evaluated is the analysis of data generated by the LHC experiments, and in particular by the ATLAS experiment. Although the framework was originally developed for analyzing textual files, it can be extended to other formats, as the structured data of the experiments (which is currently analyzed using the ROOT framework following a classical approach). Both the feasibility and the benefits of using Hadoop/MapReduce for this type of analysis have been evaluated by testing it on a real case: the top quark cross section measurement analysis performed by the ATLAS Udine Group. This analysis, which have been the first within the ATLAS Collaboration carried out by using Hadoop/MapReduce, has allowed both an in depth testing of the method and the highlight of its benefits. I presented this analysis at the "ATLAS Software and Computing week" in June 2012.

Italiano

L' elaborazione di grandi quantitativi di dati, ormai imperativo per la ricerca nel campo della fisica delle particelle elementari, è un tema sempre più di attualità anche per le industrie che lavorano nell'ambito dell'Information Technology (IT), come Google, Facebook, Yahoo etc. Questo è dovuto ad un fenomeno comunemente soprannominato *Data Deluge* (diluvio di dati) o *Big Data revolution* (rivoluzione dei grandi quantitativi dati), la cui causa è legata all'estremo avanzamento tecnologico dell'ultimo decennio, che ha portato a sorgenti di dati sempre più diffuse ed a sensori a sempre più alta risoluzione. In questo contesto è comune dover trattare insiemi di dati di dimensioni nell'ordine di diversi Terabyte (TB), spesso in maniera iterativa e con un tempo di processamento limitato. Nel caso si utilizzino degli approcci convenzionali sorgono dei problemi: l'utilizzo di una base di dati relazionale non è infatti soddisfacente né in termini di flessibilità né soprattutto in termini di costi (o, se vogliamo, di risorse richieste per soddisfare i tempi di processamento richiesti), mentre l'utilizzo di un sistema distribuito richiede pesanti e freqenti trasferimenti di dati che possono causare congestione sui canali di comunicazione.

Questo tipo di analisi ha delle similitudini con l'analisi dei dati in un esperimento di fisica delle particelle, come per esempio gli esperimenti di fisica delle alte energie all'acceleratore Large Hadron Collider (LHC) di Ginevra. Nel modello computazionale dagli esperimenti LHC i dati, dopo una iniziale fase di ricostruzione, sono organizzati in dataset di dimensioni spesso dell'ordine di decine o centinaia di TB, ed il tempo di processamento è critico per permettere di affinare e finalizzare i risultati scientifici. C'è tuttavia un'importante differenza tra i due settori, che risiede nel fatto che i dati in un esperimento LHC sono altamente strutturati e richiedono complessi programmi per accedere all'informazione fisica.

Nel settore IT, uno degli approcci ideati per minimizzare la congestione sui canali di comunicazione consiste nello sfruttare la *località dei dati*, ovvero nel portare la computazione il più vicino possibile a dove questi risiedono. Questo tipo di approccio viene solitamente implementato tramite un modello originariamente introdotto da Google e chiamato *MapReduce*. In questo modello computazionale l'analisi viene parallelizzata in modo particolarmente efficace poiché viene organizzata e pianificata in funzione di come i dati sono distribuiti in termini di distanza dai processori, riducendo al minimo i trasferimenti con un conseguente incremento nell'efficienza dell' infrastruttura di calcolo. Il paradigma MapReduce prevede pertanto che i dati siano analizzati in parallelo direttamente sui processori delle macchine che li ospitano (fase di *Map*), e che il risultato finale sia poi ottenuto combinando in cascata quelli parziali (fase di *Reduce*). Non sono quindi le risorse computazionali a guidare l'analisi, ma quelle di storage (si parla infatti di parallelizzazione *data-driven*). Una delle implementazioni più diffuse di questo modello è il

framework Hadoop/MapReduce, che verrà quindi preso a riferimento.

In questo lavoro di tesi valuto due applicazioni di Hadoop/MapReduce: una standard legata ad analisi tipiche del settore IT, ed una del tutto innovativa legata all'analisi nel campo della fisica delle alte energie.

La prima consiste nel monitoraggio della "storia" del cluster di storage dedicato all'immagazzinamento dei dati generati dagli esperimenti LHC. Questo tipo di analisi si basa sul processare i messaggi di log generati in passato dagli oltre 1500 componenti del cluster. Nonostante quest'ultima sia un'operazione relativamente semplice, dato il grande quantitativio di messaggi genearti essa poneva seri problemi poiché effettuata tramite un database relazionale. Il lavoro da me svolto in questo campo ha spaziato dalla sostituzione del sottostrato di trasporto dei messaggi di log, passando per la completa riprogettazione del motore di analisi (sviluppato per operare sia in tempo reale che sullo storico), fino alla configurazione e messa in produzione di un cluster Hadoop che ha permesso di risolvere i precedenti problemi.

La seconda applicazione del framework Hadoop/Mapreduce che presento consiste nel suo utilizzo per l'analisi dei dati generati dagli esperimenti LHC, ed in particolare dall' esperimento ATLAS. Infatti, nonostante il framework sia stato concepito per l'analisi di file testuali, può essere esteso a formati diversi, come i dati strutturati degli esperimenti (che sono attualmente analizzati tramite l'ambiente ROOT seguendo un approccio classico). Verranno studiati sia la fattibilità che i benefici dovuti all'adozione di Hadoop/MapReduce per questo tipo di analisi, testandolo su di un caso reale: l'analisi della sezione d'urto del quark top eseguita dal gruppo ATLAS Udine. Questa analisi, la prima ad essere realizzata all'interno della Collaborazione ATLAS usando Hadoop/MapReduce, ha sia permesso un test approfondito del metodo che messo in risalto i suoi benefici. E' stata inoltre da me presentata alla "ATLAS Software and Computing week" nel Giugno 2012.

Chapter 2

CERN, LHC and ATLAS

The European Organization for Nuclear Research (CERN) is one of the world's largest and presently the most renowned centre for scientific research. Its core activity is in the field of the fundamental physics, to find out what the Universe is made of and how it works. At CERN, the most complex and up to date scientific instruments are used to study the basic constituents of matter. Founded in 1954 to create an European scientific centre of excellence, after the dark years of the II World War, the CERN Laboratory sits in between the Franco-Swiss border, close to Geneva. It was one of Europe's first joint ventures and now has 20 Member States [1]. The instruments used at CERN are particle accelerators and detectors: accelerators produce collisions of particles (protons, ions) at very high energy, while detectors observe and record what is produced from these collisions.

In this Chapter, I introduce the particle physics in Section 2.1, the LHC accelerator in Section 2.2, and the ATLAS experiment including the detector components and the data acquisition schema in Section 2.3.

2.1 Particle Physics: the Standard Model an beyond

At present, the best description of the subnuclear world, the fundamental components of the Universe and their interactions, is provided by a theory called Standard Model (SM). In the SM the building blocks of matter are 12 fermions (spin 1/2 particles). These particles are six leptons which include the electron, the muon, the tau and the corresponding neutrinos, and six

quarks. Both quarks and leptons occur in pairs, differing by one unit of electric charge e, and are replicated in three generations with a strong hierarchy in mass. The fermions and gauge bosons included in this theoretical framework are listed in Figure 2.1.

The forces among the fundamental fermions are mediated by the exchange of the gauge bosons of the corresponding quantized gauge fields. The gravitational force cannot be included in the SM, but its strength is in any case small, compared to that of the other interactions at the typical energy scales of particle physics field.

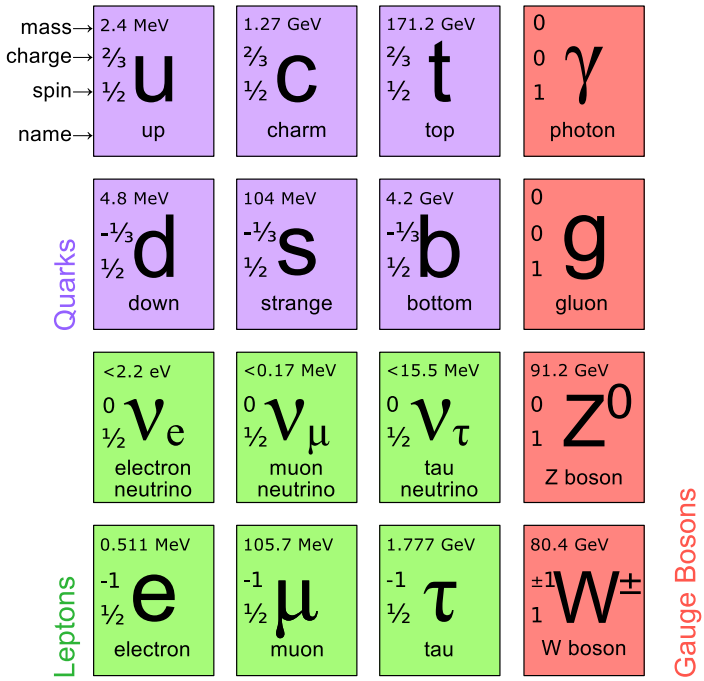

Figure 2.1: The known fundamental fermions and gauge bosons and their properties: mass, charge and spin.

The SM is a particular quantum field theory, which includes the strong interaction and the electroweak interaction theories. The strong interaction theory, coupling three different colour charges ("red", "green" and "blue") carried by the quarks and the eight massless gauge bosons (gluons), is called Quantum Chromodynamics (QCD). The gluons carry both a colour and an anticolour charge, and at increasingly short distances (or large relative momenta), the interaction becomes arbitrarily weak (asymptotic freedom), making possible a perturbative treatment. Via the strong interaction, quarks

form bound colour-singlet states called hadrons, consisting of either a quark and an antiquark (mesons) or three quarks (baryons).

The proton can be considered to accommodate three "valence" quarks (*uud*, see Figure 2.1) which dictate its quantum numbers. These valence quarks typically carry much of the momentum of the proton. In the proton are also present virtual or "sea" quarks and gluons. When two protons (or a proton and an antiproton) collide, a hard interaction occurs between one of the constituents of the first proton and one of the constituents of the second proton, which are called *partons*. The soft interactions involving the remainder of the hadron constituents produce many low energy particles which are largely uncorrelated with the hard collision.

An important experimental consequence of the fact that only colour-neutral states and no free quarks are observed in nature (which is commonly referred to as the "confinement" of quarks in hadrons), is that quarks produced in high energy particles interactions manifest themselves as collimated streams of hadrons called *jets*. The energy and direction of a jet are correlated to the energy and direction of its parent quark. The process by which the quark evolves into a jet is called "hadronization", and consists of a parton shower, which can be perturbatively calculated, and a fragmentation process, which is a non-perturbative process modelled using Monte Carlo (MC) techniques.

2.2 The LHC collider

The LHC collider is currently the largest and highest-energy particle accelerator in the world. It started its operations in 2008 and can provide both proton-proton (*pp*) and heavy ion (HI) collisions, by smashing two beams of particles circulating in opposite directions. In the LHC, the beams cross in four points, where four big experiments (detectors) have been built: ATLAS [2] at Point 1, CMS [3] at Point 5, LHCb [4] at Point 8 and ALICE [5] at Point 2. ATLAS and CMS are multi-purpose experiments, designed to study high transverse momentum events for the search of the Higgs boson and new physics beyond the SM. LHCb and ALICE are instead physics-specific experiments: the first is dedicated to study the physics related to the b-quark, one of the fundamental blocks of matter as foreseen in the SM, while the latter has been designed for studying the formation of the so-called quark-gluon plasma (a "soup" of asymptotically free quarks and gluons which is predicted at extremely high temperature and/or density),

by analyzing HI collisions.

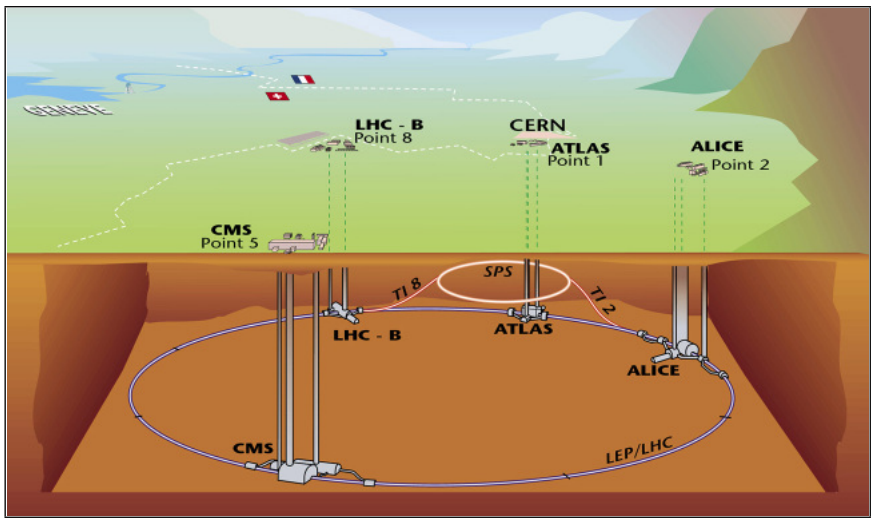

Figure 2.2: Schematic view of the CERN and LHC facilities.

The 27 km long circular tunnel at a depth varying between 50 and 175 meters below the ground, which was originally built for the Large Electron Positron Collider (LEP), houses today the LHC. The accelerator is mainly composed by two beam pipes in which the particles are kept on their circular trajectory by superconducting dipole magnets, operating at a temperature of 1.9 K thanks to a cryogenic system based on liquid Helium, and by a 400 MHz superconducting cavity system which gives the boost. In the four different collision points, where the two beams cross in a straight section, quadrupole magnets are used to keep the beams focused close to the interaction points.

Before being injected into the LHC, particles are accelerated step by step up to the energy of 450 GeV by a series of accelerators, as schematized in Figure 2.3. The very first step consists in generating the particles, and here I will take the protons as exemple. They are obtained by ionizing Hydrogen atoms (the proton source is shown in Figure 2.4) and then accelerated by the linear accelerator LINAC2, the first element of the accelerating chain, which brings them to an energy of 50 MeV. From LINAC2 protons are injected in the Proton Synchrotron Booster (PSB), which gives them an energy of 1.4 GeV, and then into the Proton Synchrotron (PS), where they are accelerated to 26 GeV. Finally, the Super Proton Synchrotron (SPS) raise their energy to 450 GeV before the last injection step, in the LHC.

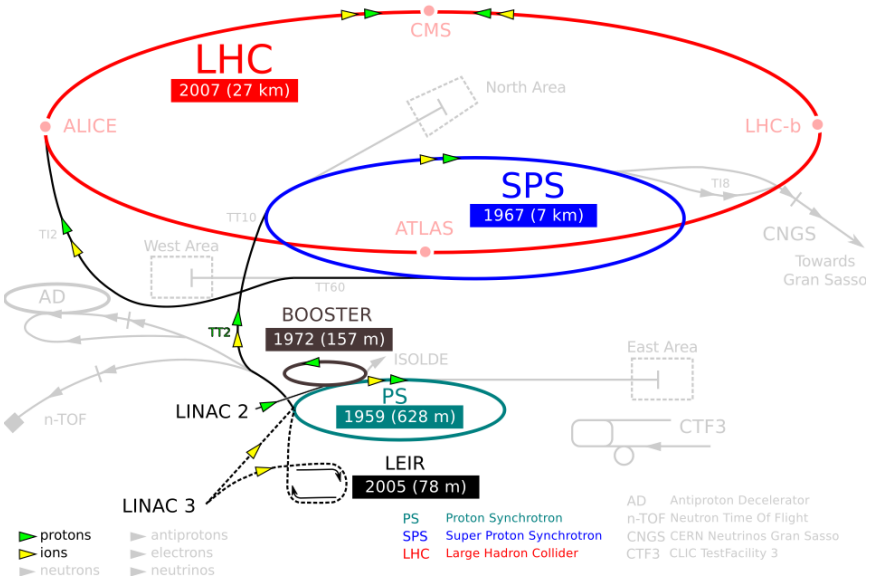

Figure 2.3: Schematic view of the CERN accelerators complex. Sections relevant for LHC operations are highlighted with different colours.

Figure 2.4: The LHC proton source.

Colliding particles are grouped together into bunches, each containing $\sim 10^{11}$ protons. The design number of bunches is 2808, which corresponds to a collision between two of them every 25 ns. During the commissioning phase, this number has been progressively increased: for example, at the end of 2010 the maximum number of colliding bunches was 348, while 1092 has been then reached in June 2011. Presently there are over 3250 bounches. For pp collisions, the design luminosity is 10^{34} cm^{-2}s^{-1}. The maximum instantaneous luminosity that has been reached in 2010 is slightly above $2 \cdot 10^{32}$ cm^{-2}s^{-1}. Higher peaks have been achieved in the next years: in 2011 of $\sim 4 \cdot 10^{33}$ cm^{-2}s^{-1}, and in 2012 of $\sim 7.7 \cdot 10^{33}$ cm^{-2}s^{-1}. The design centre-of-mass energy for the collisions is 14 TeV and collisions at 7 TeV centre-of-mass energy have been provided during 2010 and 2011 runs. In 2012 the machine has worked at 8 TeV, collecting up to now 5.6 fb^{-1} of data. A total of about 20 fb^{-1} of data have been collected since the first run in 2010.

2.3 The ATLAS experiment

The ATLAS (A Toroidal LHC ApparatuS) experiment is positioned in an underground cavern at a depth of 100 m. With its height of 25 m, its length of 44 m and its weight of about 7000 tons, it is one of the biggest detectors ever built. The construction started in 2003 after the completion of the cavern, and went on until July 2007. At the beginning, and for several years, ATLAS has been recording cosmic-ray events for testing and calibration purpose. Since November 2009, pp collision events from LHC started to be studied for physic analysis by the experiment.

ATLAS has a cylindrical symmetry and within the detector a right-handed cartesian coordinate system is used, where the x-axis points towards the centre of the LHC ring and the y-axis points upward, as detailed below.

- The nominal interaction point is defined as the origin of the coordinate system.
- The z-axis is parallel to the beam and the x- and y- axes are perpendicular.
- The x-y plane is called the transverse plane.
- The azimuthal angle ϕ is measured around the z-axis, the polar angle θ is measured from the z-axis.

- The pseudorapidity, defined as $\eta = -\ln\tan(\theta/2)$, is often preferable as a polar coordinate for massless objects or objects whose momentum is much higher than their mass, since the difference in pseudorapidity of two particles is a Lorentz invariant.
- The distance ΔR in $\eta - \phi$ space is defined as $\Delta R = \sqrt{\Delta\eta^2 + \Delta\phi^2}$.

Particles are often characterized by their transverse momentum p_T and transverse energy E_T (which are the projections in the transverse plane of the momentum and energy), since these variables are a better indicator of interesting physics than the standard energy and momentum and since they are assumed to be null for the colliding partons in the initial state.

The ATLAS detector is composed of different sub-detectors, as shown in Figure 2.5. Each of them plays an important role in reconstructing the products of collision events.

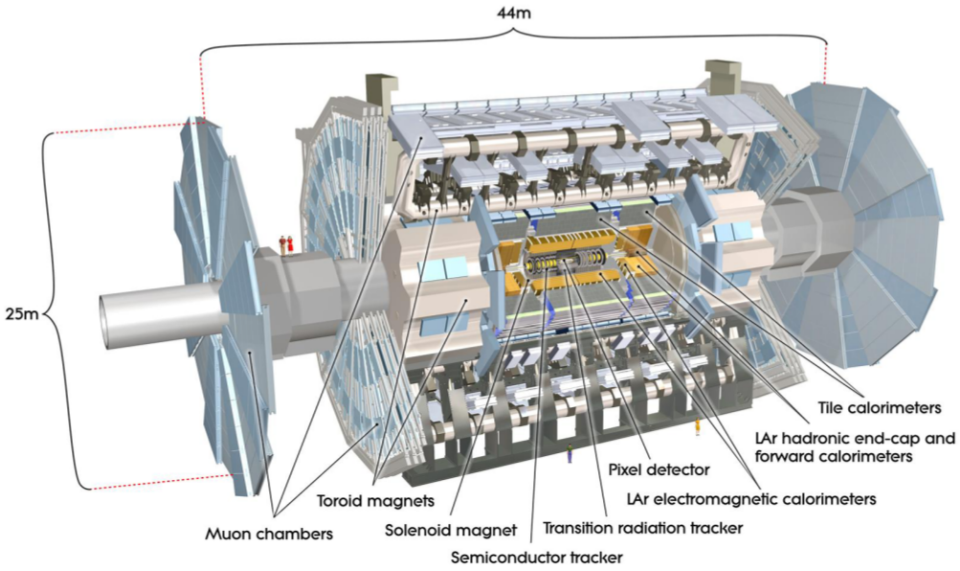

Figure 2.5: Schematic view of the ATLAS detector and sub-detectors.

2.3.1 Detector components

The sub-detectors are arranged in cylindrical layers around the interaction point, and the innermost one is enclosed by a solenoidal magnet, which provides a strong magnetic field to bend charged particles and measure their momentum and charge. In the following, the main ATLAS components are described.

The *Inner Detector* (ID) is the closest system to the beam pipe. It is used to reconstruct the trajectory of charged particles, and is divided into the Pixel, the SemiConductor Tracker (SCT) and the Transition Radiation (TRT) detectors. The Pixel detector is designed to reconstruct the primary and secondary vertices of collisions. To obtain the best resolution, it directly surrounds the Beryllium beam pipe and is composed by three cylindrical layers and two end caps, each consisting of three disks. Approximately 80.4 millions data channels are provided by 1744 silicon sensor modules. These modules are identical for the barrel part and the disks, and each of them is made of 47232 pixel sensors to perform precise measurement points for reconstructing accurate tracks. The SCT detector follows, from the beam pipe outwards. It is composed by four cylinders in the barrel region and two end caps in the forward region, each composed of nine disks made out of silicon strips. Counting 4088 modules, it provides measurements for particles originating in the beam-interaction region. The last and outermost component of the ID is the TRT detector, consisting in 298,384 proportional Drift Tubes (DFT) straws which provides approximatively 351000 data channels. The straws in the barrel region are arranged in three cylindrical layers, while in the end-cap regions are radially oriented in 80 wheel-like structures.

The *Calorimeters* surround the ID. These two detectors are made to measure the energy and position of particles. Both the calorimeters consists of a dense absorber material to fully absorb incident particles and an active material to produce an output signal proportional to the energy of the incident particle. The position measurement is achieved by registering which module and cell of the calorimeter was hit. The *Electromagnetic Calorimeter* (EM) is the innermost, and it is a Liquid Argon (LAr) sampling calorimeter dedicated to the precise measurement of electrons an photons with approximatively 170000 channels. Then follows the *Hadronic Calorimeter* (Had), a Fe-scintillator calorimeter dedicated to the measurement of hadrons and the reconstruction of jets of particles with approximatively 20000 channels.

The *Cryostat* is situated between the EM and the Had calorimeters and serves for cooling the LAr. It has a thickness of about half an interaction length and it can lead to significant energy losses in hadronic showers.

The *Muon Spectrometer* (MS) encloses the calorimeters and is designed to reconstruct and identify muons, which usually escape the previous detector layers. The MS is embedded in a toroidal magnetic field and consists in tracking chambers, to provide precise measurements of momentum and charge, and detectors used for fast triggering. These measurements are based on the reconstruction of muons trajectories curved by the magnetic field which, thanks to its configuration, it is mostly orthogonal to the trajectories, centered on the beam axis and perpendicular to the solenoidal field that serves the ID. Monitoring DFTs along the detector and Cathode Strip Chambers close to the interaction point, with high granularity, sample the muons passage. For what concerns the triggering, this feature is useful for spotting interesting physics signals, since isolated muons frequently indicate an interesting event. For this purpose, Resistive Plate Chambers (RPC) are used in the barrel region while Thin Gap Chambers (TGC) are used at the end-caps.

2.3.2 Trigger

Summing up data channels from every ATLAS sub detector means that, for every collision, something like 100 million channels have to be read out by the data acquisition software, resulting in ~ 1.5 MB events. Protons collide in ATLAS every 25 ns (corresponding to 4×10^7 collisions per second), and these values would lead to a data flow of ~ 60 TB per second from the detector. Anyway, interesting physics events occur mostly at rates of 10, 1 or < 0.1 events per second, which is a really tiny fraction of the total collisions produced. This means that even if storing and processing the ATLAS detector data flow without any filtering would be an impossible task with current technologies, it instead becomes possible by applying a proper selection of interesting events.

For evaluating and recognizing only the most interesting events, the ATLAS Collaboration has developed a three-level trigger system, configurable at every level to provide a constant stream of data under any beam conditions. Since interesting events must be quickly identified (looking for known signatures), the design of the trigger is itself a challenging task: a rapid decision must be made for each event, taking also into account that rejected events are, of course, lost forever.

The ATLAS trigger system [6] is designed to record events at a rate of up to 400 Hz, with a reduction of more than five orders of magnitude with

respect to the collision rate. At each level, physics objects are reconstructed with improved granularity and precision over a larger fraction of the detector, ending up in a complete event reconstruction in the final trigger stage. In the following, the different trigger stages are described:

The first level (L1) trigger is a pure-hardware trigger designed to make a decision on each event in less than 2.5 μs, providing an output at a rate up to 75 kHz.

The L1 provides regions of interest (RoIs) to the next level by making an initial decision based on timing from an electrostatic beam pick-up, coarse detector information from muon trigger chambers and towers of calorimeter cells, together with multiplicity information from the Minimum Bias Trigger Scintillators (MBTS) and very forward detectors (The MBTS detectors consist of 2 cm thick polystyrene scintillators mounted 3.6 meters from the nominal center of the detector [7]).

The second and third levels are software high-level triggers (HLT):

- The second level (L2) triggers make a decision in less than 40 ms and provide output rates at up to 3.5 kHz. They run a simplified version of the event reconstruction software in the RoIs defined by the L1 trigger. Events are then skimmed by applying improved calibrations and selection criteria, for example distinguishing electrons from photons by track matching.
- In the third trigger level, called the *Event Filter* (EF), the event is completely reconstructed offline and the decision made in less than four seconds. It provides output rates at 200-400 Hz.

A full sequence of triggers, from L1 through the EF, is called a *trigger chain*. After the EF, the events are divided into *streams*, each containing the outputs from several different trigger chains. On these streams the full offline event reconstruction is run, and the output is stored for further analyses. Calibration streams are processed first in order to provide new calibrations for the sub-detectors within 24-hour periods.

2.3.3 Object reconstruction

Here the way the physics objects are reconstructed with the ATLAS detector is briefly described. Only the objects used in the analysis presented in

Chapter 7 are considered, and only general reconstruction and identification algorithms used in ATLAS are mentioned.

The reconstruction of what happened in a collision event is a complex task, also because in addition to the main hard process which characterize the collision, further semi-hard interactions may occur between the other partons of the two protons colliding. Their products can overlap the main quark/gluon collision, leading to the so called "pile-up" phenomena which causes the detector to consider the two separate (hard and semi-hard) processes as part of the same collision.

Electrons

Electrons interacts with the detector material by producing an electromagnetic shower composed of electrons and photons ($e \rightarrow e\gamma$, $\gamma \rightarrow e^+e^-$) of decreasing energy as soon as the shower develops. After a while the electrons and photons produced are of such low energy that, since they are instead absorbed by the calorimeter, the shower stops.

Electron reconstruction is based on the identification of a set of clusters, where energy has been released, in the EM [8]. For every reconstructed cluster, the reconstruction algorithm tries to find a matching track in the ID. Electron's energy is then determined using the calorimeter information, and the angular information is extracted from the ID track. The algorithms for reconstructing and identifying electrons are designed to achieve both a large background rejection and a high and uniform efficiency for isolated high-energy ($E_T > 20$ GeV) electrons coming from the decay of a massive particle (e.g. a W or Z bosons) over the full detector acceptance. Once an isolated electron has been identified, it needs to be separated from misleading hadron decays in QCD jets and from secondary electrons (originating mostly from photon conversions in the tracker material).

The ATLAS electron identification algorithm can provide a good separation between isolated electrons and these fake signatures, by taking into account the information coming from the calorimeter, the tracker and the matching between tracker and calorimeter. This information allows to apply the selection based on several parameters:

- the energy released in the Had Calorimeter inside a cone drawn around the electron energy deposits,
- the shape of the electromagnetic shower,

- the value of the track impact parameter,
- the number of hits in the different layers of the ID,
- the difference between the position in the calorimeter cluster and the extrapolated track positions,
- the ratio of the cluster energy to the track momentum ratio ($E/p < 1$).

Electrons passing all the identification requirements are called *tight* electrons, while *loose* and *medium* electrons pass only some of the above listed requirements.

Muons

Muons loose energy in the detector by ionization. Their reconstruction is based on the information coming from the MS, the ID and the calorimeters. Depending on how the detector information is used in the reconstruction, different kinds of muons candidates can be identified. In the analysis described in Chapter 7, the so called *combined* muons candidates are considered: these are designed by combining the information from the MS and from the ID, through a fit to the hits in the two sub-detectors to derive their momentum and direction.

Two different algorithms are used in ATLAS to reconstruct the muons: both create combined tracks out of pairs of MS-only and ID-only tracks, matching via a χ^2 test and applying energy corrections due to losses in the calorimeters.

- STACO [9] performs a statistical combination of the track vectors to obtain the combined track vector;
- MuId [10] re-fits the combined track, starting from the ID track and then adding the MS measurements.

The two algorithms have shown very similar performances and can be both used for the analyses.

Jets of particles

When quarks or gluons are produced in the collisions, they can not be observed as free particles or through their decay products. As already

mentioned in Section 2.1, they manifest themselves in collimated streams of hadrons called jets. The energy from these hadronic particles is mainly deposited in the calorimeter system, and the resulting energy deposits are grouped into objects which identify the jets. These objects partly save the information of the energy and direction of the originating particles coming from the hard scatter. Thanks to the high granularity of the ATLAS calorimeters and to their high sensibility, high quality jets can be reconstructed. Cells are collected into larger objects like towers or topological cluster (topoclusters), because of two factors:

1. single cells signals can't be directly used because of noise effects that can alter the value (which could also happen to be negative);

2. determining the source of a signal without using informations from neighbor cells is complicated.

Calorimeter towers are built by projecting the cell energy onto a two-dimensional space, while topological clusters reconstruct three-dimensional energy deposits. The cluster is built starting from cells with a high signal-to-noise ratio, and by iteratively adding neighboring cells with a signal-to-noise ratio above a given threshold.

Jets from quarks b

If the jets are coming from the fragmentation of a quark b, they may have a distinct signature. Aim of the b-tagging algorithms is to identify and reconstruct jets containing b-flavored hadrons. Discrimination of b-quark jets from other light quark jets is mainly possible because of the relatively long life time of b-flavoured hadrons, which results in a flight path length (referred as L) in the orders of millimeters. Such a significant flight path leads to measurable secondary vertices and impact parameters of the decay products. The distance in the transverse plane (x,y) between the point of the closest approach of a track to the primary vertex is refereed as d_0, while the same parameter in the longitudinal one is refereed as z_0. By choosing different ways in which to evaluate the discrimination parameters (L, d_0 and z_0), secondary vertex properties and the presence of leptons within b-quark jets, various b-tagging algorithms (or "taggers") can be defined. In general, each of them defines a weight w which reflects the probability of the jet to have been generated by a b-quark.

Missing transverse energy

The presence of an unbalance in the total transverse momentum of all the particles produced in the collision ($\sum p_T \neq 0$ where the sum is performed on all the reconstructed objects in the event) is and indicator of the presence of neutrinos or other particles which are not expected to interact with the detector (possibly coming from new physics processes not foreseen in the SM). The procedure should take into account the difference between the initial state and final state total momentum, but since the initial momentum of the colliding partons along the beam axis is not known a priori and the initial momentum in the transverse plane is in good approximation null, a loss in the total energy can be measured just on this plane.

The missing transverse energy ($\not{E}_T$) is simply defined as:

$$\not{E}_T = \sqrt{(\not{E}_x)^2 + (\not{E}_y)^2}, \tag{2.1}$$

where $\not{E}_x$ and $\not{E}_x$ are the spatial components on the transverse plane. According to the reconstruction method presently used in ATLAS, both the x and y components include contributions from transverse energy deposits in the calorimeters, corrections for energy losses in the cryostat and measured muons:

$$\not{E}_{x(y)} = \not{E}^{calo}_{x(y)} + \not{E}^{cryo}_{x(y)} + \not{E}^{\mu}_{x(y)}. \tag{2.2}$$

The calorimeter term $\not{E}^{calo}_{x(y)}$ is built starting from calorimeter cells belonging to topoclusters (see b-jets reconstruction). Specific calibrations for cells energy are provided for every high-p_T physics reconstructed object, like electrons, photons, hadronically decaying τ-leptons, jets and muons. This is the so called *RefFinal* calibration, the most refined scheme developed in ATLAS for calculating the calorimeter missing transverse energy.

The $\not{E}_T$ muons term $\not{E}^{\mu}_{x(y)}$ is calculated from muons momenta, combining the information from MS and ID for isolated muons with $|\eta| < 2.5$, or using the MS information only for non-isolated muons and for muons outside the η range of the ID. The energy lost by the muons in the calorimeters ($\not{E}^{\mu(calo)}_{x(y)}$) is added to the calorimeter term in the latter case.

The $\not\!\!E_T$ cryostat term $\not\!\!E^{cryo}_{x(y)}$, calculated exploiting the energy correlation between the last layer of the LAr calorimeter and the first layer of the Had calorimeter, takes into account the energy losses which can occur in hadronic showers as previously explained.

2.3.4 Data taking and formats

A single data taking run in ATLAS can last for many hours. Typically, one long run is taken during an LHC fill, and if necessary the run is stopped between the fills for detector calibrations. In the ATLAS computing model [11], these runs are divided into luminosity blocks that are a few minutes long each. Luminosity blocks are the smallest units of data for an analysis, and each of them can be included or excluded in the final analysis. Data which survives the trigger cuts, divided in streams according to the triggers fired by the event, is collected by using various formats at different levels to fulfill the requirements of several kind of analyses: development of reconstruction algorithms, detector calibrations, debugging, and physics analysis.

The first level formats keep all the possible information about the collisions provided by the EF. The very first step handles the data in Byte Stream (BS) format, which is a RAW, low level format. Data is then converted into the Raw Data Object format (RDO), a structured representation of the BS data. From the RDO format onwards, data is stored in a structured way, using a C++ object-oriented data structure centered on the ROOT framework[1]. Starting from this point the first pass reconstruction of events take place, generating an intermediate format, the Event Summary Data (ESD). These files still contain all the information about the "history" of the event inside the detector, as the energy released in each cell of the calorimeter, but also provide information about reconstructed physics objects like jets, electrons, etc.

Following the natural evolution of the chain, the next format does not carry low-level informations anymore and provides only a summary of the reconstructed events. This format, the Analysis Data Object (AOD), is the starting point for all physics analyses. Two more versatile formats can be extracted from the ESD and the AOD: the dESD and the dAOD, respectively. They contain just a subset of the events matching some criteria, for example the ATLAS TOP Working Group ask for subsets containing one electron or

[1]ROOT is an object-oriented program and library developed by CERN, which has been designed and is mainly used for particle physics data analysis.

one muon (because of the final state signature from the Top quark decay, see Chapter 7), which correspond to events involving the "Egamma" and "Muon" trigger streams.

Given the huge amount of data produced by the detector, the ATLAS computing model relies on a lightened format final users specific analyses, the D3PD [12]. This format is obtained by running over dESD/dAOD, and consists in flat ROOT n-tuples. It is in practice the most common format used for physics analyses, since it is generated by skimming, thinning and slimming the original dESD/dAOD data sets to keep only events and informations interesting for a particular analysis and so reducing noticeably their size.

- Skimming is the selection of only desired events from a larger data set;
- Thinning is the cutting of unnecessary objects from the desired events, as the ones which are not involved in the decay to be studied;
- Slimming is the dropping of proprieties not needed for the analysis from objects which have been already skimmed and thinned.

ATLAS data for physics analysis needs to be filtered according to detectors conditions and is available for access and computing to collaboration's members through the Worldwide LHC Computing Grid. Several data quality flags are assigned for each sub-detector and for each reconstructed objects, in each detector region, on a luminosity block basis. These flags are assigned by the data quality shifters, according to the status of the systems. Automated procedures have been developed to speed up the process, but the flags still needs to be assessed by a human being.

Chapter 3

LHC data computing model

In this Chapter the computing model behind the LHC and its experiments is discussed. This infrastructure allows to store and analyze the huge amounts of data generated by the LHC experiments.

In Section 3.1 the *Worldwide LHC Computing Grid* is presented, in Section 3.2 the data distribution policies are covered, fundamental for distributing the workload around the globe, and finally in Section 3.3 a more in depth dive in the data storage and distribution techniques at CERN is given.

3.1 The *Worldwide LHC Computing Grid*

The challenge of analysing the volume of data produced at the LHC is an immense task. In the design phase of the LHC, it rapidly became clear that the required computing power to deal with the huge amount of data which was going to be produced by the experiments was far beyond the capacity available at CERN. In 1999 the idea of a computing system spread around the world to combine resources from all the participating institutes, for meeting the data analysis challenge on this unprecedented scale, began to emerge: the "LHC Computing Grid" aim was to link Grid infrastructures and computer centers worldwide to distribute, store and analyze LHC data.

This approach rapidly evolved from being just a concept and today the *Worldwide LHC Computing Grid* (WLCG) combines massive multi-petabyte storage systems and computing clusters with thousands of nodes connected by high-speed networks, from over 170 sites in 34 countries [13]. This distributed, Grid-based, infrastructure provides to more than 8000

physicists around the world near real-time access to LHC data and the power to process it, equally and regardless of their physical location.

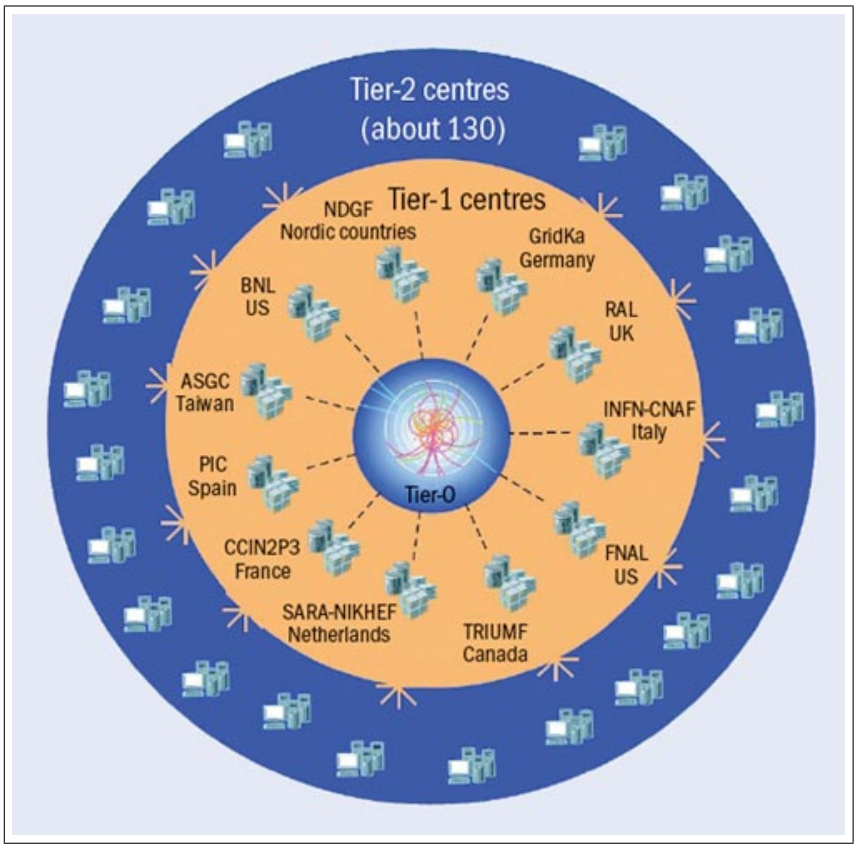

Figure 3.1: WLCG Tier structure.

The WLCG is managed and operated by a worldwide collaboration between the experiments and the participating computer centers, and it is intentionally spread worldwide for funding and sociological reasons. The WLCG is now the world's largest computing Grid and provides all the production and analysis environments for the LHC experiments. It's layout is based on the two main global Grids currently in operation, the European Grid Infrastructure (EGI) in Europe and the Open Science Grid (OSG) in the United States. The many associated regional and national Grids across the world are organized in four layers or *Tiers*: Tier 0, Tier 1, Tier 2 and Tier 3, as shown in Figure 3.1, which are shortly described in the following.

Tier-0:

This is the CERN Computer Centre. All data from the LHC passes through this central hub, but it provides less than 20% of the total computing capacity. CERN is responsible for the safe-keeping of the RAW data (first copy), first pass reconstruction, distribution of raw data and reconstruction output to the Tier-1s, and reprocessing of data during LHC down-times.

Tier-1:

These are eleven large computer centres with enough storage capacity and with round-the-clock support for the Grid. They are responsible for the safe-keeping of a proportional share of RAW and reconstructed data, large-scale reprocessing and safe-keeping of corresponding output, distribution of data to Tier-2s and safe-keeping of simulated data thereby produced.

Tier-2:

The Tier-2s are typically universities and other scientific institutes, which can store sufficient data and provide adequate computing power for specific analysis tasks. They handle analysis requirements and proportional share of simulated event production and reconstruction. There are currently around 140 Tier-2 sites covering most of the globe.

Tier-3:

The Tier-3s are not officially part of the WLCG, but they are de-facto part of the computing model, since are widely used by physicists to access WLCG data and to run their own analyses. They consists in local computing resources, which are mainly small clusters in university departments research institutes. There is no formal engagement between WLCG and Tier-3 resources.

3.2 Data distribution

The data distribution over the WLCG reflects the hierarchical structure and availability policies. Starting from CERN Tier-0 which holds and distributes the original RAW data with near 100% uptime, as moving outwards in the layout low-level data is processed giving way to higher-level structured formats and less strong uptime requirements. This holds up to the very last

step of the Tier-3, where only hard filtered data targeted on well defined analyses is available without any uptime requirements.

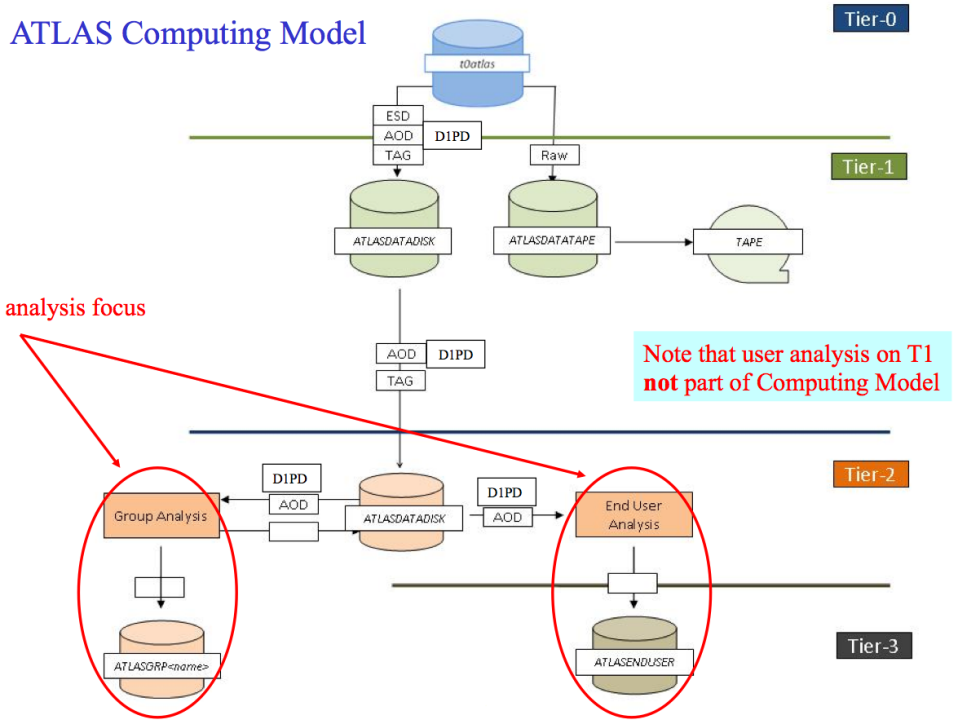

Figure 3.2: The ATLAS computing model (Image credits: James Cochran, ISU).

The ATLAS computing model, schematized in Figure 3.2, follows these criteria. The main formats involved (already introduced in Section 2.3.4) are summarized in the following, together with the distribution policy:

RAW

One copy is kept at CERN (tape) and one distributed over Tier-1s (disk). Small quantities can be copied to Tier-2/group space for special studies.

ESD

Produced from RAW at Tier-0 (first pass reconstruction) and Tier-1 (reprocessing). One ESD data copy is kept at CERN (tape), two are distributed over Tier-1s (disk). Small quantities can be copied to Tier-2. Files derived from ESD for detector and performance studies in the dESD format are distributed in ten copies across the Tier-2s.

AOD

Produced from ESD at Tier-0 (first pass reconstruction) and Tier-1 (reprocessing). At most two versions on disk at any given time can be stored. There are two plus one copies at Tier-1s and ten plus ten copies at Tier-2s. Files derived from the AOD in the dAOD format, targeted toward physics analysis and defined by needs of group analyses, are to be stored on the Tier-2 group space.

D3PD

D3PDs are normally produced by various working groups (for example by the ATLAS TOP Working Group running over dESD/dAOD containing one electron or one muon). They are under group/individual control and stored in group space or locally, at Tier-3s.

3.3 Tier-0: data storage at CERN with CASTOR

The LHC experiments produce roughly 15 PB of data every year, and the main task of CERN Tier-0 is to store and make it available to Tier-1s for backup and further elaboration. When LHC was still in the design phase, each experiment was asked to write a requirement sheet specifying the resources needed for handling its data rates and the reliability level needed by the collaboration. For ATLAS, the baseline model assumed a single, primary stream containing all physics events flowing from the the EF (see Section 2.3.2) and several other auxiliary streams, the most important of which containing calibration trigger events to produce calibrations of sufficient quality to allow a useful first-pass processing (at Tier-0) of the main stream with minimum latency. The expected data transfer to the Tier-0 was of about 320 MB/s and the target was to process 50% of the data within eight hours and 90% within 24 hours [14].

It is clear that satisfying the ATLAS requirements is a complex task, and once taken into account that CMS, ALICE and LHCb had similar needs, the task becomes really challenging. To achieve the high data rates with the low latencies required and to store this immense volume of data, while at the same fitting in the available founds, the CERN IT department developed *CASTOR*.

The CERN Advanced STORage manager (CASTOR) [15] is a hierarchical storage management system which uses an hybrid technology: disks and tapes. Disks are used for fast access tasks (incoming data, low latency processing) while tapes are for slow (in the order of hours) access tasks,

which consist in mainly archiving (migrating) files. Disks are also used as a cache of the tape pool: files which are frequently requested from tapes (recalled) are "elected" to be moved on a disk storage element to minimize the latency of the system. Tapes are stocked into *libraries*, which are composed by shelves, one or more tape readers and a robotic arm to move the stocked tapes. The reading of a tape consists in a first stage in which the robotic arm takes out the tape from its shelf bringing it to a tape drive, and a second stage in which the actual reading takes place.

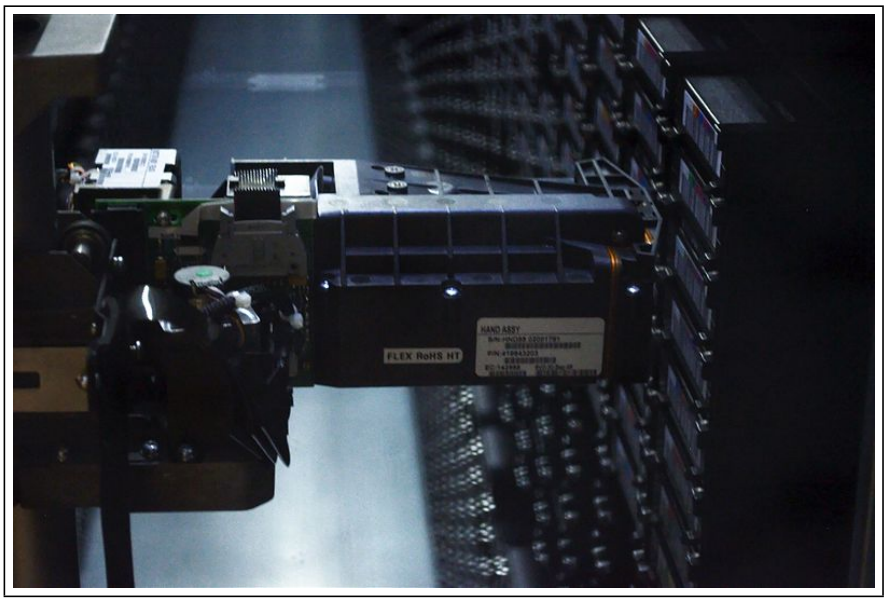

Figure 3.3: Closeup of a robotic arm in a StorageTek tape library which is in use at CERN (Image credits: Derrick Coetzee).

Relying on both disks and tapes brings various benefits: tapes costs in the order of ten times less than disks, they do not need power supply or wiring when stocked and their average fault ratio is very low compared to disks. Moreover, differentiating the storage between two technologies permits to handle market fluctuations due unexpected factors, like the recent alluvion in Thailand that almost doubled the price per disk.

On the other hand, a system like this is very complicated to manage. The design is based on a component architecture using a central database to safeguard the state changes of the CASTOR components. Access to disk pools is controlled by the *Stager*; the directory structure is kept by the *Name Server*. The tape access (write and recalls) is controlled by the *Tape Infrastructure*. These components have to cooperate to handle request for

files that could be located both on disk or tape. If the file is already on the disk, the system just provides it to the client. If the requested files is on the tape, then the system queues the request: it will be served as soon as possible, but trying to aggregate several requests per tape. This aggregation is important since the tape has to be physically moved, which is a very heavy operation from the time consumption point of view. The study of the algorithms to decide which files has to go on tape and which has to stay on disk, and how to group requests to minimize the response time of the system, is an important subject actively studied.

Chapter 4

Hadoop/MapReduce

In this Chapter, I give an overview of Hadoop/MapReduce, a technology born to address a recent phenomena known as *Data Deluge* or *Big Data* revolution.

In Section 4.1, I explain the benefits of this new technology, why it is will be so important in the coming years and why there is so much worldwide interest around it. In Section 4.2, I provide a brief description of Hadoop structure and of its components.

4.1 Context and motivaton

The last decade has ben characterized by a constant growth in technological innovation. Today, almost every application running on last generation mobile operating systems (on a smartphone, on a tablet, on a music player, etc.) is designed to connect to the internet: downloading a song from a portable music player, buying a book from an e-book reader, sharing a picture from a smartphone are all operation achievable in just few "taps". Moreover, the use of the World Wide Web is something which is become natural in everyday life: posting articles to a blog or a social network, reading newspapers online, searching flights and hotels online, etc. are nowadays common tasks. All these user interactions generates data which is extremely precious for market analysis, trend previsions, and in general for the business: it is a gold mine for data analysts. From another point of view, the internet content is exponentially growing and is becoming harder and harder to process by the search engines. In addition to this global trend, the capture devices (as sensors, cameras, GPS, etc.) are constantly both increasing

their resolution and becoming more and more pervasive, and therefore generates more and more data. The data sources just sketched above can be easily divided in two big, well defined categories:

1. Data from user interactions (comment and article, buy a book, like a post, etc.)
2. Data from capture devices (sensors, cameras, GPS, etc.)

The consequence of this technological innovation and its related exponential increase of the data flows is a phenomena which is commonly referred as the *Data Deluge* or *Big Data* revolution.

The widespread mutual interest in being able to analyze these huge amounts of data is today a central matter (in Figure 4.1 just two of the large number of evidences about this fact are reported), and it has lead to a boost in the technologies addressed to achieve this common goal.

Figure 4.1: Covers of the Economist and of The Harvard Business Review entirely dedicated to the data deluge or big data revolution.

Form a computing point of view, for high intensive cpu tasks it is common to think of solving them in parallel, using a cluster. Today the most common scenarios when talking about distributed computing models, regardless of their purpose, is to consider the storage and the computational resources as two independent, well logically-separated components. This implies the presence of a communication channel between the two, which

usually becomes a bottleneck that can be easily saturated by I/O bound applications especially when scaling up.

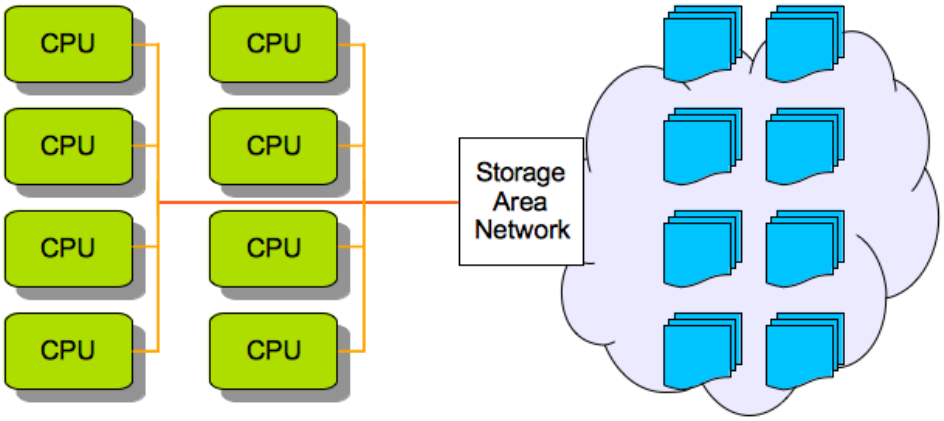

Figure 4.2: "Traditional" computing model with separate computing and storage resources.

For parallel, I/O intensive tasks (as Big Data analyses), another concept becomes therefore fundamental: *data locality*. Data locality means to let the data to be spread across all the computing nodes, allowing to analyze it within the nodes, from the local file system (and thus avoiding network data transfers). Given the context of an exponential growth in the data streams to analyze, this concept is become the the primary, fundamental requirement for developing the future distributed computing frameworks and clusters. Arguments in support of this statement can be found in almost every recent reading about distributed computing models, and in particular in "The Importance of Data Locality in Distributed Computing Applications" [16] where the authors (including Ian Foster, known as "the father of the Grid computing" [17]), explain that:

"As CPU cycles become cheaper and data sets double in size every year, the main challenge for efficient scaling of applications is the location of the data relative to the available computational resources – moving the data repeatedly to distant CPUs is becoming the bottleneck."

An extremely important concept to keep in mind when evaluating a data locality approach is that it does not necessarily speed up the computation. Data locality is about scaling up, is a concept at the level of cluster architecture, and assuming to have an infinite network bandwidth connecting the computing and storage resources it would bring no benefits at all. The real

world is anyway completely different, the network bandwidth is finite, and its value usually depends on the founds available for building a cluster. If the computation is slowed down by the time to access the data to analyze over the network, a data locality approach provides an immediately tangible speedup. Today, the real world situation is a mixture of data centers which invested in their networks and can handle Big Data analyses without taking advantage of data locality, and of data centers which could not invest in their networks and are already suffering from the data deluge. When taking into account that data flows are exponentially growing, it is evident that sooner of later data locality approaches will be the only possible approach to analyze the Big Data. The real metric when evaluating a data locality approach should be then the value of bandwidth saved in comparison to a standard computing model. This is the reason why, when giving the final performance results at the end of Chapter 7, only the bandwidth consumption is taken into account.

In this thesis I will cover two cases of sensors-generated Big Data, where a computing model taking advantage of data locality can bring huge benefits.

- The first case is the monitoring of the CASTOR cluster at CERN, where the sensors are the logging daemons of the 1500+ CASTOR nodes. The data produced is textual and its analysis, very similar to the common Big Data challenges, it is discussed in Section 5.4.
- The second case is the field of High Energy Physics (HEP) analyses, where the sensors are the detectors, which are producing huge amounts of data constantly increasing thanks to the increasing luminosities of the accelerators (especially at the LHC experiments, see Chapter 3). This particular type of analysis is more complicated that the usual Big Data analyses and will be discussed in Chapter 6.

4.2 Introducing the Hadoop/MapReduce model

Hadoop/MapReduce [18][19] is a top-level Apache project being built and used by a global community of contributors written in Java, inspired by Google's MapReduce [20] and Google File System (GFS) [21] papers. It's bigger goal is to avoid the distinction between storage and computing resources, overlapping them and bringing data locality. The components of Hadoop/MapReduce are:

- **Apache Hadoop**, a *software framework* that supports data-intensive distributed applications under a free license. It enables applications to work with thousands of nodes and petabytes of data. It provides a job manager and a location-aware[1] distributed file system, the *Hadoop Distributed File System* (HDFS).

- **Hadoop MapReduce**, a *programming model* and *software framework* for writing applications that rapidly process vast amounts of data in parallel on large clusters of computer nodes by spotting data locality of the HDFS. It runs on top of the Apache Hadoop software framework.

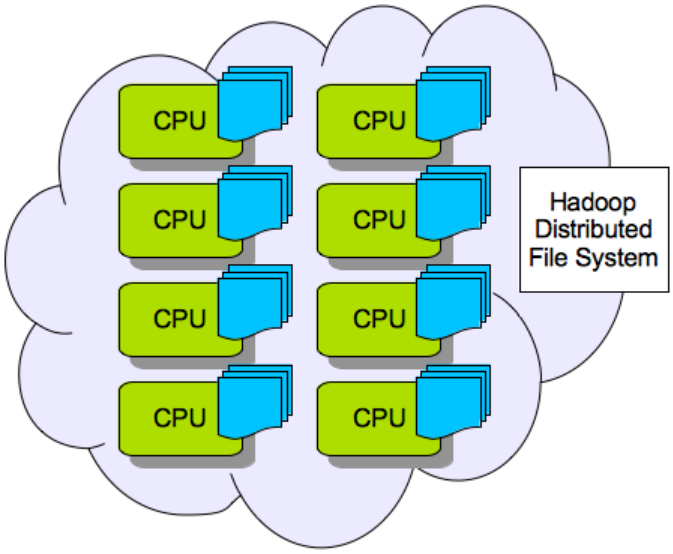

Figure 4.3: Hadoop/MapReduce computing mode with data locality.

A schematic view of the Hadoop/MapReduce architecture is shown in Figure 4.3, and a more detailed description of its components is provided below.

The HDFS is a distributed, scalable, and portable filesystem written in Java. It stores large files across multiple machines and achieves reliability by replicating the data across multiple hosts (the default replication value is three). Every file is split into chunks (HDFS blocks), usually of 64 or 128 MB. It provides location-awareness, which is used by both HDFS when replicating data to try to keep different copies on different racks or nodes, and by Hadoop MapReduce to schedule the applications on the nodes as

[1] In computer science, a location-aware file system is a file system which knows on which storage element a given file resides.

close as possible where the data is (data locality), therefore reducing backbone network traffic.

Above the file systems comes the job manager engine, which consists of one *Job Tracker*, to which client applications submit MapReduce jobs, and the *Task Trackers* running on the cluster nodes. With a location-aware file system (HDFS is the most common, but there are alternatives), the Job Tracker knows which nodes contains the data, and which other ones are nearby. The Job Tracker pushes work out to available Task Trackers trying to keep it as close to the data as possible. If the the node where the data resides is already occupied, priority is given to closer nodes[2]. If a Task Tracker fails or times out, that part of the job is rescheduled. The Task Tracker on each node spawns off a separate Java Virtual Machine (JVM) process to prevent the Task Tracker itself from failing if the running job crashes the JVM. The Task Tracker queries the Job Tracker every few minutes to check its status, and both the Job Tracker and TaskTracker status and information can be viewed from a Web browser.

The MapReduce framework is designed for compute highly distributable (or *embarrassing parallel*[3]) problems which have to process huge data sets, using a large number of computing nodes and processors. It is based on the MapReduce model, which consists in two fundamental steps (Figure 4.4): the *Map* and the *Reduce*.

Map step:
the master node takes the input, partitions it up into smaller sub-problems, and distributes them to worker nodes. The worker node processes the smaller problem, and passes the answer back to its master node.

Reduce step:
answers to all the sub-problems are collected by the master node and then combined in some way to form the output, which is the answer to the original problem.

This parallelism offers also some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails,

[2]Closer in terms of a network metric.

[3]In computer science, an *embarrassing parallel* problem is a problem which can be divided into a number of uncorrelated subproblems which can be solved independently.

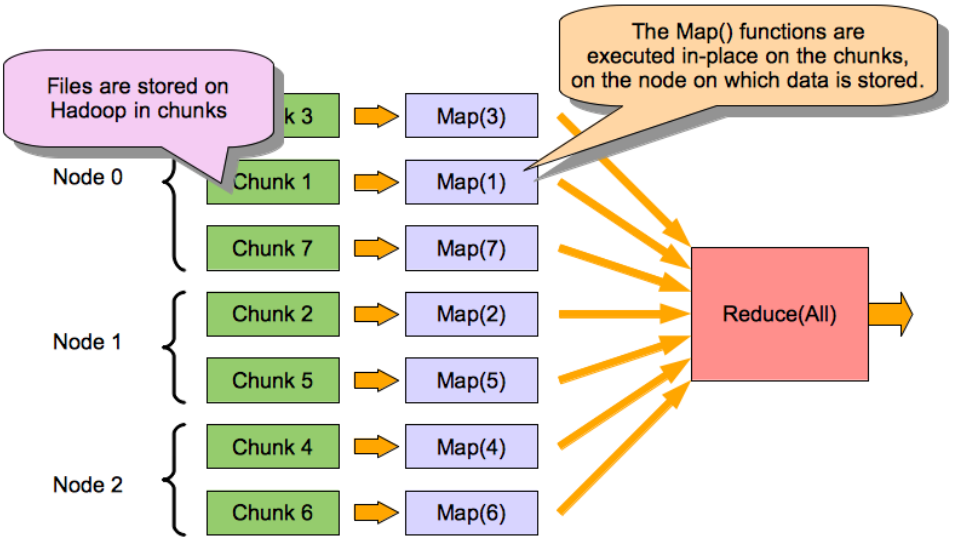

Figure 4.4: The Hadoop MapReduce model.

the work can be rescheduled, assuming the input data is still available.

As job scheduler, by default Hadoop uses FIFO, which provides five optional scheduling priorities to schedule jobs from a work queue. In version 0.19 the job scheduler was refactored out of the Job Tracker, and the ability to use alternate schedulers (such as the Fair scheduler or the Capacity scheduler) was added. The Fair scheduler was developed by Facebook, its goal is to provide fast response times for small jobs and QoS for production jobs. The Fair scheduler implements three basic concepts:

1. Jobs are grouped into Pools.
2. Each Pool is assigned a guaranteed minimum share.
3. Excess capacity is split between jobs.

By default jobs that are uncategorized go into a default pool. Pools have to specify the minimum number of map slots, reduce slots, and a limit on the number of running jobs. The Fair scheduler permits to fairly share an Hadoop cluster among a community of users, and it can be tuned to optimally allocate the computing resources allowing to maximize the number of Map tasks which can access the data locally.

Chapter 5

Monitoring of the CASTOR data storage system

As introduced in Section 3.3, CASTOR is a complex System. It therefore needs to be monitored: an online, real-time cockpit can spot errors or slow downs, while an offline repository for keeping history of what happened in the system can provide useful answers about what went wrong. In this Chapter I present the work done by me in this field at CERN IT Department.

In Section 5.1.1, I give an overview of the pre-existent CASTOR monitoring strategy, its related problems and its limitations. In Section 5.2, I describe how I replaced the log transport and aggregation layer to prepare the ground for the implementation of both an online and offline analysis framework, which I named the *Metrics Analysis Engine*. This framework, now in production at CERN, is presented together with a proof of concept Web interface in Section 5.3. In Section 5.4, I present how Hadoop has been used for Storing and analyzing CASTOR historical log data, solving the pre-existent problems.

5.1 The pre-existent implementation

5.1.1 Overview

The CASTOR monitoring system is based on a three layers model: these are the *producers*, the storage and analysis, and the consumers layers. The producers are basically CASTOR daemons running and producing log messages, which are aggregated by the transport substrate and transferred to

the storage and analysis layer. Here messages are stored and analyzed, generating a set of interesting measures on one or more parameters: these are so called *metrics*. In performance analysis a metric defines a piece of data, how to compute, how to save and how to display it; and this is the way in which the concept of metric will be used from now on. The *consumers* makes then the computed data accessible by the user, i.e. by displaying it on a plot.

In the pre-existent implementation of the monitoring chain, the transport substrate was implemented by a software named *Rsyslog* [22], which was taking care of aggregating and transferring log messages to the storage and analysis layer. Here, messages were parsed on the fly by a component named *Logprocessor daemon*, and every parameter was inserted into the *Distributed Logging Facility* (DLF) [15] database (DB) with its value. On the DFL DB *Procedural Language/Structured Query Language* (PL/SQL) [23] procedures were run to compute the metrics and the results were inserted again in the same database. The final part of the chain involved the *LHC Era MONitoring* (LEMON) [24] as the consumer, which was in charge of gathering the computed metrics with its *sensors* and of inserting their values in its internal database, to be afterwards displayed by a web interface. These components are listed in detail below.

- **Rsyslog** is an open source software utility used on UNIX and Unix-like computer systems for forwarding log messages in an IP network. It implements the standard basic syslog protocol for logging system and applications messages, extending it with important features such as using TCP/IP for transport.

- The **Logprocessor deamon** (or *logprocessord*) is a real time log stream analyzer framework structured in a plugin-oriented fashion. The input and output plugins are written in Python and set up in the Logprocessor damon configuration.

- The **DLF** is a framework designed to centrally log messages and accounting information from CASTOR related services. It consists in three major components: an API to allow clients to write messages, an Oracle database where data is stored and analyzed (the DLF DB) and a Web Interface for graphical interruption and visualisation of the stored data.

- **LEMON** is a client/server based monitoring system. On every monitored node, a monitoring agent launches and communicates using a

push/pull protocol with sensors which are responsible for retrieving monitoring information. The extracted samples are stored on a local cache and forwarded to a central Measurement Repository using UDP or TCP transport protocol with or without authentication/encryption of data samples. Sensors can collect information on behalf of remote entities like switches or power supplies. The Measurement Repository can interface to a relational database or a flat-file backend for storing the received samples. Web based interface is provided for visualizing the data.

- **PL/SQL** is the Oracle Corporation's procedural extension language for SQL and Oracle relational database. That is, an application-development language which is a superset of SQL, supplementing it with standard programming language features.

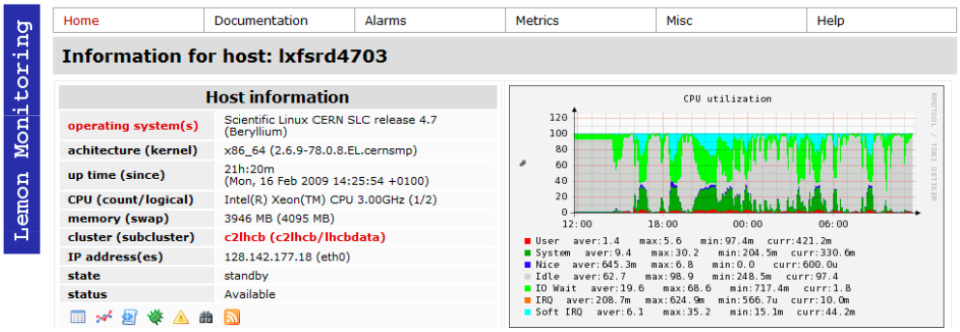

Figure 5.1: The LEMON web interface.

5.1.2 Problems and limitations

Rsyslog has its main feature and problem in being integrated into the operating system: it works out of the box in nearly all the use cases, but if something goes wrong with the message logging, then the entire logging process gets affected. We have encountered this problem at CERN: a particular mix of Rsyslog misconfigurations and network problems resulted in applications freezes, since they were not able to log messages anymore. Another issue encountered concerns the monitoring system, schematized in Figure 5.2, which was reaching its limits in terms of database performance. The problematic part was the analysis of the data within the DLF DB using PL/SQL, an inefficient implementation for computing online metrics which was overloading the database. Besides, even if the source and computed

data were handled with just few delays, the analysis was performed asynchronously only every five minutes due to its heaviness, and therefore the system was not capable of going beyond this latency threshold.

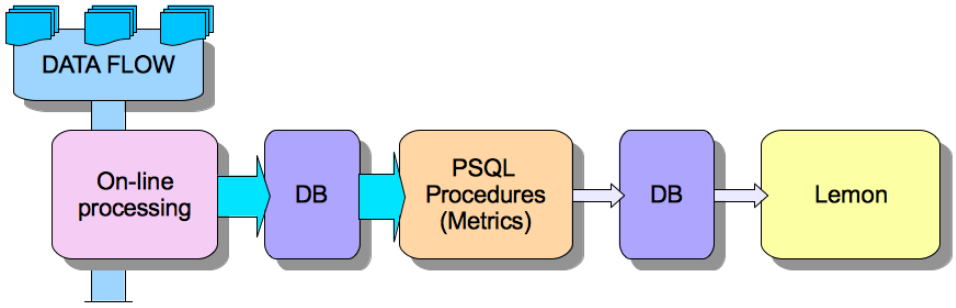

Figure 5.2: The pre-existent online monitoring chain.

Concerning the approach for analyzing and retrieving informations from CASTOR history, it was based on two methods:

1. query the database;
2. run a `grep/cut/sort` chain on the log files on every CASTOR node via `wassh`[1] and then collect the results.

Querying the database on indexed fields works fine, but when one tries to query for non indexed ones, the system just cannot perform a full text search on such a huge amount of data which is the CASTOR history. On the contrary, by running a `grep/cut/sort` chain via `wassh` on CASTOR nodes, log files can be analyzed for custom fields and patterns, but the query is not deterministic (some nodes might be offline, dead or just in maintenance) and one is limited in going back in time by the log rotation policies (due to the limited disk space, log files are usually kept for 100 or 200 days). The pros and cons of the two methods are summarized below.

The main feature of the database approach is to have a centrally managed repository of data, which can be queried in a deterministic way. But it can be queried only on pre-defined, indexed fields: a complete analysis is not possible.

The main feature of running a `grep/cut/sort` chain via `wassh` on every CASTOR node is to distribute the computation among all the nodes,

[1] wassh is an utility to run commands via ssh in parallel

which are going to analyze their own log fils, from the local file system, therefore taking advantage of data locality. But the result is not deterministic and the history is limited by the log rotation policy.

5.2 Using Scribe for the transport layer

As already introduced, the main source of problems in using Rsyslog is its integration into the system. We thus decided to look for non-intrusive solutions which could allow to decouple system monitoring from the service specific log transport and aggregation. Moreover, the alternative solution had to be able to write on the HDFS, since as will be discussed later in Section 5.4 Hadoop had been designed as the new system for storing CASTOR log messages history.

I identified *Scribe* [25] as this alternative. Scribe is a tool for aggregating streaming log data developed by Facebook recently distributed as open source. In production at Facebook from years, it is:

- decoupled from the system;
- flexible, chainable;
- fault tolerant, scalable.

Scribe works with two concepts: *messages* and *categories*. In the CASTOR context, messages are log lines, and categories are the various components of the system. Scribe takes as input messages tagged with a category, and processes them trough an abstract object, the *store*. A store can be of several types, only the ones interesting for the CASTOR use case are here listed:

1. the *Network* store (forwards incoming messages to another Scribe instance)
2. the *Standard FS* store (writes to the local File System)
3. the *HDFS* store (writes to Hadoop's Distributed File System)
4. the *Buffer* store (writes to a primary store, if this is not available switches on a secondary store)

In particular, Scribe's feature of being chainable through the network store makes it very flexible and scalable. Another important feature is that since

Scribe knows where a message starts and ends, the rotation of the files written to the standard or the HDFS file system store (which can be handled on size or time basis) won't truncate any message.

Concerning its integration with the other applications, Scribe provides both C++ and Python APIs. Integrating it with a Python code is as simple as typing "import scribe". Just to give an idea, a Python call to send a message to Scribe looks like:

```
log_entry = scribe.LogEntry(category="category", message="message")
```

To use Scribe in the CASTOR environment, it had to be installed on every CASTOR node, which are more or less 1500, and a main Scribe aggregator had to be set up to receive and aggregate the log files from the nodes. I have developed a *Scribe Injector* support script (in Python) to be installed together with a local Scribe server on every node. The Scribe Injector tails the configured list of log files and sends every new log line to the local Scribe server. The local Scribe server forwards them to the main Scribe aggregator, which stores the data locally for online analyses and on HDFS for offline analyses. For handling the possible network failures between CASTOR nodes and the main Scribe aggregator, I configured the local Scribe servers to buffer locally if they cannot connect to the main Scribe aggregator, using the Buffer store. Using this store, if Scribe cannot use the primary store it switches to the secondary, buffering messages and keeping on trying to connect to the primary one. Then, when the primary becomes available again, Scribe synchronizes by forwarding to the latter the buffer from the secondary store, and continues with the normal operational mode. The complete layout is schematized in Figure 5.3.

To test this layout before deploying it on the entire system, I used the CERN batch cluster to run 1750 jobs for simulating the logging activity of CASTOR. Every job was generating log messages within a predefined set. The timing between messages generation was driven by a Probability Density Function to simulate a real load. The test was configured as a stress test: the total mean network traffic was about 40 times the CASTOR's expected one and the network connection to the main Scribe aggregator was interrupted for 2 minutes every 5 (a longer network failure period of about 20 minutes was tested as well). The following plots show the network and the memory utilization on the main Scribe aggregator. In the network plot (Figure 5.4) the simulated network outages (valleys), the buffer replaying (peaks) and the restoring of the normal operational mode (constant lines) can be clearly distinguished. In the memory usage plot (Figure 5.5) it can be seen that at

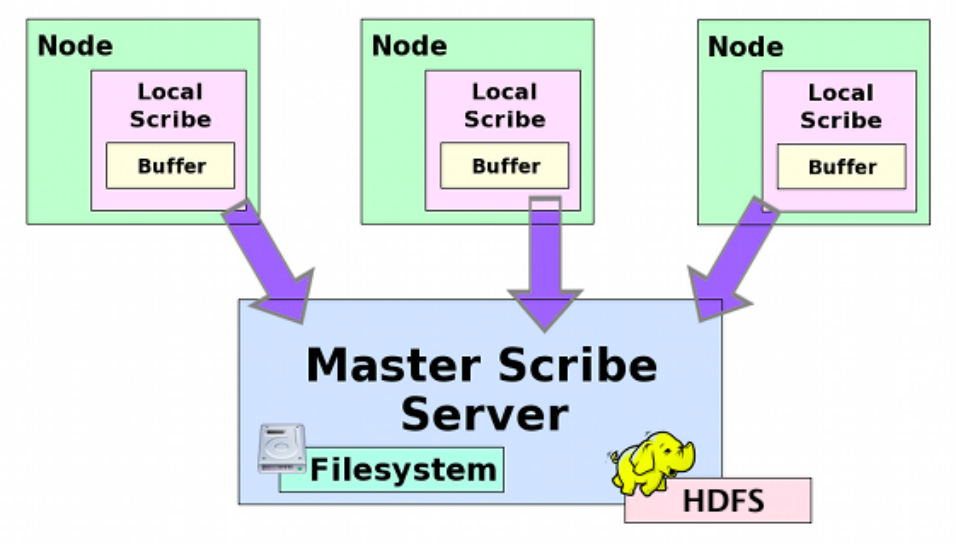

Figure 5.3: The Scribe layout.

the beginning the memory use increases in correspondence of buffers replying and that it becomes constant after a while. This is because Scribe tends to leave in memory data structures (created to handle the buffer replying) for further usage, until the configured limit (see Appendix, Section A.1.4).

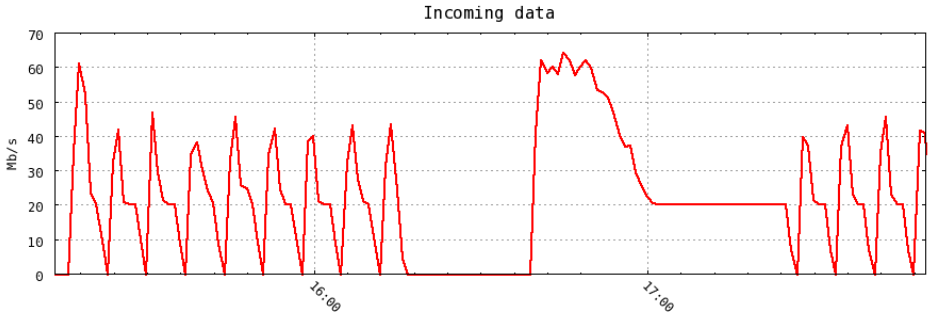

Figure 5.4: Scribe stress test: network usage (incoming) on the main Scribe aggregator, in MB/s.

Once all the tests worked as expected, Scribe was deployed on every CASTOR node. An important parameter to set and evaluate was the outgoing bandwidth when recovering the buffer from the local Scribe servers (replying it to the main Scribe aggregator), which has been limited to 50 Kbpbs per node. This limit is quite low and causes a particularly slow buffer

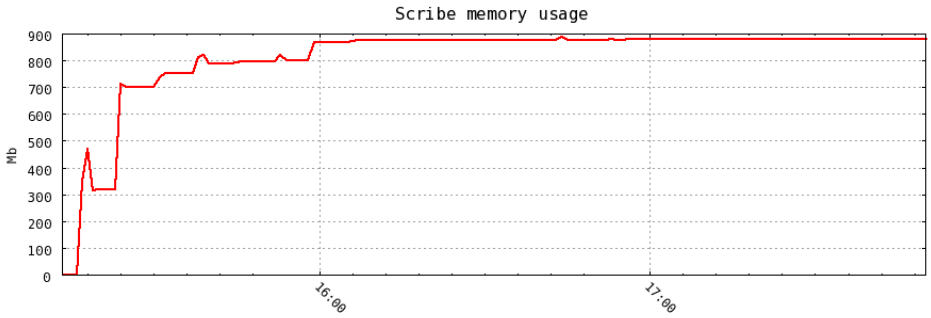

Figure 5.5: Scribe stress test: memory usage on the main Scribe aggregator, in MB.

replying on some nodes which have a huge amount of log files. On the other hand, it is a safe setting to prevent overloading the CASTOR network by Scribe after a failure, which would interfere with LHC data taking. The latter is the first and imperative aspect to keep in mind when working on the CASTOR system.

5.3 Online monitoring

One of the goals of the work described in this thesis was to evolve the pre-existent monitoring system and compute the metrics on the fly, without relying on a database. The computed data should be available to every consumer and for multiple types of use, as long term archive, plotting, etc. The overview of the new layout is summarized in Figure 5.6, its main component being the Metrics Analysis Engine which computes the metrics on the fly.

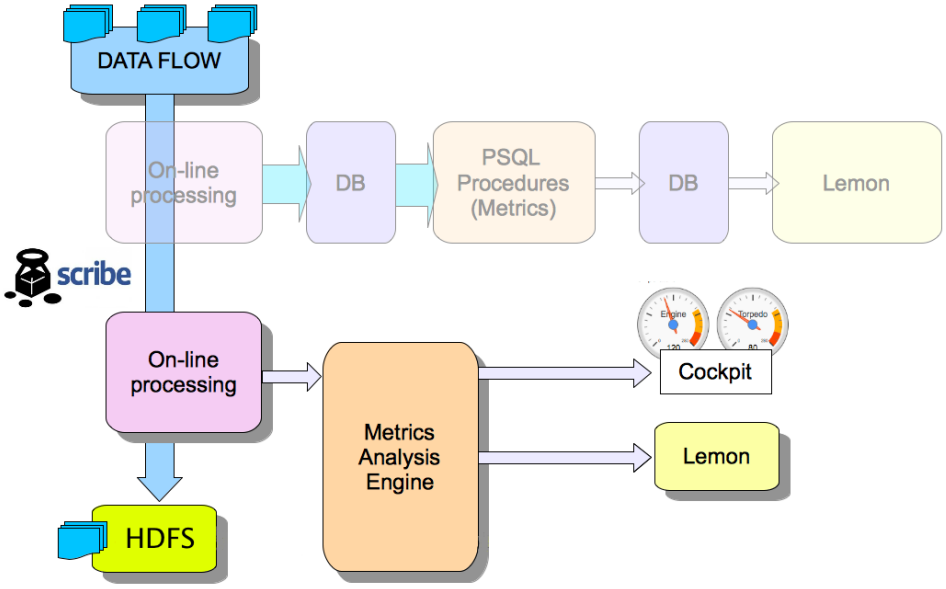

Figure 5.6: The pre-existent (grayed out) and the new online monitoring chain.

5.3.1 Introducing the Metrics Analysis Engine

This component computes the metrics by analyzing log lines previously split in key-value pairs: from now on I will in fact talk about keys and values instead of log lines. The Logprocessor daemon scope is to make it possible, by taking log lines as input, splitting them into key-value pairs according to the *source* plugin, and feeding the *destination* plugin. The latter then feeds the Metrics Analysis Engine which works on the back and can operate on whatever kind of key/value pairs[2] passed by the source plugin. Usually splitting of a log line in key value pairs is trivial, like in the following example: the keys will be simply HOSTNAME, RAM, and MODEL, with their respective values.

```
LINE 1: HOSTNAME=lxbsq1204     RAM=16G     MODEL=ec_3
LINE 2: HOSTNAME=lxbsq1402     RAM=24G     MODEL=ec_3
LINE 3: HOSTNAME=lxbsq1302     RAM=24G     MODEL=ec_3
LINE 4: HOSTNAME=lxbeb2963     RAM=12G     MODEL=ya_8
```

In the Metrics Analysis Engine a metric is defined using a custom, very basic, xml-oriented meta language. Using this language criteria which spec-

[2]This is not completely true, a mandatory keyword TIMESTAMP is needed and has to be in the format 2012-01-27T16:06:11+01:00

ifies when the metric is matched and what to do with which key's value are defined. An example is the following one:

```
<metric>
    name: ListRamSizes
    conditions: "lxb" in HOSTNAME
    groupbykeys: MODEL
    data: ListUnique(RAM)
    window: 86400
    nbins: 24
    handle_unordered: skip
</metric>
```

The metric reported above will match all messages in which the filed HOSTNAME contains lxb, it will list all the unique values found for the RAM key and will group the results by the MODEL key's values. The metric is defined on a one day window, with a 24 bins resolution: it means that it will provide a one-day moving average window updated every hour. The "conditions" field is evaluated by Python, so it is very flexible and powerful. The results of this metric applied to the above log lines would be something like:

```
- ec_3:
      16G
      24G

- ya_8:
      12G
```

The Metrics Analysis Engine works incapsulated in a destination plugin of the Logprocessor daemon. This plugin uses two threads: `Analyzer` and `Timer`. The analyzer is the thread which incapsulates it, while the timer thread checks for new data from it every five seconds. Since the metrics are already computed on the fly by the Metrics Analysis Engine only the aggregated data has to be read out, which is a light operation that can be performed often. This approach let to bring the latency threshold down to five seconds. The Logprocessor daemon plugin loads the metrics to compute by reading their definitions from files, one per metric. These files (and consequently the metrics being computed) can be added, removed and modified without restarting the daemon, as shown in the output of the Metrics Analysis Engine reported in Listing 5.1.

```
Running process ComputeMetricsFromScribe ...
2012-02-02 20:13:06.441397: Initializer
2012-02-02 20:13:06.441561: Analyzer started.
2012-02-02 20:13:06.441561: Starting with metrics:
                                Checksumd
                                TotMessagesPerDaemon
                                TotMessages
                                FilesRecalledStats
                                TotFilesRecalledStats
                                ProcessingTime
                                Throughput1sec
                                ORAerrors
                                ClientVersionsStats
2012-02-02 20:13:06.443620: Timer started (5 sec)
2012-02-02 20:17:56.581593: Adding metric MinFileSizeTape
2012-02-02 20:18:01.582814: Adding metric MaxFileSize
2012-02-02 20:21:06.758198: Reloading metric TotFilesRecalledStats
2012-02-03 14:27:03.420481: Removing metric ORAerrors
```

Listing 5.1: Output of the Metrics Analysis Engine when adding, modifying or removing a metric.

Computed data is then stored by the destination plugin to the data folder in plain text files using the Pickle Python module for further processing by the consumers. In the future, the project should provide an interface queryable by the consumers, leaving to the Logprocessor daemon the only task to compute the metrics trough the Metrics Analysis Engine, no matters how the computed data is then stored.

5.3.2 Implementation details

The syntax to define in more detail a metric is explained in the following step by step:

`name`:

A name for the metric. The filename of the metric should be the same.

`conditions`:

The conditions on the message, for example `LVL=="Error"` means that the key LVL (level) has to have the value "Error". This is a Python

expression evaluated (in a safe way[3]) in the message's key-value pairs domain.

`groupbykeys`:

The keys of the message to group on, comma separated. I.e.: DAEMON, MSG.

`data`:

A data object and the key on which it has to be applied. An example is: `Avg(ProcessingTime)`. Possible data objects are listed in the following. The argument is the key of the message you want to pass to the object when it's called on a matched message. Comma separated.

`window`:

The time window, in seconds.

`nbins`:

How many bins should the time window contain.

`handle_unordered`:

Policy to use when unordered messages are encountered. This can happen because of network outages or the normal Scribe flushing delays. Possible values are:

- "time_threshold" accepts unordered messages not older than a given amount of time, that has to be set according to the transport layer chunking and timing schema to accept unordered messages but to reject old ones (caused mainly by network outages).
- "percent_threshold" will reject messages older than the 10% of the duration of the current bin.
- "skip" will reject everything.
- "rewrite_history" will try to put the messages in the right bin (even in an old one).

[3]Python `eval()` function allows to specify on which set of functions and on which variables the code can be executed: the only available functions for this Python expression are the logical operations and the only accessible variable the key-value pairs of the current message being processed.

Special Keywords:

- NONE is a keyword on the conditions to match all the messages (to be used as: "`conditions: NONE`").
- NONE is a keyword to be used in the `groupbykeys` field: it will group everything under the "Grouped by NONE" value, which will be shown on the output and on the plots. This is because of the structure of the Metrics Analysis Engine, which requires at least one grouping key.
- DELAY is a keyword dedicated for estimating messages delays with the EstimateDelay() data object.
- DATE is a keyword extracted by the TIMESTAMP keyword to permit easy grouping by the date (YEAR-MONTH-DAY)
- KEYVALUES is a keyword to be used in the dataobjects argument for passing to the object the entire dictionary of the message's key-values pairs.
- The empty keyword corresponds to an empty value, to be used with data objects which does not require an argument, like the `Counter()` object, for example.

For backward compatibility, the Scribe source plugin of the logprocessor daemon adds a keyword `type`=log. If the message is not recognized as valid by this plugin, a message containing only the keywords `type=log` and `MSG_NOT_VALID=1` is returned, which will be skipped by the MetricsAnalysisEngine destination plugin.

Data objects:

Avg(number):
Compute the average of values of the given key. Returns a list: [mean, n, sum, min, max, sttdev].

EstimateDelay(DELAY):
Special object automatically handled by the framework, the DELAY keywork is mandatory and it will be replaced by the delay of the message. it will then calculate the Average and return: [n, mean, min, max, sq.sum].

DummyStore(whatever):

Will store every value of the keyword passed as argument (even the entire Python dictionary containing the message if the keyword is KEYVALUES) and save them in a list. Returns the list.

ListOne(whatever):

As the DummyStore, but will save only the last item found.

ListUnique(whatever):

As the DummyStore, but will save only the unique values of the keyword passed as argument.

ListAndMerge(KEYVALUES):

Will store all the unique keywords found in the dictionary passed as argument, and will save as example value the last value found. Returns the Python dictionary of the unique keys found and their example values.

Counter():

Counts how many times is invoked. The argument is discarded, for nice output and plots the special keyword COUNT can be used.

MaxMsgsPerSecOverMinute(TIMESTAMP):

Has to be used with a one minute window with only one bin. Will extrapolate the seconds value from the message TIMESTAMP and increment the corresponding counter in a 60 elements array. Will then return the value of the maximum value stored in the counters of this array. Useful for calculating throughputs.

EstimateThroughputOverMinute(TIMESTAMP):

The same as MaxMsgsPerSecOverMinute but calculates the difference between the mean over the minute and the maximum value when returning data, to make much easier to spot an high throughput per second.

Adder(number):

Sums all the values of the key passed as argument.

Max(number):

Keeps the maximum value of the key passed as argument.

Min(number):

Keeps the minimum value of the key passed as argument.

The objects and the keywords above introduced can be used in every context which requires a monitoring tool, since they are generic: monitoring a batch system, for example, would not require modifications of the tool. This is one of the main features of the Metrics Analysis Engine: it can be used in every context. Just to give an idea about the way it works, the next example shows how to use it in a custom Python code:

```
# Include the framework
include MetricsAnalysisEngine

# Initialize
metrics=MetricsAnalysisEngine.loadMetrics(Path+\*.metric")

# Process
for msg in stream:
    for metric in metrics:
        metric.apply(msg)

# Get data from the metric
for metric in metrics:
    metric.getData("print")
```

5.3.3 The Cockpit web interface

The system to display the data computed on the fly by the Metrics Analysis Engine is named the *Cockpit*. Referring to the three layers model described at the beginning of this Chapter, it is a consumer. The Cockpit consists in a web interface written in Python (using Python Server Pages), which provides basic functions to plot and select the data. A demo of this first proof of concept web interface is displayed in Figure 5.7.

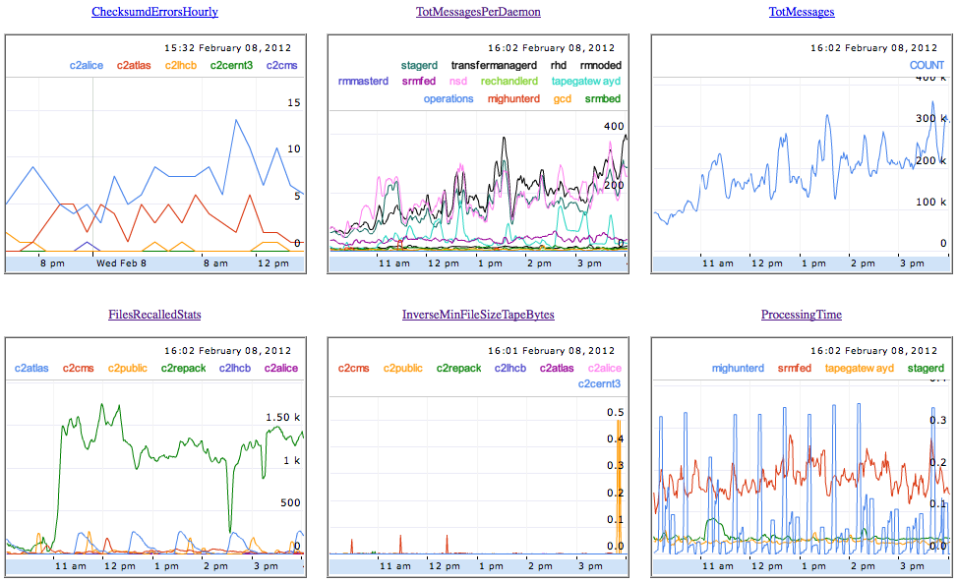

Figure 5.7: Demo of the Cockpit web interface.

5.4 Offline monitoring with Hadoop/MapReduce

Hadoop, thanks to its data locality and to its MapReduce textual analysis oriented features, allows to take advantages from both the pre-existent approaches presented in Section 5.1.2 for CASTOR offline monitoring, and avoid their limits. Hadoop/MapReduce can provide a centrally managed, *safe* storage for CASTOR history with a virtually infinite amount of space, and a way to analyze these data allowing to perform full analyses in a reasonable amount of time. The installation of the Hadoop cluster at CERN IT Department is documented in Appendix A.4. One of the goals of the work described in this thesis was to set it up and to interface it with the other components as described in the following.

As already introduced in Section 5.2, Log data from CASTOR is collected by Scribe, and stored on the HDFS. The following naming convention for storing the data had to be adopted due to organizational and partitioning reasons:

```
$INSTANCE / $NODETYPE / $DATE / logfiles
```

where `$INSTANCE` is the CASTOR instance (i.e. c2atlas, c2cms, etc.), `$NODETYPE` is the type of the node (i.e. diskserver, heaadnode, etc.) and `$DATE`

is the date. Scribe does not permit to natively specify such a structure for destination paths, so the source code had to be modified (see Appendix, Section A.1).

Once CASTOR log data is stored on HDFS, to analyze it within a MapReduce task both standard tools like `grep`, `sort` and `wc` as well as the Metrics Analysis Engine framework can be used, es explained below.

Using standard tools:

By using Hadoop Streaming [26], every line of the file to analyze is passed as standard input to the Map task. Then the Map standard output will be the Reduce standard input, and the Reduce standard output will be saved to the specified output file(s). In this way it is possible to specify as mapper a command like `grep` and as reducer a command like `sort`.

Using the Metrics Analysis Engine framework:

The metrics designed for the Metrics Analysis Engine can be computed on Hadoop via MapReduce without too much hassle (Figure 5.8). I have developed the Metrics Analysis Engine with this feature in mind and I created a dedicated package (`run-metrics-on-hadoop`) to provide all the wrappers to make it possible. For using this method, a metric has to be incapsulated in a more verbose, extended "hadoopmetric" (see Listing 5.2) which specifies also the data on which to evaluate the metric. In this context, every bin of the metric corresponds to the results of a single Map task, and the output of the Reduce task is the aggregation of all the bins in the final one (the value of `nbins` is automatically overwritten and replaced with the number of the Map tasks). Once downloaded from the CASTOR software repository, the package is ready to be used on the Hadoop cluster at CERN IT Department by an authorized user.

A custom Java MapReduce task can of course be written in case of more complex analyses.

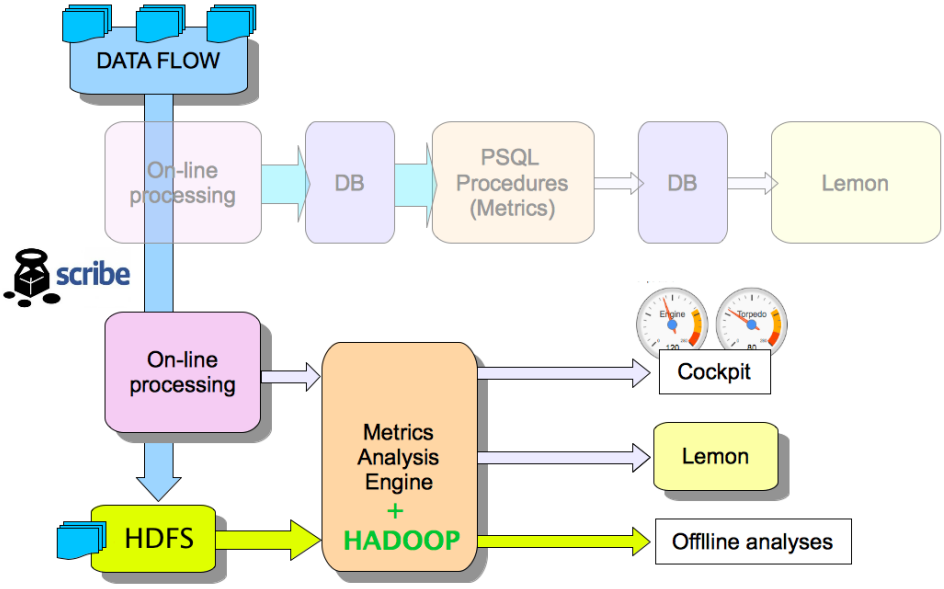

Figure 5.8: The pre-existent (grayed out) and the new online monitoring chain, including the offline metrics evaluation using Haddop/MapReduce

```
<hadoopmetric>

  inputformat: lzo
  input: /data/scribelogs/castor/c2alice/diskserver/20120201
  input: /data/scribelogs/castor/c2atlas/diskserver/20120201
  input: /data/scribelogs/castor/c2cms/diskserver/20120201
  input: /data/scribelogs/castor/c2lhcb/diskserver/20120201
  input: /data/scribelogs/castor/c2public/diskserver/20120201

 <metric>
    name: Checksumd
    window: 86400
    conditions: FACILITY=='checksumd' and MSG[0:3]=='Com'
    groupbykeys: CLUSTER, HOSTNAME
    data: Counter(), ListUnique(LOCALFILENAME)
    handle_unordered: time_threshold
    nbins: 1
 </metric>

</hadoopmetric>
```

Listing 5.2: An example of a .hadoopmetric file. In respect of standard metric, here the "inputformat" (which specifies if the data source is compressed or not) and the "input" (which specifies the data on which to evaluate the metric) parameters are added.

Chapter 6

Bringing Hadoop/Mapreduce to HEP analysis

This Chapter is the core of my thesis work. I investigate how a typical ROOT High Energy Physics (HEP) analysis can be computed on Hadoop/ MapReduce in a completely transparent way to ROOT, the data and the user. The method proposed to achieve the goal relies on a "conceptual middleware" that allows to run ROOT without any modification, to store the data in its original format, and to let the user deal with Hadoop Map/Reduce in a classic, batch-fashioned behaviour. The workflow and the solutions developed to achieve this result can be easily ported to any HEP code, and in general to any code working on binary data relying on independent sub-problems.

In the first three Sections, I explain the motivation of porting Hadoop/ MapReduce in the field of the HEP analyses, how they fit in this model, and how to achieve the result, emphasizing the problems encountered and their solutions. In Section 6.4, I discuss a possible way to evaluate the performance of this approach in comparison to the traditional computing model, also giving some arguments in support of the importance of data locality.

6.1 Motivation

HEP analyses are I/O bound applications. They have to process huge amounts of data (expecially at the LHC experiments, see Chapter 3) and they have typically to be performed several times in order to finalize the re-

sults, which means that the same data has to be accessed again and again. The last stage Tiers, as Explained in Section 3.1, regardless from talking about a Grid Tier-2 or a farm Tier-3, adopt the standard computing approach which implies the presence of a communication channel between the storage and computing elements. This channel is a bottleneck that can be easily saturated by these I/O bound applications, especially when scaling up.

As already introduced in Chapter 4, Hadoop Map/Reduce's bigger goal is to avoid the distinction between storage and computing resources, overlapping them and bringing data locality. The latter is already implemented in macro-zones by the WLCG (for example, a job sent in a USA Grid site transfers data within its own site); however Hadoop really allows to bring the computation close to where the data resides, up to the level of a processor on the same logical board where the local storage is connected to. In this way, the interface between the storage and the computing elements becomes the internal bus.

6.2 Porting HEP analyses to a MapReduce model

As already introduced in Section 4.2, when running a MapReduce job the Map tasks can be performed in parallel provided that each mapping operation is completely independent of the others. That is, computing problems involving correlations, interacting boundary conditions and so on just can't be handled by the Hadoop/MapReduce parallelism. To exploit this kind of data-driven parallelism, a problem has to be an embarrassing parallel problem, in which every sub-problem can be computed in a completely independent way from the others. In HEP analyses millions of particles collision events are analyzed , and the main hypothesis behind is that *events are independent*. A HEP analysis can therefore be split until the lower limit of one single event, and so it perfectly fits in the MapReduce model.

The simplest example of HEP analysis is the *cut-and-count* model: for every event a set of selection cuts is applied, and the events which pass all the cuts (the *accepted* events) are simply counted. These cuts involve several variables related to the objects contained in the event, allowing to decide if an event matches some criteria or not. Transposing this problem on a MapReduce task is straightforward: the Map function will consist in the analysis of a set of events, and the Reduce function in the aggregation of the partial results. The Map output for a set of events will be then the counter of

events which have matched a given criteria, and the number of events being evaluated. The Reduce function will consist in just summing these numbers to obtain the total counter of accepted events and and the total number of events being evaluated. A variation of this model is that one might be interested in already structured data. In this case the output from the Map functions would be a set of histograms, and the Reduce function should be able to merge all these partial histograms into a final one. The difference between handling set of numbers and structured data is that the latter requires the Reduce function to perform an operation more complicated than just a sum, and that it has to know how to handle the data. In this case a slightly more complex approach is required, which consists in instructing the Reduce task about how to handle the structured data (or in let it rely on an external program to achieve this goal). This is anyway still a cut-and-count model, and the same concepts apply. According to these examples, the Reduce function computational and I/O weight is near zero. Considering this hypothesis on the Reduce function is more than plausible, and reflects the spirit of the MapReduce model: as a matter of fact, Hadoop's Reduce tasks do not take advantage of the data locality which, as already explained, is an essential feature for I/O intensive tasks. There are other applications in which the Reduce function weight is not zero, and where a Hadoop/MapReduce approach could bring only partial benefits. For example the generation of the D3PD n-tuples generates as output a huge amount of data, and cannot be considered as I/O free.

The important fact here is that many of the HEP analyses performed by final users follow the cut-and-count schema: by taking it as a case study, the discussion will automatically cover a vast field of applications.

6.3 Running HEP analyses on Hadoop/MapReduce

The software for analyzing HEP data are nowadays quite complex. They use frameworks mainly centered on ROOT which are developed, maintained and used by thousands of persons. These frameworks cannot be easily modified (i.e. for adding external libraries), because of an high risk of incompatibilities. In this Chapter the ROOT framework, standard de-facto for HEP analyses, is taken as reference. The workflow to let it run on Hadoop in a MapReduce fashion are basically the same for any complex code which uses binary data formats. ROOT data is in fact binary, and dealing with binary data in Hadoop/MapReduce is itself a problem, since:

1. binary data cannot be sliced in chunks on a size-basis, because the chunks would result in corrupted data[1];

2. a standard (i.e. new-line based) record delimiter not aware of what an event is and how to read it from the binary data clearly does not work.

To solve these two problems, a solution would be to teach Hadoop how to deal with ROOT binary data, and to define a custom record able to deal with events. This approach would lead to integrate ROOT with Hadoop, which would require a complex work and long-term support. Another way would be to convert the binary files into Sequence files. A Sequence file in Hadoop is a merge of a set of files in just one big file, in which every single file of the set corresponds to a record. A Sequence file permits therefore to obtain from the Map task a binary file in its entirety as a record. This approach would require an intermediate step of conversion which would be better to avoid, and provided that HEP data files are usually comparable if not much bigger than a chunk, it would lead to lose the benefits brought by data locality[2]. The only way to preserve data locality with Sequence files would be to re-encode events, which are much smaller, as single binary files and then merge them in a Sequence file. This leads to intermediate step of conversion definitely too heavy.

None of these two methods are therefore acceptable. Moreover and in general, even assuming to find a solution for this two problems, bounding the data format with Hadoop would mean to be tighted in using Hadoop's data access methods. This constrain is too restrictive, since *ROOT binary data needs to be accessed not by the Map tasks, but from ROOT.*

Hadoop/MapReduce's native programming language is Java. Through Java, a data file can be easily accessed from a Map task in binary mode. ROOT could be then integrated with Java to use this approach, but as

[1] Actually, even cutting a textual file would result in "corrupted" data, as the size based splits can truncate a record (line, or set of lines) at any point. But in this case Map tasks can anyway *read* the chunk, and ask to Hadoop to give to them the (few) missing bytes from the previous (or next) chunk to reconstruct the corrupted record - that is how Hadoop/MapReduce works. In case of a binary file, Map tasks just cannot read only a chunk of the original file, and therefore Hadoop's procedure to deal with truncated records fails at the first step.

[2] As explained in the previous note, if the record size is comparable with the one of a chunk, the data needed to be transferred to reconstruct a record would be also comparable to the size of a chunk, loosing the benefits from data locality.

already mentioned this would require some effort since the complexity of the HEP frameworks and would risk to bring incompatibilities. Running a third party code on Hadoop/MapReduce without any modification is possible, and a number of libraries exists (Streaming, PIPES [27]). Anyway, since Hadoop was developed with textual analyses in mind, these libraries does not perform well with binary data, and their usage in this context is a bit triky (i.e.: the Streaming library passes data to a custom code via the standard input). Moreover, some changes in the ROOT code would still be required.

The solution proposed in this thesis deviates from the natural way to port an HEP analysis to Hadoop/MapReduce, but solves all these problems in one go. The idea is to store the HEP ROOT data in its original format on the HDFS, and to configure the system to have a single Map task per not chunk, but *per file*. Map tasks perform then no actions but start a ROOT instance, which takes over the analysis on the file the Map task was originally in charge of processing. In this context, analyzing just one file would mean having no parallelization. But specifying a directory as input for the MapReduce job would lead in having a Map task for every file in the directory, running in parallel. The parallizable unit has then been raised from the HDFS chunk to an entire file, and the parallization moved from a single file to a set of files, as shown in Figure 6.1. Clearly, this means that the data set to be analyzed has to fit in this schema. HEP data sets usually consist in sets of several files grouped by some criteria, so that they do perfectly fit in the schema. As already introduced in Section 2.3.4, the ATLAS experiment computing model, given the huge amount of data produced by the detector, relies on a lightened format for final users specific analyses, the D3PD. This format, which consists in flat ROOT n-tuples, is in practice the most common format used for physics analyses, since it is generated by skimming, thinning and slimming the original data sets to keep only events and informations interesting for a particular analysis and so reducing noticeably their size. D3PD data sets are stored hierarchically, organized by LHC run ($\sim 10^5 - 10^6$ events), by luminosity blocks ($\sim 10^4$ events), and only then by ROOT files, each containing a set of $\sim 10^2 - 10^4$ events [28].

To make Hadoop/Mapreduce work in the desired mode, first the HDFS chunk size has to be set equal or greater than the file size, for every file, so that they are not sliced in chunks. A custom record definition has then to

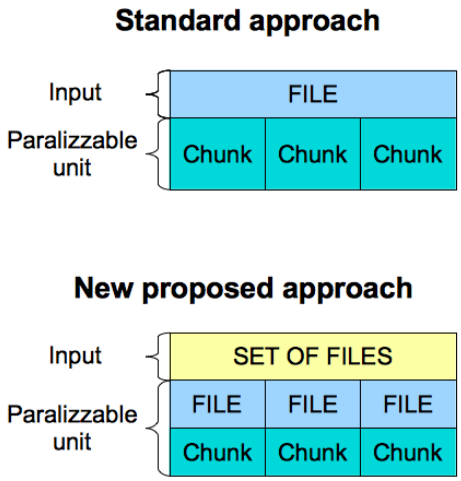

Figure 6.1: The standard and new proposed approach.

be defined to process a chunk (which now corresponds to an entire file) in one go, solving both problems 1. and 2. Summarizing, the schema is:

$$one\ Map\ task = one\ chunk = one\ file = one\ record$$

To let the analysis be performed by ROOT, a Java wrapper Map task is used to run a ROOT instance on the file to analyze. To let ROOT access this file (chunk) transparently, the latter has to be accessed from a standard file system: in this way *every* ROOT code will work almost out of the box. To access a HDFS file in this way few methods exists, which are summarized below.

- Hadoop provides command line tools for common files operations. Using these tools, a file can be copied from HDFS to Map's local sandbox. If the file has to be fetched over the network this method performs a network-to-disk copy and it works fine. But in case of data locality, instead of just using the local replica, the file has to be copied to the sandbox before being accessed. This unnecessary disk-to-disk copy wastes time and I/O resources.
- The ROOT team has developed plugin capable to read files from HDFS, which should let deal with these files in ROOT like standard files, just by using the "hdfs://" protocol in the code when loading them. Anyway, this would require a custom ROOT build.

- A FUSE[3] module is available, to allow accessing HDFS in user space. While a FUSE approach is definitely attractive, its performance is not satisfactory [29]. FUSE's main problems are its design of many layers in series which slows down the file access and introduces latency, and a heavy overhead in particular over network accesses.

All these methods work and preserve data locality[4], but their performance and compatibility are questionable. When looking for a way of improving this situation, it has to be taken into account that data locality can be achieved in nearly 100% of the cases. So, how to handle the case in which data locality cannot be achieved does not really matters, provided the very limited impact. As it just needs to work, I propose to transfer the file to the local sandbox using Hadoop command line tools in this case. Efforts should be then focused to improve the access in case of data locality.

The new access method which I propose in case of data locality is to bypass the entire Hadoop framework and point ROOT to the file on the node's local file system corresponding to the chunk (file) to be analyzed. This is possible since chunks are stored into single files on the node's local file system and since the working hypothesis assure that every file to be analyzed is contained in only one chunk. The location of the chunk on the local filesystem and whether the access can be done locally or not, as well as the number of chunks composing the file to check that the working hypothesis have been respected[5], can all be obtained from simple Hadoop tools. The entire procedure proposed is schematized below.

1. The MapReduce task is started on a data set of binary ROOT data (a set of files). Since each of these files is stored on HDFS in only one chunk, every Map task will be in charge of analyzing one entire file of the data set.

2. Every Map task checks then if a local replica is available for the file. If this is the case, it obtains the full path of the corresponding file on the node's local file system and run the user's ROOT code on it.

[3]Filesystem in USErspace (FUSE) is a loadable kernel module for Unix-like computer operating systems that lets non-privileged users create and access their own file systems.

[4]This means that if the access method is invoked from a node which has a copy of the data, the access is done locally.

[5]The system can work even if the working hypothesis have not been respected, since it can switch back on the access method for non local files, delegating Hadoop how to access the file correctly.

Otherwise, it copies the file using Hadoop command line tools to the Map's sandbox and then run the user's ROOT code on the copy.

3. ROOT starts on the given file, accessing it as a standard file from the node's local file system in both cases, and performs the analysis.

4. The output is then collected by the Reduce task(s) and merged to the final, aggregated output.

For making the user's code available to the Map tasks, two options are available.

A. Store the code on HDFS as a complete self-contained package (Grid style). Every Map task will then download a copy of the code, unpack and run it.

B. Put the code on a support network file system like NFS, accessible from every Hadoop node (Batch style). Every Map task will then just invoke the code from a standard path.

It has to be noted that since the user's code needs to be accessed by every Map task, if it becomes comparable in size with the files to analyze the consequent data transfer for accessing the code itself cannot be neglected. This data transfer has therefore to be minimized, as it risks to vanish the benefits from Hadoop/MapReduce's data locality. Here a solution for both the previous options is provided.

A. Make the first Map task on every node to download the code in a local shared location, where it will be available for all the next Map tasks. This is a rough implementation of a cache.

B. Configure the support network file system with an high caching size. For every node, the first Map task which access the user's code triggers the caching of the code on the node. The following Map tasks running on the same node will then be able to use the locally cached copy of the code, without data transfers. This approach brings also a major advantage: if the user needs to modify just few things in the code between two jobs (i.e. setting new selection criteria o modify a library), the highly cached network file system will re-cache only the modified files, and in the smartest approach, only the changed bits.

These methods are as much effective as the higher the number of Map tasks per nodes is (task belonging to the same MapReduce job). Consequentially, in case of a very small data set (or a very large cluster), the MapReduce job would end up in scheduling just few Map tasks per every node, making the access to the user's code from the nodes again problematic.

Once the Map tasks are able to compute taking advantage of data locality, the main goal is reached: as already explained, in HEP analyses the aggregation (Reduce) step is usually just a count of the selected events or an aggregation of histograms. The Reduce computational and I/O weight, compared to the analysis, is therefore near to zero.

To transfer the output data from the Map tasks to the Reduce one(s), I propose to store these data (plain text or binary) in files on the HDFS, and then to forward their paths to the Reduce task(s). The Reduce task then reads every HDFS path, access the partial results and performs the aggregation. As the input paths to the Reduce task are just textual strings, the standard MapReduce framework tools can be used. For example, an approach similar to Hadoop Streaming can be used to specify a custom program as Reduce, which would receive the paths via standard input, one per line, as the Map tasks end. A simple program in the user's preferred language can then access these HDFS files and perform the wanted merging operation. The access can be done via one of the HDFS access methods discussed for the Map task, which in this case, since the simplicity of the Reduce task and its negligible computational weight, are all suitable. Typical merging operations in HEP are retrieving from textual outputs various quantities like the total selected events, total energy, as well as standard deviations and more complex ones; or merging output ROOT histograms[6] to perform more complex operations or plots afterwards.

[6]Utilities as the "hadd" ROOT macro for merging histograms are available to manipulate ROOT data.

By putting the pieces together, a MapReduce job acting as a wrapper for the user's codes can be easily written. Users can then use this MapReduce job to run their own analyses by just specifying:

- the input data set;
- the location of the Map code;
- the location of the Reduce code;
- the output location.

User's Map and Reduce code has to be prepared following just few guidelines: the Map will receive as the first argument the file on which to operate, its output will have to follow a conventional naming schema to be uploaded to the HDFS and to be accessed from the Reduce, which will receive from the standard input, one per line, the HDFS paths of the files to merge in the final result.

6.4 Performance

As explained at the end of Section 4.1, comparing the performance between a standard computation approach and a data locality one is not easy. Running a I/O intensive benchmark on an empty standard computing model cluster with a fast network infrastructure would give more or less the same execution times than running the same benchmark on Hadoop/MapReduce. But when the cluster becomes busy, the storage element(s) overloaded and the network congestioned, then the benefits of an Hadoop/MapReduce approach would be clearly seen, since the latter permits to completely avoid these situations. That is, Hadoop is more about scaling up and having a smart cluster architecture than a pure performance boost, and this important observation will be argued in this Section. In this context, the performance is evaluated in terms of "saved bandwidth": in a distributed environment it directly reflects on both computing time and on the cost of the network infrastructure. From this point of view a key factor is given by the data locality ratio. This value represents the percent of how many Map tasks are in mean able to access their data locally. It has in fact not to be taken for granted that the job scheduler is capable to plan the computation to allow every Map task to take advantage of data locality (some of them may land on a node which does not hold a copy of the data they have to analyze). Luckily tuning Hadoop's Fair scheduler using a small delay before

allocating the resources allows to achieve a data locality ratio close to 100% on shared clusters [30].

To try to give an idea of the performance in terms of computing time, one can consider an hypothetical example analysis and evaluate how it would perform with a traditional approach and with an Hadoop/Mapreduce approach. If the execution time of the code is X seconds and the time for gathering the input data file via the network is Y seconds, the total[7] time for analyzing n files via a traditional approach would be given by:

$$t_{traditional} = X \cdot n + Y \cdot n \tag{6.1}$$

If adopting an Hadoop/MapReduce approach exploiting data locality, taking into account as approximation of the data locality factor the value of 99%, the required time for performing the same analysis would be given by:

$$t_{Hadoop} = X \cdot n + Y \cdot (n \cdot 0.01) \tag{6.2}$$

To fix some numbers, one can for example consider a case study of a commodity cluster of ten nodes, with eight cpu cores per node, and a Gigabit network interconnection. An hypothetical example data set can be assumed to have a dimension of 8 GB, divided in 100 MB files. In this framework the data set would be analyzed in parallel, and every node of the cluster would be in charge of analyzing eight files. The software for analyzing every file is supposed to run in 30 seconds. Even if the dataset is very small compared to the real ones, the execution times are realistic. The total execution time of this analysis on both a traditional and a Hadoop/MapReduce approach is below evaluated.

Traditional computing model

Since the entire data set is going to be read in parallel from each analysis task, the available bandwidth from the storage element for every core is ∼1.56 Mbit/s. This means transferring a file of 100 MB would take 640 seconds. The computing time of 30 seconds has then to be added, so that

$$t_{traditional} = 640\ s + 30\ s = 670\ s; \tag{6.3}$$

[7] If the code is smart enough, it could of course start analyzing the file as it starts to read it, which would decrease the network transfer. The aim of this quick performance analysis is anyway to just give the idea of the Hadoop/MapReduce potential.

or, assuming that the file is started to be analyzed while being accessed over the network

$$t_{traditional} = \sim 640\ s. \tag{6.4}$$

Hadoop/MapReduce computing model

The data does not need to be transferred for all the 100 analysis tasks, but just for one of them. The entire bandwidth can then be exploited to perform this single transfer, which takes only 1.25 seconds. The average computing time is then in given by

$$t_{Hadoop} = 30\ s + 1.25\ s = 31.25\ s; \tag{6.5}$$

or, assuming also in this case that the file can be started to be analyzed while being accessed over the network

$$t_{Hadoop} = \sim 30\ s. \tag{6.6}$$

Comparing the two computing times, ~ 670 vs. ~ 30 seconds , gives a clear idea of the advantages brought by Hadoop/MapReduce's data locality. One could note that by bringing up the network speed of the storage element (by installing a 10 Gigabit network adapter, for example) the execution time of the standard approach would become comparable with the Hadoop's one. But here is exactly where the real potential of Hadoop in terms of a smart cluster architecture rather than a pure boost execution times comes up, and here two scenarios in support of this observation, already introduced in Section 4.1, are given.

From the cluster architecture point of view, upgrading the network speed on the storage element implies the balancing of various components as network cards, switches and wiring. Beyond a given threshold, the bottleneck will become the storage element's internal bus and disks speed, which should be upgraded as well. All these upgrades will at a certain point hit the technological limit, and then the only way to speed up the data access will be to mirror the storage element with another one (or more). This will lead to the common problems involved in managing two storage elements. which includes the usage of a distributed file system. The bottlenecks will anyway be still present, unless pairing a storage element with only one computing node, and connecting everything in a matrix - which is the extreme, most complicated and most expensive solution. It has also to be taken into account that balancing the cluster between network speed and computing power becomes harder and harder as it grows in size. For every new computing node, a further speedup of the data transfer is indeed required.

From the software point of view, suppose that one improves the network connection on the storage element(s) to perfect balance the computing speed, optimizing the cluster for a specific analysis. What about if a different kind of analysis has to be performed on the cluster? This new analysis could compute faster, making the effort made for improving the network speed not sufficient, or it could perform slower, making the effort made (and the investments made) completely useless.

These two scenarios clearly shows that balancing and dimensioning the computing, network and storage resources on a cluster is a complicated task from various points of view. By exploiting data locality, one can completely forget about all these problems. Using Hadoop/MapReduce, tasks run at the maximum speed they can and the cluster can be expanded indefinitely, scaling it up without any constrain. The more general considerations of Section 4.1 about the exponential growth of the data flows are also to be taken into account and sums to these arguments.

Chapter 7

A real case: top quark cross section measurement in ATLAS

The approach presented in Chapter 6 for running ROOT on Hadoop with a MapReduce approach has been tested by me on a real case, which I will discuss in this Chapter. This real case is the top quark pair production cross section measurement analysis performed by the ATLAS Udine Group [31][32].

In the first three Sections, I give a brief introduction of the physics which guides the analysis; in Section 7.3, I describe the data set and the Hadoop facility used for the analysis, and finally in Section 7.4 both the physics and the performance results are reported.

7.1 Top quarks production and decay

The top quark is the heaviest quark of the Standard Model of elementary particle physics (SM). Discovered in 1995 at the Tevatron accelerator [33], has been identified at the LHC in 2010 [34][35]. The top quark mass is measured to be five orders of magnitude larger than the mass of the electron, and at least 11 orders of magnitude larger than the smallest measured neutrino mass (assumed to be massless in the formulation of the SM reported in Section 2.1). Due to its large mass, the top quark decays faster than the typical hadronization time of QCD ($\Gamma_{top} \gg \Lambda_{QCD}$), being the only quark

that does not form bound states. Its decay offers the unique possibility to study the properties of an essentially bare quark.

In the SM framework, top quarks can be produced in pairs ($t\bar{t}$) predominantly via the strong interaction, or singly via the electroweak interaction. The energies needed to produce them are currently accessible only at hadron colliders. Here just an overview of the two production modes is given. A more in depth discussion of the argument is provided in [32].

For the top pair production, at leading order (LO) two production subprocesses can be distinguished: $q\bar{q}$ annihilation and gg fusion. The corresponding relevant Feynmann diagrams are shown in Figure 7.1. At high energies, the gg fusion process dominates for both $p\bar{p}$ and pp collisions. This is the case at LHC, where in 2010 and 2011, at the centre-of-mass energy of 7 TeV, about 80% of $\sigma(t\bar{t})$ was due to gg fusion. Next-to-leading order (NLO) calculations account for associated quark production and gluon bremsstrahlung, and virtual contributions to the LO processes[1]. In the following, the theoretical predictions reported from [32] for the $t\bar{t}$ total production cross section have been obtained using the HATHOR code [36]. These theoretical cross sections were used in [32] to normalize the predicted yields obtained with the MC simulation.

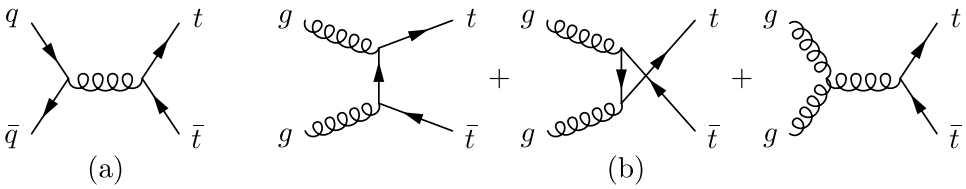

Figure 7.1: Feynman diagrams of the LO processes for $t\bar{t}$ production: (a) quark-antiquark annihilation ($q\bar{q} \rightarrow t\bar{t}$) and (b) gluon-gluon fusion ($gg \rightarrow t\bar{t}$).

The top quark can be produced, not in pairs, via the electroweak process. For this single top quark production, there are three production modes which are distinguished by the virtuality Q^2 of the W-boson ($Q^2 = -q^2$), where q is the four-momentum of the W. The dominant source of single top quarks at the LHC is the t-channel, where a virtual W-boson strikes a b-quark (a sea quark) inside a proton. The other two production modes are less relevant and consist in the s-channel and in the W-associated pro-

[1] At the centre-of-mass energy of the LHC (7 TeV), the NLO corrections to the LO $t\bar{t}$ production cross section are of the order of 50%.

duction. The Feynman diagrams representing this processes are shown in Figure 7.2. For the cross section values calculated in [32] and used in this Chapter, the single top production is considered as a background process.

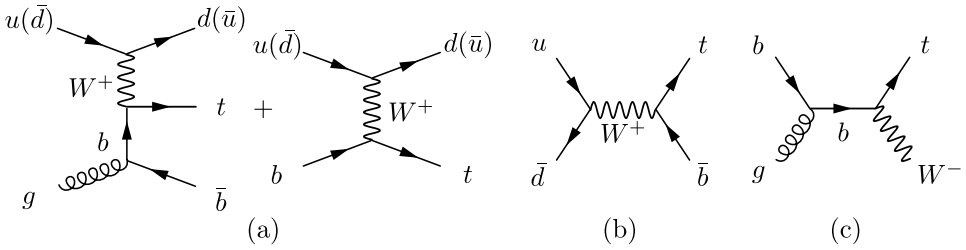

Figure 7.2: Representative Feynman diagrams for the three single top quark production modes: (a) t-channel, (b) s-channel, and (c) W-associated production process.

The dominant process at LHC is $t\bar{t}$ production, and the observation of these events was one of the milestones for the early LHC physics programme, since the measurement of the related cross section ($\sigma t\bar{t}$) in the various decay channels was and is interesting for several reasons:

- it allows a precision test of the theoretical predictions from perturbative QCD, by comparing them with experimental measurements performed in different decay channels;
- many aspects of the detector performance have been improved by exploiting the abundant $t\bar{t}$ sample which has been produced in the first years of data-taking;
- $t\bar{t}$ production is presently an important background in various investigations for physics beyond the SM, which may also give rise to additional $t\bar{t}$ production mechanisms or modification of the top quark decay channels.

Within the SM, the top quark almost always decays in a W boson and a b-quark, and the decay topologies are determined by the decays of the W bosons, which in turn can decay in a lepton and the corresponding neutrino ($\ell\nu$) and in two quarks, each of them fragmenting to give a jet (j). The semileptonic mode ($t\bar{t} \rightarrow \ell\nu bbjj$) with a branching ratio of 34.3% (combining both e and μ lepton flavours[2]), gives rise to a final state with one

[2]The branching ratios include the small contributions to $W \rightarrow l$ from $W \rightarrow \tau \rightarrow l$.

isolated high p_T lepton, missing traverse energy coming from the undetected neutrinos and jets, two of them coming from fragmentation of b-quarks. As test case analysis, only the electronic decay is covered here.

7.2 The analysis

The analysis developed by the ATLAS Udine group for the top quark search and cross section measurement was based on a code named ICToP2. The code performed a cut-and-count analysis, which as already introduced is a kind of analysis where every event undergoes a series of selection criteria and at the end is accepted or not. The cross section is then obtained by comparing the number of selected events with the luminosity, the efficiency in the selection of signal events, and the expected background events.

For selecting $t\bar{t}$ events in the electron decay channel study, the criteria are set as follows:

- the appropriate single electron trigger has fired;
- the event contains exactly one lepton (electron) with transverse momentum $p_T > 20$ GeV;
- traverse missing energy $\not{E}_t$ (indicating the presence of an escaped neutrino) > 35 GeV and Transverse mass $m_T(W)$[3] > 25GeV;
- at least four jets with transverse momentum $p_T > 25$ GeV all with pseudorapidity $|\eta| < 2.5$;
- identification of at least one jet coming from the fragmentation of the b-quark.

The ICToP2 code package comes with a plugin-oriented structure which allows using custom drivers for the analysis. The two plugins (drivers) used for the test case analysis are the *Base* driver and the *CutFlow* driver:

- the **Base** driver generates a ROOT file containing the analysis results, ready to be plotted;
- the **CutFlow** driver generates a textual "flow" of the number of events which passed the first cut, the second, the third.. until the last one.

[3] $m_T(W) = \sqrt{2{p_T}^{\ell}{p_T}^{\nu}(1 - cos(\phi^{\ell} - \phi^{\nu}))}$

The driver has to be chosen at compilation time, by using the command `make File_Base` or `make File_CutFlow`.

7.3 The data set and Hadoop

The data used for the test case has been taken with all the subsystems of the ATLAS detector in fully operational mode, with the LHC producing proton-proton collisions corresponding to a centre of mass energy of 7 Tev with stable beams condition during the 2011 run up to August. As already introduced in Section 2.3.4, given the huge amount of data to be analyzed, the ATLAS Collaboration has developed its own procedure for final users analysis which is based on a light-version of the original data, the D3PD (3rd level Derived Physics Data).

These D3PD files, which are ROOT n-ntuples already containing only "filtered" (interesting) events, are then "slimmed" (i.e. only the branches needed for the analysis are kept) to further reduce their size. The data set on which the ICToP2 code operates on is therefore a set of "slimmed" D3PD n-tuples including only events (and the per event information) needed for the top quark pair production cross section analysis. The data taking conditions described above resulted in a data set corresponding to an integrated luminosity of 2.05 fb^{-1}, with a size of 338,6 GB when considering only electron channel D3PDs. Accordingly to the ATLAS data acquisition model, this data set is structured in 8830 files, with an average size of $\sim$ 38 MB and a maximum file size of $\sim$ 48 MB, which fits in the default HDFS block size of 64 MB. The mapping between the objects involved in the ATLAS data acquisition model, their order of magnitude, their data types, and the corresponding objects on Hadoop/MapReduce is reported in Table 7.1.

The Hadoop cluster which I have used for the test case analysis is a ten nodes cluster configured with the Fair scheduler (see Appendix A.4). I have compiled the ICToP2 code without any modifications and I have copied the data set straightforward from its original location at CERN Tier-0. The nodes, which have eight CPUs, were configured to run ten Map tasks per node[4]. I have then performed the analysis using a Java MapReduce wrapper for the ICToP2 code as previously described in Chapter 6. This technique worked as expected, leading to a total of 8830 Map tasks (one per file) and an average of 883 data files analyzed per node. The aggregation of the partial

[4]This choice was made to help exploiting at maximum the CPUs, as some of the Map tasks could be waiting for I/O operations completion and therefore not using the CPU.

results was done by a simple Reduce task written in Python, which was in charge of summing the number of $t\bar{t}$ events observed in every file of the data set by the Map tasks. This sum is computed as the Map tasks progressively ends and so partial results are made available. Figure 7.3 shows the status report from the Hadoop Job Tracker while running the analysis. The data locality ratio which I have measured is 100%, confirming the expected value.

Object	Order of magnitude	Type	On Hadoop/MapReduce
Event	1	ROOT data structure	unknown (binary)
File	$10^2 - 10^4$	ROOT file, set of events	chunk,record
Lum. block	10^4	Dir., set of Files	Dir.
Run	$10^5 - 10^6$	Dir., set of Lum. blocks	Dir.
Data set	$10^5 - 10^9$	Dir., set of Runs	Dir. (input data set)

Table 7.1: Mapping between logical units of the ATLAS data acquisition model, their order of magnitude, their data types and the corresponding objects on Hadoop/MapReduce.

Kind	% Complete	Num Tasks	Pending	Running	Complete
map	48.33%	8830	4462	100	4268
reduce	16.07%	1	0	1	0

Figure 7.3: Hadoop Jobtracker output while running the analysis.

7.4 Results

The overhead by the Hadoop/MapReduce infrastructure for handling this MapReduce job has been measured to be $\sim$ 1,17 GB. The ICToP2 code size is $\sim$ 12 MB, and, as already discussed, every node performing the analysis has to transfer it only once. Since the entire Hadoop test cluster have been used for the tests, the consequent total data transfer has been of $\sim$ 0,12 GB. Given the 100% ratio of data locality and that the output data transferred from the Map tasks can be considered as negligible, these

values lead to a total data transfer of 1,29 GB across the Hadoop cluster for analyzing the 338,6 GB data set. To analyze the same data set, with the same code, on a cluster of the same size following the standard computing model[5], the total data transfer would raise to 338,1 GB. This result, which as explained in Section 4.1 is the most significative one when evaluating a data locality approach, is summarized in Table 7.2.

Data transfers:	Hadoop Computing Model	Standard Computing Model
Code	0,12 GB	0,12 GB
Infrastructure overhead	1,17 GB	-
Input data set	0 GB	336,6 GB
Output events count	-	-
Total:	1,29 GB	336,72 GB

Table 7.2: Comparison of the data transfers needed to perform the test case analysis on the Hadoop/ MapReduce computing model and on a generic standard computing model, with a ten nodes cluster. No value means negligible.

For completeness, the physics results obtained on Hadoop have been compared with the official ones, which were found using the same ICToP2 code, and it has been confirmed that they are in agreement, as explained in the following.

At the time of this thesis work, the available official, validated results [37] were computed on the 2010 data set, that had an integrated luminosity of 35.3 pb^{-1}. To be compared, the results of the two analysis (the total number of observed top candidate events) have to be rescaled taking into account the different luminosities. This can be achieved by applying a simple proportion:

$$TotEvents_{L=35.3pb^{-1}} = TotEvents_{L=2050pb^{-1}} \cdot \frac{35.3}{2050} \tag{7.1}$$

Even if this rescaling allows to compare the number of selected events of the two analyses, a direct comparison is still not possible since the number of

[5]Therefore relying on a central storage element, without data locality, and considering the infrastructure overhead in the data transfer as negligible.

selected events for the e-channel reported in Table 1 of [37] were obtained without any b-tagging requirements. Nevertheless, the number of selected events obtained using the b-tagging can be extracted a posteriori from the histograms in Figure 5 of [37], and it is has been checked that the results are in agreement. It is anyway more interesting to compare the results with an analysis which used the same method as the one used on Hadoop, as the analysis performed on the 2010 data set reported in [32] (even if minor differences in the results due the tuning applies), which will be therefore taken as reference. Besides, also the analysis performed on the 2011 data set in [32] and in the official ATLAS note [38] cannot be used for a direct comparison since the b-tagging algorithm was switched to a newer more efficient version, therefore providing a different total number of observed top candidate events.

Table 7.3 shows the results found by analyzing the 2010 data set in [32] (the "Official Observed" row), the results (rescaled) found by analyzing the 2011 data by running the analysis on Hadoop (the "Hadoop Observed" row) and the values predicted by the MC simulations. As can be seen, the value found analyzing the 2011 data set with the Hadoop approach is statistically compatible with the MC predictions and close to the value found in [32].

	$\geq$ 4-jet
ttbar (MC)	135.1 $\pm$ 23.0
QCD (DD)	8.6 $\pm$ 9.4
W+jets (MC)	15.9 $\pm$ 12.4
Z+jets (MC)	1.5 $\pm$ 1.0
single top (MC)	6.7 $\pm$ 1.7
dibosons (MC)	0.2 $\pm$ 0.1
Total background	32.9 $\pm$ 15.6
Total expected	168.0 $\pm$ 27.8
Official Observed	156
Hadoop Observed	168.9

Table 7.3: Results reported in [32] compared with the results obtained on Hadoop, for the e+jets channel, tagged, on the 2010 data set.

The top quark case-study covered in this Chapter is the first example of an HEP analysis performed using Hadoop in the ATLAS Collaboration.

Chapter 8

Conclusions

The last decade has ben characterized by a constant growth in technological innovation. Its consequence is an exponential increase of the data flows, which is a phenomena commonly referred as the Data Deluge or Big Data revolution. The widespread mutual interest in being able to analyze these huge amounts of data is today a central matter and has lead to a boost in the technologies addressed to achieve this common goal, bringing data locality as their primary key feature. Among these new technologies for distributed computing, the most widely adopted one is the Hadoop/MapReduce framework, which brings several benefits including:

- exploiting data locality allows to avoid congest the communication channels, and the average data locality ratio is shown to be nearly 100%, which reflects one-to-one on the network usage that decreases by the same factor;
- Hadoop scales linearly, which means that when more computing or storage resources are required to be added it is not necessary to rebalance the cluster, so that no bottlenecks arise when scaling up;
- Hadoop can work with heterogeneous hardware and every computing or storage resource can help in the overall computation, which makes it the perfect solution for exploiting "commodity" components;
- a world-wide community including top level companies is supporting and constantly improving the project (the new Job Scheduler algorithm was for example developed by Facebook), a project in which the global interest is growing steadily.

In thesis work I have discussed the motivations for using Hadoop/ MapReduce and the benefits which it can bring in two fields of application. In first place the IT industry, and in particular at the CERN IT Department, where I have successfully introduced Hadoop/MapReduce for the CERN Advanced STORage manager (CASTOR) monitoring, together with a set of tools as the Metrics Analysis Engine which has been taken over by the Department and is still in production.

Secondly, I investigated the usage of Hadoop/MapReduce in the field of the HEP analyses, as a possible solution for the increasing network congestion problems reported by several computing centers. The approach which I have ideated for this application takes into account several aspects of introducing a new technology which could be potentially be adopted worldwide in this field, and relies on two main features to try providing the maximum grade of transparency for both the users and the administrators. The first is to allow storing HEP data on the Hadoop Distributed File System in its original format, avoiding format conversions and making data transfers straightforward. The second is to let a user easily run its own (already existent) ROOT code, as no specific knowledge about Hadoop/MapReduce is required and changes in the computing model are minimal.

Bringing Hadoop/Mapreduce in the field of the HEP analyses is my very original work. I have tested it on a real case, an analysis to measure the top quark pair production cross section with the ATLAS experiment, and worked as expected, bringing great benefits in terms of reducing by several orders of magnitude the network usage for accessing the data. This case-study, carried out by interacting with real users on real datasets, is the first example of an HEP analysis performed using Hadoop in the ATLAS Collaboration, and I have presented it at the ATLAS Software and Computing week [39] in June 2012.

Appendix A

CERN's infrastructure

This appendix is meant to be a brief technical summary of the work I have done at the CERN IT Department. It has been fundamental for allowing an easy take over of my projects by the Department. The system in use at CERN Computing Center for the installation, configuration, and management of operating systems and application software is the Quattor tool suite [40]. In particular, in the following the *CDB* profiles and the *NCM* components, as *spma* and *filecopy*, will be assumed to be well known.

The convention used for the notation is to write in *italicus* package names, variables and parameters; to use the `monospace` for code listings; and to write paths, file names and software packages in the standard font.

A.1 The transport substrate

Chronologically, deploying a new transport substrate was the first step that I had to face before start working on the new monitoring system. This because of two main reasons:

1. a new transport layer providing a data source on which to experiment the new monitoring chain without touching the production environment would have been very handy;
2. since the future Hadoop approach was already planned, a transport layer capable of transferring data directly to it was required to start storing data for further testing.

Scribe had to be installed on all the more than 1500 CASTOR nodes, and therefore the testing, packaging and deploying phase has been quite intensive. Moreover, since a failure in Scribe could affect the LHC data taking, it had to be handled very carefully to avoid this contingency. I deployed Scribe on CASTOR by developing a *scribe-injector* support script, and by installing it together with a local Scribe server on every node. The *scribe-injector* script tails the configured set of log files and sends every new log line to the local Scribe server, which forwards the received messages to the main Scribe aggregator. The latter is running on a server named *lxbsq1204*[1], which will be referred using this shortcut in this Appendix. The local Scribe server is configured to buffer locally ifthe main Scribe aggregator is not reachable. When recovering the buffer (replying it to the main Scribe aggregator) the outgoing bandwidth is limited to 50 Kbpbs, as explained in Section 5.2. On the main Scribe aggregator another local Scribe server merges all the log messages received to a local file used as a cache for online analysis, and forwards everything (without any aggregation) to another instance of Scribe which in turn stores the data on the HDFS.

There are in detail two Scribe packages, named *scribe* and *scribe-hadoop*. The first is running on all the CASTOR nodes and on the main Scribe aggregator. Its task is to transfer data between Scribe instances. The latter is running only on the main Scribe aggregator, and its task is to transfer the incoming data to the HDFS. I deployed these two independent packages (*scribe* and *scribe-hadoop*) to make the scribe package lighter (which was an interesting feature since it had to be installed on every CASTOR node) and to allow running two instances of Scribe on the Scribe aggregator (which can be chained to achieve the total flexibility in configuring the wanted behavior).

On the main Scribe aggregator, the incoming log messages are:

1. merged in one file per day (/ var / log / scribe / castor / c2aggregated / c2aggregated / $DATE / c2aggregated_00000.log);
2. stored on the HDFS following the structure of / data / scribelogs / castor / c2$INSTANCE / $NODETYPE/ $DATE / $NODETYPE_00000;

where `$INSTANCE` is the CASTOR instance (i.e. c2atlas, c2cms...), `$NODETYPE` is the type of the node (i.e. diskserver, heaadnode...) and `$DATE` is the

[1]This name is because of the CERN computer centre naming method, to allow a quick lookup of the physical location.

date. Scribe does not permit to natively specify such a structure for the destination paths, which is required in our case for organizational and partitioning reasons. Therefore, Scribe source code had to be modified. Every part of the code which has been modified has been marked with a "CERN mod" comment. The modified source code is available at `http://svnweb.cern.ch/world/wsvn/CASTOR/SCRIBE/trunk/src`.

The sources for building the Scribe packages (in the rpm format) are located at `http://svnweb.cern.ch/world/wsvn/CASTOR/SCRIBE/trunk/rpms`. The *scribe-injector*, *scribe* and *scribe-hadoop* packages sources do not contain the real Scribe source code that should be compiled when building the rpm packages, they instead contain binary files and libraries. This is a temporary solution which was required since it is not straightforward to build Scribe: special libraries, not even shipped as rpms, are required. This solution has to be replaced by a proper packaging of libraries and by source rpms specifications. Building the rpms is just a matter of downloading the wanted subtree and then typing the "make rpm" command. After downloading a package, if one needs to make some modifications, the changes should be committed to the SVN repository, and then the new version of the rpm uploaded to the CERN software repositories. A complete sequence for modifying the *scribe-injector* rpm, suitable for all the three packages, is below reported as example.

```
svn co svn+ssh://svn/reps/CASTOR/SCRIBE/trunk/rpms/scribe-injector
# Modify what you need and update scribe-injector.spec,
# with a new release or version number.
svn commit -m "What has been done.."
make rpm
```

The rpm will be created in RPMS/x86_64 (for the x86_64 architecture). For uploading the rpm to the CERN software repositories, the `swrep-soap-client` tool has to be used. A more detailed description for each of the three packages is provided in the coming subsections.

Concerning Scribe configurations, there are few very important parameters to keep in mind:

- *check_interval*: specifies every how many seconds Scribe checks if new messages are present and every how many seconds it has to send the buffer files when recovering;

- *max_size*: specifies how big can be a file written by Scribe: after reaching this value, Scribe starts writing to a new file;
- *timeout*: specifies the maximum time (in milliseconds) which Scribe can spend for transferring a single buffer file replying, the default is 5000 (5 seconds).

By combining *check_interval* and *max_size* , the upper limit of the outgoing bandwidth over a buffer replying can be controlled, since Scribe will send a buffer file of a maximum size of *max_size* Bytes every *check_interval*. One has to be very careful here since when replying the buffer, Scribe does not handle files which end up in being partially written due to a timeout interrupt. They are just sent again, and the risk of never ending loops of data writing with a wrong set parameters is concrete (see Section A.1.4).

A.1.1 Package *scribe-injector*

```
Package name:   scribe-injector-0.3-8.x86_64.rpm
Home folder:    /usr/local/scribe-injector
Daemon binary:  /usr/local/scribe-injector/bin/scribe-injector
Config file:    /usr/local/scribe-injector/scribe-injector.conf
Log files:      /var/log/scribed.log, not rotated
Init script:    /etc/init.d/scribe-injector
```

The configuration file (Listing A.1) specifies the list of log files to tail and their category. The init script obtains the CASTOR instance from a *sysconfig* configuration file, which is written by the *CDB filecopy* component in /etc/sysconfig/scribe-injector. In case it is not able to read this file, it will try to get the instance name from the /etc/castor/castor.conf file. It is important on new installations to run the filecopy component before the spma one, otherwise the scribe-injector will not start correctly. In case this happens, just restart the injector (/etc/init.d/scribe-injector restart). If the configuration file has to be modified, a restart of the injector is required as well.

```
# Failure handling policy (to be implemented)
# "hold": do nothing
# "bufferlocal": write to a tmp buffer file
# "connectmain ip_addr:port": bypass local Scribe and try connecting
#                             to the main on given ip and port

failure_handling_policy hold

# Files to monitor:
file /var/log/castor/mighunterd.log headnode
file /var/log/castor/rechandlerd.log headnode
file /var/log/castor/schedulerd.log headnode
file /var/log/castor/c2probe.log headnode
file /var/log/castor/migrator.log headnode
file /var/log/castor/rhd.log headnode
file /var/log/castor/expertd.log headnode
file /var/log/castor/nsd.log central
file /var/log/castor/rmmasterd.log headnode
file /var/log/castor/stagerd.log headnode
file /var/log/castor/jobmanagerd.log headnode
file /var/log/castor/recaller.log headnode
file /var/log/castor/rtcpclientd.log headnode
file /var/log/castor/tperrhandler.log headnode
file /var/log/castor/gcd.log diskserver
file /var/log/castor/operations.log diskserver
file /var/log/castor/rfiod.log diskserver
file /var/log/castor/rmnoded.log diskserver
file /var/log/castor/transfermanagerd.log headnode
file /var/log/castor/tapegatewayd.log headnode
file /var/log/castor/srmfed.log srm
file /var/log/castor/srmbed.log srm
```

Listing A.1: Configuration file for the Scribe Injector.

A.1.2 Package *scribe*

```
Package name:   scribe-2.3-4.x86_64.rpm
Home folder:    /usr/local/scribe
Daemon binary:  /usr/local/scribe/bin/scribed
Config file:    /usr/local/scribe/scribe.conf
Log files:      /var/log/scribed.log, rotated every 5 days
Init script:    /etc/init.d/scribed
```

Two configuration files are provided via *CD*B for this package. The configuration file for the local Scribe servers on the CASTOR nodes (Listing A.2) sets Scribe to listen for messages on port 1464 and attempt to forward all messages to a Scribe instance of the main Scribe aggregator on port 1463. The configuration file for the main Scribe aggregator (Listing A.3) sets Scribe to listen for messages on port 1464, to store them locally (for online analysis) and to forward them to another Scribe instance on the same server on port 1464 (which will in turn store them on HDSF, for offline analysis). In both cases, if Scribe is unable to forward the messages to the other Scribe instance, it buffers them on disk and keep retrying.

A.1.3 Package *scribe-hadoop*

```
Package name:   scribe-hadoop-2.3-2.x86_64.rpm
Home folder:    /usr/local/scribe-hadoop
Daemon binary:  /usr/local/scribe-hadoop/bin/scribed-hadoop
Config file:    /usr/local/scribe-hadoop/scribe-hadoop.conf
Log files:      /var/log/scribed-hadoop.log, not rotated
Init script:    /etc/init.d/scribed-hadoop
```

The scribe-hadoop package is basically a clone of the scribe package, in which the binary and the libraries are changed to include support for the HDFS and to allow having two scribe packages running on the same server (the main Scribe aggregator). The configuration file is quite verbose since every data source has to be explicitly specified through its category, to allow storing the incoming data following the wanted schema (c2$INSTANCE / $NODETYPE/ $DATE / $NODETYPE_00000). To let the configuration be modified easier, instead of shipping the entire configuration file, a macro is provided (Listing A.4).

```
port=1464
max_msg_per_second=2000000
check_interval=10
max_queue_size=5000000 # Scribe default value

<store>
  category=default
  type=buffer
  target_write_size=16384
  max_write_interval=30
  buffer_send_rate=1
  retry_interval=10
  retry_interval_range=8
  max_queue_length=2000000 # Scribe default value

  <primary>
    type=network
    remote_host=128.142.171.200
    remote_port=1463
  </primary>

  <secondary>
    type=file
    fs_type=std
    file_path=/tmp/scribe
    base_filename=thisisoverwritten
    max_size=500024
  </secondary>

</store>
```

Listing A.2: Configuration file for Local Scribes (on CASTOR nodes).

```
port=1463
max_msg_per_second=2000000
check_interval=30
max_queue_size=50000000
num_thrift_server_threads=1
new_thread_per_category=no


# ========== Store locally c2pps-aggregated ==================
<store>
  category=c2*
  type=file
  file_path=/var/log/scribe/castor/c2aggregated
  base_filename=c2aggregated
  sub_directory=c2aggregated
  rotate_period=daily
  rotate_hour=0
  rotate_minute=0
  create_symlink=no
  max_size=10000000000000
</store>

# ========== Forward to Scribe Hadoop instance ================
<store>
  category=c2*
  type=buffer
  <primary>
    type=network
    remote_host=localhost
    remote_port=1464
    timeout=300000
  </primary>
  <secondary>
    type=file
    fs_type=std
    file_path=/tmp/scribe
    base_filename=c2forwarder
  </secondary>
</store>
```

Listing A.3: Configuration file for the Main Scribe Aggregator (on lxbsq1204).

```
#!/usr/bin/python

#======== Global Configuration ========

out = open("./scribe-hadoop.conf","w")
out.write('''
port=1464
max_msg_per_second=2000000
check_interval=30
max_queue_size=50000000
''')
out.write("\n")

#======== Per-nodetype Configuration ========

instances=['c2pps','c2cernt3','c2public','c2lhcb',
           'c2alice','c2cms','c2atlas','c2repack']
nodetypes=['diskserver','central','headnode','srm']

for instance in instances:
  for nodetype in nodetypes:
    out.write("# ====== "+instance+"-"+nodetype+" ======\n")
    out.write("<store>\n")
    out.write("  category="+instance+"-"+nodetype+"\n")
    out.write("  type=file\n")
    out.write("  fs_type=hdfs\n")
    out.write("  file_path=hdfs://lxbsq0929.cern.ch:8020/data/
                scribelogs/castor/"+instance+"\n")
    out.write("  base_filename="+nodetype+"\n")
    out.write("  sub_directory="+nodetype+"\n")
    out.write("  rotate_period=daily\n")
    out.write("  rotate_hour=0\n")
    out.write("  rotate_minute=0\n")
    out.write("  create_symlink=no\n")
    out.write("  max_size=100000000000\n")
    out.write("</store>\n")
    out.write("\n")

out.close()
```

Listing A.4: Macro for the configuration file for the scribe-hadoop package (on lxbsq1204).

A.1.4 Known problems

Despite Scribe is working fine in the CASTOR environment, it has some problems. First of all, as already mentioned, the building is not straightforward and requires some some particular libraries. This means that it cannot be easily included This facts led to some problems in maintaing it, since it cannot be easily included in a source rpm package, leading to maintenance problems. Secondly, there are two annoying bugs which were discovered in the testing phase, as described below.

The first bug is due to a Scribe limitation. The problem is that if the buffer file cannot be transferred before the time threshold specified in the *timeout* parameter, an "EAGAIN" error is generated:

```
[Mon Sep 19 16:07:53 2011] "Failed to send <8083064> messages
to remote scribe server <lxbsq1204:1463> error <EAGAIN (timed out)>"
```

This error causes the client to stop the transfer and to retry sending the entire buffer file. On the server side, data is continuously appended to the destination file in which the buffer should have transferred into (which is not deleted if a timeout occurs on the client side and is just left partly written). The result is a never ending loop of data being written, which causes the filling up of the hard disk space on both the clients (due the local buffer which cannot be emptied) and the server (due the continuos data flow). This behavior implies that timeouts have to be set really carefully. I discussed this problem with the Scribe developers and they confirmed me that this is a Scribe's known limitation [41].

The second bug is that Scribe tends to leave in memory data structures (created to handle the buffer replying) for further usage. If no limit for the allowed memory consumption is given, this behavior basically leads to a memory leaking. I discussed it with the Scribe developers and they pointed me to a patch which solves the problem by allowing to set a memory limit for the data structures used by Scribe[42].

Other minor problems are due the fact that Scribe had to be deployed very quickly: as already mentioned, it was the basic component on which to develop my work. This circumstance has led to some temporary solutions:

- the log files of the scribe daemon (/var/scribed.log) are quite big because of the logging of every message successfully sent (they are anyway rotated every five days(;

- the log file of the Scribe injector (/var/scribe-injector.log) is not rotated at all, and grows at a rate of about 30 Mb/month;
- the injector keeps monitoring every 60 seconds for new files instead of being automatically triggered.

A.2 Logprocessor daemon and Metrics Analysis Engine

The Metric Analysis Engine framework has to be used from the CASTOR's Logprocessor daemon (*logprocessord*) to compute the metrics online. The current (testing) implementation which I have deployed computes the metrics on top of a Scribe stream and saves the computed values on plain text files (Python Pickle format), to be accessed by the consumers later on.

The plugin oriented structure of the Logprocessor daemon requires a plugin for the input stream and a plugin for the output values, I therefore developed the*Scribe-src* plugin for handling the input stream (Scribe-src.py) and the *MetricsAnalysisEngine-dest* plugin for providing the output values (MetricsAnalysisEngine-dest.py). Those two plugins and the Metrics Analysis Engine framework (MetricsAnalysisEngine.py) are available under the CASTOR SVN repository[2].

The *Scribe-src* plugin just splits the incoming log messages into key-value pairs. The *MetricAnalysisEngine-dest* plugin is more complex ad make use of two threads: *analyzer* and *timer*. The *analyzer* is in charge of computing metrics through the Metrics Analysis Engine framework, while the *timer* checks for new data every 5 seconds and saves it in plain text files using the Python Pickle module. In future the project, instead of saving data, should provide through the output plugin an interface queryable by the consumers; leaving to the this plugin the only task to compute the metrics, no matters how the data is then handled.

The files and folders required by the Logprocessor daemon are:

- The config file: the configuration file for logprocessor daemon (see example A.6).

[2] `http://svnweb.cern.ch/world/wsvn/CASTOR/CASTOR2/trunk/logprocessor`.

- The metrics folder: the metrics, i.e. FilesRecalledStats.metric (see example A.5).
- The data folder: an empty folder to let the computed data be saved.

To install a Logprocessor daemon instance with the Scribe and Metrics Analysis Engine plugins (on lxbsq1204), one should follow the procedure reported below.

```
1: Checkout
   svn co svn+ssh://svn/reps/CASTOR/CASTOR2/trunk/logprocessor

2: Modify line 289 of LogginCommon.py to let it work with Scribe into:
   path += "%d%02d%02d/c2aggregated_00000" % (p.year, p.month, p.day))

3: Create a config file for logprocessor
   (you can use as template the testing one from /logprocessor/config)

4: Create directories logrprocessor/data and logprocessor/metrics

5: Put some metrics in logprocessor/metrics
   (you can just copy the testing ones from /logprocessor/metrics)

5: Run it with something like:
   /usr/bin/python -u ./logprocessord -p ComputeMetricsFromScribe
   -c config >> out.txt &
```

A logprocessor daemon is already running on lxbsq1204 as a proof of concept. It has been installed following this procedure in the /logprocessor directory and started via the command reported above, which has also been added to the /etc/rc.local init script for automatic startup at boot time.

```
<metric>
name: FilesRecalledStats
window: 300
conditions: LVL=="Info" and DAEMON=="recaller" and MSG=="File staged"
groupbykeys: INSTANCE
data: Counter(COUNT), Adder(FILESIZE)
handle_unordered: time_threshold
resolution: 5
</metric>
```

Listing A.5: Example of a metric file: FilesRecalledStats.metric.

A.3 Cockpit

The Cockpit Web interface proof of concept is installed in the directory /var/www/html/cockpit on lxbsq1204 and can be accessed from `http://lxbsq1204/cockpit`[3]. It reads the metric data from the files in the /logprocessord/data directory and generates the corresponding plots. The source code is available at `http://svnweb.cern.ch/world/wsvn/CASTOR/MONITORING/trunk/cockpit-web/`. The current implementation relies on Google Charts' Annotated Time Line , but this has to be replaced by a proper plotting engine, since using this method logarithmic scales are not available and the summary plot shown in the zoom bar is just useless and misleading with our data.

[3]Provided that the box is running the Apache web server, the mod_python is needed for the Cockpit.

```
# --- General program settings -------------------------------------

[main]
pid_file         = /var/run/logprocessord.normal.pid
log_file         = /var/log/castor/logprocessord.log
plugin_path      = /logprocessor

# --- Destination --------------------------------------------------

[dest-MetricsAnalysisEngine_dest]
module=MetricsAnalysisEngine_dest
class=ComputeMetrics

# To let it work with Scribe, change line 289 of LogginCommon into:
# path += "%d%02d%02d/c2aggregated_00000" % (p.year, p.month, p.day)

# --- Source -------------------------------------------------------

[source-Scribe_source]
module = Scribe_source
class  = ScribeLogFile

path     = /var/log/scribe/castor/c2aggregated/c2aggregated/
type     = pipe
dynfiles = true
seek     = true

# --- Processes ----------------------------------------------------

[process-ComputeMetricsFromScribe]
source=Scribe_source
destination=MetricsAnalysisEngine_dest

# The next one it's just an idea, it's not working unless you:
# - rewrite all the metrics with DLF keywords, or modify the DLF
#   plugin to provide the right ones
# - handle the nested dictionary of keyvalue pairs of the DLF plugin

[process-ComputeMetricsFromRsyslog]
source=DLF
destination=MetricsAnalysisEngine_dest
```

Listing A.6: Configuration file for the logprocessor daemon.

A.4 Hadoop

The Hadoop installation at CERN IT Department consists of 10 Data/ Worker nodes, a name node, and a MapReduce Job Tracker with a secondary name node. Each node is equipped with 8 cores, 24 GB of memory and 2 TB of disk space, for a HDFS total size of 20TB. The installed version is the Cloudera Hadoop 0.20.2-cdh3u0. Hadoop data (only on the data nodes) is stored in the /data01 and /data02 folders (1 TB each). The topology of the cluster is below reported:

```
lxbsq0929: Name node
lxbsq0930: Data/Worker node 1
lxbsq0931: Data/Worker node 2
lxbsq0932: Data/Worker node 3
lxbsq0933: Data/Worker node 4
lxbsq0934: Data/Worker node 5
lxbsq1105: MapReduce tracker and seconday name node
lxbsq1106: Data/Worker node 6
lxbsq1107: Data/Worker node 7
lxbsq1108: Data/Worker node 8
lxbsq1109: Data/Worker node 9
lxbsq1110: Data/Worker node 10
lxbsq1201: Client (Testing)
lxbsq1202: Client (Testing)
lxbsq1203: Client (Testing)
lxbsq1204: Client (Scribe, Cockpit, User client)
```

Hadoop provides two Web interfaces for monitoring the status of the jobs, the file system and in general the cluster:

- for monitoring the status of the jobs the Web interface is located at `http://lxbsq1105.cern.ch:50030/jobtracker.jsp`;
- for browsing the HDFS the Web interface is located at `http://lxbsq0929.cern.ch:50070/dfshealth.jsp`.

To add a user to the cluster, the actions required are listed below. The user "lamanc3" in the group "c3" has been taken as example.

```
1) Add the user to the prod/cluster/ahc/os/slc5/acls.tpl CDB profile.

2) On the box lxbsq1204 (the Hadoop client),
```

```
   create a fake afs home directory for the user, example:
    # mkdir -p /afs/cern.ch/user/l/lamanc3
    # chown laman:c3 /afs/cern.ch/user/l/lamanc3

3) Create Hadoop's temporary directory for the user:
    # mkdir /var/lib/hadoop-0.20/cache/lamanc3
    # chown lamanc3:c3 /var/lib/hadoop-0.20/cache/lamanc3

4) The user can now connect to lxbsq1204 and submit Hadoop jobs.
```

The HDFS is also accessible as a standard filesystem via FUSE, on every node of the cluster, from the /hfds folder. The command used to mount is:

```
hadoop-fuse-dfs dfs://lxbsq0929.cern.ch:8020 /hdfs &
```

The rpms required for loading Fuse's HDFS module has been addedd to lxbsq1204's cdb profile:

```
prod/cluster/ahc/roles/applications3.tpl:
    pkg_add("fuse");
    pkg_add("fuse-libs");
    pkg_add("hadoop-0.20-fuse","0.20.2+923.97-1","x86_64");
```

The command to mount Hadoop via FUSE has been addedd to /etc/rc.local on lxbsq1204. It has to be noted that the chunk size that applies when copying files to Hadoop using this access mode is fixed to 64 Mb.

Bibliography

[1] The CERN public website, `http://public.web.cern.ch/` [viewed 14/08/2012].

[2] ATLAS Collaboration, The ATLAS Experiment at the CERN Large Hadron Collider, JINST 3, S08003 (2008).

[3] CMS Collaboration, The CMS experiment at the CERN LHC, JINST 3, S08004 (2008).

[4] LHCb Collaboration, The LHCb Detector at the LHC, JINST 3, S08005 (2008).

[5] ALICE Collaboration, The ALICE experiment at the CERN LHC, JINST 3, S08002 (2008).

[6] ATLAS Collaboration, The Trigger for Early Running, ch. in Expected performance of the ATLAS experiment: detector, trigger and physics, pp. 550-564, CERN-OPEN-2008-020 (2008).

[7] L. Tompkins on behalf of the ATLAS Collaboration, Performance of the ATLAS Minimum BiasTrigger in pp collisions at the LHC, Proceedings of HCP 2010, Toronto, 30 September 2010, ATL-DAQ-PROC-2010-033 [arXiv:1009.6133v1].

[8] W. Lampl et al. Calorimeter Clustering Algorithms: Description and Performance, ATL-LARG-PUB-2008-002 (2008).

[9] S. Hassani, L. Chevalier, E. Lancon, J. F. Laporte, R. Nicolaidou and A. Ouraou, A muon identification and combined reconstruction procedure for the ATLAS detector at the LHC using the (MUONBOY, STACO, MuTag) reconstruction packages, Nucl. Instrum. Meth. A572, 77 (2007).

[10] T. Lagouri et al., A muon identification and combined reconstruction procedure for the ATLAS detector at the LHC at CERN, IEEE Trans. Nucl. Sci. 51, 3030-3033 (2004).

[11] G. Duckek (ed.), et al., ATLAS Computing Technical Design Report, CERN-LHCC-2005-002, ISBN 92-9083-250-9, 20 June 2005 (also available on http://cdsweb.cern.ch/record/837738).

[12] W. Bhimji, et al., The ATLAS ROOT-based data formats: recent improvements and performance measurements, Computing in High Energy and Nuclear Physics 2012, New York, NY, 21 - 25 May 2012 (also available as ATL-SOFT-PROC-2012-020, 14 May 2012, on http://cdsweb.cern.ch/record/1448601).

[13] The WLCG website, http://lcg.web.cern.ch/lcg/public [viewed 14/08/2012].

[14] I. Bird, et al., LHC Computing Grid Technical Design Report, CERN-LHCC-2005-024, LCG-TDR-001, ISBN 92-9083-253-3, 20 June 2005 (also available on http://cdsweb.cern.ch/record/840543/).

[15] G. Lo Presti, et al., Castor: A distributed storage resource facility for high performance data processing at cern. Proc. 24th IEEE Conf. on Mass Storage Systems and Technologies, 2007.

[16] A. Szalay, et. at., The Importance of Data Locality in Distributed Computing Applications, NSF Workflow Workshop 2006.

[17] Ian Foster Wikipedia page, http://en.wikipedia.org/wiki/Ian_Foster [viewed 14/08/2012].

[18] Apache Hadoop MapReduce ,http://hadoop.apache.org/mapreduce [viewed 14/08/2012].

[19] Apache Hadoop Wikipedia page,http://en.wikipedia.org/wiki/Apache_Hadoop [viewed 14/08/2012].

[20] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Communications of the ACM - 50th anniversary issue: 1958 - 2008, Volume 51 Issue 1, January 2008, Pages 107-113, ACM New York, NY.

[21] S. Ghemawat, et al., The Google File System, ACM SIGOPS Operating Systems Review - SOSP '03, Volume 37 Issue 5, December 2003.

[22] Rsyslog website, http://www.rsyslog.com/ [viewed 28/02/2013].

[23] Oracle website, http://www.oracle.com/technetwork/database/features/plsql/index.html [viewed 28/02/2013].

[24] Babik, Marian, et al., LEMON - LHC Era Monitoring for Large-Scale Infrastructures, J. Phys.: Conf. Ser. 331 Part 5, 2011.

[25] The Scribe project website, `https://github.com/facebook/scribe/wiki` [viewed 28/02/2013].

[26] Hadoop Streaming, `http://hadoop.apache.org/common/docs/r0.20.2/streaming.html` [viewed 13/08/2012].

[27] Package org.apache.hadoop.mapred.pipes, `http://hadoop.apache.org/common/docs/current/api\/org/apache/hadoop/mapred/pipes/package-summary.html` [viewed 13/08/2012].

[28] D. Costanzo, et al., Metadata for ATLAS, ATL-GEN-PUB-2007-01, 05 April 2007.

[29] MapR Technologies,Inc., MapR's Direct Access NFS vs Hadoop FUSE, Technical Brief, 23 August 2011 (available on `http://www.mapr.com/Download-document/9-NFS-Technical-Brief`).

[30] M. Zaharia, et al., Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling, EuroSys '10 Proceedings of the 5th European conference on Computer systems, Pages 265-278, ACM New York, NY.

[31] ATLAS Udine group website, `http://www.fisica.uniud.it/ATLAS/` [viewed 14/08/2012].

[32] M. Pinamonti, et al., *Measurement of the top-antitop production cross-section with the ATLAS experiment at the LHC* (PhD), CERN-THESIS-2012-082, 3 April 2012 (available on `https://cds.cern.ch/record/1460132/`).

[33] F. Abe et al., CDF Collaboration, *Observation of Top Quark Production in $\bar{p}p$ Collisions with the Collider Detector at Fermilab*, Phys. Rev. Lett. 74, 2626–2631 (1995).

[34] ATLAS Collaboration, Measurement of the top quark-pair production cross section with ATLAS in pp collisions at 7 TeV, EPJC 71, 1577 (2011).

[35] CMS Collaboration, First Measurement of the Cross Section for Top-Quark Pair Production in Proton-Proton Collisions at sqrt(s)=7 TeV, Phys. Lett. B695, 424-443 (2010), arXiv:1010.5994v1 [hep-ex].

[36] M. Aliev et al., HATHOR HAdronic Top and Heavy quarks crOss section calculatoR, Comput. Phys. Commun. **182** 1034 (2011), arXiv:1007.1327v1 [hep-ph].

[37] The ATLAS Collaboration, et al., Measurement of the top quark pair cross-section with ATLAS in pp collisions at $\sqrt{s} = 7$ TeV in the single-lepton channel using b-tagging, ATLAS-CONF-2011-035, 21 March 2011 (available on `http://cdsweb.cern.ch/record/1337785/`).

[38] The ATLAS Collaboration, et al., Measurement of the ttbar production cross-section in pp collisions at $\sqrt{s} = 7$ TeV using kinematic information of lepton+jets events, ATLAS-CONF-2011-121, 22 August 2011 (available on `https://cdsweb.cern.ch/record/1376413/`).

[39] S. A. Russo, A top quark analysis based on Hadoop (or: how to run ROOT HEP analyses on a Hadoop cluster with a MapReduce model), ATLAS Software and Computing Week, CERN, Geneva (Switzerland) 11-15/06/2012.

[40] R. Garcia Leiva, et al., Quattor: Tools and Techniques for the Configuration, Installation and Management of Large-Scale Grid Computing Fabrics, Journal of Grid Computing Volume 2, Number 4 (2004).

[41] Scribe Google group thread, `http://groups.google.com/group/scribe-server/browse_thread/thread/da26e4754d23b2cc/7c5d2cd67cd32a4e`) [viewed 14/08/2012].

[42] Scribe Google group thread, `http://groups.google.com/group/scribe-server/browse_thread/thread/9f57eaa034ab14a/d7783fae968aeee7` [viewed 14/08/2012].