An opportunistic distributed computing project for scientific applications

Author: Matteo Concas

Supervisor: Prof. Massimo Masera

Co-supervisors: Dott. Dario Berzano, Dott. Stefano Bagnasco

Reviewer: Prof. Stefano Spataro

November 8, 2016
Abstract

The computing power of most modern commodity computers is far from being fully exploited by standard usage patterns. This is true also for the research group of the Dipartimento di Fisica di Torino involved in the ALICE (A Large Hadron Collider) experiment, where I carried out this thesis project. The work I present describes the development and setup of a virtual computing cluster based on Docker containers used as worker nodes. The facility is based on Plancton: a lightweight fire-and-forget background service that spawns and controls a local pool of Docker containers on a host with free resources by constantly monitoring its CPU utilisation. Plancton is designed to release the resources allocated opportunistically whenever another demanding task is run by the host user, according to configurable thresholds: this is attained by killing a number of running containers. In ALICE, such a service is originally intended to provide resources to physicists that need to test their code either for distributed analyses or simulations. Thus, in R&D for example, they can benefit of an environment where quickly run custom jobs and access results without bothering dedicated services (e.g. WLCG, Worldwide LHC Computing Grid). Nevertheless they can run small simulations or light analyses on skimmed/partial datasets.

The resources comprising the facility are a collection of heterogeneous non-dedicated Linux hosts ideally inside the same local network, with no guaranteed network bandwidth, made available by members of a collaboration or institute. The user has agreed to donate its spare CPU cycles and remains the administrator of the involved host. Since the system is based on Docker containers performance isolation and security are guaranteed through sandboxing. Using a thin virtualisation layer such as Docker has the effect of having containers that are started almost instantly upon request. I will show how fast startup and disposal of containers finally enables me to implement the formation of the opportunistic cluster in a headless fashion, where my containers are mere pilots. As an example I am running pilot HTCondor (High Throughput Computing) containers automatically joining a given cluster and terminating right after executing a job or in a short while if no new job is available. Software is provided through CVMFS (Cern Virtual Machine FileSystem) and Parrot, making the execution environment suitable for HEP (High Energy Physics) jobs like ALICE ones. Finally, I will show how the uncomplicated approach of Plancton to containers deployment makes it suitable for setting up dedicated computing facilities too, provided that the underlying use case is sufficiently simple.
La potenza di calcolo dei computer più moderni è lungi dall’essere pienamente sfruttata dai modi di utilizzo standard. Questo è vero anche per il gruppo di ricerca del Dipartimento di Fisica di Torino coinvolti nell’esperimento ALICE (A Large Ion Collider Experiment), dove ho svolto questo progetto di tesi. Il lavoro che presento descrive lo sviluppo e la messa a punto di un cluster di calcolo virtuale basato su container Docker utilizzati come nodi di calcolo. L’impianto si basa su Plancton: un servizio da me scritto che gira in background e genera e controlla un’insieme locale di container Docker su un host con risorse libere, monitorando costantemente l’utilizzo della CPU. Plancton è stato progettato per liberare le risorse allocate opportunisticamente ogni volta che un altro compito impegnativo è richiesto dall’utente proprietario della macchina, in base a soglie configurabili: questo si ottiene spegnendo, se necessario, un numero sufficiente di container in esecuzione. In ALICE tale servizio è originariamente destinato a fornire risorse per i fisici che hanno bisogno di testare il loro codice distribuito di analisi o simulazione. Così, nella fase di sviluppo ad esempio, possono beneficiare di un ambiente dove rapidamente testare i propri programmi senza disturbare servizi dedicati (ad esempio WLCG, Worldwide LHC Computing Grid). Tuttavia questo servizio si presta solo per piccole simulazioni o analisi leggere su dataset scremati/parziali.

Le risorse che compongono la struttura sono una raccolta di eterogenea di macchine Linux non dedicate ospitate idealmente all’interno della stessa rete locale, senza alcuna larghezza di banda garantita, messe a disposizione da parte dei membri di una collaborazione o istituto. L’utente decide di donare i suoi cicli di CPU di avanzo, pur rimanendo l’amministratore della macchina interessata. Dal momento che il sistema si basa su Docker, l’isolamento, le prestazioni e la sicurezza sono garantiti attraverso “sandboxing” delle applicazioni in container. Utilizzando uno strato sottile di virtualizzazione come Docker si ha l’effetto di avere container che vengono avviati quasi istantaneamente alla richiesta. Mostrerà come l’agile avvio e smaltimento di container permetta di implementare cluster opportunistici in un modo non dipendente da una gerarchia, dove i container sono semplici "pilot". Ad esempio si possono far funzionare pilot container HTCondor (High Throughput Computing) in modo da entrare automaticamente un determinato gruppo di lavoro e chiudersi subito dopo l’esecuzione di un lavoro o dopo breve tempo, se nessun nuovo lavoro è disponibile. Il software per l’esperimento viene fornito attraverso CVMFS (Cern Virtual Machine File System) e Parrot, rendendo l’ambiente di esecuzione adatto per i lavori in HEP (High Energy Physics), come quelli di ALICE. Infine, vi mostrerò come il semplice approccio di Plancton per la distribuzione di container lo renda adatto anche per la creazione di strutture di calcolo dedicate, a condizione che il caso d’uso di fondo sia sufficientemente semplice.
Contents

Abstract iii

Sommario v

Introduction 1

1 Overview of the ALICE Experiment 3

1.1 The Large Hadron Collider at CERN . . . . . . . . . . . . . . . 3
1.2 The ALICE Experiment . . . . . . . . . . . . . . . . . . . . . . 4
1.3 The Physics of ALICE . . . . . . . . . . . . . . . . . . . . . . . 5

1.3.1 A brief introduction to the strong interaction: QCD . 5
1.3.2 QGP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.3.3 Heavy-ion Collisions . . . . . . . . . . . . . . . . . . . . 9
1.3.4 Experimental observables . . . . . . . . . . . . . . . . . . 11

1.4 The ALICE experimental setup . . . . . . . . . . . . . . . . . . 17

1.4.1 Magnet . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
1.4.2 Inner Tracking System . . . . . . . . . . . . . . . . . . . 20
1.4.3 Time Projection Chamber . . . . . . . . . . . . . . . . . . 20

Particle identification . . . . . . . . . . . . . . . . . . . 21
1.4.4 Time Of Flight . . . . . . . . . . . . . . . . . . . . . . . 22
1.4.5 Transition Radiation Detector . . . . . . . . . . . . . . . 22
1.4.6 High-Momentum Particle Identification Detector . . . . 22
1.4.7 Photon Spectrometer . . . . . . . . . . . . . . . . . . . . 22
1.4.8 Electromagnetic Calorimeter . . . . . . . . . . . . . . . . 23
1.4.9 ALICE Cosmic Ray Detector . . . . . . . . . . . . . . . . 23
1.4.10 Forward Muon Spectrometer . . . . . . . . . . . . . . . . 23
1.4.11 Forward rapidity section . . . . . . . . . . . . . . . . . . . 24
1.4.12 ALICE detector Coordinate System . . . . . . . . . . . 24
1.4.13 Trigger system . . . . . . . . . . . . . . . . . . . . . . 24
1.4.14 Data Acquisition System . . . . . . . . . . . . . . . . . 25

1.5 The ALICE Software framework . . . . . . . . . . . . . . . . . 26

1.5.1 The ROOT framework . . . . . . . . . . . . . . . . . . . 26
1.5.2 AliRoot . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
Simulation software ........................................... 27
Reconstruction software ........................................ 29
1.5.3 AliEn .......................................................... 29

2 Computing in High Energy Physics 31
2.1 Computing challenges at LHC .............................. 31
2.2 The Worldwide LHC Computing Grid .................... 31
  2.2.1 A tiered architecture ...................................... 32
  2.2.2 High Throughput Computing ............................. 34
2.3 Computing in ALICE ........................................... 35
  2.3.1 Virtualisation and Cloud Computing .................. 35
  2.3.2 Cloud Computing in ALICE ............................. 38
  2.3.3 The O² program for the upgrade ....................... 38
2.4 Volunteer computing in High Energy Physics .......... 40
  2.4.1 State of the art ............................................ 40
2.5 Opportunistic distributed computing ..................... 41
  2.5.1 A volunteer computing project for ALICE Torino ... 43

3 An opportunistic computing service based on Docker 45
3.1 Linux Containers .............................................. 45
  3.1.1 Behind the scene: cgroups and namespaces .......... 46
  3.1.2 A chroot "on steroids" .................................... 47
  3.1.3 Linux Capabilities ........................................ 47
3.2 The Docker engine ............................................ 48
  3.2.1 The Docker command line interface .................. 48
  3.2.2 A "Layered" approach to container filesystem ...... 50
  3.2.3 The Docker APIs ......................................... 51
3.3 The Plancton daemon ......................................... 53
  3.3.1 An automated Python script for container scheduling 53
  3.3.2 Design guidelines ....................................... 53
  3.3.3 Plancton configuration .................................. 57
  3.3.4 Plancton lifecycle ....................................... 58
  3.3.5 Resource monitoring and restraint system in Plancton 61
  3.3.6 Plancton policies of scheduling ...................... 63
  3.3.7 Plancton in a dedicated scenario .................... 65

4 Opportunistic computing with Plancton 67
4.1 Disposable pilots as worker nodes ....................... 67
4.2 Software provisioning with CVMFS ....................... 69
  4.2.1 Job orchestration with HTCondor .................... 70
4.3 A volunteer computing facility at ALICE Torino ........ 71
  4.3.1 Access CVMFS repositories from containers .......... 71
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Experiments at Large Hadron Collider (LHC)</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Contributions to the gluon self-interaction</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Measurements of $\alpha_s(Q^2)$ coupling running</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>QCD Phase Diagram</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>LQCD plot of the energy density $\epsilon/T^4$ against $T/T_c$</td>
<td>8</td>
</tr>
<tr>
<td>1.6</td>
<td>Spacetime evolution of a heavy-ion Collision</td>
<td>10</td>
</tr>
<tr>
<td>1.7</td>
<td>$\frac{d^2}{dN_{ch}d\eta} (dN_{part})$ in Pb-Pb and Au-Au as a function of $\sqrt{s_{NN}}$</td>
<td>12</td>
</tr>
<tr>
<td>1.8</td>
<td>Primary $\pi^-$ and $k^- p_T$ spectra in Pb-Pb at $\sqrt{s_{NN}} = 2.76TeV$</td>
<td>14</td>
</tr>
<tr>
<td>1.9</td>
<td>Anisotropic flow schema and the ALICE measurement in Pb-Pb at $\sqrt{s_{NN}} = 5.02(2.76)TeV$</td>
<td>15</td>
</tr>
<tr>
<td>1.10</td>
<td>Example of an imbalanced dijet in Pb-Pb @ $\sqrt{s_{NN}} = 2.76TeV$</td>
<td>16</td>
</tr>
<tr>
<td>1.11</td>
<td>The ALICE Experimental setup</td>
<td>19</td>
</tr>
<tr>
<td>1.12</td>
<td>ALICE ITS and TPC</td>
<td>21</td>
</tr>
<tr>
<td>1.13</td>
<td>Separating power for $\pi^- - k$ and $k^- p$</td>
<td>21</td>
</tr>
<tr>
<td>1.14</td>
<td>ROOT framework typical usage</td>
<td>26</td>
</tr>
<tr>
<td>1.15</td>
<td>Visualisation of a pp event @13 TeV</td>
<td>27</td>
</tr>
<tr>
<td>1.16</td>
<td>Schema of the AliRoot structure and interfaces</td>
<td>28</td>
</tr>
<tr>
<td>1.17</td>
<td>Data organisation in in ALICE simulations</td>
<td>28</td>
</tr>
<tr>
<td>1.18</td>
<td>Schema of the AliRoot computing facility</td>
<td>30</td>
</tr>
<tr>
<td>2.1</td>
<td>L-1 Rate vs Event Size in particle Physics experiments (2012)</td>
<td>32</td>
</tr>
<tr>
<td>2.2</td>
<td>Tiers Hierarchy for the Grid</td>
<td>33</td>
</tr>
<tr>
<td>2.3</td>
<td>Relative share of workflows in ALICE (Run 1)</td>
<td>36</td>
</tr>
<tr>
<td>2.4</td>
<td>Virtualisation layers in a Cloud computing scenario</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>A Container-based virtualisation architecture</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>Host environment VS Container Environment</td>
<td>47</td>
</tr>
<tr>
<td>3.3</td>
<td>Docker Architecture</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>OverlayFS Schematic structure</td>
<td>50</td>
</tr>
<tr>
<td>3.5</td>
<td>Docker summary scheme</td>
<td>52</td>
</tr>
<tr>
<td>3.6</td>
<td>Plancton daemon main loop</td>
<td>59</td>
</tr>
<tr>
<td>3.7</td>
<td>htop instance with fixed constraints.</td>
<td>63</td>
</tr>
<tr>
<td>3.8</td>
<td>Flowchart of the policy of scheduling in Plancton</td>
<td>64</td>
</tr>
</tbody>
</table>
4.1 Typical service based on Plancton ........................................... 69
4.2 HTCondor services running inside a Docker container ................. 75
4.3 Overview of volunteer setup on single host ................................ 76
4.4 Architecture of the volunteer cluster in Torino ........................... 77
4.5 Schema of the AliEn Work Queue Integration ............................... 83
4.6 Overview of the AliEn Work Queue virtual host ............................ 84

5.1 Condor Worker nodes as Linux containers ................................. 88
5.2 Volunteer containers as seen by Docker ..................................... 89
5.3 Job efficiencies of Grid sites at Organisation Européenne pour la Recherche Nucléaire (CERN) Tier-0 .......................... 91
5.4 Job running as a function of time .......................................... 92
5.5 Jobs done by High-level Trigger Development (HLT-DEV) in 21 days ............................................................ 93
List of Tables

1.1 LHC running conditions history . . . . . . . . . . . . . . . . . . 3
1.2 ITS layer quotas . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

3.1 Main commands available in the definition of a Dockerfile . . . 51
3.2 Setup of the test container . . . . . . . . . . . . . . . . . . . . . . 63

4.1 Average elapsed spawning time for Docker containers . . . . . . 68
4.2 I/O latency comparison in Docker container and Parrot . . . . 74

5.1 Values corresponding to the HLT-DEV Job efficiency in Fig 5.3 . 91
5.2 Values corresponding to the number of running jobs at HLT-
DEV cluster as a function of time in Fig. 5.4 . . . . . . . . . . . . 92
Alla mia famiglia, in tutte le sue variazioni sul tema
Introduction

In the years 2018-2019 the LHC at CERN will face its second one-year-long shutdown, called Long Shutdown 2 (LS2). During this time the accelerator will be maintained and upgraded, in view of the later data taking period, the RUN 3, when the luminosity delivered in heavy-ion collisions will be enhanced up to $6 \times 10^{27} \text{cm}^{-2}\text{s}^{-1}$ in Pb-Pb, corresponding to a rate of 50 kHz of interaction rate. The four major experiments: A Large Ion Collider Experiment (ALICE), A Toroidal LHC ApparatuS (ATLAS), Compact Muon Solenoid (CMS), Large Hadron Collider beauty (LHCb) will benefit from this enhancement. In particular, ALICE will perform a huge upgrade almost in every aspect, in order to benefit from the new regime and to integrate a luminosity of 13 nb$^{-1}$ in Pb-Pb collisions. This upgrade addresses precise measurements of heavy flavour hadrons, low-momentum quarkonia and low mass di-leptons. These are physics probes, characterised by a very small signal-to-background ratio, thus they require a very large statistic to be analysed. These upgrades will create great challenges for the online and offline computing systems used by the experiments. A data throughput greater than 1TB/s has been estimated for Pb-Pb events, which is an enhancement of two orders of magnitude compared with the current stream. ALICE in particular, is planning to create an unique framework for online/offline computing, called $O^2$ (Online-Offline).

New analysis paradigms and solutions are being investigated, in order to permit an efficient data analysis on those new huge dataset coming from the upgrade. Old and consolidated virtualisation solutions based on virtual machines and cloud infrastructures will be re-discussed taking into account brand new technologies. Moreover every computing resource will become useful in order to address such a great computing challenge, both in online and offline scenario.

This thesis work intends to explore the range of applicability of the Linux containers technology in high energy physics computations, and the way to implement such a technique. A Linux container is a lightweight operating-system level virtualisation method for running multiple different and isolated instances of a Linux system on the same host leaning on the same underlying Linux kernel. Linux containers are nowadays widely used in many context as a way to provide services: as databases, internet services, software build systems, sandboxing, etc. The idea is to expand their fields of applicability to
the energy physics data reconstructions, simulations or analyses.

It was developed with the main purpose to create a volunteer computing solution to be used inside the ALICE group of Torino. Volunteer computing both with opportunistic computing are two evergreen recursive themes in the research field. Those two concepts will be deeper described, defined and discussed inside this document.
Overview of the ALICE Experiment

1.1 The Large Hadron Collider at CERN

The CERN, founded in 1954, is situated near the city of Geneva, straddling the Franco-Swiss border. There is hosted the LHC, the largest and most powerful circular particle accelerator in the world, with a circumference of $\sim 27$Km. It first started up on 10 September 2008 and it will be in operation until 2035 and later, running with proton and lead ion beams, colliding $p-p$, $p-Pb$ or $Pb-Pb$. It is capable to deliver centre-of-mass energies ($\sqrt{s}$) of 14TeV in $p-p$ events and up to 5.5TeV in $Pb-Pb$.

<table>
<thead>
<tr>
<th>Year</th>
<th>pp</th>
<th>p-Pb</th>
<th>Pb-Pb</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>900 GeV</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2010</td>
<td>7 TeV</td>
<td>-</td>
<td>2.76 TeV</td>
</tr>
<tr>
<td>2011</td>
<td>7 TeV and 2.76 TeV</td>
<td>-</td>
<td>2.76 TeV</td>
</tr>
<tr>
<td>2012</td>
<td>8 TeV</td>
<td>5.02 TeV</td>
<td>-</td>
</tr>
<tr>
<td>2013</td>
<td>2.76 TeV</td>
<td>5.02 TeV</td>
<td>-</td>
</tr>
</tbody>
</table>

February 2013 - May 2015 LS1

<table>
<thead>
<tr>
<th>Year</th>
<th>pp</th>
<th>p-Pb</th>
<th>Pb-Pb</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>13 TeV</td>
<td>-</td>
<td>8 TeV</td>
</tr>
<tr>
<td>2016 current</td>
<td>13 TeV</td>
<td>5.02 TeV and 8 TeV</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1.1: LHC running conditions history
Chapter 1. Overview of the ALICE Experiment

There are in total seven experiments at the LHC located in four main sites around the circumference of the accelerator sitting underground in huge caverns, where the two counter-rotating particle beams are brought into collision. They are ALICE, ATLAS, CMS, LHCb, The Large Hadron Collider forward (LHCf), The Monopole & Exotics Detector at the LHC (MoEDAL) and Total, elastic and diffractive cross-section measurement (TOTEM). ATLAS and CMS are designed to study the widest range of phenomena and they are designed independently for cross-confirmation of any new discovery. ALICE and LHCb aim is to study specific phenomena, the remaining three smaller experiments focus either on the physics of forward-particles or on the hypothetical magnetic monopole.\footnote{A satisfactory round up on LHC experiments is available at: \cite{9}}

![Diagram of LHC experiments locations]

\textbf{Figure 1.1}: Overview of the experiments locations at the LHC. The two beams collide in four points, the sites where the ALICE, CMS, ATLAS, LHCb experiments are located.

1.2 The ALICE Experiment

The ALICE acronym stands for A Large Ion Collider Experiment. It was designed to study the nuclear matter in extreme conditions of temperature and density. At those conditions is expected a transition from the state of strongly-interacting matter to Quark Gluon Plasma (QGP). (Quark Gluon Plasma)
According to the many theoretical models\footnote{One for all: \cite{37}} formulated, such conditions will reproduce the earliest stages of the universe. One of the goals of current experiments like \textsc{alice} is to determine which is the \textit{best-fitting} one.

\section*{1.3 The Physics of \textsc{alice}}

\subsection*{1.3.1 A brief introduction to the strong interaction: QCD}

The \textbf{Quantum Chromodynamics (QCD)} is the part of the theoretical Standard Model framework of the fundamental interactions that describes the strong interaction. The strong interaction is one of the four known fundamental interactions among electromagnetism, gravitation and weak interaction. It assigns to quarks a "coloured charge", either red ($R$), green ($G$) or blue ($B$), with their respective anti-charges, and asserts that stable states exist in colourless combinations: $RGB$ (baryons) and $R\bar{R}/G\bar{G}/B\bar{B}$ (mesons). According to the quantum field theory description, the coloured interaction involves quarks and mediator bosons. The explicit symmetry group regulating the dynamics is $SU(3)$, hence there are 8 gauge bosons, the \textit{gluons}. The symmetry underlying the Lagrangian ($\mathcal{L}_{QCD}$) is non-abelian and this adds complexity with respect to the QED. This appears neatly in the definition of the tensor $F^{\mu\nu}_a$

\begin{equation}
F^{\mu\nu}_a = \partial^\mu A^\nu_a - \partial^\nu A^\mu_a - g \sum_{b,c=1}^8 f_{abc} A^\mu_b A^\nu_c
\end{equation}

where $A^\mu_a$ is the gluon field. The last term, which does not appear in the definition for the photon tensor in QED ($F^{\mu\nu}$), brings additional contributions (compared with the QED formulation for photons) in the kinetic Lagrangian ($-\frac{1}{4} F^{\mu\nu}_a F^{\mu\nu}_a$), generating

\begin{equation}
\partial_\mu A^\mu_a A^{\nu c} \quad \text{and} \quad A^b_\mu A^c_\nu A^{\mu b} A^{\nu c}
\end{equation}

that describe the self-interactions for gauge bosons\footnote{Find a derivation here: \cite{33}}. Those kind of self interactions are not observed in QED at the first order in perturbation theory.

The self interacting terms in the gluon Lagrangian introduce \textit{3-gluons} and \textit{4-gluons} vertices that lead to one loop corrections to the gluon propagation, generating to the so-called \textit{anti-screening} in colour interaction. This implies a characteristic \textbf{QCD} running coupling described by the formula (\ref{eq:alpha_s})

\begin{equation}
\alpha_S(Q^2) = \frac{\alpha(\mu^2)}{1 + \frac{3\pi - 2n_f}{12\pi} \ln \frac{Q^2}{\mu^2}}
\end{equation}
where \( n_f \) is the number of flavours \( \mu \) is the renormalisation scale and \( Q^2 \) is the transferred momentum. That is the coupling constant \( \alpha_s \) is strongly scale-dependent, as shown in figure 1.3.

Therefore it is possible to describe two limits: the asymptotic freedom and the confinement.

- **Asymptotic freedom** \( \alpha_s \ll 1 \): At high energy and high transferred momentum, the strong coupling constant \( \alpha_s \) is small enough to allow us to adopt a perturbative approach (pQCD). The pQCD standard procedure in QFT to calculate the transition element of the matrix \( S \) is used for example to evaluate cross sections. For \( Q \sim 100 \text{ GeV} - 1 \text{ TeV} \), \( \alpha_s \), in fact is about 0.1. At such energies the strong interaction vanishes resulting in a decrease of the energy needed to separate colour charges and the consequent release of quarks and gluons from their bound states.

Since high \( Q^2 \) is a synonym for short distances or high temperatures it is possible, in principle, by compressing or heating QCD matter, to melt
1.3. The Physics of ALICE

hadronic structures. The resulting state of this matter is a plasma of quarks and gluons, the QGP.

- **Confinement**: At low energy the perturbative approach is not usable. Instead numerical non-perturbative methods like the lattice QCD (LQCD) or QCD-inspired effective models must be used. In this energy range colour charges are captured and neutralised into colourless states. The strong interaction becomes really intense due to the proliferation of the high-order gluonic loops and the consequent anti-screening of coloured charge. This feature is due to the non-abelianity of the SU(3)$_C$ symmetry group. The energy needed to separate colour charges increases with the distance itself, making the system freeze in a configuration of stable equilibrium and leading it to a confinement state. Thus "confinement" indicates the absence of free colour charges.

1.3.2 QGP

As previously mentioned, the QGP is an exotic state for the nuclear matter, theoretically reachable in particular scenario. Such conditions occurred in past during the "Big Bang", when the universe itself was born, where there was a regime of almost infinite energy and temperature. Also they still occur in the current universe in massive astronomical objects like the inner core of neutron stars due to the extreme compression of nuclear matter.

The QCD Phase diagram (fig. 1.4) of strongly interacting matter describes thermodynamically the states of nuclear matter as a function of the temperature $T$ and the baryon-chemical potential, $\mu_b$. This potential represents the net baryonic density and ensures its total conservation. According to

\[\text{Figure 1.4: Phase diagram for the strongly interacting matter}\]
lattice QCD calculations, for large values of the baryon-chemical potential, a first-order transition between hadronic matter and QGP should occur, whereas a crossover is expected at lower $\mu_b$. Moreover, moving along the two axes at the two most meaningful extremes one finds two exotic behaviours:

- $T \gg 0$ and $\mu_b \approx 0$ → strongly interacting Quark Gluon Plasma ($sQGP$).
- $T \approx 0$ and $\mu_b \gg 0$ → coloured superconductor.

It is also possible to investigate the phase diagram with theoretical models, the preponderance of non-pertubative effects leave the lattice QCD as the only effective approach in providing a valid description. Calculations of the energy density trend ($\epsilon$) as a function of temperature ($T/T_c$), is depicted in fig. (1.5).

During the earliest instants in the Universe evolution, the temperature was too high for elementary partons to combine into bound states, the Universe had first to cool down in order to form hadrons and then combine them together. The temperature had to decrease to low values ($T \sim 3000K$), confining electrons and nuclei into neutral atoms to allow them to become transparent to the electromagnetic radiation. This opacity to radiation, before to the decoupling of photons, limits the maximum time one can go back with astronomical observations, thus prohibiting any direct investigation of the early stages of the evolution.
1.3.3 Heavy-ion Collisions

One of the aims in doing heavy-ion experiments is to investigate the opaque region before the photon decoupling, reproducing the QGP state in the laboratory. To recreate such a state in heavy-ion collisions, the experiments must reach some working conditions:

- The temperature has to be sufficiently high to deconfine the system.
- The system has to be in thermal equilibrium in order to adopt a thermodynamical model to describe it. Such equilibrium is reached through particle interactions, a sufficiently dense and interacting medium has to be produced during the collisions.
- The scale of the system must be larger than the range of the strong interaction ($\sim 1\, fm$), in order to describe the evolution of the system with macroscopic observables within a thermodynamic model.

The aim in heavy-ion physics is to study and understand how collective phenomena and macroscopic properties, involving many degrees of freedom, emerge from the microscopic laws of elementary particle physics. In particular, heavy-ion physics addresses these questions in the sector of strong interactions by studying nuclear matter under extreme density and temperature conditions.

Heavy-ion collisions satisfy the previous clauses, in fact the multiple collisions between nucleons, belonging to colliding nuclei, generate a high energy density region around the interaction point. The number of binary interactions between nucleons exceeds by far the number of available nucleons in the initial state.$^5$

The evolution in spacetime is a complex process and can be summarised in few phases. There are two regimes, depending on the centre-of-mass energy, usually expressed in energy per nucleon pair ($\sqrt{s_{NN}}$). If $\sqrt{s_{NN}} \simeq 10\, GeV$, the net effect is that the nucleons are stopped in the so-called stopping-region. This is called the stopping regime. The more the centre-of-mass energy increases, the less nucleons remain "trapped" in the collision region. At high $\sqrt{s_{NN}}$ almost no nucleon is left in the interaction region (that is the net average baryon density is close to zero). This is called Bjorken regime or transparent regime. Assuming to satisfy the Bjorken condition the evolution of the system can be described with the following phases (see Fig. 1.6).

- **Collision** The collision takes very short times, for example with the energy conditions at Relativistic Heavy Ion Collider (RHIC) ($\sqrt{s_{NN}} = 5\, GeV$). The average number of binary interactions in heavy-ion collision varies between 1000 and 1500.
Chapter 1. Overview of the ALICE Experiment

Figure 1.6: Space-time evolution of a heavy-ion collision.

200 GeV/c) the crossing time is

$$\tau_{\text{coll}} = \frac{2R}{\gamma} \sim 0.1 \text{fm}/c$$

- **Pre-equilibrium** ($\tau_f > \tau_{\text{coll}}$): the fireball takes shape. Quarks and gluons are yielded by initial partons scatterings. At RHIC the energy density reached (evaluated within the Bjorken model) is $\epsilon_{Bj} \sim 5.4 \text{GeV}/\text{fm}^3$ while at LHC it is $\sim 15 \text{GeV}/\text{fm}^3$.

- **QGP** ($0.6 \leq \tau_{eq} \leq 1 \text{fm}/c$ at RHIC). The quark and gluon gas evolves in time and reaches the thermal equilibrium. The QGP is formed and starts expanding.

- **Mixed phase**: the QGP expanding, starts to convert into a hadron gas.

- **Hadronization** ($\tau \simeq 20 \text{fm}/c$): as far as the system keeps expanding, its temperature decreases until quarks and gluons are again confined in hadrons.

- **Chemical freeze-out**: below a given temperature the hadrons cease their interactions. First the inelastic ones, reaching a chemical freeze-out. The chemical abundances are fixed in this phase.

- **Kinetic freeze-out**: the elastic interactions stop as well, defining the momentum spectra of every species.
1.3.4 Experimental observables

The detectors of an experiment observe the final state particles coming from the interaction zone carrying information about the fireball created after the collision process. Some typical signatures can be studied to characterise such strongly interacting medium, the QGP, comparing measured parameters with the previsions of theoretical frameworks. Thus, quantities like the energy density (equivalently the temperature), baryon density (or the chemical potential) and some other characteristics of the fluid can be calculated using the observables available in this kind of process. There are two main kinds of probes: the electroweak probes and the hadronic probes.

- The electroweak probes (virtual ($e^+e^-\mu^+\mu^-$) and real emitted direct photons by the black-body radiation of the fireball in the early stages) bring the characteristics of how the fireball was in its early stages. Since the strongly interacting medium is transparent to the electromagnetic radiation, they can easily pass through and reach unharmed the detectors. On the other hand they are less abundant than the hadronic probes.

- Hadronic probes, classified according their fraction of transferred momentum either in soft or hard probes, are influenced by the medium. The soft probes are produced either in soft processes or in the final stage of the collision. The majority of the whole particle set is composed of soft probes. The hard probes are particles generated in partonic scattering, with large transferred momentum, corresponding to a short interaction time. These probes are sensitive to all the phases of the system evolution.

The physical observables related to these probes are:

- the particle multiplicity per rapidity unit, $\frac{dN_{ch}}{dy}$. In the design phase of an experiment this parameter has to be taken seriously into account. When the ALICE experiment was conceived (neither nowadays), there was no model to safely estimate a priori the particle multiplicity. An available approach was to simulate collisions but, due to the overwhelming majority of soft interactions in the process, those simulations relies on the lattice QCD formalism. Computationally speaking, it required too high computing resources; hence, before the publication of the Technical Design Report for ALICE detectors, no safe estimate for the particle production was available.\(^6\)

\(^6\)The ALICE apparatus was designed to run with a multiplicity up to 8000 particles per rapidity unit in the central barrel. Figure (1.7) shows the $\sqrt{s_{NN}}$ dependency. The value was clearly overestimated, since measured particles per rapidity unit are $1943 \pm 54$ in $\eta < 0.5$ (in the central rapidity region). As a reference, see: \(^1\)
Chapter 1. Overview of the ALICE Experiment

This quantity is important, for example, to evaluate the energy density within the the Bjorken model through the formula (1.4).

$$\epsilon_{Bj} = \frac{\langle m_T \rangle}{\tau_f A} \frac{dN}{dy}(\tau_f)$$ (1.4)

Where $\langle m_t \rangle$ is the transverse mass (defined as $m_T = \sqrt{m^2 + p^2_T}$) and $\tau_f$ is the interaction time;

- the transverse energy per rapidity unit $\frac{dE_T}{dy}$ is used to determine the fraction of initial energy converted in particles yielded along the transverse direction.

- the nuclear modification factor is defined as the ratio between the the $p_T$ spectrum of particles in A-A and the one in pp collisions ($N_{coll}$), normalised to the average number of nucleon-nucleon interactions occurring in A-A collisions ($N_{coll}$).

$$R_{AA} = \frac{1}{N_{coll}} \frac{dN_{AA}/dydp_T}{dN_{pp}/dydp_T}$$ (1.5)

---

According to: [5]
In absence of nuclear effects the $R_{AA}$ is expected to be 1. Differences between nucleus-nucleus and a superposition of uncorrelated binary collisions between nucleons are seen as deviations from this value.

The $R_{AA}$ is sensitive to the formation of the QGP partons with high transverse momentum generated during the process loose their energy scattering with the medium, resulting in a suppression in the high $p_T$ region of the spectrum. By estimating the energy loss of hard partons using the nuclear suppression factor one can study the properties of the medium.

There are two regimes expected: a soft regime and a hard regime. The first, where soft processes dominate the low $p_T$ region and the number of produced particles depends on the number of the nucleons participating $N_{\text{part}}$ in the process, results in a $R_{AA}$ lower than one, since in A-A collisions $N_{\text{part}} \ll N_{\text{coll}}$. The second, in the region of high $p_T$, is expected to be one, according to the fact that the particle production depends on number of nucleon-nucleon collisions, in absence of further nuclear effects.

- the **identified particle spectra** is yielded for the $\approx 99\%$ by the soft processes occurring in the hadronisation phase of the collision. These particles undergo many elastic scatterings that change their momenta distributions. A measurement of the yield of the identified hadrons provides an estimation of the chemical freeze-out temperature $T_{ch}$, the baryon-chemical potential $\mu_b$ and the volume of the deconfined matter $V$. Those parameters can be evaluated by fitting the experimental data, extrapolating parameters of a statistical hadronisation model, assuming the system in a thermal and chemical equilibrium at the chemical freeze-out. The system considered is a Grand Canonical Ensemble (i.e. it exchanges energy and particles with the outside), thus the abundance of the $i$ species, with $g_i$ degrees of freedom is (1.6):

$$N_i(T, V, \mu_i) = \frac{g_i V}{2\pi^2} \int_0^\infty \frac{p^2 dp}{e^{(E-\mu_i)/T}}$$

- the **transverse momentum particle spectra** become fixed after the thermal freeze-out, when all the interactions cease. The transverse momentum spectrum at low $p_T$ of system in thermal equilibrium is described by a Maxwell-Boltzmann distribution, where $T_{fo}$ is the thermal freeze-out temperature.

$$\frac{1}{p_T} \frac{dN}{dp_T} \propto e^{-\sqrt{m^2+p_T^2}/T_{fo}}$$
Figure 1.8: Primary $\pi^-$ (left) and $K^-$ (right) transverse momentum spectra and corresponding fits in Pb-Pb collisions at $\sqrt{s_{NN}} = 2.76$ TeV [35].

However, in Pb-Pb collisions, where it is formed the QGP, it is measured a different temperature ($T_{\text{slope}}$). This deviation shows a dependency on the mass $m$. This is due to a collective motion along the transverse plane, called radial flow, superimposed to the thermal random motion generated by the internal pressure of the fireball. The $T_{\text{slope}}$ can therefore be decomposed as follows:

$$T_{\text{slope}} = T_{f_0} + \frac{1}{2} m \langle v_T \rangle^2$$ (1.8)

- the anisotropic flow is another kind of collective motion observed in nucleus-nucleus collisions. This is the more evident in semi-central collisions. When two nuclei collide the interaction volume created is almond-shaped and the spatial anisotropy is later converted in transverse momentum anisotropy due to the different pressure gradients along the orthogonal axes of the fireball. The azimuthal angle distribution of the particles is therefore measured in the final state.

Using a Fourier decomposition of the azimuthal angle distribution with respect to the reaction plane (defined by the intersection between the impact parameter and the beam direction $\Psi_{RP}$):

$$\frac{dN}{d(\phi - \Psi_{RP})} = \frac{N_0}{2\pi} \sum_{n=0}^{\infty} 2v_n \cos[n(\phi - \Psi_{RP})]$$ (1.9)

the anisotropy can be characterised with the $v_n$ Fourier coefficients defined as in:

$$v_n = \langle \cos[n(\phi - \Psi_{RP})] \rangle$$ (1.10)

The $v_2$ coefficient, named elliptic flow, is particularly interesting: it represents the ratio between the two orthogonal directions with respect to

---

8 See: [35]
9 See schema in fig. 1.9a
1.3. The Physics of ALICE

(A) Sketch of a non-central collision. The almond-shaped interaction zone and the reaction plane are specified.

(b) Measure of the anisotropic flow $\nu_n$ integrated over $0.2 < p_T < 5.0\text{GeV}/c$ as a function of the event centrality, at $\sqrt{s_{NN}} = 5.02(2.76)\text{TeV}$.

Figure 1.9

the impact parameter. Hence, the conversion from the initial spatial anisotropy to the final momentum anisotropy is strongly influenced by the QGP viscosity. The flow measurements provide parameters to theoretical models to extrapolate informations about the quality (perfect or imperfect) of the fluid created and about the thermalisation process of the fireball.

- the jet quenching is a phenomenon related to the energy loss by jets in the medium. Jets are particle showers yielded by high momentum partons in a collision. In leptonic and hadronic collisions these partons are created back-to-back and produce hadrons in the hadronisation process. Signals of this process correspond to two large energy deposits in hadronic and electromagnetic calorimeters, very close each others in the phase space. The study of these signals become very useful in studying Pb-Pb collisions to characterise the fireball. As previously mentioned, high momentum partons are created in the very early stages of the collision, so they witness all the evolution phases. Their momentum can change because of the energy loss due to the gluon emission in the medium and can be evaluated from the BDMPS formula

$$\langle E \rangle \propto \alpha_s C_R \hat{q} L^2 \quad (1.11)$$

where $\alpha_s$ is the QCD coupling constant, $C_R$ is the Casimir factor, $\hat{q}$ the transport coefficient and $L$ the distance travelled by the particle in the medium.
Chapter 1. Overview of the ALICE Experiment

Figure 1.10: Example of an imbalanced dijet emission in Pb-Pb at $\sqrt{s_{NN}} \simeq 2.76$ TeV. The plot shows the summed transverse energy in the electromagnetic and hadron calorimeters vs $\eta$ and $\phi$.

The main effect of this energy loss is the reduction of the total cross-section in jet-production. When the high $p_T$ partons are generated on the corona of the fireball the centre-of-mass back-to-back symmetry is broken. Indeed one of the partons can easily escape and travel through a non color-interacting medium, the other one must cross the fireball, loosing its energy (1.11). This results in an unbalanced dijet event.\footnote{As reference consider: [7]}

- the photons and dileptons emission is caused by the quark gluon plasma through a thermal radiation mechanism. They cross the medium unharmed and carry informations about the initial state conditions. They would give a very precise temperature measurement, unfortunately the experimental background caused by secondary decays\footnote{The background caused by the $\pi^0 \rightarrow \gamma\gamma$ secondary decays is overwhelming.} makes hard the measurement of these electroweak probes.

- the suppression of the quarkonia states. In the collision are produced, among the hard probes of the quark gluon plasma, bound states of $c\bar{c}$ (charmonium) and $b\bar{b}$ (bottomonium). The pair particle-antiparticle is due to the fact that there is no net charm or bottom charge in the initial state. A Cornell-like potential describes the the strength of the binding interaction in vacuum

\begin{equation}
V_{QCD}(r) = -\frac{4}{3} \frac{\alpha_s(r)}{r} + kr \tag{1.12}
\end{equation}
where $\alpha_s$ is the strong coupling constant, $r$ the distance between the quarks and $k$ is a string tension constant. The presence of a color-interacting medium like the QGP cause a modification to the creation of the quarkonia. There are screening effects (Debye screening) caused by free gluons and quarks. Therefore the potential form changes in a Yukawa potential:

$$V_{QCD}(r) = -\frac{4\alpha_s(r)}{3}\frac{e^{-r/\lambda_D}}{r}$$

where it appears $\lambda_D$ (Debye length) to represent the maximum distance for partons to be considered strongly interacting (At $\lambda_D \approx 3 \div 4$ the potential is almost zero). From QCD at the leading order calculations the form for $\lambda_D$ is:

$$\lambda_D = \frac{1}{g\sqrt{\left(N_c^3 + N_f^6\right)T}}$$

where $N_c$ is the number of the colour degrees of freedom, $N_f$ is the number of flavours included in the model. Because of the dependence of $\lambda_D$ from the temperature $T$, it is possible to deduce the temperature of the quark gluon plasma probing the presence or the suppression of the different excited states of quarkonia. Those states, have different binding energies (binding radii) so the suppression of some binding states give important indications about the QGP temperature (provided considering that some quarkonia states decay into another ones).

1.4 The ALICE experimental setup

The ALICE experiment is the only experiment at LHC specifically conceived to study the heavy-ion collisions. It continues the tradition of experiments with high energy heavy-ion collisions like Pioneering High Energy Nuclear Interaction eXperiment (PHENIX) and Solenoidal Tracker at RHIC (STAR) and other experiments at the Alternating Gradient Synchrotron (AGS) and at the Super Proton Synchrotron (SPS). Its goal is to study both perturbative and non-perturbative regimes of QCD collective evolution of the particles generated in the collisions, the chiral symmetry restoration and the characteristics of the QGP. The detector has been designed and built to cope with high multiplicities (up to 8000 charged particles per unit of rapidity) in heavy-ion collisions. One of the aims is to reconstruct the high number of tracks created by such multiplicity. The detector acceptance has to be large enough to provide an adequate coverage to study the QGP phase signatures. In fig. (1.11)
Chapter 1. Overview of the ALICE Experiment

is depicted the whole apparatus. It can be divided in three main parts, each one dedicated to cover a different pseudorapidity ($\eta$) window and serving complementary purposes. Hereafter is reported a list of the detectors belonging to each part, followed by a brief description:\textsuperscript{13}

1. the \textbf{central part} covers $\pm 45^\circ$ (i.e. a pseudorapidity interval $|\eta| < 0.9$) over the full azimuthal angle and is embedded in a weak solenoidal magnetic field. The full list of detectors, from the inside out:

   - the \textbf{Inner Tracking System (ITS)}
   - a cylindrical \textbf{Time Projection Chamber (TPC)}
   - a \textbf{Transition-Radiation Detector (TRD)}
   - a large area array of \textbf{Time Of Flight (TOF) counters};
   - an \textbf{Electromagnetic Calorimeter (EMCal)}
   - an electromagnetic calorimeter: \textbf{PHOton Spectrometer (PHOS)}
   - an array of counters optimised for \textbf{High-Momentum Inclusive Particle Identification (HMPID)}

2. the \textbf{forward detectors} set, constituted by:

   - a \textbf{Zero Degrees Calorimeter (ZDC)};
   - a \textbf{Forward Multiplicity Detector (FMD)}
   - a \textbf{Photon Multiplicity Detector (PMD)}

3. the \textbf{Forward Muon Spectrometer (FMS)}

4. the \textbf{ALICE Cosmic Ray Detector (ACORDE)}

5. the \textbf{ALICE Forward Hodoscopes (AD)}

1.4.1 Magnet

ALICE uses a \textit{hot magnet} to measure the momentum of the charged particles. Its magnet was built for the $L_3$ experiment at the \textbf{Large Electron–Positron Collider (LEP)} and surrounds the central barrel detectors. The trade-off between the necessity to bend high-momentum particles but at the same time to resolve low $p_T$ particles in the \textbf{TPC} led to the choice of a magnetic field of $0.2 T \leq B \leq 0.5 T$. The lowest momentum of a particle that can be resolved in the $\text{TPC}$ is $p_{\text{cut-off}} [GeV/c] = 0.3 BR \approx 0.1 GeV/c$. $B$ is the magnetic field in Tesla (0.5 T), $R = 0.9$ m is the inner radius of the TPC, the external radius is $R_{\text{ext}} = 2.5$ m. The resolution of lower momentum particles is delegated to the \textbf{ITS}

\textsuperscript{13}For further informations:\textsuperscript{17}
1.4. The ALICE experimental setup

Figure 1.11: Alice experimental setup
1.4.2 Inner Tracking System

The Inner Tracking System (fig. 1.12a) is the closest detector to the interaction region and covers a pseudorapidity window of $|\eta| < 0.9$. Its aim is to make it possible to resolve and reconstruct primary (ion-ion impact) and secondary vertices (hyperons, D and B mesons or light mesons like pions with $p_T < 80\text{MeV}/c$ decays) by measuring the track parameters as near as possible to the production vertex. It is composed of six concentric layers of silicon detectors that provide a vertex resolution better than $100\mu m$. Three different technologies in silicon detectors are exploited in the ITS, as detailed in table 1.2.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Type</th>
<th>$r$ (cm)</th>
<th>$\pm z$ (cm)</th>
<th>Area ($m^2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pixel</td>
<td>3.9</td>
<td>14.1</td>
<td>0.07</td>
</tr>
<tr>
<td>2</td>
<td>pixel</td>
<td>7.6</td>
<td>14.1</td>
<td>0.14</td>
</tr>
<tr>
<td>3</td>
<td>drift</td>
<td>15.0</td>
<td>22.2</td>
<td>0.42</td>
</tr>
<tr>
<td>4</td>
<td>drift</td>
<td>23.9</td>
<td>29.7</td>
<td>0.89</td>
</tr>
<tr>
<td>5</td>
<td>strip</td>
<td>38.0</td>
<td>43.1</td>
<td>2.20</td>
</tr>
<tr>
<td>6</td>
<td>strip</td>
<td>43.0</td>
<td>48.9</td>
<td>2.80</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Total area</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6.28</td>
</tr>
</tbody>
</table>

Table 1.2: ITS layer quotas.

1.4.3 Time Projection Chamber

The Time Projection Chamber (fig. 1.12b) is the main tracking detector in the ALICE central barrel and covers a pseudorapidity window of $|\eta| < 0.9$. It allows the reconstruction of charged particles trajectories measuring their energy loss $dE/dx$ (useful to do Particle IDentification (PID)) and momentum, and granting the two-tracks separation. The total active volume inner radius is $\sim 85$ cm and an outer radius $\sim 250$ cm, with 500 cm along the beam direction. Despite being a comparatively slow detector, with about 90 $\mu s$ drift time over the full length, the TPC can take minimum-bias collisions at a rate of 1kHz in Pb-Pb. It was ready to stand the particle tracking worst case scenario of $\left.\frac{dN_{ch}}{dy}\right|_{y=0} \simeq 8000$ for central events.
1.4. The ALICE experimental setup

Particle identification

The ALICE experiment has, among its main features, the capability of particle identification. With the ITS and the TPC together it is possible to reach excellent PID capabilities at low $p_T$, in fact it is possible to separate pions in a $p_T$ range between 200 MeV/c < $p_T$ < 1.2 GeV/c. There are further detectors situated in other places within the experiment to identify particles with high momentum, with the help of TOF pions can be identified to a $p_T$ range up to 20 GeV. In figure 1.13 there is a comparison between the "separation power", the capability of detector to distinguish between two particles, of every detector used for the PID in ALICE as a function of the $p_T$.
1.4.4 Time Of Flight

The Time Of Flight detector can identify charged particles in a momentum range of $0.2 \pm 2.5\text{GeV}$ in the central pseudorapidity range ($|\eta| < 0.9$). It keeps the cylindrical symmetry of the innermost detectors, with an internal radius of 370 cm, an external radius of 390 cm and with a total length of 7.45 m. It is made by Multi-gap Resistive-plate Chambers, which have a good time resolution with a low manufacturing cost (considering the area covered by the TOF this has to be taken in consideration at design time!). The TOF measures the time of flight of a particle, defined as the time elapsed between the particle production, provided by the T0 detector, and its detection by the MRPC. Thus, known the time of flight (so its $\beta$) and its momentum given by the TPC, it is possible to determine the mass of a particle.

1.4.5 Transition Radiation Detector

The TRD fills the space between the TPC and the TOF. It is necessary to identify the electrons with momentum higher than $1\text{GeV/c}$ since the identification through the energy-loss by the TPC is no longer efficient. For the physics studied in ALICE this capability is crucial, for example, to separate electrons yielded by the $J/\Psi$ decay from the background of Dalitz-decays. Open charm and open beauty mesons decaying in their semi-leptonic channels can be studied combining the impact parameter information coming from the ITS and the identification coming from the TRD. This detector can also provide identification for particles with a Lorentz factor $\gamma > 1000$. Pions and electrons can be well-distinguished in the momentum range $1 \leq p \leq 100\text{GeV/c}$.

1.4.6 High-Momentum Particle Identification Detector

The HMPID identifies charged hadrons with a $p_T > 1\text{GeV/c}$, extending the PID capabilities of the ITS, TPC and TOF. It has a narrow acceptance, about $\sim 5\%$ of the central barrel phase space. The detector relies on proximity-focusing Ring Imaging Cherenkov (RICH) counters and is composed of seven modules. The angular coverage is $|\eta| < 0.6$ and $1.2^\circ < \phi < 58.5^\circ$.

1.4.7 Photon Spectrometer

This detector has been designed to measure the photons emission, distinguishing the direct ones from the ones coming from the decay of $\pi^0$ and $\eta$ (which contains in its main decay channels always a $\pi^0$). It also measures the jet-quenching effect through the observation of high $p_t$ $\pi^0$ and $\gamma-$jet correlations. It has then to provide a good granularity (spatial resolution) and a good energetic resolution. The covered area is about 8 m$^2$ whilst the acceptance in pseudorapidity is in the range $|\eta| \leq 0.12$. 
1.4.8 Electromagnetic Calorimeter

The EMCal enhances the ALICE performances in the study of jet quenching in Pb-Pb collisions. It reveals direct photons, jets and electrons from heavy flavours decays. PHOS and HMPID limit the azimuthal coverage of the EMCal to $\Delta \phi \simeq 10^\circ$; the coverage in pseudorapidity is $|\eta| \leq 0.7$. Its position provides also a partial coverage, with PHOS, to the back-to-back photon emissions. The overall acceptance is about the 25% of the whole TPC acceptance.

1.4.9 ALICE Cosmic Ray Detector

ACORDE is a cosmic ray trigger detector, composed of an array of scintillator counters placed on the top of the $L_3$ magnet. It has two main tasks:

- to provide a fast trigger signal for the commissioning, calibration and alignment procedures of the other detectors;
- combine its signal with TPC TRD and TOF to measure the spectrum of the high energy cosmic rays in the correspondence with the knee of the cosmic rays spectrum.

1.4.10 Forward Muon Spectrometer

The main purpose of the Forward Muon Spectrometer is to reconstruct the di-muon pairs with a resolution sufficiently high to allow - through the invariant mass spectrum - to observe the quarkonia resonances like $J/\Psi$, $\Psi'$, $\Upsilon$, $\Upsilon'$ and $\Upsilon''$ as well as the $\phi$ meson. The efficiency in reconstructing the open-charm mesons through the coincidences $e^- \mu^-$ with the muons detected by the FMS and the electron by the TRD. This detector is placed around the beampipe and covers an acceptance region $-0.4 \leq \eta \leq -2.5$. It is made of a passive front absorber to absorb hadrons and photons coming from the interaction vertex, a high-granularity tracking system made of 10 detection planes, a large warm dipole magnet, a passive muon-filter wall followed by four planes of trigger chambers and an inner beam shield to protect the chambers from primary and secondary particles produced at large rapidities. The front absorber, whose length is 4.13 m, is located inside the solenoid magnet. The fiducial volume of the absorber is made of carbon and concrete in order to limit the small-angle scattering and energy-loss by traversing muons. The magnet produces a horizontal field, perpendicular to the beam axis. Its integrated value, from the interaction point and the muon filter, is 3 Tm. The tracking relies on cathode pad chambers, specifically designed to achieve a spatial resolution of about 100 $\mu m$. They are arranged in five stations: two are placed before, one inside and two after the dipole magnet.
1.4.11 Forward rapidity section

The high rapidity region is covered by several detectors located at small angle with respect to the beam axis. The Zero Degrees Calorimeter gives information about the centrality of Pb-Pb collisions measuring the energy of the nucleons not involved in the collision (spectators). Two calorimeters compose each ZDC, one for protons and one for neutrons. The two ZDCs are installed symmetrically at 116 m from the interaction point. The Forward Multiplicity Detector provides information on the charged particle multiplicity inside the pseudorapidity ranges $-3.4 \leq \eta \leq -1.7$ and $1.7 \leq \eta \leq 5.1$. The T0 detector generates the level 0 trigger for the TOF with a precision of $\approx 50$ ps, measures a rough vertex position and provides a first level trigger and helps to distinguish between beam-beam collisions and beam-gas collision. The V0 detector provides a minimum bias trigger for the central barrel setup and it is used as a centrality indicator. Both the T0 and the V0 consist of two modules positioned on the two opposite sides of the interaction point. For the data taking in Run 2 of the LHC, the ALICE also installed two small detectors made of scintillation plastic in the forward and in the backward rapidity regions. The two installation are far $\sim 20$ m from the interaction point (on each side). This is the AD device and covers a pseudo rapidity interval $4.8 < \eta < 6.3$ in the forward and $-7.0 < \eta < -4.9$ in the backward. With this new system, the ALICE sensitivity to diffractive masses is close to the diffraction threshold of $1.08 \text{ GeV/c}^2$ which corresponds to the sum of proton and pion masses.

1.4.12 ALICE detector Coordinate System

The coordinate frame adopted is a right-handed orthogonal Cartesian system with the origin $(x, y, z) = \vec{0}$ in at nominal beams intersection point.

- $x$-axis is horizontal, perpendicular to the main beam direction and pointing to the centre of the accelerator;
- $y$-axis is perpendicular to the $x$-axis and the mean beam direction, pointing upwards;
- $z$-axis is parallel to the mean beam direction, with its pointing given by the chirality of the coordinate system.

The central barrel has a cylindrical symmetry with respect to the $z$-axis.

1.4.13 Trigger system

The trigger system adopted in ALICE relies on three levels of hardware triggers (L0, L1 and L2) plus a fourth software trigger (High Level Trigger (HLT)).
This setup is led by considering the number and rate of events with specific signatures crucial for physical analyses, the dead and response time of each sub-detector, the bandwidth of the Data AcQuisition System (DAQ) consisting of several VME boards. All triggers are monitored by a central unit, the Central Trigger Processor (CTP) consisting of several VME boards. The CTP receives the trigger signal which propagates to the Local Trigger Unit (LTU) of each required sub-detector. Then the so-called trigger-classes are defined (high-multiplicity, minimum-bias, di-muons) as the combination of different logical trigger inputs. Trigger levels are characterised by different response time, depending on the response time of each detector. The HLT works before storing an event. Among other operations it performs a preliminary tracking and rare process selection. It performs the local reconstructions of the TPC and saves the rec-points, throwing away raw data, with the aim to reduce the amount of data stored.

1.4.14 Data Acquisition System

During Run 2 the LHC running conditions determine high data throughputs for every experiment. In p-p collisions the nominal luminosity value is \( L = 10^{30} \text{ cm}^{-2}\text{s}^{-1} \) with a bunch-crossing frequency of 50 ns. In PbPb collisions the luminosity is \( L = 10^{27} \text{ cm}^{-2}\text{s}^{-1} \), with a bunch-crossing every 100 ns. This corresponds the huge readout rate of 1.25 GB/s (in Pb-Pb), with the average event size of \( \sim 25 \text{ MB} \). This creates \( \approx 2.1 \text{ PB} \) of data yearly, with an overall amount of \( \sim 45 \text{ PB} \) of data on tape and \( \sim 55 \text{ PB} \) of data on disks so far\(^{14}\). In the scope of High Energy Physics (HEP) the minimum unit of data needed to be processed is an event, which in turn is converted to a single particle collision or multiple piled up collisions. Particle physics events have two important features which define the processing paradigm adopted: they are many and independent. In ALICE when the electronics of a detector is triggered, it sends data to a pc, the Local Data Concentrator (LDC) situated on the experimental site. Each LDC aggregates data fragments coming from detectors in sub-events. Sub-events are collected by the Global Data Collectors (GDC) that merge them into global events. Every GDC machine is able to process up to 40 events in parallel and the Global Data Storage servers (GDS) store data locally before asynchronously migrate them to the CERN computing facilities, where they become available for offline analysis.

\(^{14}\)ALICE data storage 2015-2016
1.5 The ALICE Software framework

1.5.1 The ROOT framework

ROOT is a cross-platform data-process framework, created at [CERN] in 1995 by René Brun and Fons Rademakers[15]. Currently it is composed by 3000 C++ classes grouped into 110 top-level categories. It is widely accepted and adopted by the HEP community. ROOT provides a Virtual Monte-Carlo (VMC) interface to the most important HEP simulation engines, like Geant4 (C++), Geant3, Fluka. The VMC interface allows users to build an application that simulates the behaviour of a particle detector, with the possibility to switch between different simulation engines. Comparing the results of different simulation engines allows to estimate systematic simulation uncertainties. It is equipped with a C++ interpreter called C-Interpreter (CINT) that allows the users to use the C++ language as a scripting language. Usually, most ROOT users develop programs to perform statistical data analysis of binned or unbinned data. There are included many packages for event manipulation, classification and analysis like TMVA[16]. ROOT can also be used to create event visualisation tools like in (fig. 1.15).

1.5.2 AliRoot

The ALICE Offline Project started developing the AliRoot software framework in 1998. It was born due to the need to create a simulation tool for

---


1.5. The ALICE Software framework

Figure 1.15: AliRoot event display for a measured pp collision at $\sqrt{s_{NN}} = 13$ TeV.

The ALICE Software framework is based on the Technical Design Reports of the ALICE detector using Object-Oriented (OO) programming paradigm and C++ as an implementation language, except for some other legacy large existing libraries (GEANT3-4). It is based on the ROOT framework and relies on GEANT$^1$ as particle transportation code. Physicists mainly use AliRoot for three kind of tasks: simulation, reconstruction and analysis.

Simulation software

AliRoot can plug to many Monte Carlo event generators, for example PYTHIA$^{18}$ HIJING$^{19}$ HERWIG and PHOJET; softwares like GEANT3, GEANT4 and FLUKA are used to transport the generated particles through different detectors, simulating the interaction and the energy deposition with the material constituting them. Every hit generated is labeled with Monte Carlo flags. The following step is to simulate the detector electronic response to the geometrical hits. In this phase first a data structure is produced called summable digit later merged in digit. The merging process is an actual sum, since summable digits have low thresholds, so that a digit can be created adding weaker signal coming from more particles (event merging). The resulting data have realistic thresholds and can be compared with real raw data. In fig. 1.17a is well-represented the information work-flow from simulation to the reconstruction chain. Cross-checks between different steps can be made to estimate the quality of the process. The final product of a simulation mimics real data as they

$^1$http://geant.cern.ch/
$^2$http://home.thep.lu.se/torbjorn/Pythia.html
$^3$http://ntc0.lbl.gov/xnwang/hijing/
Figure 1.16: Schema of the interface of AliRoot Software and its exposed interface to external components.

(a) Work-flow of the simulation and reconstruction chains.

(b) Comparison between real data and simulated data lifeline. *Cluster* is the first data format in common between the two procedures.

Figure 1.17
would be taken by the real detector. In addition it carries the so-called *Monte Carlo truth*, made of useful labels to diagnostic the accuracy of reconstruction software and algorithms.

**Reconstruction software**

The reconstruction (represented in the right branch of the parabola in fig. 1.17a) can be run on simulated and real data. Each detector reconstructs its own clusters/rec points autonomously. After that *track finding* algorithms reconstruct the trajectories of the particles and parametrise their kinematics. Summarising, there are three steps in data reconstruction, with different formats:

- **Raw Data** → digitised data coming from each detector;
- Reconstructed points (**Rec points**) → spatial coordinates corresponding to digitised data and energy deposition.
- **Tracks** → fitted curves defining particle’s trajectories, obtained by reconstructed points and described by five parameters (curvature, two angles with respect 3D coordinates and two positions) and their covariance matrices estimated at a fixed point in space. Information on the detectors employed in the reconstruction phase is still kept.

In order to fulfil the heavy-ion program, the tracking system has to efficiently cope with high multiplicity events (≈ 10³ particles) and must be capable to resolve low $p_T$ tracks. For these reasons ALICE juxtapose a *standalone* tracking system, using the innermost detectors (the ITS) to track charged particles, alongside the *global* tracking system that relies on data from all the other detectors. Reconstructed events are stored in two formats: the **Event Summary Data (ESD)** that can be further condensed into **Analysis Object Data (AOD)** storing all the essential data needed for the analyses.

### 1.5.3 AliEn

As it will be exhaustively described in the next chapter, data processing (simulation, reconstruction, analysis) in experiments at the LHC is mainly achieved through a distributed computing paradigm. As reported in section §1.4.1.4 the experiment produces ≈ 2.1 TB per year, which is an incredibly huge amount of data. To take advantage from the event-based computing approach, oriented more on the high data throughput than the high performance computing, it was developed a completely new infrastructure, first of its kind, called **Grid**. It is composed by a set of data-centres geographically distributed among
the participating countries with the aim to provide access to data and computing resources. All the experiments at the LHC exploits this service, implementing their own distributed computing framework interfaces. The ALICE open-source multipurpose Grid framework is called Alice Environment Grid Framework (AliEn). Its interface transparently provides a job-submission and job-execution access to the Grid. AliEn also manages schedule, keeping into account many resource-related, topological and geographical parameters to optimise the efficiency (e.g. jobs are executed as close as where data are stored as possible). ALICE data on the Grid can be transparently referenced from a global namespace independently from the actual location (and replicas location) thanks to a central File Catalog (FC).

\footnote{hereafter with job is intended every program that has to be executed on a machine.}
Computing in High Energy Physics

2.1 Computing challenges at LHC

It is well-known that alongside the important and successful results claimed by the four major experiments [1] at the LHC, CERN is a centre of research and development in many fields other than Physics. LHC was expected to be the largest data generator ever made, both in rate and overall amount ever produced [2] and the experiments had to cope with new challenges in terms of computing and storage requirements never reached before. Since the beginning it was clear that no single computing facility could have provided all the computing power required. Therefore a new infrastructure concept was conceived to deal with such a unique use case and a worldwide computing infrastructure, the Worldwide LHC Computing Grid (WLCG) was built.

2.2 The Worldwide LHC Computing Grid

Built upon existing Grid structures, such as OpenScience Grid (OSG) the former Enabling Grid for E-sciencE (EGEE) – now European Grid Infrastructure (EGI) – and additional resources provided by the nordic project Nordugrid [3], the Grid meta infrastructure currently counts 42 participating countries, 170

---

[1] As previously mentioned in §1.1
[2] An early estimation suggested that 15 PetaByte (PB) of data would have been produced per year. The first data taking period exceeded that, measuring an average of 25 PB per year.
computing centres involved with a rate of about 2 million tasks ("jobs") run every day. It shows two main features:

- it is *geographically distributed*: not a single computing centre is capable of coping with the amount resources needed to store and process such an amount of data, also taking into account the electrical consumption for running and cooling the machines.

- it is *federated*: it is composed by series of smaller computing centres whose duties are orchestrated by central points.

### 2.2.1 A tiered architecture

Computing centres participating to the Grid are organised according a hierarchical *tiered* model, depending on the roles they cover and the [Quality of Service (QoS)](https://www.globus.org) that they provide.

- There is only one **Tier-0**, located at [CERN](https://www.cern.ch) and even though it is constituted by a single computing centre it provides about 20% of the total computing resources of the whole [WLCG](https://wlcg.web.cern.ch). It stores and preserve the first copy of all *raw data*; the Tier-0 is also in charge of running the first reconstruction and redistributing RAW data and reconstruction output [ESD](https://wlcg.web.cern.ch) to the Tier1 and reprocessing of data during LHC downtimes.

- The **Tier-1** centre receive and store an amount of (RAW and reconstructed) data from the Tier-0 proportional to its capabilities in terms...

---

**Figure 2.1:** A display of the operating requirements of some major experiments in high energy Physics in the past 30 years. Arrows shows the trend expected in 2012[8]. The fastest LHC experiment is LHCb. ALICE owns the largest average event size (25MB/event in Pb-Pb during Run 2).
of resources. Tier-1s do a large scale reprocessing and secure keeping of corresponding output. In their turn they orchestrate the redistribution of these data products to Tier-2s.

- **Tier-2** centres are the environment where the users run their analyses. The Tier-2s provide a safe copy of reconstructed data and the storage needed to host the analysis result.

- A fourth division, even though it is not formally included in the hierarchy, can be represented by Tier-3s. They are considered autonomously-managed computing clusters. Often they are small computing facilities that grant access to local users only. Sometimes they can provide resources to share with the Grid.

The technological improvements in matter of connectivity performances (rate, bandwidth) and the increase in CPU density spread among sites, together with an overall reduction of the hardware cost, with respect to when the Grid was conceived, progressively blurred the borders between the previous categories. For example nowadays it may happen that a “non-neighbour” site pulls data directly from the Tier-0, as well as same-tier transfers might frequently occur.

In addition, also the essentially *datacentric* computing model - a model where data access in job execution is supposed to be *local* with respect to where the data are stored - sometimes observes *remote* accesses to data: a

---

4i.e. Tier-3s are not included in the WLCC specifications.
remote site provides data to other sites. The model is naturally evolving towards a cloud-like storage, where primary data are held mainly in few storages and several distributed non critical\textsuperscript{5} caches hosted in smaller centres.

### 2.2.2 High Throughput Computing

In \textit{HEP} the minimum amount of data needed to process is constituted by an \textit{event}. It contains all data acquired by a single detector trigger, whether it be a single particle collision or multiple piled up ones. Therefore events are independent from each other and the only things in common are the \textit{data-taking conditions}\textsuperscript{6}. This type of computing is often familiarly referred to as “embarrassingly parallel”\textsuperscript{7} problems, often referred as \textit{event-based parallelism}. Moreover, to successfully face the huge amount of events to analyse it is useful to concentrate the effort on having the largest number as possible of events processed in parallel rather than focusing on improve the processing time of a single event. This is an efficient way to deal with this problem, thanks to the large amount of available resources. Therefore in this scenario, we talk about High Throughput Computing (HTC) which is focused in efficiently process large data flows, compared to of High Performance Computing (HPC) which is focused on performances of the algorithms. In the phase of processing the events there is a set of required steps to fulfil. Each step is a summarised representation of the events in the previous one.

1. First step sees the \textit{detector hits} become \textit{raw data}. This is achieved in the data acquisition phase. Different triggers discriminate, at different levels, whether or not the measured signal is interesting enough to be propagated to the successive level of trigger. Among them a \textit{HLT} is usually implemented, performed by a dedicated computer facility. For ATLAS and CMS the median event size is about 2 MB per event.

2. \textit{Raw data} is a mere list of hits on detectors that does not provide explicit Physics meanings. Particle tracks are identified and reconstructed starting from raw data, the result is called \textit{Event Summary Data (ESD)}. Both input and output have different entries, each one being an event. The difference is in the size: reconstructed event size is about 20 times lower than the input raw data, for instance 100 kB/event.

3. A further reduction process is operated converting the \textit{ESD} to smaller files: the \textit{Analysis Object Data (AOD)}. They are collections of reconstructed events skimmed to match the interest of a particular end user.
analysis. The event size is ten times smaller than the reconstructed data ($\sim 10$ kB)

4. Last step in event processing is represented by the results: summaries of Physics events, collected in histograms, plots and fits, the actual output of a Physics analysis.

2.3 Computing in ALICE

Fast data access is a crucial point in every LHC experiment. ALICE in particular emerges for the amount of informations stored in its reconstructed ESDs. That’s why it has always been "on the edge" of data access technologies; it was the only experiment having a central file catalog, granting access to files from everywhere in the world adopting unique URLs (the so-called AliEn File Catalog).

Moreover, ALICE has been between the first experiments using XRootD\(^8\) as storage transfer protocol, providing agile and resilient mechanisms to manage distributed data.

From the tier\(^9\) perspective, the ALICE computing model comprises only Tier-1 and Tier-2 centres, local batch systems setup as Tier-3 are not officially part of it. As previously mentioned, its analysis framework is called AliRoot. ALICE production Grid environment is called AliEn\(^10\) which can be run both on laptops and on dedicated Grid sites as well. The unicity in AliEn, with respect to the frameworks of other experiments that require a login node called a user interface to access the full framework and the Grid middleware, consists of porting the ultimate user interface to user’s personal computers. So far ALICE has used AliEn for the distributed production of Monte Carlo data, reconstruction and analysis at over 80 sites. More than 280 million jobs have been successfully run worldwide from the Task Queue (TQ), resulting in the currently active volume of 14 PB of data.

2.3.1 Virtualisation and Cloud Computing

Virtualisation and Cloud Computing are two distinct concepts but strictly related. The first consists of the abstraction of resources, the second focuses on service provisioning. In computer science, the idea of Virtualisation is included in a broad semantic area, from the cloud storages\(^11\) to the entire virtual machines. The first case represents an entity transparently treated and used as a "disk" by a machine in storing data, but with no necessary correspondence

---

\(^8\)In proceedings: \cite{14}, website: \cite{44}
\(^9\)Previously described in \S\ 2.2.1
\(^10\)Official site: \cite{2}
\(^11\)Examples of distributed filesystems: Lustre\cite{27} and GlusterFS\cite{18}
between the cloud disk and some physical drive. In the second case an Operating System (OS) is run in an environment entirely constituted of emulated or virtualised hardware. The goals in adopting this technology are essentially two:

- **Resource emulation.** A virtual machine is a set of virtualised hardware resources used to run an operating system as it were hosted on a physical machine. To virtualise the hardware means to reproduce, at software level, the behaviours of real pieces of hardware. Therefore it is possible to mimic a Central Processing Unit (CPU) architecture that is substantially different from the one available in the underlying physical hardware hosting it.

- **Resource partition.** Single item of hardware resource can be partitioned into smaller virtual resources of the same kind: that is, a virtual machine can be configured with a subset of the physical CPU cores, or a physical disk can be configured with appropriate quotas to expose smaller "cloud disks" to the users, like Cernbox or other similar services do.

Many advantages in virtualisation technologies and virtual machines determined the popularity of this techniques. In particular two consequences made it possible to cloud infrastructures to grow in popularity. Two results are particularly relevant:

1. The possibility to partition a large server or supercomputer in many smaller virtual machines allows tasks with less requirements with respect

---

12 To be more specific, the correspondence may be missing already at the filesystem level, since a distributed "cloud disk" could be constituted of many (Linux) devices taken as "bricks". Devices are software partitions which abstract in turn a physical drive.

13 See: https://cernbox.cern.ch/
2.3. Computing in ALICE

Figure 2.4: Minimal scheme of a Cloud computing service setup. Virtual Machines (VMs) replicate the same underlying software overlay, spending resources to replicate virtual hardware and kernels underneath the actual applications.

to the whole resources available on the server to be executed in many parallel instances, optimising the machine utilisation, cutting the costs.

2. By executing virtual machines on physical hosts, one can effectively run different OS on each virtual machine. This has two advantages: the first is the possibility to provide a homogeneous environment on top of a heterogeneous substrate of physical hardware and/or OS, decoupling management, compatibility and monitoring of complex hardware from the level of the applications. The second is the possibility to create portable template services encapsulated inside virtual machines. This dramatically eases the service provisioning for cloud providers and creates a separation between the administrative domains, allowing cloud users to freely configure and administer their resources from inside, without interfere with the underlying hardware infrastructure.

These two features together make it possible to safely provide virtual resources as services under the form of Software as a Service (SaaS) or Platform as a Service (PaaS) whenever an entire framework is provided. This is obtained by granting isolation between different instances, whenever it is required.

The other side of the medal is that this software abstraction represents a real challenge in matter of performances as it basically requires to reproduce a set of hardware functions at software level.

It has to be pointed out that the development of virtualisation and cloud technologies with respect to the Grid technology has been carried out by a wider community and with many different aims. This led to a growing idea

---

14 e.g. even granting superuser permissions to users on a virtual machine does not allow those users to affect any other VM on the cloud unless it is explicitly permitted.

15 Both the Grid and the main virtualisation solutions are open-source projects.
of cloud computing, with a solid standardised set of tools, often inspired to
decisions taken by big companies (e.g. Amazon and its cloud Application
Programming Interface (API) called EC2). Even though cloud computing is
mainly oriented to service-provisioning (such as databases, web-servers, mail
servers, etc.) in high energy physics (HEP) it is also possible to create virtual machines to be used as worker nodes (e.g. Grid nodes). The advantage is that the remaining middleware can be virtualised as well, creating the whole environment under controlled conditions.

2.3.2 Cloud Computing in ALICE

In the present ALICE computing model there are no explicit Tier-3 sites but
several national or regional Analysis Facilities (AFs) providing a service processing subsets of ALICE data of specific interest to certain communities.

While the Grid has served the purpose well for the processing of Run 1 data for experiments at the LHC some sites are gradually embracing the model of private clouds and the virtualisation of their physical computing resources. This sounds natural, since when the Grid was first conceived the challenges of LHC were pursued mostly by the scientific community. Nowadays, however, the need for distributed computing is more common in the most diverse scenarios, from the single industry to the huge Big Data facilities and often it may happen that a single (cloud) centre harmonises resource provisioning among services exposed to different "customers". That reflects an evolution that sometimes witnesses a Tier(1-2) centre to become a specific use-case within a cloud infrastructure, alongside services for local groups (like Tier-3 services) or even for private customers.

This process is also underway at CERN and at several big WLCG Tier-1 sites; it has not introduced a shift in the Grid distributing computing paradigm already in place, since the Grid computing is run as a use case inside Virtual Machines (VMs). Using the virtualisation technology, each experiment will be able to construct its own computing cluster, spanning various Tiers and regional clouds, while continuing to use most of the existing distributed computing infrastructure already built and tested during Runs 1 and 2.

2.3.3 The O² program for the upgrade

In 2020, ALICE will start collecting data with an upgraded detector. The main Physics topics addressed by the upgrade are characterised by a very small signal-to-background ratio requiring very large statistics. This large background makes triggering techniques very inefficient, if not impossible. In order to keep up with the 50 kHz interaction rate, the TPC will also require the

\footnote{e.g. This is the case of "Centro di Calcolo di Torino" which hosts, alongside Tier-2 services, a Virtual Analysis Facility (VAF) and service for third-party customers.}
implementation of a continuous read-out workflow in order to deal with event pile-up and avoid a trigger-generated dead time. This will be significantly more challenging for the online and offline computing systems, compared to Runs 1 and 2. The overall data throughput from the detector has been estimated to be greater than 1 TB/s for Pb–Pb events, two orders of magnitude more than in Run 1. The Computing Model for Runs 3 (and 4) is designed for a maximal reduction of the data volume read-out from the detector as early as possible during the data flow. The zero-suppressed data of all collisions will be shipped to the O² facility with an interaction rate of 50 kHz for Pb–Pb collisions and 200 kHz for pp and p–Pb collisions. Detector read-out will be activated either by a minimum bias trigger or in a continuous mode. The data volume reduction will be achieved by reconstructing the data in several steps synchronously with data taking. For example, the raw data of the TPC (the largest contributor to the data volume 92.5%) will first be rapidly reconstructed using online cluster finding and a first fast tracking using an early calibration based on average running conditions. Data produced during this stage will be stored temporarily. The second reconstruction stage will be performed asynchronously, using the final calibration in order to reach the required data quality. The O² facility will perform both the synchronous and asynchronous reconstruction during pp data taking. The asynchronous reconstruction will process data archived at the O² facility or stored at the Tier-0 and Tier-1s that can provide archiving capability. The O² facility will be a high-throughput system which will include heterogeneous computing platforms similar to many high performance computing centres. The computing nodes will be equipped with hardware acceleration. The framework will also support a concurrent computing model for a wide spectrum of computing facilities, ranging from laptops to the complete O² system. Open source software conforming to open standards will be used as much as possible, either for the development tools or as a baseline for the framework itself\textsuperscript{17}.

Such an ambitious upgrade plan sets a lot of challenges in the design phase. The process of technology survey must consider every possible new scenario, machinery, or paradigm that could satisfy new goals and new design requirements. In the same way a detector (e.g. the ITS) will be completely replaced by a new one, with a new layout and with silicon detector technologies implemented, setup and design, even the computing and software sector will be refurbished and this will take into account all the bleeding-edge development in this sector, both in a hardware perspective (new and more performant architectures, CPUs, Graphic Processing Units (GPUs)) and a software perspective (new computing frameworks and languages, latest virtualisation technologies, protocols, paradigms, etc.).

\textsuperscript{17}Largely based on the Technical Design Report for the O²: \cite{6}.
The first part of the work presented in this thesis well fits this context. Its aim is to build a volunteer distributed computing infrastructure based on Linux containers, facing two particular topics: to port a distributed computing paradigm to a general-purpose setup constituted of commodity workstations in a controlled network environment and test new solutions "behind the times" and their compatibility with the HEP computing use-case.

2.4 Volunteer computing in High Energy Physics

2.4.1 State of the art

"Volunteer computing" is a type of distributed computing in which computer owners can donate their spare computing resources (processing power, storage and Internet connection) to one or more research projects. The constant increase for computing resources always encouraged the scientific communities to research and develop newer and more efficient frameworks for volunteer computing. Therefore has been possible to involve a community of non-experts which, participating to a volunteer project, in turn receive the possibility of feeling part of a broad scientific research project by donating their spare CPU cycles. Historically, the most famous platform is the Berkeley Open Infrastructures for Network Computing (BOINC)\textsuperscript{18} project, whose aim is to expose user interface to contribute to many projects listed, emulating the SETI@home\textsuperscript{19} approach, first in this sector. In 2004 the LHC came up with the LHC@home\textsuperscript{20} platform. Thus, through its SixTrack project, the LHC@home platform harnessed the power of volunteer computing to model the progress of subatomic particles traveling at nearly the speed of light around the LHC at CERN. It is still running today, and is now also being used to carry out simulations relevant to the design of the next phase of the LHC, known as the High-Luminosity LHC. Following its example many other experiments at CERN but also in many other different fields of research, developed platforms within the BOINC framework, increasing the popularity of this tool. Volunteer computing plays an important role in the computing panorama. It can be implemented in many ways, depending both on what public is intended to involve and who is the recipient of the service. In almost the entirety of cases "volunteer computing" implies an opportunistic computing model underneath.

\textsuperscript{18}Later migrated to a fully virtualised computing approach called V-BOINC\textsuperscript{29}.
\textsuperscript{19}Online: \[38\].
\textsuperscript{20}Online: \[24\].
2.5 Opportunistic distributed computing

Opportunistic distributed computing refers to a scenario where underutilized computing resources (CPUs, GPUs, etc.) are re-organised and clustered into a distributed computing service, making them available for job execution by third-party users whenever they are not busy in the execution of their main roles.

The fundamental idea is to create a service that is structured similarly to the large dedicated computing clusters previously mentioned, with the exception that the computing resources employed are likely not full-time dedicated nor performant and reliable as the supercomputers adopted in Grid sites and analysis facilities. Since volunteer hosts are managed and occasionally used by their legitimate owners, there is an additional part of this computing model to be developed which is related to the detection of a user interaction with the host owned. Also it is necessary to elaborate a strategy that determines the behaviour of the single opportunistic instance, depending on how it is designed. A volunteer computing model must grant an unconditional use by the volunteer of his/her own resources, naturally prioritising the owners against the parasitic use. Consequently, an opportunistic implementation of this paradigm must guarantee a configurable answer following a sudden reclaim of resources, being ready to safely release any resource parasitically occupied, granting the agreed computing power quota to its owner. In designing a volunteer application, some effort must be spent in figuring out which resource parameters must be taken into account and how to implement an opportunistic behaviour. A good implementation of the volunteer cluster paradigm should not leave to users the perception of the underlying structure of the service nor to the volunteers the sensation of inertia and viscosity in performance whenever they use their computers while they are participating in the volunteer project.

Other characteristics points in this model are the security and privacy pledges. At least two actors are involved in the most basic scenario: the volunteer and the end-user. The volunteer lends a workstation (resource) to the cause of volunteer computing, the end-user is anybody who benefits from the service. From a developer perspective, it is very important to consider the aspects related to the isolation of the opportunistic processes on the host, allowing no harmful degree of freedom left to a malicious end-user. On the other side, the developer should guarantee to the same end-user the least possible freedom to external agents –whether they are other running jobs or users on the volunteer workstation– to affect the execution of jobs, in order to preserve the integrity and the authenticity of the results eventually obtained.

Being slightly more technical, what is necessary to provide is isolation
for the jobs, and a **low impact** on the **Personal Computers (PCs)**\(^{21}\) in matter of resource binding (network interfaces, filesystems, additional packages). These problems are managed just fine, in cloud facilities by encapsulating every requirement inside the image of a **VM** in order not to install any further software on the hosts other than an **hypervisor**. Unfortunately there are some difficulties to import this schema *as is*. Virtual machines are a kind of heavy software, with a not negligible initialisation time (VMs boot as well as a physical computer!). In an opportunistic scenario with poor resources together with a generally non-predictable usage pattern on each host, the risk is to spend more time in booting and draining virtual machines than doing the actual computation. Also to **pause/stop VM** is not an option since it is not guaranteed that the running jobs are always **check-pointable**. It will be demonstrated why and how the most of these issues are overcome by a **fire-and-forget** computing approach based on Linux containers.

It must be anticipated that there is a *yet-not-solved* point of weakness in the volunteer computing model: the volunteer hosts are **not administered nor owned by the service provider**. This constitutes an unsolvable peculiarity in this approach, which is intrinsic in the definition of **volunteer computing** itself. **Opportunistic computing** can be done both on administered (dedicated) and volunteer hosts but **volunteer computing** itself does imply not to have full control over hosts. Full isolation and complete independence of the volunteer tasks from the environment where they are executed is a utopia, neglecting that there is always an external administrator with full rights over every instance running on the host. In any other case the hypothesis of volunteer computing collapses in a simplified opportunistic-only model. The consequence is that this boundary condition dictates the limits of this approach and, at the same time, emphasises the importance to create a network of trusted nodes or at least methods to validate final results of computations, in order to exclude the consequence of external intrusions or corruptions.

Therefore, in the particular case of this project, in the design phase has been given much attention in being as less invasive as possible on the **parasitised** workstations. In doing so, some new technologies has been investigated and implemented, evaluating **pros and cons**, and sometimes reverting decisions taken in the very first design. The most relevant milestones will be anyway reported explaining rationales and discussing consequences. Naturally it has been avoided to **reinvent the wheel** in most of cases, adopting a number of techniques and software solutions already existing, staying mainstream whenever

---

\(^{21}\) The less host resources and softwares are explicitly used, the less impact on the host is given, the less **backdoors** are exposed.

\(^{22}\) With respect to dedicated services where, by the way, this constitutes the same efficiency slowing downs issues.

\(^{23}\) The act to blank/reset a **VM** inner environment in order to accomplish new job executions without throwing away a running **VM**
it was possible.

2.5.1 A volunteer computing project for ALICE Torino

The work presented in this thesis is not strictly related to any of the projects cited in §2.4.1. Also the actors are slightly different: this project has been conceived to be implemented on a controlled environment within a trusted network and based on a trusted pool of general purpose workstations owned by members of the ALICE Group in Torino. That is the end-users and the volunteers are represented by the same group of people. Is therefore hard to expect malicious interventions on the service carried out by somebody who potentially is using the same infrastructure. The original intention was to aggregate spare resources and expose them as a distributed computing service with a broader potential with respect to the single components. With such an infrastructure a subset\(^\text{24}\) of the daily computing tasks of a generic ALICE physicist can be executed. In the next chapter I will present an approach to this topic, based on some of the newest mainstream technologies in the field of computing in \textit{HEP} and light-virtualisation solutions. Later on I will demonstrate how this approach is designed to horizontally scale without adding more complexity to the structure and how this work, originally tailored to fit the needs of the generic member of the ALICE-Torino group can be in general exported to different areas, not only \textit{HEP}-related. The second part of this thesis will demonstrate how this opportunistic and volunteer project can well-fit into a dedicated environment\(^\text{25}\) providing the sufficient backend to a Grid-like use case.

\(^{24}\)Due to the boundary conditions of this scenario not every job could be efficiently executed on this service, it will be better explained later why.

\(^{25}\)With the attribute "dedicated" here it is intended a service made of machines and networks employed full-time in \textit{HTC} and/or \textit{HPC}. 
An opportunistic computing service based on Docker

3.1 Linux Containers

A Linux Container (LXC) is an operating-system-level virtualisation method\textsuperscript{1} for running multiple isolated Linux systems on a single control host (LXC host). Thanks to the Linux namespaces it is possible to create a virtual environment has its own CPU, memory, block I/O, network, space and the resource control mechanism. A namespace is an abstraction that wraps a global system resource and exposes it to processes. Processes inside the same namespace see the same instance of that resource, but processes belonging to different namespaces see the equivalent of different instances, even if those instances are embodying the a resource supposed to be unique within the operating system. Therefore, instead of creating a full-fledged virtual machine, a Linux container relies on the underlying host kernel cgroups functionalities. The net result is the possibility to exploit such environments to create a consistent ecosystem where relegate application instances which can be agnostic to the underneath OS but not of the hosting kernel. The three main representative interfaces are Linux-VServer \textsuperscript{39}, OpenVZ \textsuperscript{32}, and Linux Container (LXC)\textsuperscript{28}.

\textsuperscript{1}Container-based virtualisation exists at the operating system level, thus allowing multiple applications to operate without redundantly running other operating system kernels on the host.
3.1.1 Behind the scene: cgroups and namespaces

The Linux kernel\(^2\) (starting from version 2.6.24) provides Control Groups \(\text{cgroups}\) a mechanism for aggregating/partitioning sets of tasks and all their future children into hierarchical groups, with specialised behaviours.

A \text{cgroup} associates a set of tasks with a set of parameters for one or more subsystems. A \text{subsystem} is a module that makes use of the task grouping facilities provided by \text{cgroups} to treat groups of tasks in particular ways. A subsystem is typically a "resource controller" that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes, e.g. a virtualisation subsystem.

A \text{hierarchy} is a set of \text{cgroups} arranged in a tree, such that every task in the system is in exactly one of the \text{cgroups} in the hierarchy, and a set of subsystems; each subsystem has system-specific state attached to each \text{cgroup} in the hierarchy. Each hierarchy has an instance of the \text{cgroup} virtual filesystem associated with it. At any one time there may be multiple active hierarchies of task \text{cgroups}. Each hierarchy is a partition of all tasks in the system.

User-level code may create and destroy \text{cgroups} by name in an instance of the cgroup virtual file system, specify and query to which cgroup a task is assigned, and list the task PIDs assigned to a cgroup. Those creations and assignments only affect the hierarchy associated with that instance of the cgroup file system. On their own, the only use for cgroups is for simple job tracking. The intention is that other subsystems hook into the generic \text{cgroup} support to provide new attributes for cgroups, such as accounting/limiting.
3.1. Linux Containers

the resources which can be accessed by processes in a cgroup. For example, cpuset[1] permit to associate a set of CPUs and a set of memory nodes with the tasks in each cgroup[2].

LXC combines kernel’s cgroups and support for isolated (kernel) namespaces to provide an isolated environment for applications.

3.1.2 A chroot "on steroids"

To make a chroot means to change the apparent root ("/") directory for the current running process and their children[3]. A program that is run in such a modified environment cannot access files and commands outside that environmental directory tree. This modified environment is called a chroot jail[4]. A Linux container is similar to a chroot[5] but offers much more isolation.

```
[user@host]$ lsb_release -ar
LSB Version: :core-4.1-amd64: ...
Distributor ID: CentOS
Description: CentOS Linux release 7.2.1511
Release: 7.2.1511
Codename: Core

[root@container]# lsb_release -ar
LSB Version: :core-4.1-amd64: ...
Distributor ID: Fedora
Description: Fedora release 24
Release: 24
Codename: TwentyFour

[user@host]$ uname -r
3.10.0-327.22.2.el7.x86_64
[root@container]# uname -r
3.10.0-327.22.2.e17.x86_64
```

Figure 3.2: Comparison between the same commands run from inside (host) and outside a (Docker) container.

The possibility of create many coexisting different namespaces for Process Identifiers (PIDs) allows tools known as "container engines" to instantiate a Linux container with inside a new process tree that fully mimics a complete one. That is, a container must have its own PID 1 (just like init or systemd) which owns a fully replicated process tree, appearing to be into an actual other environment (OS), relying on the same kernel.

3.1.3 Linux Capabilities

On Linux –and other UNIX-like operating systems– administrator privileges are held by the "root" user. Since no safe utilisation of a Linux machine foresee to run every program by root but, at the same time, some programs available to standard unprivileged users do require to perform critical actions only available for administrators, it came necessary to fragments the "power" for the root user into many capabilities in order to allow some processes whose permissions

[2]Processes spawned by a parent process which is responsible of a set tasks on their execution. [more: http://man7.org/linux/man-pages/man2/fork.2.html]
[3]It is sometimes colloquially referred as a chroot on steroids due to the further features it exposes
Chapter 3. An opportunistic computing service based on Docker

were effectively an overkill with respect to the task to be accomplished. This was achieved introducing the so-called: "Linux Capabilities". This kernel feature was implemented starting from kernel version 2.2, becoming soon the mainstream practice in most of the distributions. An explanatory example is in the /bin/ping command. It does need the possibility to open a raw socket; therefore the CAP_NET_RAW was introduced for that, fitting the original intent of the application's requirements and practicing the "principle of least privilege".

It is therefore possible to run Linux containers with a dummy root user inside, whose degrees of freedoms with the kernel interactions are managed with a good granularity through the Linux capabilities. Toggling all the capabilities for the root user in a container is equivalent to be superuser on a subsystem that belongs to a specified hierarchy, with the maximum permissions as possible on it but not necessarily congruent to the root user on the host. In fact, this kind of limitation differs from the isolation granted by cgroups since it regulates the interactions with the underlying kernel.

3.2 The Docker engine

As previously briefly mentioned, containers are mere processes running under particular conditions. The features claimed by Linux containers are provided by toggling some kernel features and few more ingredients. The process of creating a properly configured container satisfying all the requirements needed by the target task might be very tricky. In fact, the large amounts of low level parameters, together with the difficulty to create a custom consistent environment in which processes will be encapsulated and executed, did not enhance the development of software solution based on Linux containers until some automated and simplified tools appeared on the market.

A large portion of the merit for growth in popularity of Linux containers goes to the Docker engine (henceforth only Docker). Notably, three main aspects make Docker a worthy tool to design, prototype and build application based on Linux(Docker) containers. Docker in fact, provides:

3.2.1 The Docker command line interface

Docker provides a "easy-to-use" Command Line Interface (CLI). It is very straightforward and intuitive to run simple "containerised" sessions/applications with Docker. This interface is implemented through a client-server Scheme. The server is the "Docker daemon", a long running program that creates and

---

[^6]: Also to meet the set of security issues related to the exploitation of setuid-ed process
[^7]: http://man7.org/linux/man-pages/man7/capabilities.7.html
[^8]: Exhaustive documentation is available here: [12]
manages Docker objects, such as images, containers, networks, and data volumes.

A very immediate instant gratification example might be:

```
[user@host]$ docker run 
> --rm 
> -it 
> ubuntu:16.04 
> /bin/bash
```

```
root@<hash>:/# echo $$
1
```

The `docker run` command wraps some operations to enhance the final usability. A Docker container is run starting from an image, a read-only template with instructions for creating a Docker container. For instance, an image might contain an Ubuntu operating system with a web server and a web application installed. One can build or update images from scratch or download and use images created by others. An image may be based on, or may extend, one or more other images, the mechanism underneath will be briefly mentioned later. Docker images are stored in a docker registry, a library of images. A registry can be public or private, and can be on the same server as the Docker daemon or Docker client, or on a totally separate server.

In the previous example the image of the container would have been automatically downloaded if it was not already cached into the system. Moreover, Docker grants a fast access for all of those degrees of customisation specific of a container, for example the ability to mount host filesystems with a quick setup of permissions, device exposure (e.g. `/dev/fuse`) and to interface binding. The access to the docker daemon is granted only to users belonging to a specified docker Linuxgroup, due to the sensitive degrees of freedom accessible.

![Scheme of the Docker architecture](image)

**Figure 3.3:** Scheme of the Docker architecture [12].

---

Though it requires a set of further tweaks on capabilities and privileges.
3.2.2 A "Layered" approach to container filesystem

Docker provides an efficient method to organise the filesystem of a container. It adopts the union mount technology to build custom images on very lightweight basic images. Those images are "distro-flavoured" thin files represented by a read-only filesystem layer. Every modification (e.g. the creation or deletion of files, the setup of an environment variable, ...) is recorded on top of a writable layer put on top of the read-only one\textsuperscript{10}. Those layers are "union-mounted", hence exposed to host as a single one, allowing contributions to files and directories recorded on separate file system layers –known as branches– to be transparently overlaid, forming a single transparent one. Docker adopts a filesystem of this kind, called OverlayFS; newer versions are compatible also with OverlayFS-2. Once a container is run starting from an image, every modification is recorded; is hence possible to "snapshot" the equivalent image of that container by simply adding the written layer on the latest read-only one of the starting image. Therefore it is possible to create new custom container images from the default one, then they are "saved" and exportable as new images. To update an existing image Docker just pulls every layer starting from the first whose contents do not match with the existing image.

This layerised approach speeds up the distribution of Docker images, unburdening image maintenance procedures. Moreover, Docker determines which layers need to be updated at launch time.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{overlayfs.png}
\caption{Scheme of the OverlayFS structure, the Docker image corresponds to the read-only layer also referred as lowerdir, file created into a running container instance are written on the upperdir. The whole filesystem "tree" is mounted at once into a merged one \textsuperscript{11}.}
\end{figure}

Docker presents also a structured and robust method to describe how to construct container layers by possibly starting from an existing layer: this method is achieved through manifest files called Dockerfiles.

A Dockerfile is a text document that contains all the commands (see table\textsuperscript{3.1}) a user could call on the command line to assemble an image. Docker provide a tool –the command docker build– that executes the instructions

\textsuperscript{10}Presented at: \textsuperscript{22}

\textsuperscript{11}In finalising every layer, it is calculated an hash code that summarises the informations added. This eases the image management by Docker, that needs to check the equality between hashes instead direct compare the content of the layers.
3.2. The Docker engine

In a Dockerfile, it is possible to build an image step-by-step testing any change and checkpointing it. This constitutes—also in the case of this thesis work—a dramatic encouragement and acceleration in the development and debug phase of container-based applications. In fact, every command in a Dockerfile corresponds to a new layer on the image stack; whenever a command changes or its arguments change, the Docker builder has to re-execute only the commands whose layers are above the one corresponding to that command, leaving intact the underneath substrate, saving a lot of redundant operations. The only disadvantage in this is a number of layers of the created image. A multilayered image is far bigger than one obtained concatenating all the shell commands (the ones executed by \texttt{CMD}) into a single pipeline. Once the image built is proven to work properly, it is a good practice to pack all the commands into longer shell pipelines, taking care to regroup all the configuration phases (prerequisites, environment definitions, compilations) into homogeneous macro-sectors, minimising the number of layers in the final image.\textsuperscript{12}

### Table 3.1: Main commands available in the definition of a Dockerfile.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>Specify the base image.</td>
</tr>
<tr>
<td>MAINTAINER</td>
<td>Specify the maintainer of the image.</td>
</tr>
<tr>
<td>RUN</td>
<td>Run a command.</td>
</tr>
<tr>
<td>ADD</td>
<td>Add a file or directory.</td>
</tr>
<tr>
<td>ENV</td>
<td>Set an environment variable.</td>
</tr>
<tr>
<td>CMD</td>
<td>Run a program at launch.</td>
</tr>
<tr>
<td>ENTRYPOINT</td>
<td>Set an executable as entrypoint.</td>
</tr>
</tbody>
</table>

### 3.2.3 The Docker APIs

The Docker daemon exposes a \texttt{REpresentational State Transfer (REST)} API based on communications with a \textit{UNIX socket}\textsuperscript{13} through the \texttt{HyperText Transfer Protocol (HTTP)}. In short, it can both send and receive informations "listening" on a bound filesystem pathname, treating it in the same fashion as it were a web socket. Communications can be tested with ordinary \texttt{UNIX/Linux} tools—like \texttt{cURL}, \texttt{socat}, \texttt{nc}—just setting a fake \texttt{Universal Resource Locator (URL)} that points to the socket path. Supposing to find the

\textsuperscript{12}An explanatory example of a Dockerfile produced during this work is presented in Appendix \textsuperscript{5.3}

\textsuperscript{13}More info: \url{man7.org/linux/man-pages/man7/unix.7.html}
socket "docker.sock" in a standard location and to use a cURL version greater ≥ 7.40, one can run:

```bash
$> curl --unix-socket /var/run/docker.sock http://localhost/images/json
```

It will return a buffered output in the JavaScript Object Notation (JSON) format. In such a way it is possible to write programs\(^{14}\) able to interact with the docker daemon in a standard and safe fashion. On top of this "raw" approach many libraries and modules were built, using the most common programming languages\(^{15}\). In this thesis I adopted the Python version, called docker-py\(^{42}\) – which in turn relies on the solid and powerful requests Python library – that transparently provide an easy interface for programming, and maps every API call into a Python method, preserving the original effectiveness, without adding any further abstraction nor complexity.

---

\(^{14}\)Docker client itself relies on the Docker API.

\(^{15}\)Even though none is officially supported by the Docker team.
3.3 The Plancton daemon

3.3.1 An automated Python script for container scheduling

For the scope of this thesis a "missing" software component, not found in the panorama of mainstream applications in 2015, is represented by the Plancton\(^\text{10}\) daemon, a software I wrote entirely from scratch, extending a simple Python class for daemon interfaces.\(^\text{16}\) Plancton is a fully configurable daemon written in Python whose aim is to automatise the process of opportunistic spawning of Docker containers baptised in this context "plancton-workers". Plancton has been designed to be simple, customisable and robust. It lies at the same software level of the Docker daemon and uses its API to manage a set of containers. Its behaviour is fully determinable through an intuitive configuration file written with the YAML\(^\text{19}\) markup language.

3.3.2 Design guidelines

This work started with the goal of building a volunteer cluster within the ALICE group of Torino, therefore particular care was taken to the installation procedure, in a way that it can be performed ideally even by the most inexperienced user. Therefore, special focus was given to ensure that always looks primarily to the volunteer approach, whose traits are depicted in §2.5. Henceforth I will describe how I satisfied the guidelines of this paradigm, creating this application in order to be:

- EASY-TO-INSTALL and uninstall. The less effort as possible –hence the less probability to be error-prone– must be leased to the volunteer. The installation procedure evolved during the developing phase, staging mainly to two procedures. The first was in the form of a simple Bourne Again SHell (BASH) script downloaded from an online "<gitrepo>" repository and executed on-the-fly through an one-line-command as superuser:

\[
\text{# bash -x \langle(curl https://<git-repo>/mconcas/plancton/master/install) \langleproject-tag>}
\]

The last tag refers to the name of a specific setup configuration. For instance, to-infn/dev refers to the volunteer project in Torino. This implementation used to work properly; however, in a later phase of the development when the core features of the software were proven to work as expected, the daemon was heavily "refactorised" and improved, as also its installation procedure was rethought.

\(^\text{10}\)The original code is found at: \url{jejik.com/files/examples/daemon3x.py}, it was then heavily improved.

\(^\text{16}\)A daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user.

\(^\text{19}\)Yet Another Markup Language or also YAML Ain’t Markup Language.
The second and current in use installation procedure is probably more familiar—certainly more standardised—. Plancton is now installable through the **Python Install Package (pip)** utility, as a Python module. It is also published on the **Python Package Index (PyPI)**\(^{20}\). In this case the installation takes two commands:

\[
\$ \text{pip install plancton}
\]

\[
\# \text{plancton-bootstrap <project-tag>}
\]

Through the first command the Plancton Python module is installed (to install it system-wide it is necessary to run that command as superuser). The second command is needed to bootstrap the installation, applying configurations located in the repository pointed by the `<project-tag>`. In both procedures few needed prerequisites (git, the docker-py module, and Docker) are checked. Whenever some package is missing they are automatically installed. The second installation method, however, offers many advantages, for example it sets the right environment automatically, and is completely cross-platform, since it delegates to "pip" all the compatibility issues. The latter feature is also relevant when the heterogeneous population of different OSS installed on the volunteer hosts is considered, in the first case it was manually handled. Moreover, the uninstall is also managed by pip, provided that the Plancton daemon has been turned-off.

- **Lightweight.** The daemon has been written in order to use the less resources possible. That means it periodically accomplishes very simple tasks, listed into its main loop, mainly corresponding to light API calls. The duration of this loop is paced by interrupting its repetitions with sleep periods of configurable length. In such a way it is possible not to cause an excessive CPU overhead. In its default setup Plancton sleeps 30 seconds at the end of every loop, it is usually a good compromise between responsiveness and overhead.

- **Isolated, Robust and Up-to-date.** A user *ad hoc*, the **plancton** user, is created during the installation phase and added to the **docker** group. The Plancton (daemon) instance is run by that user, in order to protect the instance from every other unprivileged user active on the host. It is a mild but needed isolation, as in a volunteer configuration a real isolation from the administrator is never achievable. The program is also **stateless**, that is it does not rely on an ephemeral inner status containing runtime information. Hence, a sudden interruption of the service (reboot of the host, process kill, inner critical error) does not compromise the recovery

\(^{20}\)pypi.python.org/pypi/plancton/
3.3. **The Plancton daemon**

...In case of an update/restart of the Plancton daemon the running containers are not affected, as long as the Docker service remains unaltered. This is possible allowing the daemon to act only on a certain category of containers whose name prefix fits a particular sequence. The name of the worker containers spawned by Plancton, by default, is in the form of "plancton-slave-XXXXXX", where the last six characters are represented by a unique-generated hash. Every API call can act only on those containers starting with the "plancton-slave" prefix. It may happen that the docker daemon can be temporarily unresponsive to the API calls, for many reasons. To fix this I wrote a Python "decorator" (reported in Appendix: 1), a wrapper that force calls to be executed many times with an incremental delay, trying to improve the robustness of every call. It must be noted, for the records, that the Docker daemon heavily improved its performances in matter of responsiveness to "high frequency" API calls during the lifetime of this work. To overcome this issue I opted for a global "unburdening" of the load generated by reducing the number of calls made both by increasing the main loop sleep time and by pruning redundant queries. Plancton is designed to trap the SIGKILL and safely remove its lock-file, preserving the host from running multiple instances of the daemon simultaneously. A Cron job added to the Plancton user's crontab file restarts the daemon instance at reboot; thus, if meanwhile the Python module has been upgraded, it starts an instance with the new version.

- **EASY TO CONFIGURE and DEBUG.** Even though neither in the volunteer implementation nor in the dedicated one, the Plancton configuration file is supposed to be edited very often, Plancton periodically refreshes its rules of behaviour reading them from a config.yaml file. The file structure is very intuitive, it presents two stanzas, one for the daemon parameters and one for the container/docker parameters. In section §3.3.3 every voice will be better presented, what is worth to mention now is the fact that the daemon offers a set of default "dummy" parameters, to test with a dry-run if the setup is properly installed and working on a host. Those "defaults" are stored in a static dictionary inside the class definition and represent the whole set of configurations needed by a standard version. Note that, compared with the example of configuration file found in §5.3, the default internal dictionary shows some further key, since the defaults parameters are set to coherent values.

---

21 Processes communicate through signals [26].

22 More info about the cron daemon: [linux.die.net/man/8/cron](https://linux.die.net/man/8/cron)

23 It will be possible in the future to add a hourly-paced cron job to run the "update" and "restart" command in order to keep Plancton always at the latest version without ask to the volunteer to manually upgrade the daemon.
Plancton also provides a logging system. Toggling the preferred logging level it is possible to monitor every action made during the daemon lifecycle. This eases a lot the operations of debug both in development and in production. In the log-file a summarising table representing the status of its containers is reported among many other informations.

- **APPLICATION AGNOSTIC.** Out of the box, Plancton can be bootstrapped with a dummy configuration that does not carry particular settings nor meanings as well. By design, it is not bound to a specific use-case. It simply follows a set of rules to schedule and manage a set of containers. The isolation with the environment of a container is not broken even by the daemon, which remains at a separate level from the applications. From its perspective there could be any application running inside the containers it spawns. This very useful feature makes this software exportable in distributed computing fields even far from the high-energy physics ones. Plancton does not provide any high-level job management system, nor can communicate with other daemon instances on the network, it "blindly" spawns computing resources without any clue of the state of the whole computing network. It relies only on the information and configurations available on the host on which Linux runs. In realising a distributed computing cluster of Linux containers it is required a minimal batch queuing system (scheduler+collector), as it will be better presented in chapter in the specific case of the "Torino volunteer facility". Plancton has not to be aware of the presence of the batch system, since it plays no role nor takes control over batch systems and their jobs.

- **SCALABLE.** This minimal uncomplicated architecture does not add any further hierarchy on top as long as new hosts join the set of running containers, allowing a Plancton-based cluster to scale as well as the batch-system on top does. This model fits well in the volunteer scenario since a Plancton instance does not listen to any external input and avoids undesired external "intrusions". Plancton containers are natted and cannot have any open ports. On the other hand no restriction is applied on outbound connectivity, whose security implications must be managed by the containerized use case.

---

<table>
<thead>
<tr>
<th>n</th>
<th>docker hash</th>
<th>status</th>
<th>docker name</th>
<th>pid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29763107a43d</td>
<td>up 4 seconds</td>
<td>plancton-slave-elobwy</td>
<td>7447</td>
</tr>
</tbody>
</table>

---

Technical verb to intend a method of remapping one Internet Protocol (IP) address space into another by modifying network address information in IP datagram packet headers while they are in transit across a traffic routing device.
3.3. The Plancton daemon

| 2 | 9c0377c72540 | up 5 seconds | plancton-slave-lebf1w | 7363 |
| 3 | a3ff97652b5 | up 7 seconds | plancton-slave-aeph8k | 7303 |
| 4 | 2066f8e38cd | up 8 seconds | plancton-slave-ayr1kb | 7242 |
| 5 | d171f4dd32e1 | exited (0) less than a second ago | plancton-slave-wed5pi | |
| 6 | 896c35a44a41 | exited (0) 2 seconds ago | plancton-slave-c11p3f | |
| 7 | 0ef02bc9c7d9 | exited (0) 3 seconds ago | plancton-slave-knu491 | |

---

Log File 3.1: This Table represents the jurisdiction of Plancton. It only shows containers starting with a fixed prefix. Docker does not (yet) provide a partitioned view of containers by discerning between different users/groups.

3.3.3 Plancton configuration

The Plancton configuration is probably the part of the project that has changed the most from its early stages. Its development was strongly influenced by the real tests of the daemon and modelled according to the minimal requirements needed to work. Indeed it represents the most sensible part in the program, and the most critical from the security perspective. Configuration is stored by default in a file called `config.yaml`, written in the YAML markup language, easy and intuitive to edit. A brand new Plancton installation comes with a specific `<project-tag>` that is a reference to a branch of GitHub repository that stores configurations. To test the Plancton setup it is also possible to "bootstrap" it with a sterile configuration that adopts a "dry-run" mode:

```bash
# plancton-bootstrap mconcas/plancton-conf:dryrun
```

Plancton reads the configuration and matches every field with a default hard-coded list. An example of configuration dictionary is presented in Fig. 3.2. A Python dictionary is a set of key-value couples. Configuration is parsed reading key by key, if the same name is found in the dictionary then the default value is overridden by the config one. In such a way not only it is possible to use default values (in general dictated by common sense) wherever they are not explicitly set, moreover it sets a minimum safeguard against wrong and damaging configurations.

```python
self.conf = {
    "updateconfig" : 60,  # frequency of config updates (s)
    "image_expiration" : 43200,  # frequency of image updates (s)
    "main_sleep" : 30,  # main loop sleep (s)
    "grace_kill" : 120,  # kill containers after that many s over CPU threshold
    "grace_spam" : 60,  # spam containers that many seconds after last kill
    "cpus_per_dock" : 1,  # number of CPUs per container (non-integer)
    "max_docks" : "ncpus - 2",  # expression to compute max number of containers
    "max_ttl" : 43200,  # max ttl for a container (default: 12 hours)
    "docker_image" : "busybox",  # Docker image: repository[:tag]
    "docker.Cmd" : "/bin/sleep 10",  # command to run (string or list)
    "docker_privileged" : False,  # give super privileges to the container
    "max_dock_mem" : 2000000000,  # maximum RAM memory per container (in bytes)
    "max_dock_swap" : 0,  # maximum swap per container (in bytes)
    "binds" : [],  # list of bind mounts (all in read-only)
}  
```
"devices" : [], # list of exposed devices
"capabilities" : [], # list of added capabilities (e.g. SYS_ADMIN)
"security_opts" : [] # list of security options (e.g. apparmor profile)
}

Code Snippet 3.2: Default configuration values for Plancton, with the corresponding explanations. The "busybox" image is the smallest docker image available on the registry, with the most minimal toolkit, ideal for tests.

The comments aside the pairs `key:value` better explain the meaning of the same. Most of them acquire meaning at the light of the previous sections. The last one, `security_opts`, is just a forward of the namesake docker command. Its importance is crucial in the harmonisation with Plancton and the Mandatory Access Controls (MACs) like SELinux (for RHEL-based Linux distributions) and Apparmor (Debian-based Linux distributions). It is hence possible to run containers in given custom "Linux contexts", a set of labels owned by Linux files and processes that contain additional information, such as a SELinux user, role, type, and, optionally, a level. When running SELinux (or similar Access Control agents), all of these informations are used to make access control decisions. The MAC when active, always has the final decision (even though Linux capabilities are permitting a certain operation to be executed) in accessing a kernel resource. A few more words worth to be spent on this topic; both with SELinux and Apparmor it is possible to add new "ad-hoc" contexts to the available contexts list. This is done with the proper interfaces, the important thing to mention is the fact that these operations have to be done manually by the root user. In principle this step could be included in the Plancton "bootstrap" process which, as said, is run as superuser, thus obfuscating a potentially disruptive operation in the installation phase. However it is not always required, therefore, depending on the use-case the MAC configuration is managed time by time.

### 3.3.4 Plancton lifecycle

Plancton has a extremely simplified structure. Its member functions robustly accomplish short tasks and most of the times they are "standalone sequences" that is they do not depend from each others. Therefore even if Plancton is suddenly stopped and restarted it is almost impossible to retrieve it into an inconsistent/inoperative state. It is thought to "pick up" an interrupted workflow (plancton-slaves still running or exited) and continue it without rest an existing one. Thus, every updated version of Plancton is immediatly backward-compatible.

---

25 Red Hat Enterprise Linux (RHEL)
The Plancton daemon

3.3. The Plancton daemon

The daemon is based on a **main_loop()** repeated periodically every time interval lasting "**main_sleep**" seconds.

1. The first method called is the **overhead_control()**, its duty is to measure the CPU usage (efficiency) and to compare it with a threshold not explicitly set in the configuration file but calculated indirectly from two parameters: "**max_docks**" and "**cpus_per_dock**". Such threshold is evaluated using the formula (3.1), where \( T \) is the threshold, \( N_{cpu} \) is the number of CPUs on the host, \( N_{running} \) is the number of running containers, \( M_d \) is the maximum number of containers that Plancton is configured to spawn (equivalent to the **max_docs** in configuration), \( C_d \) is the number of CPUs per container (corresponding to **cpus_per_dock** in configuration). The \( \delta \) is an arbitrary parameter that sets the percentage of CPU efficiency that has to be tolerated as the system might be running some background routine or some user could be using the host in a very mild fashion.

\[
T = \delta + 100 \times \frac{C_d}{N_{cpu}} \times \min (N_{running}, M_d) \tag{3.1}
\]

In case the number of running containers is greater than zero, the threshold \( T \) is proportional to the minimum value between the number of running containers and the maximum number of containers that can be run.

---

\( ^{26} \text{see the configuration parameter in (3.2).} \)
on the host. The scheduling system sometimes might find that, even though there are "max_docks" or more running containers and in case they are not running at 100% of their potential, there is enough space to spawn another container. This formula ensures that no more than "max_docks" containers will be left running simultaneously if their total amount of overhead would overcome the equivalent of "max_docks" containers running at 100%\textsuperscript{27}. Moreover it ensure that in case there were no running containers the threshold would be set to $\delta$. Henceforth the T value will be used to indicate the "equivalent threshold", since it is calculated and not clearly set in the configuration.

In every other case, if Plancton finds that efficiency exceeds the equivalent-threshold, it creates a list of the present containers ordered by date of creation. Afterwards, "grace_kill" seconds it starts to kill every container in the list, starting from the youngest, until the efficiency measured become lower than the equivalent-threshold.

2. Plancton then parses the configuration file. The frequency of this operation is punctuated by the "updateconfig" parameter. The choice to force the daemon to update its configuration is useful in debug and testing phase, to change parameters with agility, especially in a dedicated scenario where configuration on the hosts has to be massively updated simultaneously. During the "bootstrap" phase Plancton pulls the configuration from a repository, to force the update from repository Plancton must be bootstrapped again. This operation requires privileges, so that a common user on the host cannot highjack the procedure making Plancton to pull another repository. Moreover, in a default setup, the configuration file is hosted in \texttt{/etc/plancton/config.yaml}, a directory owned by the plancton user, not editable by unprivileged users.

3. After the configuration, the daemon asks docker to pull the container image specified in its dictionary, if it is not present. This operation is repeated every "image_expiration" seconds, to update the image, in case it has been changed.

4. In the fourth step it checks the occupancy of the host and starts containers whenever it is possible, according to its configuration.

5. A cleanup is performed later on, to get rid of possibly stale containers in case they were not spawned correctly.

\textsuperscript{27}With 100% it is always intended the CPU usage from the perspective if the container. Resources might be capped for the container, hence the host is not observing a 100% of total overhead.
6. The last step eventually dumps the status of the plancton pool into the log file, registering the same output shown in table 3.1.

Daemon operations have a negligible impact on resources consumption (< 0% measured with ps).

### 3.3.5 Resource monitoring and restraint system in Plancton

In order to make Plancton "cross-platform", the utility to measure the CPU efficiency has been written from scratch, using what is available on every standard Linux OS.

At start Plancton stores into two data-members ("uptime" and "idle-time") two values registered into the "/proc/uptime" pseudo-file system.\(^{28}\)

The idle-time is the sum of the idletimes of every core of the CPU, hence it is reported for one core only. The instantaneous \(n\)th efficiency measurement of a multi-core architecture, with \(u_n\) and \(i_n\) are being respectively the uptime and the idle-time measured the \(n\)th time, is therefore calculated with the formula:

\[
\varepsilon_n(\%) = \frac{[(u_n - u_{n-1})N_{\text{cores}} - (i_n - i_{n-1})]}{(u_n - u_{n-1})N_{\text{cores}}} \times 100
\]  

(3.2)

Indeed, the frequency of every efficiency update is equal to the "main_sleep" (\(\sim 10\)s). Therefore the term "instantaneous efficiency" might sound inappropriate, since it is the average of two distinct values but, for the sake of a standard use, this is close enough to an instantaneous measurement. It has been also evaluated the eventuality to take into account a diagnosis of the available memory (Random Access Memory (RAM)) usage while deciding if instantiate a container or not. The measure of RAM usage is controversial on Linux systems. Linux is optimised by design to fully exploit RAM even if there are no other user processes running. Background utilities and databases, like locatedb,\(^{29}\) periodically update their caches, and store them in RAM in order to gain the best performances on call. This is an optimal use of the resources, a glaring example of opportunistic usage; the only drawback is that when memory is inspected during these operations, is found full. Those are non-priority processes and opportunistic containers could be run in their place.

A good compromise between the only taking into account of the CPU usage and the creation of a complicated system to investigate memory usage with more granularity and accuracy, is to rely on the CPU usage and at the same time to limit the memory (using the cgroups features) available for each container in order to prevent bad memory-leaking job to saturate every available resource, affecting adjacent processes. In the specific cases presented in

\(^{28}\)It doesn’t contain "real" files but runtime system information, stored into the memory by the kernel

\(^{29}\)manpages.ubuntu.com/manpages/wily/man5/locatedb.5.html
Chapter 3. An opportunistic computing service based on Docker

In this thesis, containers are usually configured to use a single core. In such way it is possible to limit the resource exploited.

A few more words worth to be spent to describe how Docker acts in limiting resources available to containers. A container can be configured to use a fraction, called "cpu_share", normalised to 1024, of the whole CPU time. That fraction is an integer greater than 2 and it is not forced to correspond to an integer number of "cores" (e.g. a container can share a cpu_share quota corresponding to less than one core). However, the proportion is only applied when CPU-intensive processes are running. When tasks in one container are idle, other containers can use the left-over CPU time, ignoring their quota. The actual amount of CPU time will vary depending on the number of containers running on the system. On multi-core systems, the shares of CPU time are distributed over all CPU cores. Even if a container is limited to less than 100% of CPU time, it can use 100% of each individual CPU core. This feature may be useful to optimise the use of resources, for example is convenient for a running container to exploit all the remaining CPU time not sufficient to trigger the spawn of another container, otherwise it will stay idle. On the other side it makes more difficult to manage the scheduling of new containers, since a measure of cpu efficiency will likely find \( \sim 100\% \) of resource utilisation whenever a cpu-voracious job is executed.

To overcome this not-completely-predictable behaviour it is necessary to manually configure the kernel default scheduler, the Completely Fair Scheduler (CFS), setting two parameters:

- **cpu.cfs_quota_us**: the total available run-time within a period (in microseconds).
- **cpu.cfs_period_us**: the length of a period (in microseconds).

The idea behind this is to subdivide the CPU time shared by each process configuring the quota and period, manageable within the cpu subsystem via cgroups. In such a way it is possible to limit the CPU time spent in a task, in this case a container, hence preventing it to flood all the available resources making spawning legit containers impossible for Plancton. To better understand these two parameters let’s assume the default cpu_period for CFS: 100\( \mu \)s. The second parameter, cpu_quota, is the total amount of computing time available summed on the whole CPU. That is, to allow a process to use a fraction \( K \) of the CPU time, in a n-core system the cpu_quota must be:

\[
\text{cpu_quota} = K \times N_{\text{cores}} \times \text{cpu_period}
\]

\[30\] Original reference: docs.docker.com/engine/reference/run/#/cpu-period-constraint
\[31\] Available documentation: kernel.org/doc/Documentation/scheduler/sched-bwc.txt
3.3. The Plancton daemon

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>Ubuntu 16.04 LTS</td>
</tr>
<tr>
<td>cpu_shares</td>
<td>1024 [a.u.]</td>
</tr>
<tr>
<td>cpu_period</td>
<td>100000 [µs]</td>
</tr>
<tr>
<td>cpu_quota</td>
<td>400000 [µs]</td>
</tr>
</tbody>
</table>

Table 3.2: Parameters used in the test container.

Figure 3.7: An instance of htop running on the host container, the 8 processes fork from the stress main instance. The average CPU load percentage is about 50% for each process, due to the limitations imposed to the CFS. Notice that the owner is "root", since inside the container the UID associated is 0. Different kernel namespaces allow the OS to create multiple users with the same id in different spaces.

This feature became necessary in configuring a dedicated site, where the resource are allocated only for job execution, in order to grant the same resources to each job and always have the same number of running container on the single host. To give an idea of the effectiveness of this features some tests have been done on a 8-core (4 physical + 4 hyper-threads) machine, by running a stress instance inside a container with this setup: The test was a success, as can be verified in figure (3.7), hence the approach was implemented and these three features inserted in Plancton.

3.3.6 Plancton policies of scheduling

An automated method is used to establish if there are enough resources to instantiate new containers. The parameter "max_docks" sets the limit for the number of plancton-slaves that can be active at the same time on each host. The common-sense assumption made while negotiating the launch of a container is that it will run at 100% of its available resources. Plancton will launch a new container whenever the number of running ones is less than the maximum, and only when the amount of overhead that a new container will create (considering it running at 100% of its cpu_quota) added up to the

---

32 This utility is useful to induce overhead on the host CPU, RAM, Caching systems, etc. In this case the command run is `stress -c 8` to overload every cpu on the host.
current CPU usage, is still less than the "equivalent-threshold". Therefore the number of "fitting docks" $F_d$, with $C_{cont}$ cores each, is evaluated starting from the available CPU efficiency on the host and is described by this formula:

$$F_d(\varepsilon_{cpu}) = \text{int} \left( A(\%) \left(1 - \varepsilon_{cpu} \right) \times \frac{N_{cores}}{C_{cont} \times 100} \right)$$  \hspace{1cm} (3.4)

Where $A(\%)$ is an arbitrary coefficient, usually set to 0.95 to take into account the oscillations of the CPU efficiency due to unpredictable background processes rising. Its aim is to avoid creating containers to be removed later on because of a sudden tiny growth in the "physiological" utilisation of the host. It basically round-down the $F_d$ number of fitting containers. The term $1 - \varepsilon_{cpu}$ is the idle time percentage. After the number of fitting container has been evaluated, there is one more step to accomplish, that is to check whether the daemon is in a "grace_spawn" cool-down state or not, depicted in fig. (3.8). This "grace" period has been included in order to smoothly avoid Plancton to create a container just after having killed another one. This artifact is useful in managing those situations where the host usage is oscillating between the possibility to host another container and the slight overcome of the cpu efficiency threshold caused by a container in excess.

A similar behaviour has been implemented in the overhead management phase. The daemon cannot kill a container while in a "grace_kill" cool-down.
3.3.7 Plancton in a dedicated scenario

An intelligent management for volunteer opportunistic computing is seen, in this context, as a small set of rules imposed to Plancton, making it to react to some external and, in principle, not forecastable high-intensity periods caused by a random utilisation. Optimise the scheduling is tough, since it must be granted the highest job efficiency possible, allowing longer tasks to be executed and not abruptly interrupted regardless their progress state. Killing containers starting from the youngest is the criterion that allows the service to waste the less work done as possible, in the eventuality of thrashing some worker node to free resources. The configurable grace periods provide effective handles to configure the tolerance and they are left as free parameters of this model.

In Chapter 4 I will show how Plancton can handle the dedicated scenario as a special case of the volunteer one, by simply acting on the configuration parameters. The minimal working conditions for this tool foresee the daemon running on hosts equipped for HTC and continuously spawning containers as worker nodes, being in charge only to reap out the exited containers and to replace them with new ones, regardless of the surrounding environment. An opportunistic dedicated computing model can be seen as a volunteer opportunistic computing scenario with inexistent control on the external overhead.
4

Opportunistic computing with Plancton

4.1 Disposable pilots as worker nodes

Docker containers have a very short deploy time, compared with virtual machines. They do not have to boot any kernel and do not virtualise any hardware resource. A test has been conducted on a workstation HP ProDesk 490 G2 MT® (available as a commodity workstation, one of the candidates for participating in the volunteer project in ALICE Torino, the other candidates have similar hardware) to estimate the average deploy time of a centos:7 container. The probe command was run ten times preceded by the `time` system utility, as follows:

```bash
$ for i in {1..10}; do time -p docker run centos:7 /bin/bash; done;
```

In table 4.1 results for three different container flavours are presented. Images used are of different sizes, busybox is the lightest. As can be seen there are no evident correlations between sizes and spawning time. Values must just be taken as an example of the order of magnitude, which is in the order of seconds. The host presents the overlay of two different filesystem: XFS for the OS and OverlayFS (v1) for the container structure. A direct comparison with a virtual machine boot time has not be made, due to the many different ways a VM can be implemented. Since a VM has to boot an entire kernel

---

2Manual reference: [linux.die.net/man/1/time](http://linux.die.net/man/1/time)
3Site reference: [xfs.org/index.php/Main_Page](http://xfs.org/index.php/Main_Page)
Chapter 4. Opportunistic computing with Plancton

<table>
<thead>
<tr>
<th>n°</th>
<th>Image</th>
<th>Avg. $[\pm0.001]\text{s}$</th>
<th>$\sigma$ $[\pm0.001]\text{s}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>centos:7</td>
<td>1.785</td>
<td>0.094</td>
</tr>
<tr>
<td>10</td>
<td>ubuntu:16.04</td>
<td>1.633</td>
<td>0.063</td>
</tr>
<tr>
<td>10</td>
<td>busybox:latest</td>
<td>1.824</td>
<td>0.111</td>
</tr>
</tbody>
</table>

Table 4.1: Average elapsed spawning time for Docker containers.

and an OS the time spent in initialisation is greater of about two orders of magnitude (~mins) with respect to the container one.

This substantial difference permits to adopt new strategies in designing an opportunistic computing service. The idea behind Plancton is to adopt a fire-and-forget approach. That is, since the daemon has no clue on the nature of the applications running inside the containers, it can not schedule containers relying on the requirements for the application itself. Therefore, Plancton blindly and continuously spawns new containers wherever there are enough resources to exploit. Moreover once containers terminate their jobs, the daemon acts as a garbage collector, removing exited images as deeper explained in chapter 3.

An implementation based on Plancton assumes that containers are mere pilots.

A pilot is a container that comes up with a set Time To Live (TTL) and patiently waits for a job to land onto it. That is, inside the container there are some utilities that take care of communicating with the proper scheduler and receive the workload. In this sense Plancton is intended to be "application-agnostic". In case the scheduler does not deliver any task to a pilot, it "dies", leaving an exited container as residual. Plancton will manage to get rid of exited containers or not running ones with bad statuses. It is worth to notice how this logic cannot be implemented as well using VM instead, since their initialisation times are not short and the overhead caused during the boot time is not even comparable with the low overhead in starting a container, that is instantiate virtual machines without be absolutely sure that they will accomplish any task is a waste of resources and computing time absolutely greater than the time spent in quickly launching low overhead containers that self-terminate if the job queue is empty. The pilot approach does require some precise circumstances and is effective only in certain cases (traditional computing fits into the model), but on the other side it naturally provides some features such as horizontal scalability and dynamic control of running containers. In particular, the horizontal scalability is achieved considering that every Plancton instance does not need to be aware of the state of the ensemble of containers. New nodes join automatically the pool without any
external request by no master. That means a cluster can horizontally scale as much as the scheduler and the routine responsible to collect results on top do, regardless of the number of Plancton instances running simultaneously (one instance per host).

4.2 Software provisioning with CVMFS

Software used by physicist is always under development. In ALICE, for instance, everyday the AliRoot framework is patched with new methods and features currently in use by the collaboration to carry out the latest analyses or simulations. The official repository for the source code for AliRoot is hosted on the web. Therefore, at the end of the day, every finalised piece of code is committed by the authors and through an half-automated check procedure is validated and authorised to be merged with the tagged development branch of the day. Then, a fully automated build system creates packages and binaries for the most used Linux and macOS flavours and publish them on the internet. This workflow is quite consolidated and used by every experiment at CERN, thus the EP-SFT department at CERN developed a common system to publish and so redistribute every daily tagged version. This network filesystem is called CernVM Filesystem (CVMFS) and its structure is based on the HTTP protocol.

Without entering in details that lay outside the topic of this chapter, a word on the architecture of CVMFS can be spent to better understand its working mechanism. CVMFS provides a scalable, reliable and low-maintenance software distribution service. At the ground level of the distribution tree there is a central repository, called Stratum-0 (Zero), where authorised people...
upload (publish) contents. Then there is then a series of mirrors that synchronise the contents of the stratum-zero, called "stratum-1". It uses outgoing HTTP connections only, thereby it can comply with firewall definitions, moreover it easily caches using mainstream proxy caching services. It transfers data and meta-data on demand and verifies data integrity through cryptographic hashes.

CernVM-FS is implemented as a POSIX read-only file system in user space (a Filesystem in User SpacE (FUSE) module). Files and directories are hosted on standard web servers and mounted in the universal namespace /cvmfs. In such a way it is possible to provide multiple software versions without installing or compiling new programs every time, but simply downloading the right one every time. CVMFS was made to distribute externally (with respect to VMs, or in this case, containers) some software whose lifecycle is quicker than the VMs/containers themselves, by bearing in mind that it is not necessary to distribute all of it everywhere. CVMFS downloads only the single files that are accessed, and caches them, solving the problem of creating consistent VM/container images.

In the case of the ALICE framework, this choice is arbitrate by the "alienv" script, which allows to enter to a transparent environment where the version of AliRoot and other components are specified at startup. Therefore CVMFS is the common denominator in these two implementations, even if the access to it is obtained in two different fashions.

4.2.1 Job orchestration with HTCondor

Another common denominator in both the use-cases presented is the HTCondor framework, even if it acts at different points in the two scenarios. HTCondor is a specialised workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion. HTCondor has been created with the aim to orchestrate opportunistic clusters, therefore it naturally harmonises with the resource provisioning done by Plancton. In the next two sections it will be presented how HTCondor has been included in the two use-cases.

4Source code available on GitHub: [github.com/alisw/alibuild/blob/master/alienv](https://github.com/alisw/alibuild/blob/master/alienv)
4.3 A volunteer computing facility at ALICE Torino

The original concept behind the Plancton daemon is related to the design and setup of a volunteer facility within the ALICE in Torino. Plancton was conceived and modelled to suit this use case, once reached this goal a new use case has been tested, simply tuning its configuration to suit an HTC scenario.

The primary goal was to create a cluster able to run some small physics Monte Carlo simulations, useful for quality-controlling development code, in order to provide a prompt "testbed" for unplanned tasks. The capabilities of this infrastructure are limited by design, since tasks more similar to the ones running on the Grid require a lot of resources such as storage, bandwidth, memory and computing power. However those resource would still be under-used, even though the machines would stay powered on all the time. Hence, this use case is very helpful to refine and stress the whole framework, in the perspective of further developments. Therefore, alongside the results and the tasks accomplished, know how, crucial for further developments in matter of job emulation with Linux containers in HEP is a by-product of this work.

In ALICE simulations are carried out using the software framework AliRoot and some more tools, as described in §1.5.1. This kind of job to generally needs a small data set in input, but can produce a huge amount of data as output. Usually, the input data are stored in the Offline Conditions DataBase (OCDB). The OCDB is the place where the calibration and alignment data for the ALICE detectors are stored. It is a set of entries in the AliEn file catalog that points to the physical entities (ROOT files stored in the various storage elements of the Grid) containing the calibration and alignment data. Then, after the simulation process whose started from few starting conditions, is finished, output data are produced and stored in some dedicated space. The size of output data in heavy-ion collision simulations may vary as a function of the number of events simulated and of the number of detector involved in the measurement simulated. It is therefore feasible to tune the parameters of a simulation in order to run it on a volunteer cluster.

4.3.1 Access CVMFS repositories from containers

Both OCDB and every "tagged" version of AliRoot is available via the CVMFS filesystem, therefore containers must have a mechanism to access the repository. Keeping in mind the guidelines to develop a low impact volunteer computing service (as explained in §2.5) it became clear that directly mount CVMFS inside containers was not the best option. This is because in that scenario the virtual filesystem has to be installed on the volunteer host. Several aspects have to be taken into account:
• More generally, installing additional software does increase the number of elements to maintain. Moreover, it implies to manage installations and updates on a heterogeneous set of many different [OSS] increasing the complexity of the installation phase. Furthermore, it cannot be done from the "outside" by design, since it requires root privileges and volunteers may not agree with that.

• Installing it on the underlying host (with respect to containers) causes CVMFS to be more sensible to the surrounding environment. It relies on a set of standard Linux tools, like the FUSE kernel module and the automount utility. The tools are shared with other common Linux software; a sudden stop for those services would result in service interruptions inside worker-containers.

• Another point is about security and integrity: the common problem of volunteer computing is that there is absolutely no way to make sure that the donator does not tamper with volunteer jobs. Our protection is given by the fact that our resources are controlled and lay in an environment whose security is granted by network providers. Obviously this is just a mild protection against inner attacks, it is more likely against CVMFS failures (if the client is stopped the virtual directory is found empty, causing errors in the job execution) than to protect against attacks which are extremely easy to carry out when the attacker is also the administrator of the host.

I tested two solutions to this problem, every one had pros and cons. They will be briefly illustrated.

**Expose a FUSE device inside a container** The most natural idea was to install the CVMFS client inside the container. To work correctly, the client requires that the FUSE device (/dev/fuse) is present in the environment. It is straightforward to do so, by specifying the list of devices to bind at startup. It also requires a specific Linux capability to mount the virtual filesystem (in read-only mode) on the container: CAP_SYS_ADMIN is easy to configure too. Even though it can be translated in moving the software requirements from a whole CVMFS client to just the FUSE module, the latter is more standard and it is likely to be already installed on the most standard Linux distributions, therefore it was investigated as a possibility. The drawback in this approach is that it is impossible to share a cache between containers, since every client exists in a different environment and, by design, CVMFS is thought to be unique on the single host. Therefore sharing a directory

---

5Automount is a utility that automatically and transparently mounts a filesystem whenever it is requested by the system. See [linux.die.net/man/8/automount](http://linux.die.net/man/8/automount)
containing the cache between many containers, will cause a concurrent write access by many clients: each one tries to lock the resource. To use the caching system in this configuration is therefore not an option, since it simply does not work. On the other hand, not using a caching system causes a great decrease in performances and job efficiency since every new container must download ex-novo the needed software from the internet, causing network overhead and bottlenecks, especially with the increase of the number of hosts participating in the cluster. On the single host another consequence is related to the disk usage: every worker-container downloads the needed software locally, thus creating a useless and redundant disk occupancy.

Use Parrot to get contents from CVMFS The other option, which has been chosen, even though it also showed some limits, was to use an utility provided by the Collaborative Computing Tools (CCTools) suite, called Parrot. Its main executable, "parrot_run", is basically a wrapper: a software that envelops the execution of another program and hooks every system call to the filesystem in order to modify the result of that call. To do that it exploits the `process trace (ptrace())` function. Therefore, in principle it is possible to wrap every program -from a BASH shell to an entire job execution- inside a `parrot_run` instance. It acts simply by converting the result of system calls on filesystem to HTTP requests towards a configured CVMFS repository, exposing results of those queries as files, like they really were on the host. It also downloads required binaries whenever the execution of a software provided by CVMFS (e.g. the aliroot binary) is required. Parrot is designed to run on the host only as an unprivileged user, in order to grant security and isolation to other processes. However, when it runs inside a container, from the OS perspective it is a child processes whose father is a container owned by the "docker" user, started by a daemon that runs with "setuid" root. Therefore, further additional capabilities must be given to containers in order to avoid the MACs blocks for `ptrace()` calls. In Appendix §5.3 an example of Apparmor profile is reported. Parrot owns also a caching system that is shareable between multiple instances also running in different containers, hence it is possible to create a directory available to every container to access, reducing useless redundancy and avoiding to cause continuous network overhead whenever a new container does its first call to download the AliRoot binary.

---

6The ratio between the time spent by the container in executing the actual job and the wall-clock time.
7The CCTools is toolkit for HPC ccl.cse.nd.edu/software/parrot/
8To access data on a file system, the kernel provides a set of calls and methods to interact with filesystems, on which the Portable Operating System Interface for Unix (POSIX) command "ls" relies.
9The same function used by the "ps" (process-status) Uniplexed Information and Computing System (UNIX) command to hook informations about running processes.
10In particular the Apparmor mandatory access control, on Ubuntu based distributions.
### Table 4.2: Comparison between latencies in I/O of (per block) in three different cases: on the host, inside a container running on the same host and inside Parrot running inside the container. The size of files used is 1GB.

<table>
<thead>
<tr>
<th>Case</th>
<th>Seq. Write [μs ± 1]</th>
<th>Seq. Read [μs ± 1]</th>
<th>Random Seeks [μs ± 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>25</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>Container</td>
<td>34</td>
<td>46</td>
<td>408</td>
</tr>
<tr>
<td>Cont+Parrot</td>
<td>71</td>
<td>141</td>
<td>274</td>
</tr>
</tbody>
</table>

However, even though this method is the one that best satisfies requirements for volunteer computing, resulting the least invasive and most resource-saving, it shows some weaknesses in managing non-trivial process topologies.\(^{11}\)

It is also known that to run jobs into a parrot\_run session adds a sensitive latency.\(^{12}\) In table 4.2 the result of an I/O benchmark of 1GB files led on the same workstation available in section §4.1 is shown. In this test Parrot is not providing data from CVMFS, it only wraps the entire bash session in which the benchmark is run, in order to quantify the latency induced in hooking every single system call to the filesystem. The overlay of different filesystems (Docker-OverlayFS and XFS on the host) causes latency both in read and write operations. For random seeks the behaviour does not follow the same trend, even if the pure host still remain the case with lowest latency. It is the sum of software processing time and the physical movement of the HDD time, in accessing data that leads to a global slow down in the execution.

Whenever performances are not the first priority, an infrastructure for code validation like the volunteer one in Torino might be the case, Parrot still represents a usable approach to access CVMFS data. Overall this approach is generally working decently, therefore I decided to adopt it in Torino.

#### 4.3.2 HTCondor inside a container

Once the container has access to the software needed through Parrot and CVMFS, the next step is to deliver jobs containers. As previously mentioned, the choice relies on HTCondor. In this implementation, however, I decided to put also the HTCondor services inside the container, in order not install them on the host, hence maintaining control on versions and installations.\(^{13}\) HTCondor services (daemons) are spawned as child processes of a parent

---

11. The ALICE job agent forks during its execution, thus creating some unmanaged issues with Parrot.
12. "Latency" is the time elapsed between giving a directive to the Hard Disk (HDD) and its actual execution.
13. HTCondor latest versions provide the possibility to run jobs wrapped inside container, however this is not an option since to fully keep control on the software used it must be provided inside the container itself.
4.3. A volunteer computing facility at ALICE Torino

**condor_master** process. Its role is to grant the availability of the other services and to manage them. There are many services available, through the HTCondor configuration files it is possible to select the needed services to be run on each host. Therefore the role of the single machine in the cluster is directly determined on the services running on it. In a minimal worker node setup (in this case each container is seen as a single worker, with its master HTCondor process running inside) only three services are chosen, after the master:

- **condor_procd**: it manages process families. If the **condor_procd** is not used, each daemon will run the process family tracking logic on its own. Use of the **condor_procd** results in improved scalability because only one instance of this logic is required.

- **port**: is an optional daemon responsible for creating a TCP listener port shared by all of the Condor daemons. In such a way it does not open any port on the firewall, which is a very good pledge in terms of security on the host. As can be seen in figure 4.2 the default value for that port is 9168, but it is indeed configurable.

- **condor_startd**: is the daemon responsible to effectively run jobs.

Docker container comes up running a pilot executable. That pilot starts the HTCondor master daemon and all the services required. In figure 4.2 it is possible to check the process topology behind the execution of a worker container which runs HTCondor. Docker container is the root process, the pilot forks from it and spawns the HTCondor master process, which starts the other three services. The pilot sleeps for a configurable interval (in this case 4 seconds) and then it sends a "graceful_kill" signal to the local HTCondor master process.

```
 Figure 4.2: This snippet from pstree shows the topology of the HTCondor processes running inside a Docker container.
```

In such a way if the master has taken workload in the meanwhile, it will finish it and then it will shutdown its services. Hence, the remote master, which diagnoses the status of the entire cluster will know that the containerised worker has been shut down. HTCondor is specifically designed for opportunistic computing, therefore to be flexible and fault-tolerant. It has *fail-over* logics...
that ensures the job completion, if suddenly interrupted. A running job that disappears, in [HPC], is an occurrence that happens more than one can imagine, and is usually managed by every distributed job batch system. This is a core feature that a job scheduler running on Plancton must provide. The overhead controller in Plancton could decide to remove a running container because of resource reclaim by the owner. This from a HTCondor perspective is the equivalent of a sudden shut-down of the worker node. HTCondor is able to face this situation, under configurable policies, and can resubmit jobs if it is the case.

4.3.3 Service Architecture

According to its configuration, which depends on the use case, every worker node queries for jobs to a designed `condor_schedd` daemon, usually running on a dedicated master node. Nodes need to be configured to contact their master on the right IP address and on a specific port. In Torino I configured connections between master and workers to be encrypted with a key injected into the container image. This is not intended to be a security provision, since the key is stored in plain text inside the Dockerfile (see Appendix §5.3); it is also true that the project is implemented in a trusted network on which only authorised users can connect, involves trusted participants, so the security aspect is granted by the network isolation. The configuration files of every worker container must be injected inside its image or mounted at instantiation time.
If there are jobs in queue they will be assigned to that worker node, otherwise it will die after a configurable TTL. In the HTCondor architecture it is not mandatory that the cluster master node is the same on which job submissions are performed, in this cases a dedicated physical node is responsible for job scheduling and cluster management.

### 4.3.4 Submission of physics simulations to the facility

In HTCondor a job is constituted by an executable (script or binary file) supposed to be run on every worker node that satisfies specific requirements. In order to submit a HTCondor job group users must provide a job submission manifest containing a number of parameters (such as the platform, executable, number of jobs to submit, resubmission parameters, etc.), every other file required by jobs and the name of output, log-file and error-file names. An example of condor configuration file for a job is reported in Appendix §5.3. The syntax is very simple and it requires a restricted number of core parameters. The HTCondor scheduler will assign the payload to one of the available resources (in this case they are pilot containers active at that moment). Once the job is assigned, all the files specified in the configuration are uploaded to the worker node. Usually an ALICE simulation is based on a macro (commonly called Sim.C and written in C++) which loads the needed...
simulation libraries and configurations (stored in another file, the `Config.C`).
In this volunteer setup, the execution of the workload runs wrapped inside a Parrot session, which is transparently done by the AliRoot framework provided by the CVMFS-aware environment. After the job finishes, results are copied to the node from where they were submitted or in any other preferred destination. In case a job fails, HTCondor local instance saves log-files and error-files, useful for debug and retransmits them to the remote master host. The volunteer computing facility in Torino has been proven to work, being capable to accomplish basic physics simulations as HTCondor job. Conclusions and considerations will be better presented in Chapter 5.
4.4 A dedicated Grid site based on Plancton

In the second part of my thesis work I demonstrate how the simple and lightweight Plancton approach in container provisioning has been exported in a more dedicated scenario.

For this purpose I used the development cluster of the HLT at the ALICE experiment. The current HLT-DEV cluster is part of the old HLT facility, used in production for the HLT of the ALICE experiment during Run 1.

Now, for the Run 2, it has been replaced by a new one with more powerful nodes. The HLT cluster has the following setup:

- 180 nodes with 2 Intel Xeon processors, 10 cores each + hyperthreading.
- 128 GB RAM.
- GPU: AMD Firepro W8000 graphics card.
- Hard disks SSD: each node has one Grid-dedicated disk.
- Network: 1 Gbit/s Ethernet and InfiniBand.
- Uplink to the CERN General Purpose Network: 40+40 Gbit/s.

Whenever these facilities are not used by the "HLT experts" nor by the experiment itself, they are converted in opportunistic AliEn Grid sites used to perform centrally-managed tasks (no user jobs). The HLT production cluster is indeed more powerful than the development cluster I used. The HLT-DEV cluster is constituted by ~30 nodes, with an overall amount of 800 cores (with hyperthreading), with a network whose nominal bandwidth is 80 Gb/s, of which 10 Gb/s are effectively available. Since it is configured with the same scheme of the current HLT production cluster and shares also the same network, my setup could, in principle, run on the production cluster when switched in opportunistic mode.

The HLT-DEV main aim is to provide a testbed where to develop and test software to be implemented on the production cluster in the next future. This kind of "opportunistic computing" is slightly different form the volunteer case previously presented. Here Plancton will not act as a smart scheduler that negotiates resources with the host node to provide container as worker nodes. When the cluster is switched to an opportunistic computing facility every resource is deployed for computing, hence restricting Plancton to a mere automation to provide worker nodes as containers.

A full description of the HLT production cluster network: [23]


Chapter 4. Opportunistic computing with Plancton

4.4.1 Pre-existing setup of the opportunistic cluster

In the HLT-DEV facility there was already an existing implementation of an opportunistic Grid node, which will be briefly described. In order to be able to nimbly and neatly switch from a use-case to another, it was adopted a cloud solution on top of the bare metal resources, moving changes and tests to a higher level. In such a way at the same time a secure and isolated ecosystem is granted, separated from the pure hardware resources, where it was possible develop and test applications.

On each physical node CERN Centos 7 (CC7) is installed. On top of them is run a virtual cluster of CentOS 7 VMs orchestrated by a private and isolated OpenStack\textsuperscript{15} instance which runs on a dedicated physical node which sees only the hypervisors. The configurations for the VM are propagated using Puppet and Foreman\textsuperscript{16}. There is also a physical AliEn Virtual Organisation Box (VOBox) to support ALICE Virtual Organisation (VO) services. This is a node on which long-lived agents and services will be deployed and it is expected to be provided at the sites. It only sees the worker nodes VMs and the Grid universe and constitutes the interface between the Grid and the virtual cluster. Jobs are run inside each virtual machine side by side, with no further isolation, sandboxing nor swap limitation, often causing a wear-out of the disks. The description of the mechanism used to dispatch jobs from the Grid to the worker nodes will be skipped, since the one adopted in the implementation with Plancton is almost the same, with the exception of some minor tweaks. Therefore it will be described once for that case which is very similar.

4.4.2 Opportunistic implementation with Plancton and containers

The Plancton capability to scale on larger scale clusters will be well demonstrated, thanks to the possibility to employ a larger number of machines with respect to the previous installation. From a configuration perspective it can be intended as a further simplification in Plancton behaviour. The configuration file is reported in Appendix §5.3\textsuperscript{17}. The very strength of containers resides in the low virtualisation overhead and in their ability to isolate processes through sandboxing.

In this case Docker containers will not run directly on the bare metal, but instead inside a CentOS 7 VM which will constitute an abstract homogenisation layer to better preserve the underlying bare metal environment. This of course will affect absolute performances but will constitute the proof of concept of a working setup, ready to be exported on the bare metal. Moreover,

\textsuperscript{15}Reference and documentation: \url{openstack.org}
\textsuperscript{16}Reference and documentation, respectively: \url{puppet.com} and \url{theforeman.org}
as previously explained before, the virtualisation layer in principle allows the "HLT experts" to easily deploy and un-deploy opportunistic facilities, without even rebooting the underlying machines. Also in this case, security is provided by the network isolation, that grants no external access at all by unauthorised agents.

**Agile cluster management with Ansible** Another aspect that is different between the volunteer approach is that now the system administrator is able to access and manage the (virtual) machines constituting the computing cluster. This eases some aspects of the problems, since it is now possible to change configurations and setup on-the-fly. There are many tools to do so, for example the aforementioned "Puppet". I chose to use Ansible\(^\text{17}\) for this purpose, since it very easy to use. Ansible has been used to:

- Install software prerequisites and the Plancton daemon.
- Manage different kind of nodes at the same time (masters and slaves).
- Manage "slave" services on every virtual host.
- Dynamically update Plancton configuration\(^\text{18}\).

**Mount CVMFS inside a container** Even in this case the software needed to execute jobs is available on CVMFS. Another advantage in owning the worker nodes is the possibility to install every requisite that is needed without exploiting any workaround (like Parrot in the volunteer setup). Therefore, CVMFS is installed directly on virtual machines, allowing them to cache contents accessible by every Docker container, gaining full advantage from that. To allow containers to mount external filesystem they are run with proper capabilities (CAP_SYS_ADMIN). An internal dedicated Squid\(^\text{19}\) server used as CVMFS proxy is also provided, granting more performances and software availability against the multiple requests.

**Plancton settings** Plancton duties in this setup are very limited. It only must take care of periodically check that every resource is in use and that containers do not survive beyond their assigned TTL. The schema adopted is again "one job per container" which allows us to isolate resources for each job. Compared with the previous opportunistic setup at HLT which did foresee to run multiple parallel jobs on the same VM with no further separation,

\(^{17}\)Official website: \texttt{ansible.com}

\(^{18}\)A running Plancton instance is able to update its behaviour simply by changing its static configuration.

\(^{19}\)See a fully documented article about using CVMFS as a High Bandwidth Distributed Filesystem for Auxiliary Physics Data. It comprises also a good explaining about the advantages in using Squid server for caching purposes.\(^\text{15}\)
this constitutes an improvement. In that scenario each job concurs against the other in utilising resources, if on at some points saturates every available resource on the host, the execution of every other job will be affected. The new "containerised" approach permits to regulate with more fine granularity the utilisation of the (virtual) host by every job. Therefore it ensures that if a single job is bad behaving (causes memory leaks or excessive resource occupancy or swaps too much on disk) it is possible to prune it, in order not to affect the execution and resource availability of its neighbours. This is a double-fold scenario: healthy jobs whose resources are limited might take longer time to be executed. Perhaps they are simply "long" to compute but not "bad-written"; hence it is necessary to take into account this in configuring TTL thresholds for jobs. Resource isolation is an advantage coming from the containerised approach not from adopting Plancton directly.

The convenience in using Plancton among other services is related to its simplicity, small footprint and low overhead combined with its responsiveness to changes and lightweight. Moreover, Plancton is a stateless daemon. For instance, it does not have an internal list of managed resources; Docker is stateful, and Plancton relies on the statefulness of other objects for working properly. In this scenario no software component more complex than that is required: using the "fire-and-forget" approach for container provisioning it is required just an automation to allocate resources. Moreover, using Ansible on the cluster, Plancton execution and configuration now becomes an easy task, it is therefore possible to modify the environment in debug phase, restarting a new version of the daemon or just changing its configurations on-the-fly, without interrupting job computations.

4.4.3 AliEn and Work Queue: an interface for job scheduling

The process of submitting jobs to the cluster is accomplished using the "AliEn Work Queue Integration".

AliEn is the necessary tool –included in the VOBox– to plug the ALICE site to the Grid. The procedure is similar to the setup of an AliEn Grid site, but using Work Queue from CCTools as a backend to manage jobs, without the need for a complex resource management system. The AliEn Work Queue setup allows us to submit ALICE pilot jobs to WQ almost natively.

20 Usually this setup is not employed to run user analyses but only Monte Carlo production code. Even though it will be better discuss later, it must be take in consideration that it is very unlikely that simulation might contain bad code, the "double-fold" is anyway presented for completeness.

21 Reference and manual: ccl.cse.nd.edu/software/manuals/workqueue.html
4.4. A dedicated Grid site based on Plancton

How does it work

The VOBox is configured as a HTCondor site, thus jobs are submitted (by AliEn) to this site through the command `condor_submit` and the job spool is checked through the command `condor_q`.

The whole point in this is that HTCondor is not needed in this site. Those commands are transparently substituted by mock-up versions that behave like the original ones from the AliEn’s perspective but they communicate with Work Queue behind the scene. The important thing of using this logic is that the results must be given in a fashion that is expected by the unmodified part of the setup. That is AliEn must receive job results and information about queues in a way that mimics the HTCondor format.

The fake `condor_submit` command in fact, instead of submitting payloads\(^\text{22}\) to HTCondor worker nodes, simply creates a set of ordered directories that contain the job payload.

Similarly, the fake `condor_q` command performs a check on the directory containing the jobs to be executed and returns true information about them in a HTCondor format, understandable by AliEn. Once Jobs are landed on the site it is possible to implement the Work Queue approach. It is again a master-slave approach, which foresees a Work Queue Master running on the host that contains the directories with jobs and is responsible to dispatch

\(^{22}\)“Payload” is the part of the job that is actually executed. It is constituted by every file needed by the computation. It differs from "metadata" which are instead the informations related to the job itself.
jobs whenever available and a foreman or a worker makes a request. There are then the Work Queue Foremen, which are an intermediate level between master and workers. Foremen are executed inside the VM and survive worker nodes lifecycle. They act as a worker with respect to the master, and as a master with respect to the workers. It is therefore possible to offload the charge allocated to the master caused by multiple simultaneous requests from workers, simply by using foremen as "proxies". At the bottom there are the workers. In this configuration they are pilots executed inside containers, hence they harmonise perfectly with the Plancton workflow. They come up with a --single-shot flag, that specifies that after the execution of a single payload it must it ought to self-terminate, hence killing its container (and cleaned up by Plancton). During their lifetime they query the master or the foremen –if they are included in the implementation– for jobs to be executed. The Work Queue Master checks if there are jobs available in the spool and answers with a task, an executable to be run on the worker that contains the job payload. The schema of AliEn Work Queue is depicted in figure 4.5.

**Worker node architecture** In this case the architecture of the single node –shown in 4.6– is slightly different from the volunteer one previously described.

![Figure 4.6: Overview of dedicated Work Queue setup on a single host. It shows the layer hierarchy for services and containers. Plancton is at the same level of Docker and Work Queue foreman, the job execution is wrapped inside the pilot container and accesses the framework directly on /cvmfs. The host used this time is a virtual one, hosted by an "untouched" CC7-based physical host.](image-url)

First of all the host is now a virtual machine –which runs CentOS 7– hosted by an underlying CC7 system. This does not affect the inner system,
which is completely agnostic of the outer architecture. The further virtualisation layer is useful whenever the cluster administrators decide to switch setup by simply converting the pool of running VMs into another.

Plancton and Docker now are manageable from the outside using Ansible, which adds more control on the facility. Alongside one or more instances of the work Queue Foremen are also installed. The CVMFS filesystem is installed on the virtual host and bind-mounted inside the container. Instead of HTCondor services now there is only the Work Queue (WQ) worker pilot running, which accesses either the WQ master or one of the WQ foremen. Job execution is completely wrapped inside the container. On each virtual host are run a number of containers equal to the number of virtual cores, which are directly mapped to the physical ones.

The whole facility has around 30 instances of Plancton running (one per node) with 800 total containers concurrently executing the same amount of jobs. These results have been achieved and the HLT-DEV cluster at CERN now up and running relying on the presented setup based on Plancton, Work Queue and Docker. Conclusions and considerations will be better presented in Chapter 5.
5.1 Plancton and Docker: an enabling technology in volunteer computing

For what concerns the project of volunteer computing at ALICE Torino, results are encouraging. The entire workflow based on HTCondor has been proven to work. Users wishing to submit jobs on the cluster can log-in to a front-end (physical) machine with their own UNIX credential created ad-hoc. Currently a workstation has been elected to accomplish this job. In the future it possibly will be replaced by a dedicated host, where users have a local storage quota, where input data and the results of their analyses will be temporarily stored. Indeed that machine should provide a storage capability greater than the quantity ordinary available on commodity workstations ($\sim 500 - 1000$ GB, nowadays). It could be also evaluated the idea to create a distributed filesystem shared across multiple hosts collaborating in the volunteer project\footnote{For example creating a distributed storage device using GlusterFS: \url{www.gluster.org/}}. The scheduler is running on a physical host and it is able to dispatch workloads to worker nodes constituted by Docker containers. Currently it is not implemented an official public login-interface reachable from the outside of the Istituto Nazionale di Fisica Nucleare (INFN) network. Intended users are supposed to be members of the collaboration to whom credentials can be easily provided by creating a UNIX account on the submitting host, which is indeed reachable from the branch (subnet) of the INFN network. The Plancton daemon behaves as planned, and its configuration provides enough granularity
Chapter 5. Results, Conclusions and Outlook

Figure 5.1: Screenshot of the `condor_status` command. It lists every worker node available in the HTCondor cluster. Containers are listed with their hostname whenever specified or with the single ID generated by Docker by default.

to properly tune the main routines. Thanks to the possibility represented by Linux containers to be used in such a _disposable_ mode, the management is drastically eased by not having to "keep alive" heavier machineries like VMs on the collaborating hosts. This unburdens the cluster management from pausing/stopping or draining any virtual machine. The utilisation of Docker containers as "pilots" allows the infrastructure to adopt different architectures and workflows. It is extremely easy to add new hosts to the project, just by installing the Plancton+Docker bundle in a new collaborating host. New spawned containers will automatically attach to the service according to their configuration (see Fig. 5.1). The HTCondor master will be automatically notified about the presence of new participants and will dynamically add them to its pool. With the (HT)Condor Connection Broker (CCB) it is possible to create unidirectional communications between master and workers: a worker receives instructions from its master by getting them through a request; to communicate some information to the master it makes a query as well. In such a way it is possible not to open further network towards the extern ports, which is both a good defence against external intrusions and a method to save "resources" (network ports on a Linux system are many, but not infinite). The latter aspect is further corroborated by the adoption of the "Shared Port" technology available in HTCondor, which allows to use only a single internal port to answer every external request mad by any HTCondor daemon.

\footnote{"Drain" a VM means to wait for the completion of its current task (if any) without enqueuing other jobs. It is usually accomplished to prevent further usage on the host or before intervene on the cluster with updates, global resets, etc. In such a way it is not necessary to remove and re-instantiate another instance, saving time spent in boot and CPU efficiency.}
At the current state the tool is usable, even though its user-interface could be largely improved: it makes more difficult to present it as a viable option to the end-user. It can be considered as a working proof of concept, on which more stress tests could be performed, perhaps submitting some more real use-cases to the platform. Those cases will be very useful to better test the software adopted, and obtain feedbacks on the design choices. As an example, the Parrot tool (see §4.3.1), even though conceptually appears to be the best solution to provide the HEP software to containers, is not able to consistently wrap the entire execution of more complex tasks. This limits its range of applicability and must be taken into account. Moreover it must be considered the overhead added by wrapping an entire job session inside a `parrot_run` command, which serialises the whole execution of parallel threads, killing every optimisation deriving from parallelisation. Furthermore the software does not correctly manage non-trivial process topologies, since it does not take care of any of its orphan children processes, this causes issues in many implementations. This incoherent behaviour, actually found in the second dedicated use-case, has determined the dismissal of the tool in the dedicated Grid environment which requires outstanding performance and reliability. If the hypothesis to "not mount any external filesystem" cannot be released, there are some workarounds that can be exploited, even though at a price. For instance it could be possible to create a manifest to list the software requires by the application, which has to be downloaded by Parrot once started a container. Workload would be run as usual, relying on local software. Moreover procedure will add more complexity to the submission process and it will sensibly increase the initialisation time at startup, reducing the neat advantages obtained by using containers against virtual machines. There are other methods that aim to act as an intermediary between containers and software for HPC that could be considered.

In conclusion the Plancton daemon satisfies perfectly the requirements for volunteer computing. The tool is general-purpose and independent from the software running on top of it, making it a perfectly application-agnostic.

\footnote{Take as an example the Shifter\cite{21} project carried out by the National Energy Research Scientific Computing Center (NERSC).}

**Figure 5.2:** Portion of the output of the `docker ps -a` command. It lists every Docker container on the single host. It is clearly visible that no further external ports are bound to running containers.
tool. It has been found stable and reliable also when running for long periods. Improvements and new solutions could be found for better setup the implementation of the application.

5.2 Plancton as a lightweight automation for containers

Used in a minimal configuration setup\(^4\), Plancton demonstrated to be a usable tool to provide containers wrapping Grid job executions. The site where it is installed is the HLT-DEV cluster, where the VOBox has been configured to accept only Monte Carlo productions. The final setup started on September 26\(^{th}\) (2016) and is still up and running. Together with the AliEn Work Queue integration, with a strict resource partitioning on every host VM, Plancton demonstrated to be reliable and robust enough to be employed in a production environment. The choice to accept only simulations does not depend on Plancton itself. It has been made for safety, in order not to saturate the entire available bandwidth of the site running reconstructions and analyses and not to run the risk to interfere with users’ analyses during tests which in principle could fail.

5.2.1 Job Efficiency

Simulation jobs are characterised by a high job-efficiency\(^5\) since they have almost zero time spent in Input/Output (I/O) operations. The plot\(^6\) in Fig. 5.3 shows how the containerised computing approach in the opportunist facility at T0 satisfies the high-efficiency expectations.

The violet line (upper part) represents the efficiency of HLT-DEV cluster over time. The other lines represent the other sites in the CERN T0. This must not be red as a comparison, since other sites run a heterogeneous set of tasks, which naturally have different job efficiencies. It would be meaningless to compare other lines with the violet one, which represents a sort of "majorant" line, where the only jobs are simulations which are high-efficiency tasks by definition. In other words it can be used to check whether the best scenario, where every job reaches 100% of efficiency due to its nature, is verified or not. The fact that points are plotted proves that Plancton is reliable and the containerised approach is working properly.

Summarising, the plot shows that containers can be successfully used as single job instances, with the expected efficiencies. There are also some evident

\(^4\)Plancton in the dedicated scenario does not take any decision based on the resource usage, since its threshold is set to 100% of availability.

\(^5\)The job-efficiency is the ratio between the amount of time spent in doing actual computations against the wall-clock time.

\(^6\)Source: http://alimonitor.cern.ch/display
5.2. Plancton as a lightweight automation for containers

Figure 5.3: Simulation jobs that run in typical conditions have a naturally high job efficiency. Measures taken form September 26th to October 17th (2016).

<table>
<thead>
<tr>
<th>Last Value [±0.01]</th>
<th>Min [±0.001]</th>
<th>Avg [±0.1]</th>
<th>Max [±0.01]</th>
</tr>
</thead>
<tbody>
<tr>
<td>95.35</td>
<td>8.896</td>
<td>93.8</td>
<td>98.55</td>
</tr>
</tbody>
</table>

Table 5.1: Values corresponding to the HLT-DEV Job efficiency in Fig 5.3

fluctuations in relatively short time intervals. In reading these deviation it is necessary to take into account the log of events happening during those periods.

5.2.2 Running Jobs

Another interesting information is the number of running jobs as a function of time. It is an indicator of the correct usage of the whole resources pool over time. Since the correspondence between jobs and containers is "$1 : 1" and there is a container running for every CPU we expect to have $\sim 800$ jobs as best scenario. In Fig. 5.4 is plotted the trend of the number of running jobs over time. There are sparse falls, are not related to any Plancton malfunction, but more likely they are related to the moment when the framework measures the number of running jobs. The index of running job is sensitive to the variations of Monte Carlo production workload to be run, if for a period there are not jobs suitable for the AliEn VOBox from the Grid, it is natural to see such drops in the graph. The system has perfectly been able to recover from a drop in the number of running jobs. Reasons are unknown, probably related to an inconstant workload. The pilot approach has been able to provide promptly all the required worker nodes in the raise.
Chapter 5. Results, Conclusions and Outlook

To summarise, the plot 5.4 shows how Plancton is able to occupy almost every CPU resource available. It is also able to robustly provide new resources after job completion, since the configured TTL is of 2 days (jobs are therefore deleted every 48h regardless they are still running regularly or they are pending due to errors), as can be seen in Appendix §5.3, where the used configuration is presented.

Finally it is presented the plot in Fig. 5.5 containing the number of jobs run on the cluster during this period. It is the integration of the number of running jobs, with the subtraction of the failed and overaged ones.

So far the cluster has completed more than 15000 jobs, corresponding to the creation and deletion of likewise Docker containers.

Plancton can be proficiently adopted in the field of computing in HEP.

5.3 A wider glance upon computing scene in HEP

Linux containers represent a viable technology usable in various aspects of a Physics experiment. The ability to provide custom consistent environments makes them a desirable alternative to VMs. They are used, for instance, in
automated software building systems, which are responsible to periodically publish latest software packages for a variety of platforms. Docker is a good tool which has been improved heavily during last years, its "facilitated" approach to containers enhances their main strengths, but it does not represent a unique approach to container provisioning. Linux containers are also used in deploying long-time services, in coordination with powerful frameworks, grown outside the strict ambit of scientific computing, like Apache Mesos and Aurora\(^7\). But containers are not to be intended as lightweight substitutes to VM since their features sometimes cannot even be compared. It was therefore not obvious \textit{a priori} to consider containers a viable solution to isolate physics jobs. This work corroborates the idea to use them in such a fashion, also covering an interesting matter, the volunteer computing, a definitely not trivial use-case to test. Also taking into consideration what has emerged from the leading sector conference in October 2016 Computing in High Energy Physics (CHEP)\(^8\), Linux containers are technology used in production, on which critical services can rely, and is acquiring more and more popularity. This work\(^9\), well fitted in that environment, proposing two new solutions to two old problems.

\(^7\)See: \url{mesos.apache.org} \url{aurora.apache.org}

\(^8\)Official website: \url{chep2016.org}

\(^9\)Presented in the form of a poster: \url{indico.cern.ch/event/505613/contributions/2227997}
List of Acronyms

ACORDE  ALICE Cosmic Ray Detector.
AD      ALICE Forward Hodoscopes.
AF      Analysis Facility.
AGS     Alternating Gradient Synchrotron.
ALICE  A Large Ion Collider Experiment.
AliEn   Alice Environment Grid Framework.
AOD     Analysis Object Data.
API     Application Programming Interface.
ATLAS   A Toroidal LHC ApparatuS.
BASH    Bourne Again SHell.
BOINC   Berkeley Open Infrastructures for Network Computing.
CC7     CERN Centos 7.
CCB     (HT)Condor Connection Broker.
CCTools Collaborative Computing Tools.
CERN    Organisation Européen pour la Recherche Nucléaire.
CFS     Completely Fair Scheduler.
cgroup  Control Group.
CHEP    Computing in High Energy Physics.
CINT    C-Interpreter.
CLI     Command Line Interface.
CMS     Compact Muon Solenoid.
CPU     Central Processing Unit.
CTP     Central Trigger Processor.
CVMFS   CernVM Filesystem.
DAQ     Data AcQuisition System.
EGEE    Enabling Grid for E-sciencE.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGI</td>
<td>European Grid Infrastructure.</td>
</tr>
<tr>
<td>EMCal</td>
<td>Electromagnetic Calorimeter.</td>
</tr>
<tr>
<td>ESD</td>
<td>Event Summary Data.</td>
</tr>
<tr>
<td>FC</td>
<td>File Catalog.</td>
</tr>
<tr>
<td>FMD</td>
<td>Forward Multiplicity Detector.</td>
</tr>
<tr>
<td>FMS</td>
<td>Forward Muon Spectrometer.</td>
</tr>
<tr>
<td>FUSE</td>
<td>Filesystem in User SpacE.</td>
</tr>
<tr>
<td>GDC</td>
<td>Global Data Collectors.</td>
</tr>
<tr>
<td>GDS</td>
<td>Global Data Storage servers.</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphic Processing Unit.</td>
</tr>
<tr>
<td>HDD</td>
<td>HarD Disk.</td>
</tr>
<tr>
<td>HEP</td>
<td>High Energy Physics.</td>
</tr>
<tr>
<td>HLT</td>
<td>High Level Trigger.</td>
</tr>
<tr>
<td>HLT-DEV</td>
<td>High-level Trigger Development.</td>
</tr>
<tr>
<td>HMPID</td>
<td>High-Momentum Inclusive Particle Identification.</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing.</td>
</tr>
<tr>
<td>HTC</td>
<td>High Throughput Computing.</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol.</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output.</td>
</tr>
<tr>
<td>INFN</td>
<td>Istituto Nazionale di Fisica Nucleare.</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol.</td>
</tr>
<tr>
<td>ITS</td>
<td>Inner Tracking System.</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation.</td>
</tr>
<tr>
<td>LDC</td>
<td>Local Data Concentrator.</td>
</tr>
<tr>
<td>LEP</td>
<td>Large Electron–Positron Collider.</td>
</tr>
<tr>
<td>LHC</td>
<td>Large Hadron Collider.</td>
</tr>
<tr>
<td>LHCb</td>
<td>Large Hadron Collider beauty.</td>
</tr>
<tr>
<td>LHCf</td>
<td>The Large Hadron Collider forward.</td>
</tr>
<tr>
<td>LS2</td>
<td>Long Shutdown 2.</td>
</tr>
<tr>
<td>LTU</td>
<td>Local Trigger Unit.</td>
</tr>
<tr>
<td>LXC</td>
<td>LinuX Container.</td>
</tr>
<tr>
<td>MAC</td>
<td>Mandatory Access Control.</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>MoEDAL</td>
<td>The Monopole &amp; Exotics Detector at the LHC.</td>
</tr>
<tr>
<td>OCDB</td>
<td>Offline Conditions DataBase.</td>
</tr>
<tr>
<td>OO</td>
<td>Object-Oriented.</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System.</td>
</tr>
<tr>
<td>OSG</td>
<td>OpenScience Grid.</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a Service.</td>
</tr>
<tr>
<td>PB</td>
<td>PetaByte.</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer.</td>
</tr>
<tr>
<td>PHENIX</td>
<td>Pioneering High Energy Nuclear Interaction eXperiment.</td>
</tr>
<tr>
<td>PHOS</td>
<td>PHOton Spectrometer.</td>
</tr>
<tr>
<td>PID</td>
<td>Process IDentifier.</td>
</tr>
<tr>
<td>PID</td>
<td>Particle IDentification.</td>
</tr>
<tr>
<td>pip</td>
<td>Python Install Package.</td>
</tr>
<tr>
<td>PMD</td>
<td>Photon Multiplicity Detector.</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface for Unix.</td>
</tr>
<tr>
<td>ptrace()</td>
<td>process trace.</td>
</tr>
<tr>
<td>PyPI</td>
<td>Python Package Index.</td>
</tr>
<tr>
<td>QCD</td>
<td>Quantum Chromodynamics.</td>
</tr>
<tr>
<td>QGP</td>
<td>Quark Gluon Plasma.</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service.</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory.</td>
</tr>
<tr>
<td>REST</td>
<td>REpresentational State Transfer.</td>
</tr>
<tr>
<td>RHEL</td>
<td>Red Hat Enterprise Linux.</td>
</tr>
<tr>
<td>RHIC</td>
<td>Relativistic Heavy Ion Collider.</td>
</tr>
<tr>
<td>RICH</td>
<td>Ring Imaging Cherenkov.</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service.</td>
</tr>
<tr>
<td>SPS</td>
<td>Super Proton Synchrotron.</td>
</tr>
<tr>
<td>STAR</td>
<td>Solenoidal Tracker at RHIC.</td>
</tr>
<tr>
<td>TOF</td>
<td>Time Of Flight.</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>TOTEM</td>
<td>Total, elastic and diffractive cross-section measurement.</td>
</tr>
<tr>
<td>TPC</td>
<td>Time Projection Chamber.</td>
</tr>
<tr>
<td>TQ</td>
<td>Task Queue.</td>
</tr>
<tr>
<td>TRD</td>
<td>Transition-Radiation Detector.</td>
</tr>
<tr>
<td>TTL</td>
<td>Time To Live.</td>
</tr>
<tr>
<td>UNIX</td>
<td>Uniplexed Information and Computing System.</td>
</tr>
<tr>
<td>URL</td>
<td>Universal Resource Locator.</td>
</tr>
<tr>
<td>VAF</td>
<td>Virtual Analysis Facility.</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine.</td>
</tr>
<tr>
<td>VMC</td>
<td>Virtual Monte-Carlo.</td>
</tr>
<tr>
<td>VO</td>
<td>Virtual Organisation.</td>
</tr>
<tr>
<td>VObx</td>
<td>Virtual Organisation Box.</td>
</tr>
<tr>
<td>WLCG</td>
<td>Worldwide LHC Computing Grid.</td>
</tr>
<tr>
<td>WQ</td>
<td>Work Queue.</td>
</tr>
<tr>
<td>ZDC</td>
<td>Zero Degrees Calorimeter.</td>
</tr>
</tbody>
</table>
Appendix

Dockerfiles

Example of Dockerfile available at [13]. This Dockerfile is used to build a CentOS-based HTCondor[20] worker node for the volunteer facility of Torino. It is intended as an example, it is likely changed meanwhile.

```
# DockerFile for Centos6 with a working parrot+cvmfs installation.
#
# See: github.com/cvmfs/cvmfs and cernvm.cern.ch/portal/filesystem
# See: github.com/cooperative-computing-lab/cctools and
# http://ccl.cse.nd.edu/software/manuals/parrot.html

FROM centos:centos6
MAINTAINER Matteo Concas mconcas@cern.ch

# Environment variables.
ENV TEST_USER="parrotuser" TEST_USER_HOME="/home/parrotuser"
    PATH=/opt/cctools/cctools-5.2.3-x86_64-redhat6/bin:$PATH
    # CCTools
    CCTOOLS_URL=http://ccl.cse.nd.edu/software/files/cctools-5.2.3-x86_64-redhat6.tar.gz
    CCTOOLS_PATH=/opt/cctools
    # Compile flag
    MJ=9
    # Generic prerequisites, currently deprecated and not installed.
    COMM_REQ="git cmake tar unzip gcc gcc-c++ patch zlib-devel openssl-devel openssl-devel make which vim libcap-2-devel"
    # Software-specific requisites
    ROOT_REQ="autoconf automake libtool libxml2-devel libX11-devel libXpm-devel
               libXft-devel libXext-devel mesa-libGLU-devel CGAL-devel subversion"
    # AliRoot
    ALIROOT_REQ="libXpm compat-libgfortran-41 tcl
               compat-libtermcap redhat-ldbl-core"
    # Condor Password
    # CONDOR_SECRET="@mycondorpassword@

RUN mkdir $CCTOOLS_PATH
    mkdir -p /cvmfs/.modulerc
    rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-6
    # Installing prerequisites.
    yum install -y which tar git
    curl -o $CCTOOLS_PATH/cctools.tar.gz $CCTOOLS_URL
    tar -xvf $CCTOOLS_PATH/cctools.tar.gz -C$CCTOOLS_PATH
```

COPY opt/p_cvmfs_env.sh /home/$TEST_USER/p_cvmfs_env.sh
COPY opt/cern-it1.cern.ch.pub /etc/cvmfs/keys/cern.ch/cern-it1.cern.ch.pub
COPY opt/p_cvmfs_env.sh /home/$TEST_USER/p_cvmfs_env.sh
COPY opt/cern-it1.cern.ch.pub /etc/cvmfs/keys/cern.ch/cern-it1.cern.ch.pub
```
# Setting up a test user

```bash
useradd -d $TEST_USER_HOME $TEST_USER && 
chown -R $TEST_USER $TEST_USER_HOME && 
chown -R $TEST_USER /etc/cvmfs/keys/cern.ch/cern-it1.cern.ch.pub && 
runuser -l $TEST_USER -c "echo \"alias ali='source /cvmfs/alice.cern.ch/etc/login.sh'\" \n>> $TEST_USER_HOME/.bashrc" && 
```

# Root & AliRoot stuff

```bash
yum install -y $ROOT_REQ $ALIROOT_REQ
```

# Condor setup.

```bash
RUN curl -o /etc/yum.repos.d/htcondor-development-rhel6.repo
yum install -y condor
```

# This tricks the alienv script...

```bash
COPY opt/os-release /etc/os-release
```

# Pilot injection.

```bash
COPY opt/condor-centos6-worker-pilot /tmp/condor-centos6-worker-pilot
RUN chmod +x /tmp/condor-centos6-worker-pilot
```
Plancton configuration examples

It is reported an example of the Plancton configuration file `config.yaml`, for the dedicated Grid HLT DEV farm.

```yaml
---
updateconfig: 10
main_sleep: 10
grace_kill: 100
grace_spawn: 100
cpus_per_dock: 1
max_docks: ncpus
max_ttl: 172800
docker_image: alisw/sl6-builder
docker_cmd:
  - /cvmfs/alice.cern.ch/bin/alienv
  - setenv
  - AliEn-WorkQueue/v1.3-1
  - -c
    - "work_queue_worker --cores 1 --debug all --single-shot \ 
      --single-task --timeout 60 10.162.223.250 \ 
      $((RANDOM % 12 + 9080))"
docker_privileged: True
max_dock_mem: 2750000000
max_dock_swap: 1000000000
binds:
  - /cvmfs:/cvmfs
```
Code algorithms and solutions

# Wrap API calls and catch exceptions to provide "robustness"
def robust(tries=5, delay=3, backoff=2):
    def robust_decorator(f):
        @wraps(f)
        def robust_call(self, *args, **kwargs):
            ltries, ldelay = tries, delay
            while ltries > 1:
                try:
                    return f(self, *args, **kwargs)
                except re.ConnectionError, e:
                    msg = "[%s], Failed to reach docker, retrying in %d seconds." % 
                    (f.__name__, ldelay)
                    self.logctl.warning(msg)
                    self.logctl.warning(e)
                    time.sleep(ldelay)
                    ltries -= 1
                    ldelay *= backoff
                except re.ReadTimeout, e:
                    msg = "[%s], Failed to reach docker, retrying in %d seconds." % 
                    (f.__name__, ldelay)
                    self.logctl.warning(msg)
                    self.logctl.warning(e)
                    time.sleep(ldelay)
                    ltries -= 1
                    ldelay *= backoff
                except de.APIError, e:
                    msg = "[%s], Failed to successfully evade API request, retrying in %d seconds" % 
                    (f.__name__, ldelay)
                    self.logctl.warning(msg)
                    self.logctl.warning(e)
                    time.sleep(ldelay)
                    ltries -= 1
                    ldelay *= backoff
                except Exception, e:
                    raise
                self.logctl.error("Call [%s] failed, after %d tentatives." % 
                    (f.__name__, ltries))
            return f(self, *args, **kwargs)
        return robust_call
    return robust_decorator

Code Snippet 1: The decorator used to wrap and strengthen the API calls.
Apparmor profile to run ptrace calls within containers

In order to make Apparmor allow setuid processes to run *ptrace calls* it is necessary to write a custom profile in which it is explicitly set.

```c
#include <tunables/global>
profile docker-allow-ptrace flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>
  network,
  capability,
  file,
  umount,
  ptrace peer=@{profile_name}, # This allows containers to run ptrace().
  deny @PROC/([^1-9]*)[^/]+rwklx,
  deny @PROC/smrq-trigger rwklx,
  deny @PROC/mem rwklx,
  deny @PROC/kmem rwklx,
  deny @PROC/kcore rwklx,
  deny mount, # This forbids containers to arbitrarily mount filesystem,
  deny /sys/[f]/[^/]+rwklx,
  deny /sys/fs/[c]/[^/]+rwklx,
  deny /sys/fs/cg/[^/]+rwklx,
  deny /sys/firmware/efi/efivars/[^/]+rwklx,
  deny /sys/kernel/security/[^/]+rwklx,
}
```

*Code Snippet 2:* Apparmor profile I wrote to explicitly grant *ptrace calls* into a Docker container.
HTCondor Job configuration file

An example of the `job.config` file used to run a simulation of pp collisions on the volunteer facility of ALICE Torino.

```plaintext
# Submit description file #
#executables = job.sh
#arguments = "2>&1"
#universe = vanilla
#transfer_input_files = sim.C,Config.C,parrot.rc
#output = job$(Process).out
#error = job$(Process).error
#log = job$(Process).log
#local_files = /tmp/mdc1/*
#queue = 10
```

**Code Snippet 3:** Example of an HTCondor job configuration file used to submit a `Sim.C` simulation.
List of References


[10] Matteo Concas. “Plancton: an opportunistic distributed computing project based on Docker containers”. In: 2016. URL: https://agenda.infn.it/getFile.py/access?contribId=189&sessionId=8&resId=0&materialId=slides&confId=10802.


[16] EGI Federation. 2016. URL: egi.eu


[18] GlusterFS.


[24] LHC@home. 2016. URL: lhcathome.web.cern.ch/

[25] Linux Kernel. 2016. URL: kernel.org/category/about.html

LIST OF REFERENCES

[27] Lustre.

[28] LXC. 2016. URL: linuxcontainers.org/


[31] Open Science Grid. 2013. URL: opensciencegrid.org

[32] OpenVZ. 2016. URL: openvz.org/Main_Page


[34] Plancton repository on GitHub.


[38] SETI@home. 2016. URL: setiathome.ssl.berkeley.edu/


[40] stress - tool to impose load on and stress test systems. 2016. URL: linux.die.net/man/1/stress


Academic acknowledgements

I miei ringraziamenti vanno innanzitutto al mio Relatore, il professor Massimo Masera (Università di Torino e INFN), per avermi dato l’opportunità di condurre questo lavoro di tesi all’interno del gruppo ALICE di Torino. A lui in particolare porgo i miei ringraziamenti per la fiducia che mi ha concesso, lasciandomi ampio margine di manovra e iniziativa personale nello sviluppo di questo progetto. Lo ringrazio per i preziosi consigli in fase di design del progetto, che hanno aiutato a definire le linee guida dello stesso, cosa fondamentale per definire un piano di lavoro. Inoltre lo ringrazio per avermi dato l’opportunità e il supporto nel presentare questo lavoro in due eventi importanti sulla scena Nazionale e Internazionale: il Workshop della Commissione Calcolo e Reti dell’INFN all’Isola d’Elba (16-20 maggio 2016) e la conferenza Computing in High Energy Physics (CHEP) a San Francisco (9-14 Ottobre 2016).

I miei più sentiti ringraziamenti vanno poi al mio Co-Relatore, il dottor Dario Berzano (CERN) per vitale apporto di conoscenza, know-how, esperienza, idee, soluzioni e per avermi insegnato a focalizzarmi sugli aspetti rilevanti nella conduzione di un lavoro di questo genere. Lo ringrazio per avere investito parte del suo tempo, anche extra-lavorativo, per seguire e partecipare attivamente lo sviluppo e la messa a punto di Plancton e per avermi fornito un caso d’uso così importante come la messa in produzione in un ambiente Grid reale.

Ringrazio ancora il dottor Stefano Bagnasco, mio Co-Relatore, per aver partecipato insieme alla dottoressa Sara Vallero e al dottor Stefano Lusso, ai meeting iniziali legati a questo lavoro, condividendo esperienze, critiche costruttive e risorse di calcolo.

Ringrazio inoltre Maximiliano Puccio (Università di Torino e INFN), per aver innescato la creazione di questo lavoro (l’idea di sfruttare macchine inutilizzate l’ha avuta lui, originariamente) e per essersi interessato alla prosecuzione dello stesso, fornendo sempre un interlocutore competente e brillante con il quale discutere.

Vorrei inoltre ringraziare tutto il gruppo di ALICE Torino, per avermi accolto in un clima positivo di collaborazione e intesa.