A ground-up approach to High-Throughput Cloud Computing in High-Energy Physics

Dario Berzano

Relatore
Prof. Massimo Masera
Università degli Studi di Torino

Correlatori
Dott. Gerardo Ganis
CERN
Dott. Stefano Bagnasco
INFN Torino

Controrelatore
Prof. Marco Paganoni
Università degli Studi di Milano-Bicocca

Anno Accademico 2012–2013
“Simplicity is the ultimate sophistication.”

Apple II commercial (1977)
Table of Contents

1 Virtualization and Cloud Computing in High Energy Physics 13
  1.1 Computing in the LHC era ................................. 13
    1.1.1 Hierarchy of computing centres ..................... 16
    1.1.2 Event processing paradigm ........................... 17
    1.1.3 Analysis framework and data format: ROOT ............. 18
    1.1.4 The ALICE Experiment ................................. 19
  1.2 Virtualization and Cloud Computing ........................ 21
    1.2.1 State of the art .................................... 23
  1.3 From the Grid to clouds? .................................. 25
    1.3.1 Grid over clouds and more ............................ 26
  1.4 Overview of this work ..................................... 27

2 Building a Private Cloud out of a Grid Tier-2 29
  2.1 Overview of the INFN Torino Computing Centre .............. 29
    2.1.1 Grid VOs and ALICE .................................. 30
  2.2 Rationale of the Cloud migration .......................... 32
    2.2.1 No performance loss through virtualization .......... 32
    2.2.2 Abstraction of resources in the Grid and the cloud .... 32
    2.2.3 Harmonization of diverse use cases in a private cloud ... 33
    2.2.4 IaaS: separating administrative domains ............... 34
      2.2.4.1 Virtual farm administrator ....................... 35
      2.2.4.2 Rolling updates .................................. 35
    2.2.5 Computing on demand and elasticity .................... 36
    2.2.6 Preparing for the near future ........................ 37
  2.3 The cloud controller: OpenNebula .......................... 38
    2.3.1 Datacenter virtualization vs. infrastructure provisioning ... 38
    2.3.2 OpenNebula features ................................. 39
  2.4 Designing and building the private cloud .................... 42
    2.4.1 Management of the physical infrastructure ............. 43
2.4.1.1 Provisioning: Cobbler ......................... 43
2.4.1.2 Configuration: Puppet .......................... 44
2.4.1.3 Monitoring: Zabbix ............................ 46
2.4.1.4 Network connectivity: NAT and proxy ......... 46
2.4.2 Classes of hypervisors .......................... 46
2.4.2.1 Working class hypervisors ...................... 49
2.4.2.2 Service hypervisors ........................... 50
2.4.3 Level 2 and 3 networking ......................... 51
2.4.4 Shared storage ................................ 52
2.4.4.1 GlusterFS .................................. 52
2.4.4.2 Virtual machine image repository ............... 53
2.4.5 Disks of running virtual guests ................... 54
2.4.6 Benchmarks .................................. 56
2.5 Performance tunings and other issues ................. 57
2.5.1 Kernel SamePage Merging issues .................. 57
2.5.2 OpenNebula backing database: MySQL vs. SQLite .... 60
2.5.3 Fast creation of ephemeral storage ................ 60
2.5.3.1 Allocation of large files ....................... 60
2.5.3.2 Filesystem formatting .......................... 62
2.5.4 Direct QEMU GlusterFS support .................. 62
2.5.5 Fast deployment: caching and snapshotting ........ 63
2.5.5.1 LVM .................................. 63
2.5.5.2 QCow2 .................................. 64
2.6 Custom virtual guest images ......................... 65

3 Running elastic applications on the Cloud 69
3.1 Virtualizing the Grid Tier-2 ........................ 69
3.1.1 Preparation and update of a worker node ........... 70
3.1.1.1 Creation of the base image ...................... 71
3.1.1.2 Contextualization ............................. 72
3.1.1.3 Dynamic configuration: Puppet .................. 72
3.1.1.4 Creating images vs. contextualizing ............. 73
3.1.2 Automatic addition of worker nodes to TORQUE .... 74
3.1.3 Nodes self registration to Puppet .................. 76
3.1.4 CPU efficiency measurements ....................... 77
3.1.5 Grid services on the cloud ........................ 77
3.2 Sandboxed virtual farms ............................ 80
3.2.1 Virtual networks: ebtables ......................... 80
3.2.2 Virtual Router: OpenWRT .......................... 82
3.2.2.1 Building a custom OpenWRT image ............... 82
3.2.2.2 Contextualization ............................. 85
CONTENTS

3.2.2.3 Virtual Routers management and security . . . . . . . 86
3.2.2.4 VPN for geographically distributed virtual farms . . 86
3.2.3 OpenNebula EC2 interface: econe-server . . . . . . . . . . . 87
3.2.3.1 EC2 credentials . . . . . . . . . . . . . . . . . . . . . 88
3.2.3.2 Mapping flavours to templates . . . . . . . . . . . . . 88
3.2.3.3 SSH keys and user-data . . . . . . . . . . . . . . . . . 89
3.2.3.4 Improved econe-server Elastic IPs support . . . . . 90
3.2.3.5 EC2 API traffic encryption . . . . . . . . . . . . . . . 91
3.2.4 Elastic IPs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
3.2.4.1 Integration of econe-server and Virtual Routers . . 92
3.2.5 Creating a user sandbox . . . . . . . . . . . . . . . . . . . . . . . 93
3.2.6 Using the cloud . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
3.3 M5L-CAD: an elastic farm for medical imaging . . . . . . . . . . . . 94
3.3.1 Challenges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
3.3.2 The WIDEN web frontend . . . . . . . . . . . . . . . . . . . . . . 96
3.3.3 Processing algorithms . . . . . . . . . . . . . . . . . . . . . . . . . 96
3.3.4 Elastic cloud backend . . . . . . . . . . . . . . . . . . . . . . . . . 98
3.3.4.1 Evolution towards a generic Virtual Analysis Facility 99
3.3.5 Privacy and patient confidentiality . . . . . . . . . . . . . . . . . 100

4 PROOF as a Service: deploying PROOF on clouds and the Grid 103
4.1 Overview of the PROOF computing model . . . . . . . . . . . . . . . . . 103
4.1.1 Event based parallelism . . . . . . . . . . . . . . . . . . . . . . . . 103
4.1.2 Interactivity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
4.1.3 PROOF and ROOT . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
4.1.4 Dynamic scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . 106
4.1.5 Data locality and XRootD . . . . . . . . . . . . . . . . . . . . . . . 107
4.1.6 PROOF-Lite . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
4.2 Static and dynamic PROOF deployments . . . . . . . . . . . . . . . . 109
4.2.1 Dedicated PROOF clusters . . . . . . . . . . . . . . . . . . . . . . 110
4.2.2 Issues of static deployments . . . . . . . . . . . . . . . . . . . . . 110
4.2.3 Overcoming deployment issues . . . . . . . . . . . . . . . . . . . . 111
4.2.4 PROOF on Demand . . . . . . . . . . . . . . . . . . . . . . . . . . 112
4.3 ALICE Analysis Facilities and their evolution . . . . . . . . . . . . . . . 112
4.3.1 Specifications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
4.3.1.1 The analysis framework . . . . . . . . . . . . . . . . . 113
4.3.1.2 Deployment . . . . . . . . . . . . . . . . . . . . . . . . . . 114
4.3.2 Users authentication and authorization . . . . . . . . . . . . . . 115
4.3.3 Data storage and access model . . . . . . . . . . . . . . . . . . . . 116
4.3.3.1 PROOF datasets . . . . . . . . . . . . . . . . . . . . . . 117
4.3.3.2 The AliEn File Catalog . . . . . . . . . . . . . . . . . . . 118
4.3.4  Issues of the AAF data model ............................... 119
  4.3.4.1 Very slow first time data access ......................... 119
  4.3.4.2 Lack of a pre-staging mechanism ...................... 120
  4.3.4.3 Data integrity concerns and resiliency ................. 121
4.3.5 The dataset stager ........................................ 121
  4.3.5.1 Features of the dataset stager ......................... 122
  4.3.5.2 Robustness of the staging daemon ..................... 123
  4.3.5.3 Daemon status monitoring: the MonALISA plugin .. 125
  4.3.5.4 The ALICE download script ............................ 126
  4.3.5.5 The dataset management utilities for ALICE ........ 127
  4.3.5.6 Alternative usage of the staging daemon .......... 127
4.3.6 A PROOF interface to the AliEn File Catalog ............. 128
  4.3.6.1 Dataset queries .................................... 129
  4.3.6.2 Issuing staging requests ............................ 130
4.3.7 The first virtual Torino Analysis Facility for ALICE ... 131
  4.3.7.1 Overview ........................................... 131
  4.3.7.2 Storage and datasets model .......................... 132
  4.3.7.3 Software distribution model ......................... 136
  4.3.7.4 Dynamically changing PROOF resources ............... 136
  4.3.7.5 Virtual machine images and Tier-2 integration .... 136
4.3.8 Spawning and scaling the TAF ............................... 138
4.4 The Virtual Analysis Facility ................................ 139
  4.4.1 Cloud awareness ....................................... 140
    4.4.1.1 PROOF is cloud aware ............................. 141
  4.4.2 Components ............................................. 142
  4.4.3 The CernVM ecosystem .................................... 143
    4.4.3.1 µCernVM: an on-demand operating system .......... 143
    4.4.3.2 CernVM-FS ....................................... 146
    4.4.3.3 CernVM Online .................................... 147
  4.4.4 External components ...................................... 149
    4.4.4.1 HTCondor ......................................... 151
    4.4.4.2 PROOF on Demand .................................. 154
  4.4.5 Authentication: sshcertauth ............................... 155
    4.4.5.1 Rationale ......................................... 156
    4.4.5.2 Components ....................................... 157
    4.4.5.3 The web interface .................................. 158
    4.4.5.4 Users mapping plugins ............................ 159
    4.4.5.5 The keys keeper ................................... 161
    4.4.5.6 Restricting access to some users .................. 162
  4.4.6 PROOF dynamic workers .................................... 163
    4.4.6.1 Chain of propagation of the new workers .......... 163
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.3.3</td>
<td>Extracting a public key to the SSH format</td>
<td>241</td>
</tr>
<tr>
<td>E.3.4</td>
<td>Writing a users mapping plugin</td>
<td>242</td>
</tr>
<tr>
<td>E.3.5</td>
<td>Pool accounts mapping plugin configuration</td>
<td>243</td>
</tr>
<tr>
<td>E.3.6</td>
<td>Configuration file for sshcertauth</td>
<td>244</td>
</tr>
<tr>
<td>E.4</td>
<td>elastiq</td>
<td>245</td>
</tr>
<tr>
<td>E.4.1</td>
<td>Environment and setup</td>
<td>245</td>
</tr>
<tr>
<td>E.4.2</td>
<td>Configuration</td>
<td>246</td>
</tr>
<tr>
<td>E.4.2.1</td>
<td>Sections</td>
<td>246</td>
</tr>
<tr>
<td>E.4.2.2</td>
<td>Sample configuration file</td>
<td>248</td>
</tr>
<tr>
<td>E.5</td>
<td>The VAF client</td>
<td>249</td>
</tr>
<tr>
<td>E.5.1</td>
<td>Loading the configuration files</td>
<td>249</td>
</tr>
</tbody>
</table>

List of Acronyms 251

References 255

Acknowledgements 267
Chapter 1

Virtualization and Cloud Computing in High Energy Physics

1.1 Computing in the LHC era

Large Hadron Collider (LHC) is the world’s largest particle accelerator with its 27 km of circumference, and the world’s largest machine ever built by the man. After an initial attempt in September 2008, continuous data taking operations began in autumn 2009 and temporarily concluded in February 2013 for the first maintenance stop, called the Long Shutdown 1 (LS1).

The main peculiarity of LHC is the capability to collide protons (pp collisions) at a centre of mass energy up to 14 TeV. LHC is also capable of colliding nuclei of heavy ions (AA collisions), and protons with heavy ions (pA collisions). Particles collide in the so-called Interaction Points (IPs), i.e. points where the two particle beams, orbitating in opposite directions, cross each other: at those interaction points, four different major particle detectors have been built, each one of them composed of several smaller detectors.

The major detectors are:

- ATLAS and CMS, the two experiments that discovered the Higgs boson[^110];
- LHCb, a detector designed for studying the physics of the quark b and CP violation signatures;
- ALICE, an experiment especially designed for heavy ion collisions and detection of low \( p_T \) particles, whose main purpose is studying the Quark Gluon Plasma and the first instants following the Big Bang.
Apart from the challenges in High-Energy Physics, these experiments posed several design problems for what concerns the computing infrastructure: in fact, LHC was expected to be the biggest data generator ever made, both in data rate and overall amount of data produced.

Data collected from LHC detectors must be transferred, stored and analyzed many times: an early estimation suggested that 15 PB of new data would have been produced per year, and the first long data taking period exceeded this estimation, measuring an average of 25 PB per year. To deal with such a unique use case, a worldwide computing infrastructure called Worldwide LHC Computing Grid (WLCG)\cite{103} has been created. The WLCG has two main characteristics:

- it is \textit{geographically distributed}: not a single computing centre is capable of dealing with the amount of resources needed to store and process such a large amount of data; this also takes into account the amount of electrical power used for powering the machines and for cooling them;
- it is \textit{federated}: it is not a single enormous computing infrastructure, but a series of relatively small computing centres whose tasks are orchestrated from single central points.

The Grid is a truly unique and very large computing meta infrastructure, as the following numbers clearly show:

- around 10000 users;
- more than 50.000.000 analysis jobs run so far;
- computing power equivalent to 340000 of our fastest cores;
- a total of 180 PB of data currently stored.

In Figure 1.1 we can see how the stored LHC data is divided between the major experiments.

The amount of data clearly shows how a computing model processing it has to be \textit{data-centric}: this means for instance that it is more convenient to bring analysis jobs near their input data rather than accessing data remotely. The reasons why the collected data rate is so high is twofold:

- the collection rate is very high: the fastest LHC experiment is LHCb, capable of collecting up to 1 000 000 events per second;
- the event size is very high as well: the event having the largest event size is ALICE, with almost 6 MB per event in \(Pb – Pb\) collisions.

An overview of the evolution of event rates in particle physics experiments is shown in Figure 1.2: all LHC experiments are on the edges of the plots, and in particular ATLAS and CMS are posing data challenges both for their event rate and size.
Figure 1.1: Repartition of the 177 PB of LHC ROOT files between the major four experiments: data in the pie chart is expressed in petabytes, and it is as February 2013.

Source: https://indico.cern.ch/event/217511/contribution/2.

Figure 1.2: Evolution of data rates in bytes in particle physics experiments: in this plot, LHC experiments are on the edges, posing significant challenges both for their large data rates and event sizes.

Source: https://indico.cern.ch/event/242881/.
1.1.1 Hierarchy of computing centres

Computing centres part of the LHC Grid are organized according to a hierarchical structure called the tiered model. Each tier has different roles and requirements in terms of Quality of Service (QoS), and types of jobs running. There are four tiers in the model[63].

- The Tier-0 is constituted by a single computing centre located at CERN. Despite it is a single tier, it provides about 20% of the total computing resources of the whole WLCG. The Tier-0 is responsible of keeping custody of the first copy of all raw data; it is also in charge of running the first pass reconstruction and distributing raw data to the next tiers.

- Centres part of the Tier-1 receive data from the Tier-0 and are responsible of simulation and reconstruction. The Tier-1 is expected to provide storage for a safe secondary copy of raw data as well and to orchestrate the distribution of reconstructed data to the Tier-2 sites.

- Tier-2 centres are considered the places where user analysis runs: they are expected to provide a safe copy of the reconstructed data, and to provide storage for the results of user analysis.

- Tier-3 centres are not formally engaged in the tiered model and are considered autonomously managed computing clusters. What runs on such clusters, what data is stored there and access policies are not part of the WLCG specifications: they are usually constituted by small computing facilities with access for local users only. Some of them might also decide to provide a resource share to the Grid when necessary.

An extended real time map of the WLCG sites is available on the web[52].

Technological evolution has improved network links and reduced their cost since the creation of the Grid, and the CPU density has increased as well. This has progressively blurred the borders between tiers and their associated roles: nowadays direct transfers between sites belonging to the same tier occur frequently (in particular at Tier-2 level); transfers between “non-neighbour” tiers occur as well (such as Tier-2 sites retrieving data from the Tier-0).

Even if the computing model is essentially datacentric, meaning that data access from the Grid jobs is meant to be local with respect to the site they are running, it might also happen to have remote data access, i.e. a remote site providing data to other sites.

The model is naturally evolving towards a cloud-like storage where there are a few “custodial” storages holding the primary copy of the data and several distributed “caches” in smaller centres that can be lost or wiped without any serious consequences.
1.1.2 Event processing paradigm

In the scope of HEP, the minimum unit of data needed to process is an event, corresponding to a single detector trigger, which in turn translates 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. It is therefore feasible a computing model where we focus more on obtaining a large number of events processed after a certain time rather than concentrating on improving the processing time of a single event: we call it High Throughput Computing (HTC), as opposed to High Performance Computing (HPC).

This kind of computing is often referred to as event based parallelism, or even embarrassingly parallel computing, since it is relatively trivial to obtain a linear speedup in the throughput by simply augmenting the number of computing nodes.

The whole WLCG computing model is essentially a large Workload Management System (WMS) where we run jobs, each one of them responsible of processing bunch of events.

The process of events processing is constituted by several steps: in each step we obtain a summarized representation of the events in the previous step. An overview of those basic steps follows.

- **From detector hits to raw data.** Not all the data collected by LHC experiments is saved to disks: in order to select specific kinds of events and to save space and processing time, there are different levels of triggers. Each level decides whether the measured event is interesting enough to pass to the next level. Trigger levels are numbered: the lowest and fastest level is \(L0\), then \(L1\) and so on. Such trigger levels are performed by the acquisition hardware: the highest and most fine grained level of trigger is called High Level Trigger (HLT) and it is performed by a dedicated farm of computers. Events selected by all trigger levels have a median size of 2 MB, corresponding to the ATLAS and CMS event size (Figure 1.2).

- **Raw data to Event Summary Data (ESD).** Raw data is constituted by the list of detector hits, that are essentially numbers without a particular physics meaning. The process of finding and identifying particle tracks from raw data is called reconstruction, and the result is called Event Summary Data (ESD). Both input and output files have different entries, each one of them being an event. On average, the reconstructed event size is 20 times lower than the input raw data: 100 kB/event.

- **ESD to Analysis Object Data (AOD).** Reconstructed physics events are reduced into smaller files: each Analysis Object Data (AOD) is a collection of
reconstructed events filtered to match the interests of a particular end user analysis. Event size is on average 10 times smaller than reconstructed data: 10 kB/event.

- **AOD to results.** Results are a summary of physics events, collected in the form of histograms, plots and fits, which constitute the final output of physics analysis.

### 1.1.3 Analysis framework and data format: ROOT

The data format used for storing nearly all LHC data is the ROOT file format\(^9\)\(^5\), and each experiment has custom analysis frameworks taking care of all the processing steps described in § 1.1.2: all of them are based on ROOT\(^2\)\(^2\)\(^2\).

ROOT offers a complete framework and environment for developing and running physics analysis, and for storing data in an efficient way. The language used to develop ROOT is C++, and the options for writing user code based on ROOT are C++ and Python, through a module called PyROOT\(^6\)\(^5\).

A peculiarity of ROOT is a certain capability of interpreting the C++ language: its working environment is in fact constituted by a prompt accepting C++ commands as input. It is also possible to run user's code in the form of interpreted C++ macros instead of compiling them. Such features are currently possible thanks to the CINT\(^\[22,109\]\) interpreter, which will soon be replaced by the Cling\(^\[135\]\) interpreter in the first release of ROOT 6, still under development at the time of writing (march 2014).

ROOT’s unique marshalling features allow to write any instance of any ROOT C++ class to a ROOT file. In particular, events are organized in structures called trees, implemented by the class TTree: a tree is an object that gets saved to disk with a structure optimized for a single write and multiple reads. While reading a tree, which usually happens sequentially, proper buffers are filled to speed up data retrieving, and only the current entry is saved on memory to prevent useless memory consumption.

A tree can span multiple files: in fact, when writing a tree, it automatically splits to multiple files when a certain size limit is reached. The division on multiple trees can be easily made transparent to users via methods like the TSelector (a class that allows to write the code necessary to process a single event without directly managing the event loop) or the TChain (a class allowing to loop over a series of trees as if they were a single one).

ROOT includes PROOF (the Parallel ROOT Facility), which provides an integrated framework for parallel and interactive analysis on either a single machine (PROOF-Lite) or multiple communicating machines. PROOF constitutes the central application of the Virtual Analysis Facility, and the main focusing area of the
work related to this thesis: we will describe in detail the Virtual Analysis Facility in § 4.4.

1.1.4 The ALICE Experiment

A Large Ion Collider Experiment (ALICE) is one of the four major LHC experiments, and it is the only one specifically designed to gather information from heavy ion collisions; like all other LHC experiments, it is capable of collecting data from \( pp \) collisions as well. ALICE does therefore provide three specific physics programs.

- The *heavy ions* program is dedicated to study the properties of a state of the matter called Quark Gluon Plasma (QGP), where quarks and gluons are deconfined. The highest center of mass energy at LHC for heavy ion collisions is \( \sqrt{5.5 \text{ TeV}} \), being about 30 times greater than the highest energy available at RHIC. The QGP is detectable through signatures left by the particles which cross it: the ALICE experiment aims at reproducing the conditions of the Universe during the first microseconds after the Big Bang, giving us hints on the formation of the matter.

- The *proton-proton* program provides a baseline for heavy ion collisions with no QGP formation. Moreover, it brings a good contribution to general LHC physics due to the unique features of ALICE: the low magnetic field, performances on low transverse momentum particles and the capability of reconstructing efficiently secondary collision vertices make ALICE capable of detecting and identifying particles over a broad momentum range, from only 100 MeV/\( c \) to 100 GeV/\( c \).

- The *proton-lead* program brings a fundamental contribution to evaluate the effects of the cold nuclear matter, i.e. nuclear effects without QGP formation.

The most peculiar detector in ALICE is the Time Projection Chamber (TPC): the TPC is a large cylinder of 88 m\(^3\), 510 m long, filled with a proper gas mixture. Particles in the TPC ionize the gas mixture: liberated electrons drift to the collecting pad thanks to an electrical field ensuring constant speed. When they are collected, two cylindric coordinates are measured: the \( z \) coordinate, parallel to the cylinder’s axis, is calculated using the drift time, while the radial \( \theta \) coordinate is directly measured by the hit.

ALICE’s TPC contributes significantly to the event size, with its 570 000 readout signals: it also negatively contributes to the event rate, restricting the trigger frequency down to 1 kHz at most, due to the relatively high drift time (\( \sim88 \mu s \)). Despite the low rate, data throughput rate remains very high because of a large
event size, as we can see from Table 1.1. Even if other experiments might reach an event size reduction of more than 90% during reconstruction, in ALICE most of the raw information collected by the detectors is still relevant after the reconstruction, leading to a reduction of only 6% in the event size of Pb-Pb collisions.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p-p</td>
<td>0.41</td>
<td>0.23</td>
<td>44%</td>
</tr>
<tr>
<td>Pb-Pb</td>
<td>5.59</td>
<td>5.26</td>
<td>6%</td>
</tr>
</tbody>
</table>

Table 1.1: Raw and reconstructed (ESD) event size in ALICE from proton and lead collisions: where other LHC experiments are capable of greatly reducing the event size during reconstruction, most of the information collected from the detectors in ALICE is still relevant after reconstruction, leading to less significant reductions.

While fast data access is critical to all LHC experiments, the amount of information still stored in reconstructed ESDs of ALICE is peculiarly high: as a consequence, ALICE has always been on the edge of data access technologies. For a long time, it was the only experiment providing a central file catalog allowing to access files from everywhere in the world using unique URLs (the AliEn File Catalog), just like many “cloud storage” providers are doing now. Moreover, it has been the first experiment adopting XRootD as storage and transfer technology, providing lightweight and resilient mechanisms to manage distributed storages.

As we have seen in § 1.1.1, the WLCG has several computing centres hierarchically divided into multiple tiers. In the ALICE model, apart from the CERN site, only Tier-1 and Tier-2 centres exist and are well defined: local batch systems configured as Tier-3 are not part of the experiment’s model (even if some of them exist in a semi-unofficial form).

The ALICE analysis framework is called AliRoot, and can be easily built and used on many Unix-like platforms. ALICE also features an easily installable Grid middleware called AliEn, that can be used on end user computers as well. While other experiments require the user to log in to “user interfaces” to use the full experiment framework and the Grid middleware, the ALICE user interface is ultimately the user’s personal computer: this is unique in the panorama of LHC experiments, and for this reason user laptops are often said to be the true ALICE “Tier-3s”.

Another possible implementation of ALICE “Tier-3s” is constituted by the PROOF analysis facilities, which are officially part of the computing model and officially supported by AliRoot. An extensive description of the PROOF-based ALICE Analysis Facility (AAF), to which the work described in this thesis contributed significantly, is described in § 4.3.
All the work described in this thesis, though generically applicable to most LHC experiments, has either done within the context of the ALICE experiment or regarded ALICE as the first community of users.

1.2 Virtualization and Cloud Computing

The main topic of this thesis is to study the benefits of virtualization and cloud computing in the context of High-Energy Physics (HEP), and to provide concrete and usable implementations that leverage such benefits.

Virtualization and Cloud Computing are two different concepts, often mistaken, but strictly correlated. A brief explanation of the difference between the two can be: virtualization focuses on the abstraction of resources, while cloud computing focuses on service provisioning.

Virtualizing hardware resources means that we overlay an abstraction layer on top of the physical resources before exposing them to the applications. Virtualization is a concept ubiquitous in computer science, and ranges, for instance, from the concept of cloud storage to entire virtual machines: in the first case we are exposed to an entire “disk” we can use to store our data, but there is likely no correspondence between the “cloud disk” and some physical disk; in the second case, an operating system is run in an environment entirely constituted by virtual or emulated hardware.

Virtualization can be essentially used for one or both of the following two purposes.

- **Emulating resources.** A virtual machine is a set of virtual resources used to run an operating system as if they were a physical computer. Virtual machines can expose virtual pieces of hardware, including the CPU architecture, that are different from the ones available on the physical hardware hosting them: this is why in some cases we talk about emulation.

- **Partitioning resources.** A single piece of an hardware resource can be partitioned into many smaller virtual resources of the same kind: for instance, a virtual machine can be configured with a number of cores not corresponding to physical CPU cores, or a set of physical disks can be configured with appropriate quotas to expose smaller “cloud disks” to the users, like Dropbox or other similar services do.

Fully emulating a CPU architecture is a challenging task from the performance point of view, as it is basically required to reproduce at software level a series of functions which are proper to hardware devices. Although it is possible to emulate an entire architecture on a completely different hardware architecture,
for the purposes of our work we are more interested in virtualization as a way of partitioning large physical resources into smaller virtual ones: that is, in our work we are going to virtualize the same architecture we are running onto.

Virtualizing the same architecture roughly means that, instead of running the code of the virtual machine entirely in a software emulated architecture, it is possible to pass it directly to the underlying hardware with no translation necessary. In 1974, Popek and Goldberg defined a strict set of rules that an architecture must comply to in order to be virtualizable\cite{90}: the evolution of architectures has made the x86 architecture the most widespread in the world, although it does not comply to such requirements, making it impossible to be efficiently virtualized without a great performance cost. With the raise of virtualization technologies, both Intel and AMD implementations of the x86 architecture have provided “extensions” to the architecture specifications (respectively, Intel VT-x\cite{133} and AMD-V\cite{97}) allowing x86 to be efficiently virtualized.

There are several reasons why virtual machines and virtualization technologies became popular.

- With the increase of hardware density, it is common to buy servers with 48 or more physical cores on board: as many use cases are simply less resource intensive than that, partitioning those cores into smaller virtual machines has become a necessity to cut the costs.

- By means of virtualization technologies we can use computing resources more efficiently: if we pack virtual machines with low requirements, we can overcommit resources instead of partitioning them: this means that we can expose more virtual resources than physical ones, by sharing them among the virtual machines.

- With the possibility of running virtual machines, we effectively have a different operating system installation for each virtual machine. The operating system only sees “virtual” resources which are usually more uniform than physical resources: this means that it becomes easier to create a portable service in virtual machines. It is also possible to decouple the management and monitoring of complex hardware from the applications, by creating virtual machines that simply do not have to care about hardware failures, because they are dealt with at another level.

Virtualization technologies are the base of cloud computing. Precisely defining what cloud computing is and what it is supposed to provide was not an easy task\cite{73}: we will try to provide a simplified definition with specific references to the scope of this work.

Cloud computing is a way of leveraging virtualization technologies in order to provide computing resources (CPU, RAM, disk...) as if they were services. In
particular, in cloud computing:

- there is an absolute abstraction between physical and virtual resources;
- there is a clear separation between different administrative domains, meaning in practice that people responsible for configuring a certain virtual service are not the same people configuring the hardware infrastructure;
- there is a certain accent on the ubiquity and distribution of services, meaning that cloud services usually expose standard interfaces in order to be accessed transparently from everywhere;
- cloud services are meant to be scalable, and, being virtual machines the keystone of cloud computing, scalability practically translates to an increase or reduction of the number of running virtual machines.

1.2.1 State of the art

While designing and developing the Grid, the main use case was so unique and peculiar to a certain community that many of the solutions adopted, as we will see (§ 1.3), are far from being general purpose.

On the other hand, the technologies behind virtualization and cloud computing are addressing a broader number of users from different communities, and are therefore meant to be more generic in their application scope. This means that cloud computing, unlike the Grid, has a more diverse community whose developers come from industry, science and even private enthusiasts.

For what concerns virtualization, we will focus on virtualizing the x86 architecture. Several technologies currently exist: the most important ones, used at large scales, are VMware, KVM (the Kernel-based Virtual Machine)\(^{[62]}\) and Xen\(^{[14]}\). Apart from VMware, which has a commercial license, the other two are open source. All of them have similar performance behaviours: while we based our previous work\(^{[15]}\) on Xen, we have decided to use KVM for our new work (§ 2). The main difference is that Xen boots its own kernel and loads the hypervisor's kernel as a privileged virtual machine, while KVM runs inside the Linux kernel and treats virtual machines as kernel processes.

Even if the tools used for the Grid are formally open source, there has never been a notable contribution from communities outside of the Grid. This led to problems of maintenance of several tools, which unfortunately often feature very low “bus factors”\(^{[24]}\).

On the other hand, cloud tools were born in a world where the open source development model is firmly recognized as essential for most success projects even by industries: from a seemingly disorganized open source community\(^{[96]}\), often steered by the decisions of big companies, cloud computing is nowadays a
growing idea with a solid base of standards and tools, coming in several different 
flavours.

The most important actor so far in the definition of cloud computing is Amaz-
on, that in mid 2006 started selling computing resources with a pay-per-use 
billing model. Such products primarily targeted small companies selling web 
services that could not afford the costs and personnel of maintaining a physi-
cal infrastructure for High Availability: Amazon has essentially created the first 
large computing cloud, featuring a web interface and a RESTful\(^{[97]}\) API (the EC2 API\(^{[8]}\)) to manage the lifecycle of virtual machines over an invisible set of com-
puting resources.

From the experience of Amazon, several open source clones have been created 
to manage your own “private” cloud, using the same paradigms (and sometimes 
the same open interfaces) that were de facto standardized by Amazon. So far, 
we are focusing on the lowest level of service that a private cloud can offer: 
Infrastructure as a Service (IaaS), i.e. the ability to manage virtual machines and 
clusters like the equivalent physical ones.

Some notable private cloud management tools, or cloud controllers, close to 
this paradigm, are Eucalyptus\(^{[82]}\), which offers an open source implementation 
of the Amazon EC2 API, CloudStack\(^{[10]}\) OpenStack\(^{[120]}\) and OpenNebula\(^{[75]}\); all 
of them offer similar functionalities with slight differences in terms of what use 
case is best supported, as we will see in § 2.3.1.

Apart from the simplest use case, i.e. running virtual machines, clouds are 
about offering services that leverage unique features of virtualization. Many of 
the services we use on a daily base are based on cloud resources, and the best 
clue to suspect that is that there is no clue: web services such as Google’s Gmail 
or the popular Dropbox “cloud storage” application are cloud services, that most 
likely scale under the hood by increasing or decreasing the number of virtual 
machines involved. In this case we talk about Software as a Service (SaaS), or, 
if the provided service is more like a framework to run end user applications 
on the cloud, a Platform as a Service (PaaS): to this respect, we can define any 
possible implementation of scalable analysis clusters on the cloud for HEP as PaaS 
providers.

As we have said, the cloud, differently from the Grid, has several open source 
and commercial competitors offering their services and platforms. The paradigm 
inherited from the open source communities is to avoid the so-called “vendor 
lock-in”, i.e. avoid being stuck to a certain cloud platform because it would be too 
hard to completely rewrite applications to support another cloud. The necessity 
for creating simple, accessible and open standards for communicating with cloud 
controllers is fundamental: in particular, all cloud vendors currently offer HTTP-
based RESTful control interfaces, all of them implementing their standard—but 
despite the difference in flavours, it must be said that:
the number of operations you can do on a cloud controller are so restricted that it is not impossible to make two clouds interoperate;

- all popular cloud controller tools offers, apart from their proprietary API, another standard API alongside: which is likely to be Amazon EC2, since it is one of the simplest and it was the first de facto standard in terms of cloud APIs;

- solutions to write a cloud application supporting different cloud providers exist and are actively developed: libcloud\textsuperscript{[67]} and Deltacloud\textsuperscript{[111]} are two notable examples.

The cloud, differently from the Grid, has therefore a plethora of different implementations, but it is easy to recognize in each one of them the trend of moving towards common paradigms and to open for portability and interoperability. This necessity comes from the industry world, and one of the reasons why it is not the case with Grid interfaces (which are considered complicated and lack a variety of implementation) is the lack of diversity of the supported community.

### 1.3 From the Grid to clouds?

If we compare the features of the Grid\textsuperscript{[42]} with the characteristics of cloud computing (we have named some in § 1.2), we cannot help but noticing certain similarities: in particular, as the Grid was conceived to be like a “power grid” where you connect your computing application when you need computing power\textsuperscript{[42]} but you do not care where this computing power comes from, the cloud extends the concept to flexible and generic virtual machines instead of a series of computing jobs.

Moreover, where the Grid is essentially focused in providing computing power for computing intensive applications, the cloud is meant to run any kind of services, and it is not limited to the mere number crunching.

Even if the Grid was born to address a specific computing challenge (the computing for HEP experiments and in particular LHC experiments), the idea was to address the whole scientific community to provide seamless access to computing resources for everybody from everywhere, introducing the concept of abstract and geographically distributed computing resources, which was eventually borrowed by cloud computing.

In practice, the development directions of the Grid made it too tailored for HEP and despite its adoption from smaller communities was supposed to be “as easy as plugging an household appliance to the power grid” to get computing power instead, in reality the complexity of the components involved prevented single users and small communities from ever effectively adopting it.
In spite of the fact that there is a single major use case currently based entirely on the Grid, it is still a peculiar use case whose computing requirements are enormous: from this point of view, and with the experience of the first years of data taking from LHC, the Grid has to be considered a very successful computing model, which, despite its shortcomings, is currently providing the computing power for LHC and it also paved the way for cloud computing by providing a huge background of concepts and user experience.

When the Grid was first conceived in early 1990s, the challenges of LHC were still far away and peculiar to the scientific community. Nowadays, the need for distributed computing power is common for many large to small industry companies, and many technologies that are not tied to HEP have been developed (§ 1.2.1), and eventually converged into what we currently call cloud computing. The industry standard of providing an abstraction layer for distributed resources is now cloud computing: being the HEP use case no longer peculiar to this respect, more and more computing centres and people from the HEP community are joining the efforts on industry cloud computing development, with the idea that the cloud will gradually and seamlessly integrate with the existing Grid.

1.3.1 Grid over clouds and more

As we will see in § 2, despite the Grid is the computing model for HEP, various solutions to exploit cloud resources and the cloud computing model in HEP exist.

There are essentially two different aspects to which the computing community in HEP is focusing:

- **applications**: providing new classes of applications capable of running on clouds;
- **infrastructure**: converting existing Grid tiers into cloud providers.

Despite the fact that the cloud computing model sounds attractive and more sustainable than the Grid, it is just impossible to get rid of a well established computing model. Therefore, several Platform as a Service (PaaS) solutions (as seen in § 1.2.1) currently exist to exploit cloud resources through the Grid without changing the interfaces to the end user. We identify two classes of solutions.

- The **datacenter virtualization** technique simply consists in converting an existing Grid computing centre to virtual machines, which provide no particular extra “cloud” functionality with respect to the old physical centre. Two advantages of this approach are the harmonization of the update process (all virtual machines are cloned from a single one), and the possibility to make room for other completely different and Grid-unrelated use cases using the same physical infrastructure (multi tenancy).
This approach basically maintains the Grid and cloud acting in their separate worlds: with a more flexible resource abstraction layer, the Grid becomes a special case of a more generic infrastructure.

- The second approach requires some sort of communication between the Grid and the cloud, and it consists in dynamically launching Grid virtual machines by monitoring a submission interface which stays the same from the user’s point of view.

A Grid-like approach to the problem is constituted by WNoDeS\textsuperscript{[100]}, developed by INFN CNAF, where a single submission queue is monitored, and when a job is assigned to a certain “bait” node, a special virtual machine is launched (or reused) and the job is migrated inside it, while remaining controlled by the same batch system. This approach has the advantage of being closely integrated with current complex Grid infrastructures, and requires no installation of additional cloud controllers to exploit virtualization—however, it is difficult to support other non-batch models alongside.

A more generic approach is constituted by CloudScheduler\textsuperscript{[11]}, which integrates to the HTCondor\textsuperscript{[115]} queue and requests the instantiation of virtual machines by communicating with an existing cloud controller. The presence of an external cloud controller is what makes CloudScheduler more generic.

All these approaches are either related to the infrastructure or the application level, or both, but they do not imply any change in the user’s interfaces: this means that users can possibly benefit from cloud computing without noticing its presence. There are several studies however which inspect the problem of giving users the “power” and relative complexity of launching their own virtual machines: as part of these studies, the Virtual Analysis Facility was born (§ 4.4), aiming to be a personal and virtual analysis cluster very easy to launch by anyone and whose computing resources are accessible in a PaaS fashion. That is, once the Virtual Analysis Facility has been launched, the user forgets about cloud computing and gets her own scalable virtual cluster.

## 1.4 Overview of this work

The work described in this thesis has been motivated by the growing interest in cloud computing by the HEP community. Being a relatively young field to explore, there is finally the occasion to develop new computing models based on the new technology, or bridges to the existing computing models that rely on cloud computing as resource providers.
From the original intention to share resources of the Tier-2 of INFN Torino between Grid and PROOF virtual machines\cite{15}, a more complex and generic work has been developed. The work is essentially divided in two parts.

- The first part of the work was done at the INFN Torino computing centre and focused on infrastructural problems: that is, modifying an existing Grid computing center in order to make it capable of supporting different non-Grid use cases by means of virtualization technologies. This work is described in § 2.

  This part of the work also focused on the conversion and adaption of the preexisting use cases to the private cloud: the migration of Grid worker nodes and Grid services has been particularly dedicated, but it finally allowed to get a robust infrastructure aimed to multi tenancy. Conversion and creation of specific “elastic” applications to run on the aforementioned infrastructure are covered in § 3.

- The second part of the work was done at CERN in collaboration with the PH-SFT group, and in particular the ROOT project for PROOF and CernVM: this phase has been devoted to the realization of the Virtual Analysis Facility, a cloud-independent analysis cluster in a box, and the study and development of appropriate access interfaces to make feasible for the end user to create her personal cluster by eliminating the most common complications.

  The realization of a truly cloud independent Virtual Analysis Facility capable of automatically scaling and giving user “just enough” control but not to the point of making it complicated to use was tricky, and involved setup and modification of existing components, and the development of new ones, including a lightweight method for scaling a virtual HTCondor cluster automatically. The Virtual Analysis Facility and the steps that brought to it are extensively covered in § 4.

The scope of this work in the context of cloud computing is vast, and it covers almost every aspect of it, from the realization of a physical infrastructure of hypervisors to the development of cloud applications. Issues of dealing with distributed storage systems and data access have been addressed as well.

While the findings and results can be considered of a generic application, the primary use case is HEP, and all the testbeds and user support for the new cloud use cases were done within the context of the ALICE experiment.
Chapter 2

Building a Private Cloud out of a Grid Tier-2

2.1 Overview of the INFN Torino Computing Centre

The Computing Centre at INFN and University of Torino is divided into two different sections.

- The “services” section is responsible of hosting and maintaining all the generic University and INFN computing services as well as the physical network infrastructure. Examples of the hosted services are: a mail server, a web server, some SSH public login machines, Windows domain servers, a DNS server and a DHCP server. Services are also responsible of managing fourth-level domains (*.to.infn.it) and firewall policies.

- The “computing” section hosts all the institute’s high-performance computing farms: the Grid Tier-2 farm, which is the largest, is also managed by the same section, whereas custom computing farms for other non-Grid users are merely hosted there.

The work concerning this thesis purposes the harmonization of both Grid and non-Grid use cases within the same physical infrastructure.

As of October 2013, the sole Grid features the following physical resources.

- **Storage.** 552 SATA SAS 7200 RPM disks, ranging from 1 to 3 TB each, for a gross (non-RAID) total of ~1060 TB.

- **Computational resources.** 28 pre-2009 computing nodes with 8 cores, 16 GB of RAM each and 38 post-2010 computing nodes with 24 cores, 48 GB of RAM each (2 GB per core). The latter feature two AMD Opteron 6168
“Magny Cours” processors, with 12 cores per socket and one thread per core. The full processors topology along with the caches topology is depicted in Figure 2.1.

- **Networking.** 140 1 Gbps links and 22 10 Gbps links for the internal network. 1 Gbps link towards the WAN (soon to be migrated to a 10 Gbps uplink).

Figure 2.1: Processors and caches topology of the AMD Opteron 6168 “Magny Cours” nodes. Output generated by the likwid-topology -g command\[^{116}\].

As the core density increased dramatically in the last years, a larger amount of cores can be packed in a smaller space, even requiring less electrical power per core: Torino’s computing center went from a computing power of less than 1 kH$^2$O\[^{39}\] in early 2008 (around 100 job slots) to more than 14 kH$^2$O in late 2013 (around 1400 job slots), as we can see from Figure 2.2. From the same plot we can see an increase of the gross storage in the same period from 50 TB to more than 1000 TB. Its computing capacity is compatible with an average WLCG Tier-2 site.

### 2.1.1 Grid VOs and ALICE

Torino’s Grid computing centre supports four Virtual Organizations (VOs): the major user is the ALICE experiment, which is given a quota of about 90% of the job slots. Quota repartition is shown in Figure 2.3.

Apart from the job slots, ALICE has a Storage Element (SE) hosted in Torino, which currently counts over 550 TB of storage. Torino’s ALICE storage usage and status is reported, along with all the other storages, on the MonALISA\[^{28}\] monitoring web interface\[^{77}\].
2.1 – Overview of the INFN Torino Computing Centre

Figure 2.2: Growth of computing power (measured in HEP SPEC06) and gross disk capacity of the Tier-2 Grid site in Torino from September 2006 to August 2008.

Figure 2.3: Quota of assigned job slots per VO at the Grid Tier-2 in Torino.
2.2 Rationale of the Cloud migration

Starting from December 2011, all the new computing nodes acquired to replace old ones at the Torino’s Tier-2 computing centre were configured as KVM hypervisors (or Virtual Machine Monitors (VMMs)), i.e. hosts capable of running virtual machines. Hypervisors are not operated directly; all cloud actions are orchestrated by the OpenNebula\textsuperscript{75,119} cloud controller.

Our infrastructure of hypervisors running virtual machines can be classified as a private cloud specifically oriented for Infrastructure as a Service (IaaS) applications. It is a cloud because it provides a layer of abstraction between hardware resources and the virtual machines. The cloud is private as it satisfies the computing needs of a restricted community, i.e. all the members of our Institute and University. Finally, it is IaaS-oriented as we provide users a near-low-level service giving the possibility to build their own bare computing infrastructure, i.e. a virtual farm including virtual machines, networks, disks and memory—all of them using software interfaces instead of physically accessing the hardware rooms.

2.2.1 No performance loss through virtualization

The lowest-level component of a cloud infrastructure is the hypervisor running virtual machines: in our cloud only QEMU backed by KVM is used. Migrating an entire computing centre specifically oriented to High Throughput Computing would be pointless if the virtualization layer added a noticeable performance loss: it has been however extensively proven in our very computing centre that virtualizing our typical distributed High-Energy Physics (HEP) applications adds an overhead which is negligible\textsuperscript{15} while performing local disk I/O and RAM access. Such overhead cannot be even measured for long-running applications (a typical Grid job might run for 15 hours continuously) accessing remote data.

Further measurements on the jobs CPU efficiency were conducted during the last two years of running the Grid site almost fully on virtual machines: as we will see in § 3.1.4, such measurements confirm early studies by showing no slowdown due to virtualization as well.

2.2.2 Abstraction of resources in the Grid and the cloud

The main idea behind the Grid was to make a clear distinction between the physical hardware and the “resources”, by interleaving a layer of abstraction which would ultimately allow an application to run and return its results without being concerned by the geographical location and physical configuration of the underlying hardware. It is the same paradigm adopted when we connect an appliance to the electrical grid through an outlet without being concerned either by the
source or the availability of energy\cite{42}.

Currently the Grid is behind the computing model of most major physics experiments (including all LHC ones). Notwithstanding that going beyond the local computing farm by providing some “resources” to minor use cases which could not afford one was amongst its original intentions, a high level of complexity in its implementation effectively prevented small organisations and experiments to adopt it.

Moreover the Grid was specifically designed to run batch applications, i.e. applications that can be placed in a waiting queue and executed as soon as there are enough free resources, while the number of HPC applications incompatible with such paradigm has increased.

If we consider the resources abstraction level provided, the cloud can be considered the natural evolution of the Grid, where we instantiate entire virtual machines instead of running jobs.

2.2.3 Harmonization of diverse use cases in a private cloud

In our specific use case, we are dealing with a private cloud, where our set of resources is concentrated in a single computing centre, differently from the distributed concept behind the Grid. As we have seen, there are many sparse local farms in our computing centre, with the main user being the Grid.

The presence of many small local farms presents a series of problems. Small farms are usually managed by non-expert personnel, requiring a certain time before the resources become operative. Moreover, our minor users usually need high computing power for a small amount of time, meaning that expensive hardware is acquired to stay idle for most of the time. Additional costs, such hardware maintenance, room cooling and power, are to be taken into account. Finally, any hardware problem, such as a broken disk, has a minor impact on a large farm, but it is likely to interrupt the workflow of all the small farm users.

The cloud, as we have seen, constitutes a way to abstract the resources: differently from the Grid, the approach is less invasive, not requiring any special operating system or packages installed to work. The model we are at the time of writing rolling out in Torino is the following: whenever a small research group has some computing needs and invests on new hardware, such new machines become part of the cloud infrastructure by being configured as hypervisors. Small organisation will receive in exchange a number of resources equivalent to the invested money in the form of a virtual farm.

Such approach is an evolution of what has been proposed in the past with the Grid model, but it has not been effective for small organisations. We believe that the adoption of the cloud model presents a number of advantages that effectively ease the configuration of a virtual farm, as we will see in § 2.2.4, whereas
adopting the Grid was considered too complicated and invasive in terms of requirements for many smaller use cases.

Both small use cases and the Grid can run on the same private cloud, sharing the same set of resources. Whenever a small use case is not needed, its virtual farm can be easily turned off or reduced, and the resources thus freed are assigned to something else: it is evident how a cloud infrastructure can greatly help minimizing the waste of resources.

One of the biggest problems of the Grid is the relative difficulty of temporarily reclaiming Grid resources for local usage: by configuring the Grid on the cloud, it becomes instead very easy to turn off some Grid worker nodes to make room for local use cases.

### 2.2.4 IaaS: separating administrative domains

Torino’s computing centre, as many small similar computing centres, supports a variety of use cases within the same physical location. As we have seen, the Grid is the major user, and it used to be maintained by the same personnel working for the computing centre. Other use cases are supported as well: before the adoption of the cloud model, this meant a number of sparse and small computing farms installed in the same location but managed by external personnel—which usually lacked system administration skills and operated the machines on a best effort basis.

One of the cloud models we adopt is the Infrastructure as a Service (IaaS) model: in practice, we provide resources in the form of virtual machines to the end users. Such approach leads to a clear separation of what we call administrative domains: the personnel of the computing centre is now solely responsible of installing, connecting and configuring the hypervisors, while end users can start and configure their own virtual machines.

The major advantage of such separation is that users do not need to access the hardware rooms anymore: they can concentrate straight to their specific operations, such as installing their software, without bothering on mounting the hardware into the racks, connecting redundant power supplies, or connecting their machines to the physical network. Granting access to the hardware rooms to non-expert personnel also exposes a security risk, including the user unplugging the “wrong” cable and causing a downtime on the Grid infrastructure.

From the user’s point of view, the virtual machines start with a preconfigured minimal operating system. There’s no need to install on them special kernel modules or drivers to support some hardware, as the only hardware they see is virtual. Virtual machines come connected to the network, and as soon as they boot they can be accessed through the tools users are currently employing, such as SSH.
In addition, users belonging to small communities do not deal anymore with hardware problems: the hardware their virtual machines see is “virtual” and it never breaks, and whenever a hardware failure occurs, trained computing centre personnel can mitigate the problem transparently, with zero or little notice by the virtual machine user. We will also see in § 2.4.2.2 how even a complete replacement of the hardware can occur without stopping the virtual machines, which can be transparently migrated to other hypervisors when needed.

2.2.4.1 Virtual farm administrator

Separating administrative domains effectively requires an intermediate specialist between the system administrator (taking care of configuring the farm) and the user: the virtual farm administrator practically configures the software for the users on top of an abstraction layer which makes the configuration independent from the hardware, and from the cloud itself.

The virtual farm administrator can typically use a feature which comes for free with the cloud: after completing the configuration of a virtual machine, it can be snapshotted, saved, and replicated as many times as we please. In practice, even a relatively unexperienced administrator can easily create a scalable virtual farm by being sure that the environment on each virtual machine is consistent.

This feature is particularly useful for unexperienced administrators as it does not require any special knowledge to be put into practice. Larger farms (physical or virtual), such as the Grid farm, can also rely on distributed installation and configuration tools to ensure consistency. For such use cases, management software like Cobbler[^30] and Puppet[^93] can be used, and they are still used in Torino to manage the hypervisors infrastructure.

2.2.4.2 Rolling updates

The approach of cloning a single, “golden” virtual machine turns out to be useful even for the Grid services, which we used to manage by means of Cobbler and Puppet when it ran on physical machines. Within the context of a cloud, the Grid is just another IaaS application sharing the same physical resources with the others, and configured by a special virtual farm administrator. Installing, configuring and updating a Grid worker node has historically been an unwieldy task, due to the large number of package dependencies and the vast set of configuration files: due to the frequency of package updates, installing two Grid worker nodes in different days inevitably leads to two virtual machines with different versions of the same packages, making very difficult to track down a security problem whenever it occurs.

In practice, being the configuration of a Grid worker node an error-prone
operation, the advantage of using the IaaS approach on top of a private cloud reduces the update workflow as follows. First, an offline virtual worker node is selected and upgraded properly: if something goes wrong, we can safely eliminate the virtual machine and start anew. When the upgrade is successful, we can save the updated snapshot. Old virtual worker nodes are put in “drain mode”: they will not accept any new job and they will turn off when all current jobs finish running. When each old virtual worker node is stopped, a freshly updated virtual worker node is started to replace it. A specific program to automatically perform the drain and shutdown operation has been created: a description is available in § A.5.

2.2.5 Computing on demand and elasticity

So far we have depicted the migration to a private cloud as something that can help doing the same things that we have done before in a more rational and clear way. However, the essence of cloud computing is in how quick it is to change the way we use the infrastructure, which becomes as easy as turning off some virtual machines and deploying some other virtual machines.

A new class of applications can be adapted to rely on such feature: instead of configuring a virtual static farm, it is possible to have an application that expands and reduces automatically, depending on its “load”, by varying the number of running virtual machines. Applications which can automatically change used cloud resources on demand are called elastic, and they are described in detail in § 3 and § 4.

An elastic application does not usually expose the IaaS interface to the end user: the user does not need to know that its application is running on top of virtual machines. Instead, it exposes the cloud using an additional layer.

A first example, related to the HEP, is the Platform as a Service (PaaS) approach: users submit their own complex compiled programs using the provided platform, which is usually a “batch system” or Workload Management System (WMS). Such WMS can exploit elasticity by requesting more virtual machines in cases it sees too many jobs waiting in the queue, and turn off idle virtual machines in order. This is exactly what the elastiq\textsuperscript{[49]} daemon does, as we will see in § 4.4.7.

Another example is the Software as a Service (SaaS) approach: for instance, a web application running on several virtual machines, with a caching frontend such as Varnish\textsuperscript{[130]}; the schema is illustrated in Figure 2.4. The workload is balanced among the virtual machines: whenever the frequency of accesses increases, we can think of placing a sensor that decides to start new virtual machines for offloading. End users do not need to be aware of the infrastructure details, as they are only see the software interface.
2.2 – RATIONALE OF THE CLOUD MIGRATION

A virtual farm can also be constituted of a static part and an elastic part which is used only during peak loads. In the cloud computing model, it is common to think of applications “overflowing” to the cloud (or “cloud bursting”), or even from a private cloud to a public cloud, such as Amazon EC2[8]: this is possible because our virtual machine ensures a consistent environment which is independent from the cloud we are running on.

Elasticity becomes a very important factor on public clouds when considering their billing model: all the major cloud providers, including Amazon, charge with a per-use policy, meaning that an efficient model to expand the application only when needed ultimately saves money.

2.2.6 Preparing for the near future

Embracing the cloud model by means of a private cloud means being prepared for one of the possible futures of distributed computing in HEP. It is likely that the current model will evolve in a PaaS transparent for Virtual Organization (VO) users where they still submit jobs, but, under the hood, virtual machines will be submitted over a distributed (and maybe federated) infrastructure. The difference between models which keep the presence of virtual machines transparent as well, such as WNoDeS[100] and Cloud Scheduler[11] is that the latter is geographically distributed and the base virtual machines are supposed to be provided by the VOs themselves.
Moreover, such VO-provided virtual machines completely eliminates configuration hassles from the local sites, giving more time to sites administrators to dedicate to the hardware infrastructures and ensuring an environment which remains consistent even across different computing sites.

While a private cloud, as we have seen, can harmonize the exploitation of resources by many differently sized local users, all current private clouds have the ability to expose public interfaces to standard API languages: opening such interfaces to the VOIs is the first step to be prepared for direct virtual machines submission, which will be able to coexist seamlessly with the local use cases by setting up nothing more than proper access credentials and quotas, making the cloud hybrid.

The resulting global infrastructure can be considered a better federated model than the Grid, as no particular requirements other than some hypervisors are required as long as standard interfaces are exposed. The federated model also emerges when it comes to reclaim a certain degree of autonomy on our own local resources: organisations can submit their own properly configured virtual machines running everything they want, and that can be allowed without yielding the complete control on our private infrastructure.

2.3 The cloud controller: OpenNebula

OpenNebula\(^\text{75}\) was the cloud controller of our choice, since the first deployment of our private cloud in December 2011, which used version 2.0. As of January 2014, our current production version is 3.8, while version 4.4 is being tested.

OpenNebula is an open source software having most of the code written in C++ and Ruby. While there are several open source alternatives (OpenStack\(^\text{120}\) being one of the most popular at the moment, which is currently in production at CERN), there are several reasons why we opted for OpenNebula.

2.3.1 Datacenter virtualization vs. infrastructure provisioning

Cloud computing world is settling to two different cloud models\(^\text{68}\): datacenter virtualization and infrastructure provisioning. The first cloud model describes the major computing centre's use case (the Grid), while the harmonization of several minor use cases fits in the second category.

The infrastructure provisioning model is in fact the Infrastructure as a Service (IaaS) model, where there is a very clear separation between the different cloud layers, and in principle a IaaS virtual farm can run on any IaaS provider (it is said it is cloud-independent). On the other hand, the datacenter virtualization model usually features virtual machines which take direct advantage of the
peculiar features of this particular cloud. A virtualized datacenter is usually not independent from the underlying cloud: in our cloud, for instance, we have an external script to “drain” and shutdown Grid worker nodes by communicating both with OpenNebula and with the Grid head node (§ A.5).

OpenNebula features an hybrid, in-between approach which makes it appropriate for the coexistence of both cloud models, as OpenStack does: the situation is shown in Figure 2.5.

![Figure 2.5: Classification of four different cloud products according to their “cloud model” and “flexibility”.](http://blog.opennebula.org/?p=4042)

**Figure 2.5:** Classification of four different cloud products according to their “cloud model” and “flexibility”.

*Source: the OpenNebula Blog, [http://blog.opennebula.org/?p=4042](http://blog.opennebula.org/?p=4042).*

### 2.3.2 OpenNebula features

Compared to other similar products like OpenStack, OpenNebula’s less obtrusive approach and degree of flexibility makes easier the progressive migration of an existing datacenter, which is exactly our use case. A list of what in our opinion are the strong points of OpenNebula follows.

- OpenNebula has to be installed only on a *single* server, which will be the cloud controller: no OpenNebula agent is needed on the hypervisors.

- All monitoring and controlling is performed, in the default configuration, via simple SSH commands. In general, OpenNebula relies very much on
standard Linux facilities, preferring the “reuse” approach to the “rewrite” one.

- OpenNebula is a “real” open source project, meaning that it is not only possible to inspect and modify the source code, but it is also feasible to do. What makes the codebase extremely proper and clear is the appropriate usage of essentially two languages: C++ for the performance-critical parts, and Ruby for higher-level functionalities.

- It features a highly modular architecture, where every component can be customized. A non exhaustive list of the available modules covers: monitoring, cloud backends (KVM, Xen, even Amazon EC2), and datastores (like iSCSI, LVM).

- OpenNebula is robust: the OpenNebula daemon has been always stopped because we needed to upgrade it. We never experienced a single crash. The longest running time for the OpenNebula daemon on our infrastructure was around 8 months.

- OpenNebula is flexible: while it exposes a high level of complexity when defining physical and virtual resources, it also gives the possibility to add higher level interfaces concealing such complexity: this is the case of the EC2 API server (§ 3.2.3) that we use to expose a simpler interface to our end users, without losing in flexibility on the administration’s side.

As all major cloud controllers, OpenNebula exposes a custom RESTful API called OpenNebula Cloud API (OCA), which is exploited by the supplied command line utilities. Using the Ruby OCA bindings[83] we have created many convenience scripts we use for cloud management (§ A). In addition, a web interface called Sunstone is available, which also features the JavaScript noVNC[80] client which uses websockets to perform remote virtual guest access using a modern web browser (Figure 2.6).

Moreover, OpenNebula has a scheduler which is customizable as well: requested instances are placed in a queue and executed according to match making algorithms and quota policies. Usually, cloud controllers closer to the “infrastructure provisioning” model do not feature any queue: when no resources are available, requests simply fail. There are projects to implement a batch-like model on other cloud controllers, such as QUACK[99] for OpenStack.

One of the reasons why we need a queue is to perform rolling updates on our virtualized Grid farm, as described in § 2.2.4.2: “fresh” virtual machines are enqueued and OpenNebula takes care of booting them as soon as “old” virtual machines are shut down.
2.3 – The cloud controller: OpenNebula

(a) The Sunstone dashboard.

(b) A noVNC session in Sunstone showing the OpenWRT welcome screen.

Figure 2.6: Screenshots of the Sunstone OpenNebula web management interface provided with version 3.8.
After three entire years of experience with OpenNebula, we are definitely happy with our choice. It has to be said that in our opinion no cloud solution is better than another: for the moment, OpenNebula appears to match our requirements, and the rapid evolution of cloud products is clearly going to a diverse ecosystem on one side, and towards an harmonization and integration of the access APIs on the other. Most of the cloud infrastructures nowadays use OpenStack: the main reason why we picked OpenNebula instead is that, when we started migrating the infrastructure (late 2011), OpenNebula did not have comparable competitors, and OpenStack did not exist yet.

It is very common nowadays to find cloud products that support other cloud products both in the frontend and in the backend: to make an easy example, OpenNebula can expose an EC2 interface as frontend (§ 3.2.3), and it is also capable of running virtual machines on a cloud speaking EC2 using the Amazon AWS backend.

2.4 Designing and building the private cloud

The Grid Tier-2 computing centre at INFN Torino does not have enough resources to host a test infrastructure alongside a production one: the first **caveat** when building a private cloud in such a computing centre is that we are forced to share some resources between the “old” Grid physical nodes and the new arising cloud—all without disrupting the production services, exploited by a worldwide community.

In particular, the hypervisors, disk servers and cloud controller share:

- the same network infrastructure;
- the same range of IP addresses;
- the same Linux provisioning server (Cobbler);
- the same distributed configuration server (Puppet).

Apart from the particular attention that must be kept in order to avoid a major service disruption, another reason for sharing the same network and services infrastructure is that cloud services were expected to gradually take over Grid services, that in turn were migrated to virtual services—and this is what eventually happened.

Before diving into the details of how our private cloud is designed, it is useful to give an overview of the Grid-independent services already in production that manage the deployment and ensure the configuration consistency of our physical infrastructure.
2.4.1 Management of the physical infrastructure

The physical infrastructure of the Grid Tier-2 was, at the time of the first installation of the hypervisors, already backed by four different classes of services.

- Installation and deployment of new machines is managed by a Cobbler\[^{30}\] service which also keeps the mapping between MAC addresses, IP addresses and domain names.
- Runtime configuration changes are managed by a Puppet\[^{93}\] service.
- Outside network connectivity is managed by a single node performing Network Address Translation (NAT) and acting as an HTTP caching proxy.
- Status of the infrastructure is monitored by a Zabbix\[^{131}\] service.

Such services share the same physical network and the same IP addresses subnetwork. The aim is to use the same services used for the Grid worker nodes to support the hypervisors.

2.4.1.1 Provisioning: Cobbler

Cobbler\[^{30}\] is a Linux provisioning and installation tool. In our computing centre, every physical host is registered to Cobbler with its MAC address, IP address, Fully Qualified Domain Name (FQDN). Cobbler gives the possibility to pick a “profile” to associate to each host: a profile is essentially constituted by an operating system (both RHEL-based and Debian based operating systems are supported) plus some configuration scripts. There is, for instance, a “Worker Node” profile and a “KVM Hypervisor” profile.

The only operating systems deployed in our site are CentOS 6 plus a few nodes which retain CentOS 5 (which are binary compatible respectively with RHEL 5 and 6) for legacy reasons.

In order to install a new physical node, it is sufficient to turn on the Preboot Execution Environment (PXE) boot on it, then tell Cobbler that we want to install the operating system on that node.

Cobbler uses a single database to keep other network databases consistent:

- when PXE boot is on, the \texttt{tftp} service\[^{128}\] is told to serve the operating system to that node;
- the \texttt{dnsmasq} service\[^{37}\] is used to assign fixed IP addresses via DHCP by mapping MAC addresses to IP addresses in the \texttt{/etc/ethers} file, automatically updated;
• the same dnsmasq service is used to map IP addresses to FQDN and to forward external resolve requests by acting as a DNS for the whole Tier-2 subnetwork.

Our CentOS variants are installed automatically on target nodes by using custom Anaconda scripts. The system is configured to boot only once through PXE: if the installation process completes correctly on the target node, Cobbler server is called back and turns off network boot by removing the MAC address from the tftpdboot server.

Thanks to the Cobbler provisioning service, the procedure of installing a new physical node can be fully scripted and automated, and an entire set of new nodes can be put in production consistently in a small amount of time.

The automation provided by Cobbler “forces” system administrators to insert each node with its full information in the Cobbler database, which then acts as a complete and always up-to-date registry of all the hardware installed in the computing centre.

We still need to pass through Cobbler for assigning IP addresses and FQDNs to the virtual machines, since it has full control over the DNS and DHCP databases. For this reason, we have created a “dummy” Cobbler profile and a series of “fake” nodes to make sure that proper addresses are assigned when virtual hosts are up.

Since the mapping between IP addresses and MAC addresses is also separately maintained by OpenNebula in the virtual networks, we have settled a convention for assigning private IP addresses (illustrated in Table 2.1) and we created a Ruby helper script to create an OpenNebula virtual network and the corresponding Cobbler host profiles in one go. The script onvnet-new.rb is illustrated in § A.6.

<table>
<thead>
<tr>
<th>IPv4 Address (decimal)</th>
<th>XX.YY.ZZ.TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC address (hexadecimal)</td>
<td>02:00:XX:YY:ZZ:TT</td>
</tr>
</tbody>
</table>

Table 2.1: Mapping between IP addresses and MAC addresses in the private Tier-2 network for virtual hosts: the prefix 02:00 in the host’s MAC address identifies a virtual host.

2.4.1.2 Configuration: Puppet

Puppet\[^{93}\] is a tool to configure a distributed set of machines. We use Puppet to provide configuration for the Grid physical and virtual worker nodes, as well as hypervisors and service nodes.

Puppet allows to assign configuration profiles to a list of nodes. Such profiles are modular, and modules can be shared across profiles. Each module defines
a configuration bit, as well as the actions to perform when the configuration changes: for instance, it is possible to trigger a daemon restart when something has changed in its configuration. It is not possible to specify the order in which the actions will be executed: order is decided automatically based on service dependencies.

Configuration files for Puppet modules are meant to be independent from the operating system: a module installing a missing package can trigger, for instance, `yum install` on RHEL operating systems, and `apt-get install` on Debian flavours.

Puppet consists of a daemon running on a server, plus some agents running on target machines. No configuration needs to be performed on the agents, as long as the Puppet daemon runs on a machine whose alias is `puppet.domainname`: indeed, such address is contacted by default by a running agent.

The Puppet server keeps a list of configuration files and modules to distribute: whenever a piece of configuration is changed, it is automatically and gradually deployed to all the concerned nodes. Puppet can also be used to ensure that a certain service is running, and to bring it up in case it is not.

Since Puppet is meant to configure and distribute critical pieces of configuration files (including, for instance, private keys or CA certificates), there is a form of authentication which prevents unknown hosts to connect to the Puppet daemon. The normal workflow when adding a new Puppet host is to first connect to the new node and run manually the Puppet agent:

```
puppet agent --onetime --verbose --ignorecache --waitforcert=120
```

Since the new node does not have a certificate, the Puppet client automatically generates and signs a certificate request and sends it to the Puppet server; then, it stands 120 seconds for the signed certificate to come. To customize such waiting time, the switch `--waitforcert` is used.

At this point, we must connect to the server to list and sign the pending certificate request manually:

```
puppetca --list
puppetca --sign --all
```

We will also use Puppet for configuring virtual worker nodes which come up automatically. Since we want those virtual nodes to come up automatically, the manual signing procedure cannot be used. For such cases, Puppet allows to specify a list of “trusted” FQDNs whose certificate requests are signed automatically. This procedure is described in § 3.1.3.
2.4.1.3 Monitoring: Zabbix

The Zabbix\textsuperscript{[131]} monitoring service is used to monitor the status of the infrastructure. Zabbix consists of a service running on a separate machine, saving in a database the status of a set of distributed sensors which periodically collect information from the Zabbix agents running on the monitored hosts. Different information is collected from hosts belonging to different classes.

Zabbix has a discovery feature that we use to automatically add virtual hosts to the monitoring infrastructure. Subsequently, we have extended the auto-discovery to the full infrastructure so that a new node is added automatically whenever a problem is detected.

Our Zabbix has a dashboard configured to show the general status of the infrastructure using the classic “red” and “green” lights. Zabbix is also capable of generating plots from the history of collected data. Screenshots of the Zabbix web interface are shown in Figure 2.7.

2.4.1.4 Network connectivity: NAT and proxy

All hosts with no public IP address within the Grid and hypervisors network can do outbound connectivity using a single host configured to perform Network Address Translation (NAT). The host also logs and monitors the originating connections.

A HTTP caching proxy has been configured on the same host primarily for the purpose of caching package repositories: this is extremely useful in speeding up the download process of new packages during an update. The proxy server used is Polipo\textsuperscript{[89]}, and it is restricted to point to the allowed package repositories.

Yum is configured on every concerned node to point to the proxy server. Moreover, mirrors are turned off and Yum always points to the same location, in order to benefit from the speedup of caching.

2.4.2 Classes of hypervisors

Ideally, when designing a new cloud, we would like to benefit from all the advantages of virtualization. One of the most interesting features is the possibility to migrate a running virtual machine from one hypervisor to another one, in a more or less seamless way, i.e. with zero or little notice from the user.

There are many reasons why one would like to perform virtual machine migration.

- High availability. We may want to turn off a faulty or obsolescent hypervisor with running virtual machines without interrupting the workflow: virtual
(a) The Zabbix dashboard showing a summary grouped by host groups, latest issues and discovered hosts.

(b) A Zabbix graph showing a report of the Grid jobs queue, plotted using the history of collected data.

Figure 2.7: Screen captures of the Zabbix monitoring web interface showing the dashboard and the capability of plotting the history of collected data.
machines can be migrated on a fresh hypervisor while fixing or updating the node.

- **Green computing.** By adopting a packing policy, we can move as many running virtual machines as we can to the fewest possible hypervisors, in a way that we can turn off unused ones by saving some energy.

- **Better performances.** In cloud computing, it is common to present to the virtual machines physical resources that do not actually exist: in other words, the sum of the virtual CPUs running on a single hypervisor might be greater than the total number of physical cores. When some load burst is detected in one of the packed virtual machines, we can just move them to a less loaded hypervisor, and move it back to the “slow” hypervisors when it becomes idle again. It is believed that this technique is common in commercial public clouds when using “small” virtual machine instances.

Being our private cloud essentially oriented for running High Throughput Computing applications, we will not perform any resources overcommit; moreover, since a consistent part of the private cloud is dedicated to running Grid jobs, we could never reach a situation where we have idle hypervisors to turn off. Achieving High Availability (HA) is instead a very interesting application of the migration in our infrastructure.

There are essentially two types of virtual machines migration.

- **Pseudo-migration.** Virtual machines are first suspended to the source hypervisor’s disk (or hibernated). The suspended image is transferred to another hypervisor, where the virtual machine is then resumed. This technique is as slow as the time required to transfer the virtual machine image, and it can pose problems if we are currently running operations that time out.

- **Live migration.** Virtual machines are running with their block devices stored on a shared filesystem with caches disabled. All the hypervisors mounting the same shared filesystem always have at their disposal all the running virtual machines, meaning that it is roughly sufficient to suspend the machine in one place and resume it elsewhere. Some transfer time is still required for the memory and processors state, but it is usually negligible with respect to transferring the whole virtual machine disks.

KVM supports migration without involving the guest operating system\[^{74}\]. Unfortunately we deal with a set of resources which is far from ideal, and we cannot run all virtual machines on a shared filesystem: as we said, our applications are essentially HTC, and running from a shared filesystem with caches
turned off has serious implications on the performance side. Moreover, the availability of high speed (10 Gbps in our context) network interfaces on servers, storage and switch is limited.

It must be said however that live migration is an unneeded feature when running HTC applications: in case of a virtual worker node failure it is generally easier to resubmit the lost jobs somewhere else than maintaining a more complicated (thus more error prone) system for migrating virtual machines before a problem occurs.

After going through considerations
We have therefore divided our hypervisors into two clusters:

- 35 working class hypervisors, running “unimportant” virtual machines from their local disks;
- 4 services hypervisors, running HA services from a shared and redundant filesystem.

A detailed description of the two hypervisor classes follows.

### 2.4.2.1 Working class hypervisors

The so-called working class hypervisors constitute the 90% of our installed hypervisors. As the name suggests, they are meant to run performance intensive applications. The configuration of such hypervisors penalizes reliability and HA in favour of performances. A list of features of such hypervisors follows.

- **Local and non-redundant virtual machine storage.** Working class hypervisors store disks of the running virtual machines locally. In addition, the majority of working class hypervisors does not have any form of disk redundancy (typically RAID 1) due to cost constraints. Any critical failure to the hypervisors and the disks are likely to cause the loss of the running virtual machines.

- **Base images cache datastore and QCow2 snapshotting.** The most commonly used virtual machine base images (for instance: Grid worker nodes) are cached on each hypervisor instead of being retrieved from the shared image repository. Such local image repository constitutes a separated OpenNebula datastore, i.e. a storage dedicated to host virtual machine images. Images are stored as files in the QCow2 format\(^{[122]}\) and they are started as snapshots to reduce the deployment time, as we will describe in more detail in § 2.5.5.

- **Caching policy.** Virtual machines disks are usually stored in two different files: one, the root disk, which contains the base operating system, plus an
additional ephemeral storage. The root disk, which does not happen to be written many times, is configured from QEMU to have a writethrough\textsuperscript{[61]} cache (i.e., read cache on, but no write cache): all write operations are blocked until data is effectively dumped on the hypervisor’s disk. On the other hand, ephemeral disks, where the majority of I/O operations occur, are configured with a writeback\textsuperscript{[61]} policy (i.e., write cache on), which may lead to data loss in case of power failures, but it improves performances.

- **Private networking.** Working class hypervisors have one 1 Gbps network interface which connects them to the private Tier-2 network. All network interfaces of guest virtual machines are bridged to this network. Virtual machines running on these hypervisors cannot have public IP addresses.

Given the list of features of the working class hypervisors, they are ideal for running high performance applications such as many identical worker nodes of a virtual farm. Normally in such cases losing one job’s processed data in case of hardware failures does not matter, as jobs can be resubmitted and input data is stored in external and secure storages: it would be in practice worthless to sustain the costs in terms of money and performances to make such applications more reliable.

### 2.4.2.2 Service hypervisors

Service hypervisors are constituted by high-end servers and they are oriented to run applications where High Availability (HA) is fundamental. Features of service hypervisors follow.

- **Shared and redundant virtual machine storage.** Virtual machines running on service hypervisors have their disks stored on a shared filesystem with no write cache enabled. The filesystem technology used is GlusterFS: there are two identical data servers with two identical disks serving the same data, making the storage fully redundant and fault tolerant. I/O performances are lower than working class hypervisors.

- **Live migration.** The presence of a shared filesystem allows for the live migration of virtual guests between all the hypervisors of the same class. No automatic migration policy is in place: it is common to manually remove all the running virtual machines from one physical host to update its operating system without interrupting any service. The live migration of a running guest takes on average less than 15 s and a user connected via SSH only notices a temporary glitch.

- **Private and public network interfaces.** Apart from a private network interface, service hypervisors have a second network interface connected to a
switch uplinked to the public network. Service hypervisors themselves don’t have any active IP address on such interface, which is however bridged to virtual machines in need of a public inbound network connectivity: this is the common scenario for a Grid service. While the public interface runs at 1 Gbps, the private network interface is a 10 Gbps and it is therefore used for mounting the shared filesystem.

- **Host disks and power supplies redundancy.** Disks containing the host’s operating system are in mirror (RAID 1), and each host has two power supplies connected to two different power sources. Each service hypervisor is thus completely fault tolerant.

At the time of writing, the oldest running service guest has an uptime of over 450 days, which is greater than the uptime of all service hypervisors.

Due to the presence of a public and a private network interface, service hypervisors run Virtual Routers, which are used to provide Level 3 network connectivity to isolated virtual farms. Virtual Routers are described in § 3.2.2.

### 2.4.3 Level 2 and 3 networking

The Ethernet private network infrastructure of the private cloud is managed by a single switch logically divided into three sets of modules:

- a 1 Gbps set of ports for the Grid physical worker nodes and worker hypervisors;
- a 10 Gbps small set of ports for service hypervisors and storage servers;
- a 10 Gbps set of ports uplinked to the public network.

At Ethernet level (OSI layer 2), the first two sets of modules are interconnected to form the private network, while the third is separated.

At IP level (OSI layer 3), there is a variety of networks running on the same private level 2 infrastructure. Only one level 3 network is managed centrally by Cobbler, as explained in § 2.4.1.1: this private network is a Class C network used by all Tier-2 services, Grid nodes, PROOF nodes and cloud services. Network addresses are described in Table 2.2.

The reason why only one network has been kept for a wide range of diverse services is to facilitate access and interoperability between all the hosts and the shared storages during the migration phase: this network is generally considered trusted as all the physical and virtual machines running into this network are operated exclusively by the computing centre’s personnel. Specific firewall and authentication policies are implemented on the single services hosts.
### Table 2.2: IP addresses description of the hypervisors, Grid services and storages private network.

<table>
<thead>
<tr>
<th>Network class</th>
<th>C (with a wider mask[91])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network address</td>
<td>192.168.0.0/21</td>
</tr>
<tr>
<td>Network mask</td>
<td>255.255.248.0</td>
</tr>
<tr>
<td>Hosts IP range</td>
<td>192.168.0.1 ÷ 192.168.7.254</td>
</tr>
<tr>
<td>Number of hosts</td>
<td>2046</td>
</tr>
</tbody>
</table>

Many other small and isolated level 3 networks run on the same physical infrastructure: in the Virtual Router model, described in § 3.2.2, each virtual farm has a set of IP addresses managed by each Virtual Router, and we have configured OpenNebula to isolate such networks from the rest of the infrastructure using `iptables` on the hypervisors: guests belonging to the same virtual farm can only communicate between themselves.

### 2.4.4 Shared storage

The Grid Tier-2 had already a deployed shared storage based on Lustre[117] when we started building the private cloud. Our Lustre deployment included a single Meta Data Server (MDS) and a series of servers configured as Object Storage Server (OSS). Lustre was used essentially to export a directory containing experiments software to the Grid worker nodes, and as a backend for the ALICE Tier-2 storage served by `xrootd`[38,140].

With the creation of a private cloud, we took the opportunity to renew our storage services by providing GlusterFS[114] support.

#### 2.4.4.1 GlusterFS

GlusterFS is a network distributed software-defined storage system capable of aggregating several simple disks (also called bricks) to export a single mount point which scales up to several petabytes. A list of distinctive features follows.

- **Horizontal scalability.** In a set of GlusterFS servers, no server has a “master” role. All servers are considered equal and exchange information between them using a peer-to-peer model, whereas in Lustre the MDS constitutes both a bottleneck and a single point of failure. In principle it is possible to make a GlusterFS setup scale by simply adding new nodes to the cluster.
- **FUSE and NFS-mountable filesystem.** Clients can mount a remote GlusterFS filesystem using the FUSE backend. GlusterFS does not provide any form of local caching, but it exposes a NFS frontend: if mounting a GlusterFS filesystem using NFS we can benefit from caching.

- **Failovers and redirection.** Any of the GlusterFS servers part of the same pool can be contacted, and the client automatically switches to another in case of problems. When a server is contacted to retrieve a file which is not there, it communicates the client which server to contact to retrieve data directly, using a redirector model similar to xrootd’s. The redirector functionality is not available if mounting the filesystem through NFS.

- **RAID-alike functionalities.** GlusterFS provides the functionality to mirror and stripe a filesystem over different disks. It also features a self-healing mechanism that rebuilds filesystem consistency after major failures.

- **Just a Bunch of Disks (JBOD).** Any POSIX directory on a filesystem supporting extended file attributes can be turned into a GlusterFS brick. Files stored by GlusterFS are generally accessible in read mode by directly reading the brick, and extended attributes are used to store GlusterFS-specific metainformation.

- **Easy storage maintenance.** GlusterFS features a set of tools which make easier to replace a brick or to rebalance existing data evenly on every brick. All maintenance operations can be performed online, i.e. without stopping GlusterFS services.

The connection model of GlusterFS, involving many clients and many servers on the same network, is represented in Figure 2.8.

In the following sections we will see how we used the sole GlusterFS filesystem to cover two different use cases in our private cloud: an image repository and a shared filesystem for critical running virtual machines. The connection schema is illustrated in Figure 2.9. A third use case concerning data analysis from a local GlusterFS storage is covered in the context of the “old Torino Analysis Facility (TAF)” (§ 4.3.7.2).

### 2.4.4.2 Virtual machine image repository

GlusterFS is used to export the repository of virtual machine images to every hypervisor. A single storage server is used to export one brick, made of a single logical unit from a storage rack, as represented in Figure 2.9. The rack is connected to the server with a fiber optics interface.
No replication of striping are in place, since they are already available at RAID level.

Before exporting the image repository with GlusterFS, which is seen by OpenNebula as the default datastore, we used OpenNebula’s capabilities of transferring images by means of SSH. Even if this use case does not leverage the peculiar capabilities of GlusterFS, migrating to a non-encrypted file transfer solution greatly speeded up virtual machines deployment, as we can see in Table 2.3: tests were conducted copying the same 4 GB file using either scp or cp.

A more exhaustive benchmark on the GlusterFS filesystem has been conducted: results are presented in § 2.4.6.

2.4.5 Disks of running virtual guests

As already explained in § 2.4.2.2, a shared filesystem based on GlusterFS is used to store the disks of those virtual guests running on the service hypervisors. As those hypervisors are configured to run HA-oriented virtual guests, and since the disks are the most critical part of the system, disk servers are configured to avoid
Two storage servers with 10Gbps interface provide some of the LUNs through GlusterFS. Services System Datastore is shared to allow live migration of the machines. Workers System Datastore is local to the hypervisors disks in order to increase I/O capacity. Images are cached locally to increase startup speed. An ad-hoc script synchronizes the local copies using a custom "torrent-like" tool when new versions of the images are saved. All the virtual machines run on RAW or QCOW file images.

Figure 2.9: Connection schema illustrating how the different GlusterFS filesystems are exported to the different hypervisors. The cached datastore is not a shared filesystem: instead, it is a local filesystem manually synchronized using custom tools.

<table>
<thead>
<tr>
<th></th>
<th>Time taken</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSH</td>
<td>228 s</td>
<td>17.5 MB/s</td>
</tr>
<tr>
<td>GlusterFS</td>
<td>36 s</td>
<td>111.1 MB/s</td>
</tr>
<tr>
<td>Speedup</td>
<td></td>
<td>6.3x</td>
</tr>
</tbody>
</table>

Table 2.3: Transfer of a 4 GB image from the cloud controller to a hypervisor by means of SSH (using scp) and GlusterFS (using cp). The speedup obtained using GlusterFS is substantial.

as much as possible any bottleneck and with many components redundant.

There are two identically equipped disk servers providing the storage for service hypervisors: both of them have redundant power supplies. Servers provide the storage by exporting the local disks (no external disk pool is attached): each disk is mirrored in a RAID 1 configuration.

There are in total two 1 TB bricks, configured in replicated mode on the two different disk servers: this means that if one server goes down, the other takes its place. GlusterFS automatically performs load balancing for disk reads, while disk writes are slower because each write is synchronous between the two nodes.

A mechanism of self-healing is in place: if one node or disk suffers a major failure, it will be automatically rebuilt when coming back online. Self-healing is a very CPU-expensive operation, and GlusterFS automatically divides the process into many threads: for this reason, the two disk servers are two 24 cores systems (hyperthreading is on).

The gluster volume info on such volume shows the two replicas:
A schema of this setup is illustrated in Figure 2.10. In Figure 2.9 the connection schema of both this datastore and the image repository datastore is shown. Benchmarks are available in § 2.4.6.

**Figure 2.10:** A GlusterFS replicated setup consisting of two bricks residing on two different disk servers: this is the setup in place for serving disks of the running virtual guests on the service hypervisors.


### 2.4.6 Benchmarks

Disk benchmarks have been run on the aforementioned volumes by means of the *Bonnie++* tool. The command used to perform benchmarks is the following:

```bash
def i in {1..10} ; do
    bonnie++ -d/dir/of/mntpoint -s5000 -r2500 -f
done```
where 5000 is the size, in MB, of each written file. Bonnie++ is executed ten times and the average and standard deviation of the results is calculated. Bonnie++ normally uses techniques to rule out system caches when reading and writing each file, such as dealing with files which are double the size of the RAM memory. In this case we are using a filesystem which does not perform any caching, thus we pretend that the RAM size was 2500 MB (the -r2500 parameter). Results are reported in Table 2.4 and plotted in Figure 2.11.

The image repository performs faster than the virtual guests volume for several reasons: first of all it is backed by a RAID 5 SAN whose aggregated read performance is better than the one measured on the “raw” local disks installed on the GlusterFS servers for the virtual guests volume. When writing, software synchronization locks between the two GlusterFS replicated bricks are to be taken into account. The overhead added by the FUSE layer is the same in both cases.

<table>
<thead>
<tr>
<th>Volume</th>
<th>Write</th>
<th>Rewrite</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image repository</td>
<td>64.4 ± 3.3</td>
<td>38.0 ± 0.4</td>
<td>98.3 ± 2.3</td>
</tr>
<tr>
<td>Running VMs</td>
<td>47.6 ± 2.2</td>
<td>24.8 ± 1.5</td>
<td>62.7 ± 0.8</td>
</tr>
</tbody>
</table>

Table 2.4: Bonnie++ benchmark results of the comparison between the GlusterFS Image Repository and the Running VMs replicated volume. Results are in MB/s. All tests perform sequential block I/O operations. The Bonnie++ “rewrite” test performs, in order: a read, a write and a read operation on each block before proceeding to the next.

2.5 Performance tunings and other issues

During the first runs of our private cloud, two critical issues were encountered: a performance problem with the ksmd daemon running on the hypervisors, and a database consistency problem with the SQLite backend on OpenNebula.

Alongside, it has been observed that there was room for improvement concerning the deployment speed, i.e. the time before a requested virtual guest is effectively booted—mostly dominated by the time taken to transfer the image to the destination hypervisor.

We describe such issues and their implemented solutions in the following sections.

2.5.1 Kernel SamePage Merging issues

Right after the first setup of the private cloud, a series of Grid virtual worker nodes was deployed on the new hypervisors. After an initial ramp up of the
number of running jobs, a sudden lock up of the worker nodes and the hosting hypervisors was observed: as shown in Figure 2.12, the number of running jobs in the whole site fell from almost 700 down to nearly zero (Figure 2.12(a)), whilst the load (i.e. the number of processes waiting for a blocking kernel operation to finish) on the involved hypervisors went up to over 300 (Figure 2.12(b)).

After two consecutive failed restart attempts of the virtual worker nodes evidenced by two small peaks of ALICE running jobs in Figure 2.12(a), the problem was tracked down to the presence of the ksmd process consuming 100% of a single core and responsible of the locks in Figure 2.12(b).

Kernel SamePage Merging (KSM) is a Linux kernel facility which continuously scans host memory pages searching for identical ones. Memory pages are therefore deduplicated by adding references and marked as Copy on Write (COW): as soon as the page is dirtied (i.e., writing occurs), the reference is dropped and the page is duplicated back again. KSM appeared in Linux kernel 2.6.23 and it is active by default on CentOS 6.

In KVM virtual guests are treated as host processes, and guest memory pages correspond to host memory pages: the KSM facility aims to greatly reduce memory usage when running a large number of “almost identical” virtual machines, such as web servers.

In environments where memory pages change dramatically and continuously, the KSM facility takes an entire core identifying identical pages: not being fast enough, this results in locking all the memory operations for a long time, which
2.5 – Performance tunings and other issues

(a) MonALISA ALICE monitoring showing a decrease in the running jobs.

(b) Zabbix reporting an increase in the load of the involved hypervisors.

Figure 2.12: Monitoring tools showing major issues related to the presence of the Kernel SamePage Merging facility.
is seen as an increased load. Grid worker nodes run jobs that are very likely to trigger a misbehaviour of KSM.

To prevent this problem from ever occurring again, a special Puppet configuration is in place to ensure that the ksm and ksmtuned services\(^\text{[60]}\) are disabled and not running, which translates into the following commands:

```bash
1. service ksm stop
2. service ksmtuned stop
3. chkconfig ksm off
4. chkconfig ksmtuned off
```

2.5.2 OpenNebula backing database: MySQL vs. SQLite

OpenNebula supports different databases to save the current cloud status and the full history: by default it comes configured with a SQLite\(^\text{[125]}\) database on a file.

We have noticed some problems with SQLite on old versions of OpenNebula (2.0, 2.2), that led to corrupted data. Problems were probably due to abrupt interruptions of the oned daemon, also due to power cuts.

However, in order to avoid losing data, we have decided to back OpenNebula with a simple MySQL database hosted on the same node: for a production infrastructure, MySQL scales better and has enough protections against data loss. Moreover, an automatic backup of the database has been set up.

2.5.3 Fast creation of ephemeral storage

In the most common scenario, a virtual guest has a root disk for the operating system and an ephemeral disk for “extra” space.

OpenNebula allows to provide such extra storage as a mere raw space (in this case the guest operating system must take care of formatting it) or as a formatted disk.

Operations that can be sped up are:

- space allocation on the filesystem (we are using files on the hypervisor’s filesystem as disk backends);
- filesystem creation.

2.5.3.1 Allocation of large files

The standard Linux method to allocate some space is using the `dd` command. For instance, to allocate 2 GB writing zeroes in blocks of 1:
The command takes some time to complete because zeroes are actually written on the destination disk.

There are however different techniques to create a large file without actually initializing the space—either by creating sparse files, initialized at the first read, or by using special low-level functions which must be supported by the hypervisor’s filesystem. A sparse file can be spotted by noticing that its actual file size (i.e., the actual size of the blocks allocated on the filesystem) is less than the declared size. While the latter is the simple output of `ls`, the apparent size can be checked with `du`; in bytes:

```
du -B 1 large-file
```

A list of various file allocation techniques follow: all of them will result in a file whose apparent size is 2 GB (2 \( \cdot 10^9 \) bytes).

- **`dd with seek`**. Creates a sparse file: the GNU version uses the low-level `ftruncate()` function. Space is not really allocated until it’s used, and not a single byte is written.

  ```
  dd if=/dev/zero of=large-ddseek bs=1 count=0 seek=2GB
  ```

- **`truncate`**. Uses `ftruncate()` as well to create a sparse file.

  ```
  truncate -s 2GB large-truncate
  ```

- **`fallocate`**. Requires support from the filesystem: XFS and ext4 support it, ext3 does not\(^41\). Invokes the low-level `fallocate()` function: nothing is actually written, but the blocks appear to be allocated.

  ```
  fallocate -l 2000000000 large-fallocate
  ```

A comparison between allocation methods is reported in Table 2.5. It must be noted that neither `truncate` nor `fallocate` offer a protection against allocating more than the available free space, which must be checked separately.

OpenNebula uses the `dd seek` method by default, which however might give some problems if the file has to be used as a swap partition: Linux swap implementation is meant to be efficient and does not support if the file has holes on the disk, which is the case for sparse files.

We therefore patched the appropriate OpenNebula command to use the `fallocate` method instead, but only when we are on a local supported filesystem: our working class hypervisors use the ext4 filesystem, but service hypervisors use GlusterFS which does not support it. In such case it falls back to the `dd seek` method.
Table 2.5: Comparison between different file allocation methods. The real size is the size actually taken on the disk right after its creation (multiple of the block size), as reported by `du -B 1`. The `fallocate` method must be supported by the filesystem: XFS and ext4 do support it.

2.5.3.2 Filesystem formatting

We have decided to use XFS as our default filesystem for ephemeral storages. The filesystem is directly presented as formatted to the virtual guest: OpenNebula takes care of creating it.

XFS was favoured over ext4 because it is faster to create a large filesystem: in particular, ext filesystems initialize the inode bitmap and inode tables during formatting by default. Creating a XFS filesystem almost does not depend on the size, as shown in Table 2.6.

Table 2.6: Time required to format a XFS or ext4 filesystem of various sizes.

<table>
<thead>
<tr>
<th>Size</th>
<th>Time: ext4</th>
<th>Time: XFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 GB</td>
<td>13.1 s</td>
<td>1.5 s</td>
</tr>
<tr>
<td>40 GB</td>
<td>20.0 s</td>
<td>1.6 s</td>
</tr>
<tr>
<td>120 GB</td>
<td>59.8 s</td>
<td>1.7 s</td>
</tr>
</tbody>
</table>

It is also possible to achieve a near-XFS formatting speed by disabling inode tables initialization during formatting with the `lazy_itable_init` option:

```
mkfs.ext4 /dir/ephemeral.img -E lazy_itable_init
```

In this case, initialization will occur upon the first mount: the filesystem will mount and it can be used immediately, because initialization will lazily occur in the background.

2.5.4 Direct QEMU GlusterFS support

Service hypervisors currently run virtual guests on a shared GlusterFS filesystem mounted on the hypervisor via FUSE: the presence of the FUSE layer might con-
siderably slow down I/O operations.

Starting from version 1.3, QEMU integrates direct GlusterFS backend support: at least version 3.4 of GlusterFS is required\textsuperscript{94}. This means that it is possible to use an image on GlusterFS without mounting the filesystem and thus without passing through FUSE.

This is possible because GlusterFS provides \textit{libgfapi}, a library which exposes POSIX-like functions to GlusterFS. To use GlusterFS volumes with QEMU:

\begin{verbatim}
qemu-kvm -drive file=gluster://<host>/dir/vm.img,if=virtio,cache=none
\end{verbatim}

Unfortunately the production infrastructure uses an old version of QEMU (0.12), as it is the one supplied by CentOS 6. It is planned to migrate the hypervisors to alternative operating systems in order to have newer features available for testing more promptly.

2.5.5 Fast deployment: caching and snapshotting

We combine two techniques together to reduce the deployment time:

\begin{itemize}
\item \textit{caching} the most commonly used images on hypervisors;
\item starting the virtual guests as a \textit{snapshot} of the cached images.
\end{itemize}

We initially implemented the caching and snapshot technique by backing our guest disks with LVM logical volumes: this required some heavy modification in the OpenNebula 2.0 LVM Transfer Manager (TM) driver. To ease the maintenance and upgrade process, we migrated to filesystem-backed QCow2 images, which is currently in production.

2.5.5.1 LVM

Early deployments of our private cloud, running OpenNebula 2.0 (subsequently upgraded to 2.2) had their hypervisors configured to store the disks of the virtual guests as LVM logical volumes.

A single \textit{volume group} was dedicated on each hypervisor to the virtual guests. The creation of LVM partitions occurs by means of the \texttt{lvcreate} command and it is almost immediate, like the \texttt{fallocate} or \texttt{truncate} methods described in § 2.5.3.1. At the end of each virtual machine lifecycle, logical volumes were removed with the \texttt{lvremove} command.

At that time, OpenNebula didn’t have multiple datastores. Images were stored in the raw format as files on a single image repository, and virtual machines were started as LVM snapshots. The deployment procedure goes as follows.
1. On the destination hypervisor, we check for a LVM LV containing the dump of the base image: in case it does not exist, a new LV is created with `lvcreate`, and the `dd` command is used to dump on it the image from the GlusterFS image repository. The cached image is immutable.

2. A new LVM snapshot is created for each virtual guest: snapshots have a Copy on Write (COW) model, meaning that only the differences from the originating LV will be written. An example of creating a snapshot:

   ```
   lvcreate -L 4G -s -n one-<VMID> one-<IMAGEHASH>
   ```

3. When the virtual guest is destroyed, only the snapshot is removed: the originating disk is left there for caching purposes.

   We have completely rewritten the OpenNebula LVM Transfer Manager in order to support the aforementioned workflow. When OpenNebula completely restructured its deployment model by introducing datastores and separating the transfer model from the backend, the patched LVM TM was discontinued and we started using QCow2 images to preserve upstream compatibility and reduce our maintenance work.

### 2.5.5.2 QCow2

Our OpenNebula setup is configured, starting from version 3.6, with three different datastores used for image repositories:

- **shared datastore**, a GlusterFS filesystem mounted on every hypervisor, described in § 2.4.4.2;
- **two local datastores** used for caching respectively QCow2 and raw images.

Such cached datastores (`cached_qcow` and `cached_raw`) are used to store the most frequently used images directly on the hypervisors: the presence of multiple datastores allows us to decide selectively which images to cache. When an image is registered in any datastore whose name starts with `cached_`, it is not immediately available on the hypervisors: we have written a custom utility to synchronize the caches, presented in § A.2.

Since the images are cached, deployment time is considerably reduced. Image files are available in the same location on different nodes, being different files: for this reason it is not possible to store persistent disks on caches.

Whenever we need to take a snapshot (or save back) a virtual guest originating from a cached datastore, its image is saved only on the hypervisor. The same
synchronization utility (§ A.2) is used to manually propagate changes on all the concerned hosts.

Cached datastores are represented, along with the other datastores, in Figure 2.9.

The QCow2 (QEMU Copy On Write) format is a file format that can be used to store virtual machine images to be used with the QEMU hypervisor. Virtual machine images can have most of their space empty: the QCow2 format does not actually allocate, at image file level, the empty space, but it delays the allocation for when the space is actually claimed. As a result, QCow2 images vary in size as they are used, and write performances are generally worse than with preallocated space.

Cached images in QCow2 format can be used as the base for creating snapshots: this is permitted by the QCow2 image format, and it is directly supported by the OpenNebula QCow2 datastore from version 3.4 (prior versions also supported it but did not have the concept of datastore).

Although it is possible to store QCow2 images in a “raw” OpenNebula datastore, in such cases the image will be copied to the destination by means of the cp command. When using the special QCow2 datastore, the following command is used instead:

```
qemu-img create -b base.img -f qcow2 vm.img
```

The base image might as well reside on the GlusterFS-mounted image repository, but for performance and stability reasons we prefer to cache the image beforehand: if the image repository becomes unavailable for some reason, running virtual guests are not affected.

### 2.6 Custom virtual guest images

Even if many Linux distributions already come with a ready to use virtual machine base image (for instance, Ubuntu has a dedicated website\footnote{\url{https://help.ubuntu.com/community/UbuntuVirtualBox}} while CernVM\footnote{\url{http://cernvm.fnal.gov/}} is an operating system made explicitly for the cloud), in many cases we still need to perform some customisations or modifications for improving performances and integration with our datacenter.

- **No LVM.** If we create the virtual machine from scratch, our disk layout will be plain and won’t use LVM: in our case it constitutes an additional layer which does not give us any advantage.

- **Serial console.** Even if OpenNebula comes with a web interface supporting VNC access to the virtual machine, QEMU supports mapping a guest serial
port to a virtual console accessible for instance via the `virsh console` command.

Serial consoles need to be enabled on the guest operating system: in most cases they are disabled for security reasons, or because (physical) serial ports are left free for other use cases. On a RHEL-based operating system, using the event-based `upstart` replacement for `init`, serial consoles are enabled by adding appropriate lines in `/etc/init/serial.conf`.

The bootloader should be also configured to output on the serial console, and to tell the kernel to do it as well. If using Grub on RHEL, the configuration resides in `/etc/grub.conf`.

- **No hardcoded MAC addresses.** It happens frequently that the system maps a certain MAC address to a network interface name: when restarting the virtual machine, it will remember the former MAC address and the new network interface will be named `eth1` instead of `eth0`.

  On RHEL 6 systems, the `HWADDR` line should be removed from `/etc/sysconfig/network-scripts/ifcfg-eth0` and udev should be told not to generate such mappings anymore:

  ```
  1 $ echo -n '' > /etc/udev/rules.d/75-persistent-net-generator.rules
  2 $ rm -f /etc/udev/rules.d/70-persistent-net.rules
  ```

- **Mount disks without barriers.** We have noticed that filesystem barriers, on by default on Linux, can cause problems on virtual guests. This problem has also been reported by other sources.[34]

  Barriers are a form of protection to protect data integrity. A barrier forbids all further block writes, until previous transactions are reported on the filesystem journal and committed to the underlying disk. Barriers also come at the cost of performances.

  On virtual guests, barriers might make the virtual guest wait for too long: the guest then thinks that an hardware failure occurred, and remounts the affected filesystem read-only.

  To disable barriers, in `/etc/fstab` add the `barrier=0` mount option:

  ```
  1 /dev/vda2 / ext4 defaults,barrier=0 0 0
  ```

- **Remove SSH host keys.** One of the worst practices in cloud computing is leaving the SSH host keys saved inside the base virtual machine image. SSH host keys must be unique to the host for obvious security reasons, and they must be therefore removed prior to saving the image:
The CloudInit\cite{29} contextualization tool automatically removes SSH host keys at boot, triggering the automatic regeneration performed when starting sshd.

- **Add hooks to fetch OpenNebula “contextualization”**. Contextualization is the process of configuring a virtual machine upon boot according to some custom data provided when deploying it. Usually, contextualization consists in either a shell script or a configuration file to be interpreted by some program.

  According to the cloud controller, contextualization can be made available to the virtual machines in different ways: for instance, on Amazon EC2 the virtual machine must download a file (the “user-data”) from a private HTTP server on a special fixed IP address. OpenNebula instead burns a virtual CD-ROM image containing the script in the root directory and attaches it to the virtual machine.

  The CloudInit\cite{29} contextualization mechanism is capable of probing for different sources (including the OpenNebula’s virtual CD-ROM) and treats the file as a configuration file divided in “modules”.

  Most of our contextualization procedures do not use CloudInit: instead, a simple shell script is provided to the virtual machine. In such virtual machines, CloudInit is not installed, so we have added some custom lines in the /etc/rc.local file, which is the last script to be executed at boot. OpenNebula also provides packages for the major Linux distributions that add appropriate hooks\cite{106}.

- **No swap partition**. It is unnecessary to save a swap partition in the base image, since it only contains dynamic data which makes sense only when the machine is running. No references to a swap partition will be placed in the fstab file either: we assume that the contextualization should take care of creating one from the ephemeral storage, as well as enabling it and modifying proper configuration files.

1. `rm -f /etc/ssh/ssh*key*`
Chapter 3

Running elastic applications on the Cloud

3.1 Virtualizing the Grid Tier-2

Since we have transformed a Tier-2 Grid computing center into a private cloud for multiple tenants, the first use case we needed to support was the Grid itself, this time in the form of virtual machines. We have already discussed the difference between datacenter virtualization and Infrastructure as a Service (§ 2.3.1): in this case we fall into the first category, meaning that our virtual machines were created specifically for our cloud site.

Although the idea of having a virtual cluster running on every cloud is appealing, there are several clear reasons why a site-independent solution was not feasible in our case.

- First of all, the amount and complexity of services running on a Grid site is too high and complicated to configure to allow for a quick migration.
- The migration of the Grid site to virtual machines was expected to happen in a gradual fashion: newly acquired physical nodes become hypervisors for running virtual worker nodes, while old physical worker nodes will never be converted to hypervisors and will just phase out with time. We need an infrastructure capable of seamlessly supporting a mixture of physical and virtual Grid worker nodes.
- When the migration started in late 2011, we were not sure about the quality of service delivered by certain configurations of a cloud infrastructure: being the Grid the main use case of our computing centre, we wanted to test the cloud share of the worker nodes, and provide a fallback to the physical ones in case something went wrong (as it did at the beginning, § 2.5.1).
In this section we will cover all the aspects related to the creation of virtual Grid worker nodes integrated with the existing Grid. We start with the combined methods we use to prepare a single Grid worker node (§ 3.1.1). How we deal with the automatic addition and removal of worker nodes to the local Workload Management System (we use TORQUE\footnote{127}) is covered in § 3.1.2, and in particular the automatic registration of nodes to the Puppet configuration server is described in § 3.1.3. We will also show some efficiency measurements proving how the migration from physical to virtual machines was invisible from the performances perspective (§ 3.1.4). Finally, we will cover the migration of some services to virtual machines in § 3.1.5.

### 3.1.1 Preparation and update of a worker node

Due to constraints related to Grid packages, available only as RPMs, we have chosen SLC 5 as base operating system for our Grid worker nodes (which was subsequently updated to SLC 6 once available). Worker nodes require a 64 bit architecture.

We use and mix three different configuration techniques for our Grid worker nodes:

- **Separate image.** We maintain a specific base image of SLC 5 that we use as worker nodes: starting from a SLC 5 installation, the heaviest components are installed and then the image is snapshotted.

- **Contextualization.** Some host-specific parts are configured at boot time during the contextualization.

- **Dynamic configuration.** Some other configuration parts are maintained by Puppet (§ 2.4.1.2), used both for boot-time configuration and for dynamically changing it while the node is running.

Despite the apparent lack of harmonization of using three different techniques together for configuring a virtual machine, this was necessary to seamlessly integrate the virtual and the physical worker nodes, the latter being already managed by Puppet.

When we performed the first migration, the other potential customer of the cloud was the old Torino Analysis Facility (TAF), which we will see in detail in § 4.3.7. Since it had to share in part the configuration and installed components with the Grid worker nodes, it was easier to create a single base image and specialize it through contextualization. In other words, the contextualization part of the configuration procedure is important as it decides if the virtual machine will become either a Grid or a PROOF worker node.
Figure 3.1: Three configuration techniques combined together used for the virtual Grid and PROOF nodes, which share the same base image: the contextualization decides the node type, while enabling Puppet allows for massive live configuration updates.

The workflow of the three configuration techniques working together is illustrated in Figure 3.1.

3.1.1.1 Creation of the base image

Periodically we prepare a new minimal installation of SLC 5 for general purposes, containing the most recent version of the operating system: this is performed by starting from a previous installation and running:

```
yum upgrade
```

and saving back the upgraded image. Customizations specific to our cloud are performed as of § 2.6.

For configuring the Grid worker node, we always start from a base SLC 5 image. We boot it, then we configure the following additional components:

- the Lustre\[117\] kernel module, which, before CernVM-FS (§ 4.4.3.2), was used to mount the shared software repository;
- TORQUE\[127\], the Workload Management System in use in our Grid site;
- monitoring (Zabbix) and configuration (Puppet) clients.

After this initial step, a standard procedure to make it a Grid node\[56\] is followed. Eventually a Grid base worker node image is snapshotted from the configured “golden” virtual machine.

All the extra components are installed and configured to not start at boot (including TORQUE), and no credentials to access services are stored there: last
minute configuration and authentication credentials will be supplied during the contextualization.

Whenever we want to update the base software of the Grid worker node, we always start over from the updated SLC 5 image: due to the complexity of the dependencies and installation procedure of the Grid packages, it has been noted that it is easy to start over rather than upgrading the old worker node with Yum. The “rolling updates” procedure used to substitute old worker nodes with new ones is explained in § 2.2.4.2.

3.1.1.2 Contextualization

As we can see from Figure 3.1, the same base image with Grid software will become either a Grid worker node or a PROOF node part of the “old” Torino Analysis Facility: PROOF configuration will be covered in § 4.3.7.

Main purposes of the contextualization are to:

• provide the virtual machine proper authentication details to communicate with site services: these include the TORQUE key to join the cluster, and a list of valid SSH public keys to store in the known hosts for trusting some hosts during some automatic operations;

• start the appropriate services: we must configure TORQUE, for instance, to start at boot in this case (it will be left dormant in the PROOF case);

• partition the ephemeral storage (§ 2.5.3) created by the cloud controller into a swap space, a temporary space and the /home for pool accounts;

• create pool accounts (several for each Virtual Organization) and set up quotas on their homes.

The contextualization, as we have seen in § 2.6, is executed synchronously after all other boot operations as a shell script. At the very end of the contextualization we also execute Puppet synchronously and manually, before starting it in background, since many configuration bits are managed there.

Another important aspect of the contextualization is the dynamic “self” addition of the new virtual machine to the TORQUE cluster: the contextualization provides all the necessary credentials to do that. The procedure is explained in detail in § 3.1.2.

3.1.1.3 Dynamic configuration: Puppet

Although a great part of the configuration is performed via the contextualization phase, we give ourselves the degree of freedom of changing the configuration of
the worker nodes when they are running, without redeploying them or revert to contextualization procedures.

The reason why we want to do this is that our Grid worker nodes are not supposed to represent an “elastic” use case (like the ones we will describe later on in § 4.4.7.8), but much simply the virtualized version of our former physical Tier-2, constituted by long-living machines which are rarely rebooted. Via Puppet, we essentially update all the authentication credentials used for the Grid by “ensuring” (in Puppet’s speech) that the latest version of the RPMs containing valid CAs and CRLs are installed. However, other configuration options can be added dynamically on the Puppet master upon need, and the agents will pull them periodically. Our Puppet master is shared between several services and clusters, as seen in § 2.4.1.2.

### 3.1.1.4 Creating images vs. contextualizing

One of the advantages of using cloud infrastructures and virtual machines is the possibility to easily replicate a single configuration as many times as we want. The most straightforward approach is to create a base image containing all that is needed: we will then follow only once a complex list of installation procedures, then we save back the base image and start it many times.

This approach has advantages and disadvantages. Apart from the clear advantage of scalability, other specific advantages are:

- a straightforward configuration approach, requiring standard installation procedures;
- it is guaranteed that the “whole package”, *i.e.* the virtual machine base image, is consistent.

However, this might lead to maintenance problems when it comes to updates. If we consider one of the “worst case scenarios”, *i.e.* the preparation of a Grid worker node base image, the installation and configuration of a consistent list of packages is a tedious task which must be accomplished manually each time a relevant upgrade comes out. It is true that once configured the Grid node will always work as expected, but due to the limited amount of manpower it is a problem not to have a fully automatic procedure creating new Grid virtual machines with the new Grid software.

On the other hand, the philosophy is to:

- prepare only base installations of the operating systems—or even download the versions already prepared by some distributors like Ubuntu\[132\];
- configure *everything* by means of the contextualization only.
In some cases, like when using µCernVM virtual machines (§ 4.4.3.1), it is not even possible to save back a customized base image: in this specific case using the contextualization is non-optional.

The clear advantage of this approach is that the administrator is “forced” to create an automatic procedure for the installation, making feasible to have always fresh and secure software, which is essential on Grid worker nodes. Disadvantages however are as clear:

- the larger the number of operations to do (packages to install), the higher the probability for something to go wrong;
- the cluster will be inconsistent, being formed by virtual machines with different versions of software, making more difficult to associate a problem or a security incident to a certain configuration snapshot;
- installing a large number of packages increases the boot time by orders of magnitude.

An attempt has been made to migrate the whole Grid node configuration procedure during contextualization, leading to boot times that easily trespass 20 minutes—which is unacceptable in a dynamic environment.

### 3.1.2 Automatic addition of worker nodes to TORQUE

The Workload Management System in use in our Grid Tier-2 is TORQUE[^127], is a very popular batch system and it is one of the most widely supported batch systems from the EGI software. The Maui Cluster Scheduler[^118] is used on top of TORQUE.

For compatibility we do not change such setup: new worker nodes will attach to the central TORQUE manager like if they were physical worker nodes. The only difference is that in our new configuration we have part of the cluster virtual and dynamic, meaning that we are potentially adding (and removing) new nodes constantly.

Unfortunately, the combination of TORQUE and MAUI does not play very well with dynamic environments: for instance some services need to be restarted after modifying the list of nodes, as even recommended by the user’s manual of TORQUE[^2], which might cause in some cases temporary problems in job submission. Apart from that, the biggest problem is that new nodes do not autoregister to the central manager: instead, the list of nodes must be manually modified to make the pbs_server aware of the changes, which involves a series of workarounds in our current dynamic setup. Likewise, when nodes are no longer available they are not automatically garbage collected.
3.1 – Virtualizing the Grid Tier-2

Other batch systems as HTCondor (see § 4.4.4.1) play instead very well with dynamic environments (by featuring for instance autoregistration of new nodes): unfortunately it is considered not being supported equally well by the EGI software.

The mechanism of autoregistration and removal of nodes is performed at the end of the contextualization, and it consists of:

- a server-side script, running on the TORQUE central manager, managing the technical details of the addition and removal of nodes;
- a client-side contextualization invoking such scripts after boot.

Scripts on the server are associated to specific SSH keys for a certain user: this means that the client simply connects via SSH using a certain key to trigger the command. Appropriate SSH credentials are installed during contextualization:

- the SSH public key of the central manager, to be placed in the list of known hosts;
- the SSH private keys used to trigger the addition and removal of nodes.

When adding a new node, it communicates the server-side script its number of cores. The information is trusted, as the node must have embedded an appropriate private key to connect. The server-side script:

- updates the list of host SSH public keys including the one from the new node: if a previous entry with the same hostname existed, it is removed first;
- removes possible stale node with the same name from the list;
- adds the new node, with the appropriate number of cores, to the list: the node is initially added as “offline”, and the status is changed afterwards.

The hostname of the new node is retrieved automatically from SSH environment variables. Details on the script used for automatic addition of nodes are available in § B.1.

When removing a node gently, the node itself calls a script by performing SSH using a different private key associated to the deletion script. The node is deleted from the list and its public key is removed from the list of known ones. The script used for deleting the nodes is described in § B.2. It is also possible to remove a node from the list by using the “drain nodes” functionality that we have integrated with OpenNebula: the drain script is presented in § A.5.

In none of the cases we restart the pbs_server on the central node: after some minutes, the situation settles automatically. Problems occur especially
when removing nodes, as the MAUI scheduler might keep trying to assign jobs to the removed ones for a while. From our experience we have noticed that it is safer to keep the situation settle by itself rather than abruptly terminating the pbs_server, which might cause instability in the nodes already running jobs.

We have no cleanup method for nodes that terminate abruptly, i.e. without a graceful shutdown: however, TORQUE puts them automatically offline after a while, even if it keeps them in the list. In any case, when a new node having the same name of an old offline node comes up, the old entry is eliminated from the list by the nodes addition script (§ B.1).

3.1.3 Nodes self registration to Puppet

Puppet has an authentication mechanism to distribute configuration snapshots. In a normal, static workflow, the Puppet agent requests a certificate the first time it starts to the Puppet master, which in turn puts the request in a queue.

The system administrator must then connect to the Puppet master, list the pending certificate requests, and sign them one by one or all at the same time:

```
puppetca --list
puppetca --sign [HOSTNAME|all]
```

This manual workflow is secure and it works very well with statically deployed machines. In our environment however we have many virtual machines coming up at different times, so we would like to have a workflow to add them automatically to Puppet, yet in a secure way.

The /etc/puppet/autosign.conf configuration file is optionally used by Puppet to specify, line by line, a list of FQDNs whose certificate request is automatically signed. In our setup, the full chain of MAC addresses, IP addresses and corresponding FQDNs are assigned by trusted services, so adding virtual machine names to the autosign list is considered safe.

Self registration of nodes in Puppet turns out to be pretty straightforward. The problem is that nodes die over time, and new nodes will take over their name, resulting in a clash that will prevent Puppet from autosigning their certificate requests. This means that we must find a way of automatically delete a node, if it exists, before autosigning its certificate.

We have created a script to do so. The script uses the same techniques as the ones used for automatic nodes addition to TORQUE (see § 3.1.2): it is installed on the Puppet master and associated to a certain SSH public key, and it is invoked from the remote virtual machine as soon as it starts during contextualization by performing a ssh using the corresponding private key.

The script, detailed in § B.4, simply tells puppetca to remove the calling node from the list of known ones. After that, autosigning will automatically work.
The reason why a separate script has been created, instead of directly authorizing the puppetca command, is that it has lots of dangerous options that must not be accessible from a remote client. Instead, we want this command to do one thing only: remove a node, which is the node invoking the script.

### 3.1.4 CPU efficiency measurements

Although extensive tests were conducted to prove that for our HEP use cases there is no relevant performance impact due to virtualization (§ 2.2.1), we wanted to collect an important parameter from actual running jobs on the long run.

Being ALICE the main customer of our Grid infrastructure, we compared the CPU efficiency:

\[
\frac{\text{CPU time}}{\text{Total running time}} \tag{3.1}
\]

of our site with another ALICE Tier-2 site non-virtualized. At the time of the measurement, about 70% of the jobs in Torino were run on virtual machines.

Results of the comparison are represented in Figure 3.2: we took the data for one year, starting from November 2011 up until October 2012. The migration happened between December 2011 and January 2012: no effect due to the virtualization is visible in this plot.

We believe that comparing the CPU efficiency is an important value: the greatest impact of virtualization technologies on performances is on I/O, which would normally cause the wall time of jobs to increase without an increase of the time spent in the CPU. In practice, an impact of virtualization can be possibly seen by poorer CPU efficiency results.

From the plot we see that the CPU efficiency varies a lot over the year: this is due to several factors, such as data distribution and job types (user jobs tend to be less efficient than “analysis trains” endorsed by the collaboration, for instance). Being both the Torino and Legnaro site similar, they run the same types of jobs, and this is why there is a certain overlap to the two curves.

During the reference period the average CPU efficiency is even slightly higher in Torino, as we can see from Table 3.1.

### 3.1.5 Grid services on the cloud

The migration of Grid worker nodes to the cloud infrastructure began in December 2011 and, as we have already seen, it is proceeding gradually: old hardware running Grid worker nodes will not be configured as hypervisors. Instead, it will gradually phase out, while new replacement hardware is immediately configured as cloud hypervisors.
Figure 3.2: CPU efficiency of ALICE jobs running on two Tier-2s running similar jobs: Torino and Legnaro. The two plots were taken from November 2011 to October 2012, a period that includes the transition of the Torino site from physical to virtual worker nodes, showing no measurable impact of the migration.

Source: Generated from the MonALISA ALICE repository: http://alimonitor.cern.ch.

<table>
<thead>
<tr>
<th>Grid site</th>
<th>Avg CPU eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legnaro</td>
<td>56.48%</td>
</tr>
<tr>
<td>Torino</td>
<td>58.79%</td>
</tr>
</tbody>
</table>

Table 3.1: Average of the CPU efficiencies in the Legnaro and Torino sites in a period of 12 months, from November 2011 to October 2012, displayed in Figure 3.2.

Source: Generated from the MonALISA ALICE repository: http://alimonitor.cern.ch.
So far we have described the configuration of virtual Grid worker nodes. There are however many services related to the Grid, that we wanted to migrate to the cloud for at least two reasons.

- **High availability.** Virtual machine migration techniques might grant us an improved high availability. Moreover, if a virtual machine running a service is down for some reason, it can be easily restarted from scratch, in the worst case.

- **Pack tiny and little used services.** Being a Grid site requires having a set of services installed and running on the site. Many services are little used by the supported Virtual Organizations, but still we need to run them in order to be considered a Grid site. Requiring little resources, dedicating entire “dense” physical nodes with many cores would be a waste: moreover, it is difficult if not impossible to pack several services on a single operating system installation, due to incompatibility in the installation procedures. By virtualizing such services, we can dedicate many small virtual machines to them, and even use overcommit techniques.

We have already seen how we feature two classes of hypervisors: worker hypervisors tuned for performances and running our Grid worker nodes (§ 2.4.2.1), and service hypervisors (§ 2.4.2.2) tuned for high availability. The latter are the ones we use for our Grid services.

As of February 2014 the following services have been migrated to the cloud and run by means of virtual machines.

- **MyProxy.** The MyProxy[^81] Grid service is used to securely store a long-living user Grid proxy to retrieve derivate proxies whenever it is needed.

- **User interface.** This Grid service runs GSI-OpenSSH[^51] (also known as gsissh) and it is used by some minor VOs to access Grid services. Major VOs, such as ALICE, do not need a Grid user interface, as all Grid services are directly accessible from the client’s machine.

- **BDII.** The Berkeley Database Information Index (BDII) is a hierarchical mechanism to expose information on the site via a LDAP database. On our BDII we store local information, which is then periodically propagated to a central database.

- **CE.** The Computing Element is the machine used by the Grid to transform remote submissions to local submissions. Since the Grid now uses almost exclusively pilot jobs for this purpose, our CE is essentially the central manager of our TORQUE cluster.
**ALICE VO Box.** The “VO Box” is a node supposed to support a VO inside a site, by providing specific services. In the ALICE case, the VO Box is a node with a special installation of AliEn\(^98\), taking care of local ALICE services. The VO Box is in particular responsible of submitting pilot jobs to the CE.

**SE.** Our Grid Storage Element is a machine providing access to data via the SRM\(^{126}\) protocol using an implementation called StoRM\(^{31}\). Access to the GridFTP\(^{50}\) service is provided as well. Those services are not used by ALICE, but they are required to a Grid site nevertheless.

Keeping those services on shared disks that require a certain amount of network bandwidth was feasible thanks to a network upgrade (service hypervisors and shared disks have 10 GbE interfaces) and to an accurate inspection (and reduction) of all the I/O intensive activity of the virtual machines.

Machines providing high data throughput services, such as the actual ALICE SE (which is not the one running Grid services, but a series of XRootD interfaces), will not be migrated to virtual machines for performance reasons.

### 3.2 Sandboxed virtual farms

The provisioning model we adopt is that of the virtual farms, each one associated to a different use case. Some virtual farms are managed directly by the computing centre’s personnel, while other small virtual farms are intended to be managed by the end users.

End users merely see the computing farms (being them real or virtual) as mere tools to achieve as fast as possible their results: given that, and their average little experience in administering systems and networks (which usually boils down to installing a Linux operating system), it is to be expected that they will overlook many important security aspects.

We are currently adopting the IaaS model with some less degrees of freedom: external users running virtual machines on our cloud can start and stop virtual machines as they please, but they cannot, for instance, upload their own image. Base images are administered by the computing centre, giving users a starting point for their customizations which is guaranteed to work on any cloud.

Other barriers are in place, like a network isolation layer, to contain as much as possible potential security holes and misconfigurations.

#### 3.2.1 Virtual networks: ebtables

OpenNebula provides virtual network drivers capable of performing forms of network isolation. To enable isolation, the parameter `VLAN=Yes` must be added to
3.2 – SANDBOXED VIRTUAL FARMS

the OpenNebula network template.

The reason behind isolating virtual farms is to provide a strong layer of security which is always effective, even in the presence of loose security precautions on virtual guests which are not under our control.

We utilise `ebtables`\(^{[112]}\) as a form of network isolation. `ebtables` is a filtering tool to add a level of firewalling in Linux bridges. Every hypervisor has a network bridge that connects all guests running on it to the same private network: using `ebtables`, a hypervisor can restrict at MAC address level the allowed inbound and outbound traffic.

To tell OpenNebula to use `ebtables` on a specific hypervisor, its Virtual Network Manager driver must be set to `ebtables` when creating the host. Whenever a guest is deployed (either following a boot or after a migration), the following `ebtables` rules are added:

```
1   -s ! <mac_address>/ff:ff:ff:ff:ff:0 -o <VNET_IFACE> -j DROP
2   -s ! <mac_address> -i <VNET_IFACE> -j DROP
```

The first rule drops all the incoming traffic not originating from a device belonging to the same virtual network: as the MAC mask suggests, using this method OpenNebula supports a maximum of 256 addresses per virtual network, which is more than enough for our use cases. The second rule prevents MAC spoofing: all the traffic originating from the `VNET_IFACE` are expected to have a fixed MAC address; if the guest attempts to change its MAC address, its traffic will be automatically dropped.

We use class B\(^{[91]}\) networks limited to 254 hosts per virtual farm: the layer 3 structure is reported in Table 3.2. Being a virtual network completely isolated, the only way to access it is through a Virtual Router. Virtual Routers are described in § 3.2.2.

<table>
<thead>
<tr>
<th>Network class</th>
<th>B (with a class C mask(^{[91]}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network address</td>
<td>172.16.0.0</td>
</tr>
<tr>
<td>Network mask</td>
<td>255.255.0.0</td>
</tr>
<tr>
<td>Each farm’s mask</td>
<td>255.255.255.0</td>
</tr>
<tr>
<td>Hosts IP range</td>
<td>172.16.0.0/12-172.16.255.254</td>
</tr>
<tr>
<td>Number of hosts</td>
<td>65534</td>
</tr>
<tr>
<td>Number of hosts per farm</td>
<td>254</td>
</tr>
</tbody>
</table>

Table 3.2: IP address space of the virtual farms.
3.2.2 Virtual Router: OpenWRT

A Virtual Router is a virtual appliance providing functionalities similar to a residential router to a virtual farm. We created a custom Virtual Router appliance in order to satisfy our primary need: giving access to completely isolated virtual farms.

The operating system of our choice is OpenWRT\textsuperscript{[121]}, which is based on Linux and it is primarily intended to run on devices. Because of this, it has an extremely low memory footprint. OpenWRT features exactly the functionalities we need on our virtual networks:

- **DNS forwarder and DHCP server** for the virtual farm (which has a separate domain);
- **Network Address Translation (NAT)** for outbound connections;
- “**port forwarding**” (i.e., DNAT) to expose selected internal services;
- **trivial administration** via both SSH and a web interface;
- **Linux-based**: well-known configuration procedures.

A Virtual Router is started with two network interfaces: one public interface, and the other connected to the isolated virtual network. For this reason they can only be started on the services hypervisors (§ 2.4.2.2). A Virtual Router is completely invisible to the end user, who has no access credentials to its functionalities: our default configuration forwards incoming SSH connections to one virtual farm guest transparently.

Our Virtual Routers have been modified to provide a functionality similar to Amazon’s “Elastic IP”, as we will see in depth in § 3.2.4.

3.2.2.1 Building a custom OpenWRT image

OpenWRT features the same build system as the Linux kernel. To configure, then build it, it is sufficient to do:

```
1  make menuconfig
2  make
```

The first command opens a text-mode menu shown in Figure 3.3. We created a custom build because we are using the trunk version, which provides bugfixes more quickly.

The following options are customized before building.
3.2 – Sandboxed Virtual Farms

Figure 3.3: Configuration menu of the OpenWRT build system, based on the Linux kernel’s `make menuconfig`. As the picture shows, building OpenWRT as a KVM guest is supported.
• We configure it as a 32 bit KVM guest: this way we can use the virtio paravirtualized network driver instead of an emulated network card.

• We enable the acpid daemon to provide support for the “power button” shutdown action: this way the router gently shuts down by reacting to a “software shutdown” signal. We prefer this method to “destroying” the virtual machine abruptly.

• The web interface (Figure 3.4) is configured with a more modern theme (the Bootstrap theme) and with Secure Sockets Layer (SSL) support, not enabled by default.

• CD-ROM support is added: this is the way OpenNebula passes contextualization information to the virtual machine. We need it to configure network-specific parameters at boot time.

![Figure 3.4: A screenshot of the OpenWRT web interface, featuring the LuCI Bootstrap theme, which shows the Port Forwarding configuration section.](image)

When the build is complete, we inject a custom boot time configuration script which mounts the contextualization CD-ROM created by OpenNebula, retrieves specific network information and performs the initial router configuration. Contextualization variables and procedure are explained in § 3.2.2.2.

The contextualization script is injected through the guestfish\textsuperscript{[53]} command line tool, which helps modifying the filesystem of a guest image without starting it snapshotting it again. guestfish is very convenient for small edits and for scripting
modifications: a custom build script which uses it has been created to facilitate the OpenWRT image creation, as described in § A.9.

3.2.2.2 Contextualization

Our custom OpenWRT images support OpenNebula contextualization: we can therefore configure a Virtual Router and its corresponding virtual network entirely from the OpenNebula virtual machine template. Instantiating a Virtual Router instance is the way to create a fully fledged virtual network.

The contextualization procedure consists of four different configuration files.

- **OpenNebula template.** The OpenNebula virtual machine template defines all the parameters and requirements of the virtual machine we are creating. The template in use for our Virtual Routers is reported in § A.7.1.

  The template is interpreted by OpenNebula: in particular, a REQUIREMENTS parameters is in this case used to specify that our Virtual Router can only be executed on “services” hypervisors (§ 2.4.2.2), as it needs to have both a public and a private network interface.

  A special section, named CONTEXT, will be passed as-is to the virtual machine. OpenNebula will burn a virtual CD-ROM with a context.sh shell script in its root directory, along with all the files specified in the FILES variable. The Virtual Router is configured to read this file and act accordingly.

  As already pointed out, this is the only file available to the administrator when configuring the Virtual Router. In addition, a tool that automatizes the creation process in an interactive fashion has been written (see § 3.2.5).

- **Contextualization script.** This is the script performing the actual initial configuration. During the creation of the custom OpenWRT build (§ 3.2.2.1), a hook has been inserted to mount the virtual CD-ROM and run this script.

  The script starts the dnsmasq service for DNS and DHCP. The DHCP service is non-authoritative and binds specific MAC addresses to IP addresses; moreover the virtual network is isolated with ebtables (§ 3.2.1), in a way that no interference with other DHCP services might ever happen.

  Another contextualization task is setting the administration password and moving the administration ports to 60022/TCP for SSH and 60443/TCP for HTTPS. This has the consequence of freeing ports 22 and 443 which can be used for forwarding connections to the interal network in a natural way.

- **SSH public key.** A SSH public key is written in the authorized_keys file to allow for passwordless (and automatic) external administration operations.
This feature is currently in use by the Elastic IP functionality (§ 3.2.4).

- **Elastic IP setup script.** This shell script is bound to the authorized public key and it is used to change which internal IP address is associated to the public “elastic” IP. It makes use of the uci[129] command-line tool of OpenWRT to modify the configuration consistently. Details are given in § 3.2.4.1.

### 3.2.2.3 Virtual Routers management and security

Creating a virtual network is as easy as starting a new Virtual Router, which ultimately constitutes an effective solution for introducing a form of Software Defined Network (SDN) without messing with our physical router configuration.

In fact, our custom OpenWRT Virtual Router has both SSH and HTTPS (Figure 3.4) interfaces for management which are completely independent from our physical router. Forwarding policies and firewalls for each virtual network are defined at Virtual Router’s level by means of those interfaces.

Administration is only possible by the computing centre’s personnel for security reasons. It is generally not ever needed to connect to the administration interface: forwarding the SSH port to a certain host is an operation which can be securely performed by the end user by means of the Elastic IP functionality, as we will see (§ 3.2.4).

For special configuration needs (such as forwarding of ports different than SSH), one has to contact the computing centre’s personnel, which is ultimately in control of the security policies.

Being each Virtual Router associated to a single public IP all the outgoing connections from the corresponding virtual network are seen as originating from the single public IP. Separating virtual networks public IPs from the Tier-2 single public IP makes much easier to identify a potential threat or offense generated by misuse of our computing infrastructure—and it also makes shutting down the offending network easy as destroying a Virtual Router.

We can therefore conclude that Virtual Routers are a powerful tool which gives unexperienced end users the complete freedom of running custom virtual machines while moving the security to an upper layer which is in control of skilled personnel.

### 3.2.2.4 VPN for geographically distributed virtual farms

Virtual Routers have also been tested with success to create a single transparent Virtual Private Network (VPN) between our cloud and INFN Roma Tor Vergata’s OpenNebula infrastructure[72] by means of OpenVPN[105]: unmodified virtual guests running on both infrastructures were able to communicate to each other
3.2 – SANDBOXED VIRTUAL FARMS

as if they were in the same LAN, with two invisible virtual routers tunneling the connections. The resulting layout is similar to the one illustrated in Figure 3.5.

Figure 3.5: Layout of a typical OpenVPN setup providing OSI Level 2 connectivity tunneled between two remote LANs. This setup has been tested using OpenWRT Virtual Routers to tunnel a private network in our OpenNebula cloud to an equivalent network in the INFN Roma Tor Vergata OpenNebula cloud.


3.2.3 OpenNebula EC2 interface: econe-server

OpenNebula, as all cloud services, features a custom HTTP RESTful API called OpenNebula Cloud API (OCA). All the OpenNebula one* control utilities use that, and OCA bindings for many programming languages exist: all our custom OpenNebula scripts make use of the Ruby OCA bindings\[83\].

OpenNebula gives the possibility to expose the Amazon EC2 interface through a separated Ruby daemon, called econe-server, which translates EC2 requests to appropriate OCA requests. The process is represented in Figure 3.6 Once the daemon is started, it processes incoming HTTP RESTful requests on a separate port.

While OpenNebula API is a complex set of procedures allowing for a complete management of every detail of the cloud infrastructure (including datastores, hypervisors and virtual networks), the EC2 interface is more user-oriented and exposes simplified functionalities: we found the standard OpenNebula interface being too complex for the unexperienced user, while EC2 commands are more straightforward. Moreover, EC2 is a de facto standard supported by many cloud controllers, including OpenStack\[120\]: this means that in case we will migrate to a different cloud controller we will still be able to retain the same interface, reducing the impact for the users.

Exposing a standard interface is also the first step of making our private cloud
Figure 3.6: Schema representing how the OpenNebula econe-server translates incoming EC2 requests to OCA requests. econe-server is backed by the HTTP Thin\textsuperscript{[123]} web server and the Sinatra\textsuperscript{[124]} web applications framework.


ready to be integrated in a federation.

3.2.3.1 EC2 credentials

The EC2 API needs a pair of credentials to authenticate, which correspond to their OpenNebula counterparts.

- **EC2 Access Key ID.** It is mapped to the OpenNebula username. This information is not secret.

- **EC2 Secret Access Key.** It is mapped to the SHA1-hashed plaintext OpenNebula password. For instance, the OpenNebula password “password” corresponds to the EC2 Secret Access Key given by:

  ```
  > echo -n 'password' | sha1sum | cut -d' ' -f1
  5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
  ```

3.2.3.2 Mapping flavours to templates

OpenNebula uses a text configuration file called virtual machine template to configure all the aspects related to the instance, including:

- base image;
- number of CPUs and amount of memory;
- destination hypervisor;
• virtual networks to use;
• virtual machine architecture (e.g., x86_64 or i686);
• extra disks to attach and their size.

Although this is very powerful for the administrator, it introduces an excessive level of complexity for end users. The EC2 approach of defining a virtual machine is, on the contrary, very straightforward, and it consists in mapping a “base image” to one of the available “flavours”, which roughly corresponds to the amount of “resources” (memory, CPU, disk) to assign. Flavours are conventionally given self-explanatory names: it is clear that a flavour named m1.small will assign less resources than the m1.large one.

OpenNebula supports custom flavours through the EC2 interface by mapping each one of them to a “template of a virtual machine template” in the ERB (embedded Ruby) format. An example is given by the following snippet, mapping the base image specification into an OpenNebula DISK section:

```erb
[1] DISK = [
[2] IMAGE_ID = '<%= erb_vm_info[:img_id] %>',
[3] TARGET = "vda",
[4] CACHE = "none"
[5] ]
```

### 3.2.3.3 SSH keys and user-data

The most common way of creating a virtual guest from EC2 is to supply four parameters:

• the base image (mandatory);
• a flavour (mandatory);
• a SSH public key name (optional);
• a text file called the user-data (optional).

The idea behind supplying a SSH public key is that it should be immediately possible to connect to the virtual guest using the corresponding private key right after its creation.

EC2 manages a database of SSH public keys in a keystore: the econe-server uses metainformation in the OpenNebula backing database to save a keystore for each user. A keypair (a public and a private) is created: the public part is saved in the keystore and it is given a name, while the private part is saved to a file and must be kept confidential by the user.
The public key is “passed” to the virtual machine, which should have in place appropriate mechanisms to fetch it and store it in some SSH authorized_keys file (usually the one belonging to the root user). This mechanism is emulated by using the OpenNebula CONTEXT section in virtual machine templates. An example of the CONTEXT section created follows:

```plaintext
CONTEXT = [
  EC2_KEYNAME="MyPersonalKey",
  EC2_PUBLIC_KEY="ssh-rsa AAAA...5i3hGn MyPersonalKey", # shortened
  ...
]
```

The key will end in the /context.sh file burned on a virtual CD-ROM as the EC2_PUBLIC_KEY variable.

The so-called user-data is some “extra data”, usually in plain text format, that will be passed as-is to the virtual guest, which should have appropriate hooks to interpret it correctly. Such user-data is used to trigger the contextualization: it is for instance possible to supply a CloudInit\(^{29}\) configuration file if the guest supports it, or an amiconfig\(^{47}\) file for CernVM\(^{23}\) virtual machines.

The file is converted to base64 and it is written in the OpenNebula CONTEXT section like this:

```plaintext
CONTEXT = [
  EC2_USER_DATA="IyEvYmluL3...RuYW1lXQoK", # shortened
  ...
]
```

The size limit for the raw (non-base64) version of the user-data is 16 KB\(^7\). It is a common technique to supply data which does not fit from external sources (like a configuration drive or a web server).

### 3.2.3.4 Improved econe-server Elastic IPs support

The OpenNebula econe-server supports EC2’s Elastic IPs, a feature that our model relies on, as explained in § 3.2.4. The default econe-server supplied with OpenNebula 3.8 allows for mapping a single OpenNebula virtual network to the pool of Elastic IPs for every user. Virtual network is specified by its ID (as returned by onevnet list) in the econe-server configuration file:

```plaintext
:elasticips_vnet_id: 12345
```

Unfortunately this method exposes a series of IP addresses to all users, while we want to pin a different, single and well known IP address for each user, while preventing her from using the others.
Our solution was to create a “public” virtual network for each user containing a single IP address, and patch the econe-server to include per-user Elastic IP pool support. The patched version is publicly available on our GitHub OpenNebula fork\footnote{84} and allows for specifying templated network names in the configuration file:

```
:elasticips_vnet_name: <USER_NAME>-Pub
```

where `<USER_NAME>` will be substituted with the username making the request.

### 3.2.3.5 EC2 API traffic encryption

econe-server does not support HTTPS out of the box: we therefore mapped a base URL on a HTTPS Apache 2 server to econe-server using a reverse proxy technique. We have used the same HTTPS server we use for the Sunstone web interface.

The base URL exposed to the external world must be communicated to econe-server using the following configuration directive:

```
:ssl_server: "https://one-master.to.infn.it:443/ec2api/"
```

Securing the EC2 API is fundamental as sensitive data (like a user-data containing credentials, see § 3.2.3.3) is sent through this channel.

### 3.2.4 Elastic IPs

The Elastic IP addresses functionality\footnote{6} is a feature supported by the Amazon EC2 cloud that allows the dynamic mapping of a known static and public IP address to the private address of any of my instances.

With the introduction of virtual farms (§ 3.2), each user can start instances in a sandboxed environment. The following two problems arise:

1. user has no control over the assigned addresses: they are assigned in a round-robin fashion from an availability pool;
2. assigned addresses are private, therefore no communication is possible with the external world.

In our model, we assign a single public IP address to each virtual farm: the address is always assigned to the Virtual Router (§ 3.2.2), which is transparent to the end user. Whenever the user creates an instance, she can map the Elastic IP address to that instance by using a standard EC2 client: what happens under the hood is that the Virtual Router is reconfigured in order to forward (DNAT) the SSH port to the specified instance.
This is not a 1:1 mapping, as the sole SSH port is forwarded for security reasons. It is also possible to manually add other port forwardings to the same host, as we have seen in § 3.2.2.3. After that, whenever the Elastic IP is remapped, all port forwardings mapped to the original host will be assigned to the new one.

This model allows users to change port forwardings by themselves without ever touching the Virtual Router interface, and without asking system administrators. A high degree of autonomy is achieved, however the limited and selected number of ports forwarded, security is not compromised. The flexibility of our Elastic IPs model is limited by design, as the users we are targetting are mostly unexperienced.

### 3.2.4.1 Integration of econe-server and Virtual Routers

Whenever a user request to associate or disassociate an Elastic IP to a certain instance, econe-server triggers external actions specified in its configuration file:

```bash
# Args: elastic_ip private_ip vnet_template(base64_encoded)
:associate_script: /path/to/custom/econe-associate

# Args: elastic_ip
:disassociate_script: /path/to/custom/econe-disassociate
```

Commands are invoked by passing the parameters specified in the comments above. We have written custom association and disassociation scripts which communicate the command to the corresponding Virtual Router by means of SSH. When associating an address:

```bash
echo <PRIV_IP> | ssh -p60022 -i<PRIV_KEY> root@<ELASTIC_PUB_IP>
```

When disassociating:

```bash
echo 'disassociate' | ssh -p60022 -i<PRIV_KEY> root@<ELASTIC_PUB_IP>
```

As we have seen in § 3.2.2.2, the Virtual Router has been created with an appropriate script handling the commands received on stdin; plus, the authorized injected public key is bound to one command, and no login is possible. In the authorized_keys file:

```bash
command="/sbin/update-elastic-ip.sh" ssh-rsa AAAAB3NzaC1yc2...
```
3.2.5 Creating a user sandbox

The creation of a user sandbox by the cloud administrator consists of a series of repetitive steps. For convenience, a Ruby script that creates the environment has been written. The script takes only the following three parameters as input:

- the user name;
- public IPv4 address (e.g. 193.205.66.123);
- private IPv4 network (e.g. 172.16.123.0/24).

The following operations will be automatically performed in sequence according to the input parameters.

1. The user is created.
2. A template for the Virtual Router is generated, like the one in § A.7.1.
3. A default limiting quota is set.
4. A template for the private network (e.g. 172.16.123.0/24) is created: the private network will have a maximum of 254 hosts (256 minus broadcast and Virtual Router).
5. A template for the public network, containing one address only, is created: that template will be used by the econe-server Elastic IP functionality, as in § 3.2.4.
6. An Access Control List (ACL) allowing the user to use addresses from the private network is added.

Note that the above operations are not committed by default: instead, proper template files are created, and instructions on what to do are printed on screen for convenience. A special switch makes the script commit the changes immediately: that includes instantiating the virtual router right away.

More technical information on the sandbox creation script is reported in § A.7. Proper HTML documentation is also available on the web\cite{58}.

3.2.6 Using the cloud

The EC2 API can be contacted by any host connected to the Institute’s network, provided that its URL is known. In order to provide a ready to use client environment, euca2ools\cite{40} have been installed on the public login machines. euca2ools are a set of EC2-compliant command-line clients part of Eucalyptus\cite{82}.

euca2ools needs at least the following three environment variables set:
• \$EC2\_ACCESS\_KEY and \$EC2\_SECRET\_KEY, whose purpose and format is described in § 3.2.3.1;

• \$EC2\_URL, the URL used to contact the EC2 API server.

A helper script named cloud-enter installed on the public machines prompts user for username and password and sets the environment for controlling the cloud. Since we are using HTTPS, the issuing CA certificate has been added to the list of authorized ones in the user’s environment.

For the moment, the following user operations can be performed via EC2 on our private cloud:

• creation, deletion and listing of SSH keypairs (see § 3.2.3.3);

• listing current base images;

• creation, termination and listing of instances;

• allocation and association of Elastic IPs (see § 3.2.4).

A user’s guide is available on the web\cite{58}. The available virtual machine flavours (§ 3.2.3.2) are reported in Table 3.3.

<table>
<thead>
<tr>
<th>Flavour</th>
<th>CPUs</th>
<th>RAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.tiny</td>
<td>1</td>
<td>512 MB</td>
<td>3 GB</td>
</tr>
<tr>
<td>m1.small</td>
<td>1</td>
<td>2.6 GB</td>
<td>20 GB</td>
</tr>
<tr>
<td>m1.medium</td>
<td>3</td>
<td>7.7 GB</td>
<td>60 GB</td>
</tr>
<tr>
<td>m1.large</td>
<td>6</td>
<td>15.4 GB</td>
<td>120 GB</td>
</tr>
</tbody>
</table>

Table 3.3: Flavours currently available on the INFN Torino private cloud, corresponding to different virtual hardware configurations. Disk refers to the ephemeral storage.

3.3 M5L-CAD: an elastic farm for medical imaging

One of the first test use cases of the private cloud in Torino, which was directly supported as part of this thesis work, is the use of a virtual and scalable farm for unattended processing of sensitive medical images\cite{16}.

Input data is constituted by lung CTs of several patients: automatic analysis, via appropriate combined algorithms, should determine whether there is evidence of lung cancer, and inform the medical doctor treating the patients accordingly.

The project is composed of three distinct parts:
• a front end web interface where medical doctors upload CTs from their patients, called Web-based Image and Diagnosis Exchange Network (WIDEN).

• three different processing Computer Assisted Detection (CAD) algorithms combined together, developed by the MAGIC-5 project of INFN;

• an elastic cloud computing backend running the algorithms over the CTs.

3.3.1 Challenges

Lung cancer is one of the deadliest forms of cancer, yet it is considered to be one of the most curable cancers if detected in its earliest stage. Current problem with cancer detection is that it mostly works manually, i.e. a multi-layered CT must be examined by a medical doctor.

Examining CTs manually is a slow and error prone process: if we imagine a mass lungs screening to prevent lung cancer, we would not have enough time from the medical doctors to examine all the CTs; moreover, the accuracy of detection of some shapes in a CT greatly depends on the level of attention of the medical doctor, that can get easily confused with a great number of images per CT and an even greater number of CTs to process.

Solutions for automating CT analysis exist, and they go under the name of Computer Assisted Detection (CAD): such solutions target medical doctors with all-in-one computers already containing the required hardware and software.

What we propose is instead a cloud solution providing “CAD as a Service”, and has at least two advantages over a “boxed” product:

• all-in-one solutions are difficult to upgrade, as they are usually sold as computers without an appropriate form of Internet connectivity, usually for patient confidentiality reasons: any upgrade in the algorithms and software has to be pushed manually to the machines, and the kind of users we are dealing with are usually not inclined to allow any upgrade;

• any kind of hardware rupture requires maintenance and effectively stops the analysis workflow;

• a single computer, even when well equipped, cannot perform as good as a cluster of computers.

A company called diXit solutions\textsuperscript{[35]} has been founded as a spin off of Università di Torino and INFN Torino, with which they maintain a close collaboration: the company is providing access to the Web-based Image and Diagnosis Exchange Network (WIDEN)\textsuperscript{[138]} web interface. The challenges faced by the collaboration are multifold:
• the “CAD as a Service” must be immediate to use for a doctor, both for uploading patient data and for retrieving at a glance the list of suspect cases;

• such service should provide an easy way to define a workflow when a suspect CT is examined by a panel of doctors working for distant institutes;

• computing resources must be taken partly from the cloud, and the underlying resources (amount and nature) should be completely concealed to the eyes of the medical doctors;

• patient privacy and confidentiality must be preserved: this is especially important when going to heterogeneous environments like clouds.

In the following chapters, the aspects representing a challenge are briefly described.

3.3.2 The WIDEN web frontend

The WIDEN web frontend is the entry point of the whole project. A web interface, accessible from every browser of desktop computers and mobile devices, provides an easy way to upload patient images in the standard DICOM format. The medical doctor first enters her credentials Figure 3.7(a), then selects an archive containing all the DICOM images concerning the patient case Figure 3.7(b), that are subsequently uploaded to the remote server Figure 3.7(c).

After uploading, images are stripped from their personal data for patient confidentiality, as we will see in detail in § 3.3.5, and they are processed using appropriate algorithms § 3.3.3. This thesis work helped in setting up a cloud backend for processing, as we will see in § 3.3.4. Once processed, results are collected back by the web interface, and proper notifications occur via email and short text messages; notifications are handled by the WIDEN interface as well.

For every “patient case”, the uploader has the possibility to define a series of reviewers which will be notified in case a suspect case has been found. The presence of a web interface and an efficient notification system defines a common platform for distant medical doctors, providing them tools to work efficiently together despite their distance.

3.3.3 Processing algorithms

Lung nodules detection is usually performed by doing low dose CTs at a high resolution: this can produce something like 500 2D images per patient, all of them affected by a high noise ratio due to the low dose.
3.3 – MSL-CAD: AN ELASTIC FARM FOR MEDICAL IMAGING

(a) Login to the WIDEN web frontend.

(b) Archive containing the patient case is selected.

(c) Images are uploaded to the server.

Figure 3.7: Images showing the workflow of uploading patient DICOM images to the WIDEN server for processing. The medical doctor will be eventually notified when processing has been completed.
On top of that, nodules can be of a very small size, or attached to the pleura surface, making the identification even harder. It has been shown that combining results from different algorithms yields a greater detection rate and lowers the number of false positives.

There are three different algorithms in use: all three of them are used together on each patient case.

- **The channeler ant model**[^26], using colonies of virtual ants moving in the CT 3D space where each voxel (a volume element in a three-dimensional space) has a density value representing the amount of available food. Ants leave virtual pheromone trails while they move: when all virtual ants have died, the shapes of such trails will be classified by a feed forward neural network based on their features: a certain combination of features is considered a signal of a nodule.

- **The voxel-based neural analysis** is a special approach used to identify nodules attached to the pleura by attempting to map groups of voxels to spheres with a gaussian profile, while groups of voxels in the paraenchyma are mapped to cylinders. Both raw methods yield a large number of false positives, smoothed out by using a neural network classifier.

- **The region growing volume plateau**[^33] segments the paraenchymal volume and applies a region growing algorithm iteratively to detect nodules. False positives are reduced by eventually applying the same neural network classifier used in the channeler ant model.

The M5L-CAD, standing for MAGIC-5 Lung CAD, combines the results of the three algorithms and assigns a weight to them: this has been shown to increase the detection accuracy and reduce false positives[^25].

### 3.3.4 Elastic cloud backend

The WIDEN interface feeds the computing backend with a list of patient cases to process: the way of communicating to the computing backend is constituted by a simple shell script. In our case, the backend is a simple batch system, and the shell script invoked by WIDEN performs the following operations:

- sends data to the backend’s head node via SSH;

- launches a script to enqueue the processing on a batch system;

- periodically pings the results directory on the backend’s head node to verify if processing has finished;
• collects data and publishes it on the web interface.

The first prototype presented\cite{16} used SLURM\cite{104} as computing backend. In the initial situation the computing facility as no cloud nodes: as the number of requests increases, a Ruby script triggers the execution of new computing nodes and turns them off when unused.

This Ruby script worked only with OpenNebula and its OCA\cite{83} API, but the prototype showed, after the first trial with PROOF on Demand (PoD)\cite{17}, that managing the elasticity of virtual farms from inside the virtual farm itself is feasible and greatly simplifies the setup and maintenance cost.

The idea of providing resources in a cloud form is that, being the company developing the product a commercial company\cite{35,138}, it has direct costs to sustain: it has been noted how renting cloud computing resources on commercial clouds (such as Amazon EC2) might be a sustainable possibility for emerging startups.

The service sold by the company must be fast enough in providing the results, and affordable enough to keep the cost low.

An elastic mechanism absorbing peak loads and turning virtual machines off when unused greatly addresses both issues.

3.3.4.1 Evolution towards a generic Virtual Analysis Facility

The ideas behind the virtual M5L farm eventually converged into the Virtual Analysis Facility, extensively covered in § 4.4. The Virtual Analysis Facility, unlike the M5L cloud, communicates to the cloud controller using the EC2 interface, and it is not tied to OpenNebula. Moreover, being based on \(\mu\)CernVM (§ 4.4.3.1), its entire configuration is performed during contextualization, making the update of the operating system easier. The batch system chosen for the Virtual Analysis Facility is HTCondor (§ 4.4.4.1), which deals very well with the dynamic addition and removal of worker nodes.

Even if the Virtual Analysis Facility was essentially born to find a sustainable way of running PROOF using PoD, they only constitute what the client decides to launch on top of it. Under the hood, the Virtual Analysis Facility as we will see is nothing but an elastic HTCondor-based virtual cluster, which is capable of running as-is on every cloud supporting the EC2 API.

The applications processing patient cases using the algorithms described in § 3.3.3 is already configured to run on batch systems: it requires changing a single line of code for allowing the submission on HTCondor instead of SLURM to adapt it to the Virtual Analysis Facility.
3.3.5 Privacy and patient confidentiality

Unlike HEP use cases, computing for medical applications has stricter constraints for data security, primarily due to higher concerns in privacy justified by the protection of the doctor-patient confidentiality, and in possible data tampering that, if occurs, might have consequences on the patient’s health.

Since we are using a computing backend in an opportunistic manner, security concerns are more important than ever: we should enact multiple levels of security in order to limit as much as possible data tampering and leaks in case one of the layers fails its purpose.

There are in particular three aspects that must be guaranteed by appropriate security policies.

1. **Data integrity.** We must make sure that data is not altered by any means during transfers from the WIDEN interface to the cloud backend, and vice-versa.

2. **Confidentiality and privacy.** Algorithms should process images without using any data which might relate them to a specific patient. The association between input images, the automated diagnosis and the patient should be only made by selected human reviewers, i.e. authorized physicians.

3. **Protection from data retention.** Patient data must not be retained, either intentionally or accidentally, on the cloud backend. If this happens, there must be a set of policies allowing the patient to know where her data is located, and giving her the possibility to request the removal.

The security policies used by the M5L virtual farm and WIDEN cover all these aspects very carefully. First of all, data integrity and tamper protection is guaranteed by security tunnels: all the communication layers involved use SSL, from the WIDEN web frontend to the communication between the interface and the cloud backend, which uses SSH and data transfer tools related to it. Secure connections not only prevent eavesdropping on data, but also check for and correct accidental damages that might have occurred due to a bad network link.

From the very source of data (the WIDEN interface), sensitive patient information is stripped off from all input DICOM files: such information include name, gender and age. Instead, a unique identifier is inserted in place: the only component knowing the association between the identifier and the original data is the WIDEN interface, and appropriate policies allow only the submitter and selected reviewers to retrieve it.

This essentially means that:

- data is sent to the remote cloud infrastructure already deprived of sensitive information: even if someone will be able to retrieve it, in spite of the
security layers, data will just be meaningless;

- since an association is maintained in a single secure place, in case of requests it is possible to identify with precision the exact virtual machines that have processed it and either remove traces of the data, or remove the entire virtual machines for increased security.

Such security measures are enough to guarantee all the aforementioned policies on every cloud, even a commercial one not directly under our control. However, it is worth reminding the feature of the private cloud in Torino that helped in providing an additional layer of security during the testing phase: network isolation of virtual farms by means of ebtables and virtual routers (§ 3.2.2).
Chapter 4

PROOF as a Service: deploying PROOF on clouds and the Grid

4.1 Overview of the PROOF computing model

PROOF: The Parallel ROOT Facility\textsuperscript{[13]} is a parallel framework facility distributed with ROOT\textsuperscript{[22]}. Being ROOT the framework for scientific analysis which constitutes the base for the software of all LHC experiments, all LHC physicists use it on a daily basis. PROOF can therefore count on a vast installation base.

PROOF is capable of performing analyses in a parallel and interactive fashion: a digression on the meaning of the terms is presented in the next sections.

4.1.1 Event based parallelism

PROOF works in parallel because it has several working units, called workers, executing calculations at the same time.

The type of parallelism exploited is called event based parallelism. PROOF and ROOT are primarily oriented for physics, and in particular for HEP analysis, where input data is logically divided into several independent physics events, each one representing particles collision.

Since input data is constituted of independent units, the most trivial way of parallelising it is by analysing independent events on many independent workers: this type of parallelism is also called an embarrassingly parallel workload, as it requires nearly no effort to be parallelised. Since it is extremely cost effective and almost not error prone, it is always the first “dimension” of parallelism being explored\textsuperscript{[59]} when dealing with a demanding computational problem.

It must be noted that event-based parallelism does not increase the execution speed of the analysis process: instead, the throughput is increased. This means
that, assuming that each event takes the same computation time, and assuming that we have $n$ identical computing workers, after the time required to process a single event we will have $n$ events processed.

This kind of parallelism is very effective with large input or output datasets: we often refer to it as High Throughput Computing (HTC).

PROOF is not the only computing model leveraging event based parallelism: in HEP, the largest deployment of such computing model is the Grid. What distinguishes PROOF from the Grid is interactivity.

### 4.1.2 Interactivity

PROOF is defined as an *interactive* computing model, although the definitions of “interactivity” are diverse.

PROOF was originally created to be interactive from the user’s point of view. The idea was that the user sitting in front of her computer screen launches a PROOF task and sees either the progress of the analysis, or the results, or both, in real time. As a matter of fact, PROOF features a graphical control interface showing such progress (see Figure 4.1), and a function continuously sending partial output data to the client.

**Figure 4.1**: PROOF progress status, showing real time information on the current data rate. The interface also controls PROOF execution, and gives access to performance measurement plots and logs.

Another aspect of interactivity is the ability to send commands and execute them in parallel. ROOT has a command line interface where the user can type her commands: while a PROOF session is opened in the background, the command used to run her analysis locally can be used with some degree of transparency to run the same analysis in parallel on PROOF.

Current real use cases of PROOF in LHC experiments often regard an inter-
active model as the opposite of a “batch model”. The latter is characterized by a submission queue where a large analysis is pre-divided into many independent “jobs”: each job is executed when a scheduler starts it (roughly speaking: when there are enough free resources and by respecting certain user policies and quotas).

In respect of this, PROOF’s advantage is that you do not need to submit jobs to exploit computing resources: opening a PROOF session means acquiring the control over a distributed set of resources for some time, hence the “user interactivity”. Once a PROOF session is opened, many tasks can be executed without losing the hold on the obtained resources.

All of the various meanings of interactivity given so far concern the interaction between users with the parallel facility. We however prefer to consider the interactivity between the PROOF workers: while batch jobs are being executed independently from each other, PROOF workers feature a constant communication with an head node, called the PROOF master.

This two tier architecture along with continuous network communication allows for interaction between the master and the workers. The first visible advantage of such interaction is that different pieces of the output data produced by several workers are collected automatically and presented to the user as once, directly on her client. On the contrary, batch models produce independent output data which must be both collected and merged manually: a PROOF set of workers really gives the impression of working together as once, and automatic data collection and merging is one of its most appreciated features.

We will see later on how interactive communication is used to achieve a complete independence between data and resources leading to a more optimal exploitation of the latter: in fact, PROOF features a dynamic scheduler assigning input data to workers in real time, as well as the new ability to add new workers dynamically to offload a currently running task.

4.1.3 PROOF and ROOT

PROOF is distributed as a part of the ROOT framework, and it is built on top of many ROOT features.

PROOF workers are, under the hood, independent and distributed ROOT sessions, each one of them having its own command line interpreter, where input and output streams are connected to network sockets. From this perspective, PROOF is a way of controlling many parallel ROOT sessions, therefore it is possible to execute interpreted snippets of C++ code by means of the CINT\textsuperscript{[22,109]} or the Cling\textsuperscript{[135]} C++ interpreters, the latter being more modern and robust, and available from the forthcoming version 6 of ROOT.

ROOT’s object model allows for platform independent serialization of ROOT
C++ objects: that is, the instance of a given class can be converted to a stream of bytes, saved or sent somewhere, and restored afterwards. ROOT marshalling is the technique used to save objects to files, and it can be very powerful for end users, as they don’t have to bother too much writing special methods to save their custom classes to disk.

PROOF uses ROOT serialization for socket communications between the master and the workers: for instance, output data is streamed back from the workers to the master which restores it in memory and eventually merges it.

ROOT has a convenient structure for storing data: the tree, a database table containing many entries having the same structure, where each one of them represents a physics collision event. Data in trees is organized to optimize the readout. Since it is not possible to define relations between trees (in the database terminology), they can be considered object databases optimized for fast and repeated readings.

PROOF has been essentially designed to process input data in the form of ROOT trees: by writing analysis code as a subclass of a ROOT selector (represented by the TSelector class), it is possible to run exactly the same analysis code either sequentially or in parallel with PROOF. A selector is simply a class which logically divides the code into an initialization function, an event loop (i.e., a function invoked for each event) and a finalization function.

Finally, PROOF uses ROOT as the connection client: any user having ROOT installed can connect to a PROOF cluster without any additional dependencies.

4.1.4 Dynamic scheduling

PROOF features a scheduler that dynamically assigns data to workers, while the analysis is progressing. PROOF’s scheduler is called packetizer, as input data is divided into work units called packets.

A packet is a set of physics events, or, by referring to a more general data structure, a set of ROOT tree entries. A single packet has a variable size, but it is always smaller than the number of events contained in a file.

The PROOF master keeps track of the data events to analyze: workers process packets, then, when idle, ask the master for the next packet. The master never assigns the same packet twice: the process is represented in Figure 4.2. Since the workers are the ones requesting data to process, the packetizer is a pull scheduler: the fact that packets are in general small allows for a more fine-grained workload distribution.

Even if there are several packetizers available for PROOF, the one we are describing here is the adaptive packetizer: such packetizer was written\textsuperscript{[45]} with the observation that:

- not all events take the same time to process: for instance, in some analy-
4.1 – OVERVIEW OF THE PROOF COMPUTING MODEL

Figure 4.2: The PROOF packetizer assigns data to workers using a pull model: pointers to data to process are sent whenever workers are finished processing old data.

ses of rare events, some events are simply skipped taking almost no time to complete, while others undergo some deep calculation lasting several seconds;

• computing resources are not uniform, leading to some PROOF workers being faster than others.

When performing analysis interactively, it is a clear waste of resources having to wait for the slowest, or most overloaded worker to finish before collecting the results. The adaptive packetizer purposes the uniform completion time of workers by assigning the workload non-uniformly.

Plots in Figure 4.3 show how the adaptive packetizer accomplishes its task egregiously.

4.1.5 Data locality and XRootD

XRootD\textsuperscript{[38]} is a storage solution capable of exporting and aggregating filesystems from a distributed set of hosts in an efficient way. XRootD is meant to be a scalable and safe storage solution.

Scalability is achieved via a redirector architecture: a typical XRootD setup is constituted by a head node, called the redirector, and many XRootD servers exporting their filesystems. The set of filesystems altogether forms a disk pool, and both the redirector and the disk servers need to be network accessible from the client.
Figure 4.3: Measurements on the PROOF’s adaptive packetizer, showing that workers complete their tasks almost together (in a time window of around 20 s), while the number of packets processed is not assigned uniformly.
When the client connects to the redirector to request a file, the redirector will not send back the file: instead, it will forward the file query to the servers, and it will tell the client to contact directly another file URL, the *endpoint URL*. Subsequent requests of the same files will be faster as the redirector also caches the queries.

The redirector model is exploited in some dedicated PROOF farms by configuring the PROOF master as a XRootD redirector, and PROOF workers as XRootD file servers. If files to process are on the XRootD pool, the PROOF scheduler performs a preliminary step where it queries each URL to find its endpoint: when the analysis starts, PROOF will try to assign as much as possible data packets to the nodes hosting them.

The final result will be a considerable reduction of the local network traffic, as every file transfer will benefit from the speed allowed by its locally mounted storage.

This setup is in particular in use by the ALICE experiment in the ALICE Analysis Facility (AAF) model (§ 4.3.3). We have written a daemon to resiliently transfer data to PROOF local storages, as we will see in § 4.3.5.

### 4.1.6 PROOF-Lite

It is worth mentioning the possibility to fully exploit the capabilities of PROOF on a local multicore machine: the technology used is a subset of PROOF called PROOF-Lite\[^{92}\], also distributed as part of ROOT.

PROOF-Lite can be used and controlled just as PROOF from a local ROOT session, but it does not need any network connectivity. A PROOF-Lite session has no dedicated master, as the client session performs master’s functions: separated ROOT sessions are launched in the background (by default, one per core) and they are connected to the client-master by local Unix sockets.

As the name suggests, PROOF-Lite is a lightweight version of PROOF, as it does not need many parts of PROOF to work: for instance, a large part of the code is dedicated for the networking and multiple users management, which are not needed for a local multicore session.

### 4.2 Static and dynamic PROOF deployments

PROOF has originally been intended for the deployment on a local dedicated computing farm: the head node is configured as PROOF master, while other nodes are workers.

It has been observed that configuring a dedicated PROOF farm may present administrative difficulties: to improve the user’s experience, PROOF evolved to a
zero-configuration model which allows users to run it on non-dedicated, or even opportunistic, computing resources.

In the following sections we will describe how static PROOF farms work (§ 4.2.1) and what kind of issues they present (§ 4.2.2). A digression on the methodology adopted to overcome such problems follows (§ 4.2.3). Finally, PROOF on Demand is introduced as the first technology enabling dynamic PROOF deployments (§ 4.2.4).

### 4.2.1 Dedicated PROOF clusters

In a static PROOF deployment, the xproofd daemon runs in background on every node, as a privileged Unix user. xproofd is based on the XRootD communication layer, which allows for very resilient network connections (both client to master and master to workers): it has in fact the capability of reestablishing a lost TCP connection transparently to the application layer.

Whenever a user connects, the xproofd daemon on the master performs authentication, then it forks to a ROOT interactive and remote session. All the worker xproofd daemons fork to ROOT sessions as well: a single xproofd worker spawns as many ROOT sessions as configured, commonly corresponding to the number of available cores on that node. ROOT PROOF sessions communicate via XRootD sockets, and xproofd daemons multiplex and deal the effective TCP connection.

In practice, users need different ROOT versions when performing analysis. Even if a xproofd daemon comes with a certain ROOT version, it has the ability to fork to a ROOT session coming from a different version if requested.

Even if this approach is still largely adopted by some experiments (especially the ALICE experiment, § 4.3), it poses very practical and critical issues of scheduling, stability and maintenance.

### 4.2.2 Issues of static deployments

PROOF is essentially a tool to reserve computing resources and use them many times: in a multiple users environment it always assigns the same number of workers per connected user, meaning that the number of total running workers might be greater than the number of total cores. The problem is mitigated by the fact that in real deployments PROOF is configured to assign less workers than the available cores, and many assigned PROOF sessions are opened but idle.

We do not consider this workaround satisfactory, as a more appropriate way of solving the problem would be having some sort of user scheduling mechanism.

Having too many users connected may pose problems of stability: PROOF sessions run custom user’s compiled code, which is not always perfect. Crashes
and excessive memory usages are frequent in user’s code, and a crash in a user’s PROOF session might propagate to the controlling xproofd daemon, leading to an abrupt termination of all users sessions.

A mechanism providing an isolated environment for each user would be desirable: in addition to improving the stability, it would also mitigate code security concerns.

In spite of the ability of xproofd daemons to fork different ROOT versions, in practice there might be some incompatibility in the communication protocol used by mismatching xproofd and ROOT versions. Unfortunately, in a static deployment there always is one main ROOT version which is assured to be compatible: the one running the xproofd daemon.

A static PROOF deployment also poses administration and support problems: when a PROOF crash occurs, services must be restarted manually, unless an appropriate monitoring and restart method (such as Monit\(^{[79]}\)) is in place.

On top of that, PROOF is, as we have seen, an interactive computing facility, primarily meant for running relatively short tasks. This roughly means that a PROOF static deployment would be mostly unused overnight, leading to the need for an easy way to turn PROOF resources into something else when possible.

### 4.2.3 Overcoming deployment issues

We believe that all the issues encountered in static PROOF deployments were due to the diverse and vast nature of tasks PROOF is expected to accomplish. We therefore reviewed PROOF features and weak points to understand what has to be retained, what needed to be improved, and what could be simply delegated to external components.

The main strength of PROOF is the computing model it implements: similarly to other non-HEP specific solutions like Hadoop, PROOF is essentially a pull scheduler (§ 4.1.4) which efficiently deals with data locality (§ 4.1.5). To this respect, a feature that was missing prior to this work was the ability of adding new workers to a running analysis: the implementation of such feature (§ 4.4.6) has made PROOF a computing model for HEP which outperforms the Grid (§ 4.4.9.1) by simply scheduling resources in a more efficient way.

PROOF has also been used by the ALICE experiment for performing data management, despite it is essentially a HPC tool: a solution which delegates data management by retaining an improved interface for accessing ALICE files has been implemented, as we will see in § 4.3.6).

To benefit as much as possible from such an advantageous computing model, we addressed the deployment issues, purposing the idea that PROOF should be available with no configuration at all: we will see in § 4.4 how combining existing tools with the emerging cloud computing technologies makes possible, even
for the most inexperienced user, to run her scalable PROOF cluster with no administrative efforts—ideally bringing to the cloud the excellent work done with PROOF on Demand (§ 4.2.4).

4.2.4 PROOF on Demand

PROOF on Demand (PoD)\textsuperscript{[71,88]} is a program which makes PROOF run without any static deployment or administrative privileges. PoD is capable of doing so by wrapping xproofd daemons into “agents”, and by submitting them either on a batch farm or via SSH. A plugin to use it on the Grid also exists for the ATLAS experiment\textsuperscript{[136]}.

The advantage of PoD is that any user can use PROOF provided that she has some resources at her disposal: no intervention from a system administrator is needed. PoD effectively takes care of the PROOF deployment at user level without the complications of a dedicated PROOF farm.

Concerning the issues raised with static PROOF deployments (§ 4.2.2), PoD provides a solution for most of them. Since each user has her own PROOF set of daemons, failures and crashes do not propagate to other users. With a batch farm at our disposal, PoD agents are submitted as jobs, one per worker: this solves the problem of scheduling, as the workers are started only when there are enough resources to do so; moreover, a batch farm can orchestrate resource sharing between PROOF and other non-interactive use cases, eliminating the problem of PROOF dedicated facilities which stay unused overnight.

Finally, since no administrator is needed to configure PROOF, in case of problems users can restart their own PROOF resources without any external intervention.

Given all of that, PoD can be seen as a way of reserving interactive resources on a non-interactive batch farm by means of PROOF. We will see how the Virtual Analysis Facility (§ 4.4) extends this concept to clouds.

4.3 ALICE Analysis Facilities and their evolution

Among all the major LHC experiments, ALICE explicitly supports PROOF in its computing model\textsuperscript{[107]} and has production deployments of PROOF since 2010. ALICE provides both specifications for PROOF deployments and a framework which makes the development of analysis code independent from the platform it will run onto—namely on the Grid, locally or on PROOF.

The set of specifications a PROOF deployment must follow is described in § 4.3.1, and goes under the name of ALICE Analysis Facility (AAF). The technical aspects of authenticating users from the ALICE collaboration are covered in
§ 4.3.2. A discussion on the AAF data model and its limitations are presented respectively in § 4.3.3 and § 4.3.4.

The work described in this thesis also involved contributions to the AAF model, namely: the dataset stager, used to resiliently orchestrate data transfers towards the analysis facility storage pool and presented in § 4.3.5, and an interface to make the ALICE File Catalog accessible from PROOF, covered in § 4.3.6. The experience acquired in setting up and using PROOF facilities for ALICE is what eventually led to the Virtual Analysis Facility (VAF) model (§ 4.4).

4.3.1 Specifications

All ALICE Analysis Facilities are defined by a set of specifications to be followed by a PROOF deployment in order to be compatible with the ALICE analysis framework.

4.3.1.1 The analysis framework

Within the ALICE analysis framework, a “user analysis” is referred to as an “analysis task”. ALICE’s C++ framework (AliRoot) provides a base class, whose name is AliAnalysisTask: user analysis is written in the form of a C++ class inheriting from such parent class. The analysis task structure allows to divide user’s code into three main member functions: an initialization run at the beginning of the analysis, an event loop called for each event, and a termination called at the end of the analysis. The analysis task per se is independent from the way it is executed, and the presence of an event loop naturally allows for the code to be run on PROOF as on any other infrastructure without any modification.

ALICE has its own way of calling the event loop, namely through the analysis manager, represented by the class AliAnalysisManager. The analysis manager makes the analysis run on Grid, locally or on PROOF: concerning the latter, it practically provides a layer of compatibility between the analysis task and PROOF, which does not directly support ALICE-specific classes.

ALICE uses a Grid middleware called AliEn for user authentication, to orchestrate Grid analysis and to access the file catalog and the files themselves. AliRoot provides a convenient interface for running analyses called the AliEn Analysis Plugin: with a single steering macro it is possible to define all the analysis details and data sources. By means of the AliEn Analysis Plugin, running on multiple infrastructures is a matter of changing a single parameter. The AliEn Analysis Plugin, originally conceived for simplifying Grid job submission, supports PROOF analysis as well.
4.3.1.2 Deployment

AAF deployments have a series of commonalities which altogether concur to allow any ALICE member to use any AAF in the world using the same set of credentials; moreover, it should be possible to run the same analysis on different AAFs by simply changing the AAF name in the PROOF connection string:

\begin{verbatim}
TProof::Open("user@aaf.fully.qualified.domain.name");
\end{verbatim}

The first thing to be noted is that ALICE PROOF deployments are static deployments as already discussed in § 4.2.1: this means that there is a single head node per analysis facility having the xproofd daemon listening for connections and forking PROOF masters.

There are several items covered by the specifications: in all cases their specific implementation may vary, as long as the interface for accessing the features remains consistent throughout all AAFs.

- **User authentication and authorization.** All AAFs are accessible by any ALICE member from anywhere in the world by using the same set of credentials. More details on the authentication procedure are available in § 4.3.2.

- **Software packages distribution.** All AAFs must have the same analysis software framework available. AliRoot comes in different versions, with several releases per month, each one of them depending on different versions of ROOT and Geant3.

AAF deployments from 2010 to early 2013 used only Packman (part of AliEn) as software distribution method: the same used on the Grid. Packman downloads and installs full AliRoot packages on each PROOF node: AAF procedures require the software update procedure to be triggered manually.

Due to constraints in disk capacities, the CernVM-FS4.4.3.2 remote filesystem is also supported: on CernVM-FS new versions are immediately available, and only the files effectively accessed are downloaded and cached locally, greatly reducing the amount of disk space used and network usage as well.

- **Data model.** Effective URLs pointing to data are not specified manually, as they vary per analysis facility. Instead, users specify input data by using PROOF datasets § 4.3.3.1. Data is served via XRootD, where every PROOF node is usually used to store data: this is to take advantage of local file access when possible. Other analysis facilities prefer using an external XRootD storage, or other data access methods different than XRootD. AAF
data model and datasets handling, along with some considerations on its usability, is presented in § 4.3.3.

- Monitoring. AAFs are monitored by MonALISA\textsuperscript{[28]} in real time: in particular, the number of connected users, total number of available PROOF workers and storage usage and availability is reported for each facility. All this information, along with the service status, is reported on a dedicated web page\textsuperscript{[78]}.

A guide for system administrators willing to set up an ALICE Analysis Facility (AAF) is provided\textsuperscript{[4]}, along with an installation script automatizing most of the installation procedure.

### 4.3.2 Users authentication and authorization

As we have already noted, AAFs are static PROOF deployments with a single head node. Such node runs the xproofd daemon, listening on port 1093/TCP for incoming connections from the PROOF clients. Since every AAF has to be accessible from every ALICE user, there is no firewall limiting connections based on their origin.

Since PROOF’s communication layer is based on XRootD, PROOF also relies on the authentication mechanisms that it offers. In the AAF case, the xproofd daemon performs authentication and authorization using the Grid Security Infrastructure (GSI) plugin: the client (i.e., ROOT) supplies a proxy certificate issued by means of the user’s certificate, which must be in turn issued by a Certification Authority (CA) authorized by the server. A digitally signed message whose signature is verified is supplied as well\textsuperscript{[137]}. It must be noted that in PROOF, like in XRootD, the initial handshake happens securely, while the rest of the communication is unencrypted.

Once the user is authenticated, authorization is performed by checking the certificate’s subject against all the certificates registered to the ALICE VO: this is done by querying an appropriate ALICE LDAP service.

PROOF also uses the LDAP to perform mapping between the user’s certificate subject and the AliEn user name: user’s PROOF master and workers will be run as a Unix user corresponding to the AliEn user name.

In order to make possible the use of AliEn user names as Unix user names, PROOF hosts are configured to use the AliEn LDAP as users database.

One of the advantages of performing authentication using GSI is that we can use the delegation mechanism to create additional proxies, one for each PROOF host. These proxies will be used to create AliEn authentication tokens, used in turn to access files from the Grid.

More in-depth configuration details are covered in § C.1.
4.3.3 Data storage and access model

The PROOF AAF data model requires that input data is located as close as possible to the computing nodes. This means one of the following situations.

- Input data is evenly distributed on the nodes performing the actual computation: using XRootD[^38,140] to manage the storage, this is usually achieved by configuring the PROOF master as XRootD redirector and making the nodes part of the XRootD pool. This is the case for the majority of the ALICE Analysis Facilities, namely CAF, SKAF, KIAF and JRAF.

- Data is located on a separated local storage. Such storage, independently from the actual underlying storage technology, always exposes an XRootD interface for compatibility and accessibility—therefore using only the file serving part of XRootD and not its storage management and aggregation features, managed by the cmsd daemon. The separate storage model has been adopted by the LAF (now dismissed) and both TAFs (the original “AAF version”, described in § 4.3.7 and the “elastic version”, covered in § 4.4).

Storage in the AAF data model is an unmanaged buffer between PROOF and the AliEn File Catalog: remote AliEn files are accessed using the local storage’s URL instead of the alien://, in a way that they are downloaded and cached on the local storage; subsequent file accesses are expected to be substantially faster. On the other hand, the cache is unmanaged in a way that when the storage is running out of space it deletes “old” files automatically. Both features are attained by using XRootD and configuring it appropriately.

XRootD’s ability to fetch data from third party storages is called Virtual Mass Storage System (vMSS)[^43], and works as follows. When a client tries to access a XRootD path, XRootD checks if the file is available on the exported partition. In case it’s there, the file is immediately served to the client; in case it’s not, an external command (usually a shell script) is called to download the file or return a failure code in case the file cannot be retrieved.

In the ALICE model with one redirector and many servers (each of them being a computing node as well), the redirector receives the request and asks which servers owns the file. If none of the servers has the file, one server is selected in a round-robin fashion to manage the file retrieval. In practice, file retrieval is managed by a single given data server and not by the XRootD redirector.

It is also possible that many servers have the same file cached: in such a case, the redirector returns the first server that replied.

The download script effectively provides a mapping between the XRootD exported storage and one or many third party storages. For instance, in the ALICE use case, a XRootD input URL like the following:
will be mapped to the corresponding AliEn file catalog URL:

```
root://alice-srv-11.to.infn.it//alice/data/2011/LHC11h/000170315/ESDs/pass2_muon/11000170315082.45/AliESDs.root
```

The corresponding file will be retrieved by means of the `alien_cp` command.

XRootD’s disk quota management is complementary to the vMSS and prevents disks from getting full. It is possible to configure a percentage threshold of disk usage on each server: when the threshold is reached, XRootD picks those files whose access timestamp is the oldest and erases them, until the disk usage is once again below threshold.

### 4.3.3.1 PROOF datasets

PROOF datasets are lists of files with a name and metainformation: each dataset points to all the files which are usually processed together, e.g. all the ESD files from a single LHC run. Per-file metainformation includes:

- default tree name
- number of events in the default tree
- file size
- integrity information: “is my file corrupted?”
- locality information: “is my remote file available on a local storage?”
- “originating” URL (where does the file come from)
- “local” URL (pointer to the closest storage)

As we have seen in § 4.3.1, AAF specifications are intended to provide uniform working environments where the user can switch from one facility to another by simply changing the Analysis Facility name in its configuration file.

PROOF datasets are part of this specification: instead of directly specifying the local URLs of the files to analyze, one specifies only the dataset name. The same dataset in different analysis facilities will point exactly to the same file, but in each case the closest copy is referenced.

While describing the dataset stager (§ 4.3.5) we will see how datasets are also used as a communication layer between PROOF and the stager itself: staging
requests can be issued by saving specific PROOF datasets which are monitored by the stager. The stager will save the download progress back on such datasets, which can then be read from PROOF.

PROOF datasets are handled by the TDataSetManager ROOT class, which acts as a collection of datasets. Such abstract class used to have a single implementation, which stores each dataset as a marshallled TFileCollection instance inside a ROOT file: the TDataSetManagerFile class.

A more modern datasets implementation is available for interfacing PROOF with the AliEn File Catalog: the interface, which is compatible with the former dataset model, is presented in § 4.3.6.

4.3.3.2 The AliEn File Catalog

The ALICE experiment stores its data in a storage distributed on many computing centres, each one of them exposing a XRootD interface.

All data is stored under a single filesystem-like namespace called the AliEn File Catalog: without knowing the exact location and URL of the desired file, called the Physical File Name (PFN), files are always accessed via a meta-URL, or the Logical File Name (LFN). LFNs start with alien://, while all PFNs start with root://, and there might be several PFNs for each file, one for each mirror copy.

Whenever we access a remote file pointed by an alien:// URL, the following sequence happens behind the scenes.

1. A query to the AliEn File Catalog is issued to retrieve the list of PFNs.
2. Which PFN to use is left to the client: usually the “closest storage element” policy is adopted.
3. The corresponding XRootD URL is contacted. If the URL corresponds to a XRootD redirector, then another URL (the endpoint URL) is returned, and the file is finally accessed from there.

It is important to point out how it is up to the client to decide which URL to use. When performing analysis on the Grid, the destination site is chosen among the sites having input data: when those Grid jobs attempt to access files, the access will always be local.

Normally, in AAFs analyses, alien:// URLs are never accessed directly, even if it is technically possible to do so: the XRootD redirector is accessed directly instead. It is important however to understand how the File Catalog works, because AAF XRootD storages configured in cache mode with vMSS perform queries on behalf of the user to fill the cache with ALICE data.
Moreover, as we will see in § 4.3.4 catalog queries might have a profound impact on each file’s access time, and this is especially noticeable with small files.

4.3.4 Issues of the AAF data model

As we have seen, the original AAF data model implies the presence of a AAF-dedicated XRootD caching storage. Working as a cache, the storage is filled while accessing files for the first time, while files which have not been accessed for a while are periodically removed to reclaim the space when the storage is about to get full.

Several practical issues were encountered while putting this model in production.

- **Much slower than expected.** It is expected that a certain analysis running on an empty cache would take more time than the same analysis on cached data. Measurements and tests show how many factors contribute to make this time difference much greater than expected, as we are about to see.

- **No pre-staging mechanism.** The transparent cache as-is does not allow users to pre-stage data that they foresee to process: in practice it is not possible to request today some data that you would like to analyze tomorrow.

- **Data integrity concerns.** No mechanism to ensure the integrity of the downloaded data was present in the original AAF data model. Methods to retry downloads upon temporary failures were not present as well.

In the following paragraphs we will see the issues in greater detail.

4.3.4.1 Very slow first time data access

The biggest practical issue encountered was the time taken for an AAF analysis when no data is cached. We attempted to give a rough estimate of the time differences by running a specifically chosen sample analysis on a “critical” yet very common dataset.

The sample analysis is a $J/\Psi$ analysis filling invariant mass histograms from dimuon events: the analysis is I/O bound as almost no computation is performed apart from filling histograms, and it has been chosen to underline bottlenecks in data access.

Input data are filtered muon AODs (*from the AOD set num. 134, special muon pass 2 reconstruction*) from several runs of the LHC13d data taking period: such AODs are very small, weighting less than 12 MiB each on average. Given their small size, these files are perfect to emphasize the effects of slow file catalog queries compared to the data transfer.
Our tests were specifically run on two runs of the said LHC period (namely, 195725 and 195682). We took note of performance data returned by PROOF: the total analysis time, and the average file transfer speed. The test has been run twice on the new PoD TAF in Torino (§ 4.4) on 24 PROOF workers accessing data from the local and dedicated XRootD storage in two different conditions:

- in one case, data to process was already available locally, making it not necessary to download data remotely and accessing the AliEn File catalog;
- in the other case, data was not available locally: this implies a query to the File Catalog for each file, plus the time taken to download the file from the remote location (namely, the ALICE EOS storage at CERN for our dataset).

Results of our test are shown in Table 4.1. The average download speed is around 4.5 times lower for remote data access, which is perfectly acceptable considering that we are comparing a remotely accessed, multipurpose and busy storage with a local and dedicated storage.

Total analysis time however is 12 times slower with an empty cache: this effect is due to latencies introduced when querying the AliEn File Catalog for each file to process.

<table>
<thead>
<tr>
<th>Storage</th>
<th>Total time [m:ss]</th>
<th>Data rate [MiB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cached data</td>
<td>1:06</td>
<td>56.6</td>
</tr>
<tr>
<td>Empty cache</td>
<td>12:43</td>
<td>12.5</td>
</tr>
<tr>
<td>Speedup</td>
<td>12x</td>
<td>4.5x</td>
</tr>
</tbody>
</table>

Table 4.1: Cached vs. uncached data access through the XRootD’s vMSS interface on the new PoD-based Torino Analysis Facility (TAF). Test was conducted by running a sample I/O bound analysis on 24 PROOF workers. Note that in the empty cache situation there is a slowdown effect due to the latency of AliEn File Catalog queries, which makes the analysis 12 times slower whereas data transfer is only 4.5 times slower.

4.3.4.2 Lack of a pre-staging mechanism

If a user or group of users are planning to perform repetitive analyses on certain datasets, it is a good idea to pre-stage the data on the local AAF storage in advance. In general, having a pre-staging mechanism is a good idea for two reasons.

- As we have measured in § 4.3.1.2, first-time data access is extremely slow. This is the main issue reported by users when the first AAF was put in production (SKAF in 2010).
A busy analysis facility might deal with several concurrent requests for data which is not there during working hours, whereas there are periods of the day when no data access is performed. Having a pre-staging mechanism working, for instance, during the night helps to better distribute and schedule the downloads.

A partial solution initially proposed was to run a “dummy” PROOF analysis merely accessing data to trigger the downloads. Unfortunately, this requires that the PROOF client stays connected during the whole pre-staging phase which might last for hours: if any problem occurs on the client (e.g. like a simple connection lost), then the staging will fail.

Moreover, due to the issues of static PROOF deployments already covered in § 4.2.2, any general problem to the xproofd daemons would affect the pre-staging.

The operation of pre-filling a storage with remote data should be performed by an asynchronous and separated service.

### 4.3.4.3 Data integrity concerns and resiliency

The staging scripts used by XRootD’s vMSS functionality for AAFs do not enact any particular mechanism to deeply check data for defects.

Unfortunately it has been observed in ALICE that some ROOT files contain corrupted data, which might negatively impact a PROOF analysis up to the point of producing crashes.

A partial solution to the problem would be to place into the vMSS script proper checks which communicate to XRootD the download failure in case of corruption, or even retry immediately in case it was due to a temporary failure. The vMSS feature however has been designed for synchronous data access: the attempt to retrieve a file happens while a client accesses it. If the temporary failure lasts for a long period of time, it is up to the client to “reschedule” the file access for later and try with some other file in the meanwhile.

A system relying on random clients for scheduling and monitoring file downloads is weak, whereas a separate and dedicated staging service is more appropriate, as we are going to see in § 4.3.5.

### 4.3.5 The dataset stager

One of the main components of the ALICE Analysis Facility (AAF) model is a daemon running in background called `afdsmgrd` (short for analysis facility dataset manager daemon), also known as the dataset stager. The presence of such daemon amends the original AAF data model by adding asynchronous pre-staging features.
on top of any storage (and in particular XRootD with vMSS) and works around
the issues extensively covered in § 4.3.4.

Its purpose is to orchestrate the transfer of a very large amount of data upon
request by triggering downloads and verifying retrieved data in a resilient way.

The dataset stager is at the time of writing (early 2014) currently in produc-
tion on all AAFs, and it has been since mid 2010.

4.3.5.1 Features of the dataset stager

The dataset stager, or afdsmgrd, is a daemon written in C++ whose purpose is to
orchestrate the transfer of a large number of files from diverse storage systems by
guaranteeing data integrity and by providing resilient failovers.

The daemon scans PROOF datasets in search for files which are marked as not staged and not corrupted and enqueues them. Files are then picked in an orderly
fashion from the transfer queue and a custom download script is invoked for each
one. When transfer is completed and successful, the “staged” bit is flipped in the
original dataset to signal the success; some additional metadata retrieved from
the file is saved as well. A description of the workflow in detail follows.

1. All PROOF datasets are scanned in search for non-staged and non-corrupted
files: such files are pushed at the end of the transfer queue. Since a file
might exist in different datasets, a check is performed to avoid inserting
the same file twice.

2. Files are picked from the FIFO transfer queue to fill the transfer slots, whose
number is configurable.

3. For every new file inserted in the transfer slots, the custom download and
verification script is invoked: a separate process is spawned for each file,
and the dataset manager takes note of the proces IDs. The standard output
is captured and saved on external temporary files.

4. The program checks if the spawned processes are still running: for transfers
that are not running anymore, the captured standard output is parsed to
check for a special string signalling a successful transfer. Transfers running
for too long hit a timeout limit and get killed.

5. Successful transfers are marked as such on the transfer queue.

6. Unsuccessful transfers are moved to the end of the queue with an increased
error count. After a certain number of failures, the transfer is eventually
marked as failed on the transfer queue and no further retries occur.
7. Periodically, PROOF datasets are re-scanned to save the status of completed transfers: successful files are marked as *staged*, whereas unsuccessful transfers are marked as *corrupted*. Each file status is updated in all the lists containing the same file.

8. All the finished transfers are deleted from the in-memory transfer queue. There might exist completed files whose datasets have disappeared in the meanwhile: such unclaimed files are removed from the transfer queue as well.

What we have called *in-memory transfer queue* is in practice a temporary SQLite database. SQLite has a very efficient way of handling which portion of the database to keep in RAM and which one to keep on disk, on a temporary file. Using SQLite allows for having a very large queue with constant RAM memory consumption. A test has been conducted with the following conditions:

- 10 000 000 URLs
- each URL 100 characters long

If all those URLs with additional metadata are stored in memory, they would take more than 1 GiB. Instead, SQLite maintained the overall afdsmgrd Resident Set Size (RSS) below 300 MiB.

The whole daemon has been written in plain C++03 using only the Standard Template Library and POSIX functions to minimize the dependency set as much as possible. SQLite has been included in a format called *amalgamation*, where the whole source code is merged in a single C source file. The only external dependency is ROOT, needed for accessing the datasets: starting from ROOT v5.30.00, released on Jun 28, 2011, afdsmgrd is distributed as part of ROOT. It is not enabled by default, but it can be enabled during the configuration phase:

```
./configure --enable-afdsmgrd
```

or, with CMake:

```
cmake <root_source_dir> -Dafdsmgrd=ON
```

ALICE Grid builds of ROOT have afdsmgrd enabled by default.

### 4.3.5.2 Robustness of the staging daemon

Since the task of transferring large amounts of data is extremely critical, the whole daemon has been developed with *robustness* in mind.
The daemon is completely \textit{stateless}, meaning that in the unfortunate event of a system failure, its operations can be simply resumed by restarting it. The only “stateful” database is constituted by the PROOF datasets, which are not scanned at every loop since the scan operation is costly, especially with a large number of datasets: \texttt{afdsmgrd} keeps an in-memory state of the completed files, which is dumped every once in a while, meaning that the amount of download statuses lost is minimal, constituting a good compromise between performances and robustness.

The daemon uses a single configuration file: all configuration variables can be changed in real time, without restarting the daemon, which automatically detects if the configuration file has changed. In case of errors in the configuration file the daemon remains up and running but pauses its operations. This feature allows to tune some parameters in real time (for instance, the number of parallel file transfers) without stopping current downloads.

First versions of the daemon used threads and the linked ROOT libraries to open and verify ROOT files. It might happen, however, in some cases, that badly written ROOT files lead to memory leaks and crashes, that constantly affected the whole daemon’s operations. For this reason, and also to give more flexibility in files transfer and verification, downloads are actually performed by a configurable external command run in a separate process. Since the external command might crash when opening corrupted files, the operation is considered successful only if a specific string has been produced in the standard output. Exit code is not checked. The string format in case of success is the following, and it has been chosen because it is compatible with the output produced by XRootD’s \texttt{xrdstagetool}:

\begin{verbatim}
OK <filename.root> Tree:<default_tree> EndpointUrl:<endpoint_url>
Size:<file_size_bytes> Events:<count_tree_entries>
\end{verbatim}

In case of a failure which did not lead to a crash, the following string is returned instead:

\begin{verbatim}
FAIL <filename.root>
\end{verbatim}

The fields have the following meanings.

- **Tree**: name of the default tree found in the ROOT file, or name of the first tree found

- **EndpointUrl**: URL to access the file directly. In case the input URL points to an XRootD redirector, the endpoint URL will point to the server actually hosting the file.
4.3 – ALICE Analysis Facilities and Their Evolution

- **Size**: the file size, in bytes.
- **Events**: number of “events”, *i.e.* number of entries of the default tree.

Such metadata is synchronized back to the PROOF datasets.

One of the problems XRootD’s vMSS does not deal with are temporary failures of the remote storage. To be as sure as possible that remote data will actually be retrieved, and to overcome resiliently any download client problem or remote storage issues, a *timeout* for each download can be specified: after the timeout, the download is considered failed.

Each entry has a failure counter, and it is possible to set up a maximum number of failures: upon each failure the counter is increased, but the download is not immediately retried: instead, the file is pushed to the end of the transfer queue, to avoid blocking the whole process on downloading the same file over and over. When the error limit is reached, the file is ultimately considered as *corrupted*.

Separating the daemon from the scripts and ROOT sessions actually downloading and accessing the files has the benefit of making the number of internal operations very reduced, meaning that the codebase is lightweight and therefore less error prone. Apart from standard Valgrind memcheck passes, a memory monitoring facility has been integrated in the daemon to spot possible memory leaks: the longest afdsmgrd run was in the TAF in Torino where the daemon’s RSS has not gone above 400 MiB after six months of continuous operations. The daemon was then restarted to allow for a system upgrade.

### 4.3.5.3 Daemon status monitoring: the MonALISA plugin

Following the exigences of the ALICE experiment, the staging daemon can optionally notify its status to an external monitoring facility. Compiled plugins (in the form of *shared objects*) are used for monitoring: plugins can be loaded and unloaded in real time without restarting the daemon by simply editing the configuration file. The following information is monitored:

- Resident Set Size (RSS) in *KiB*
- virtual memory in *KiB*
- wall time (uptime), user time and system time in *s* (to calculate the absolute CPU percentage used since daemon has started)
- delta time, user time and system time since last main loop call in *s* (to calculate the CPU percentage used by the last main loop call only)

In addition, the list of PROOF datasets is sent. For each dataset the following information is provided:
dataset name
number of files
number of staged files
number of corrupted files
default tree name
number of entries in the default tree
total size of the dataset (in bytes)

The only plugin distributed with afdsmgrd is the MonALISA monitoring plugin, which sends information using the ApMon client library. Monitored variables are accessible using the MonALISA client, and graphic reports and tables are periodically generated on the ALICE MonALISA dashboard.

4.3.5.4 The ALICE download script

The download script used in AAFs (afdsmgrd-xrd-stage-verify.sh) takes two arguments as input:

- the destination file URL (i.e., root://server//alice/data/...);
- the default tree to check (i.e., /aodTree).

The following operations are performed in order.

1. The command xrdstagetool -p is invoked to tell XRootD to prepare the file. XRootD will use its internal vMSS facility to retrieve the file from the remote location. The -p switch tells the command to wait until the transfer is over (synchronous execution) instead of exiting immediately: xrdstagetool will periodically poll XRootD to see the download status.

2. Once the download is OK, a ROOT macro is called to test if the file is accessible, if the default tree actually exists and if the file is not corrupted: impossibility to read the default tree is seen as a strong indicator of file corruption. Such macro is used also to return metadata such as the endpoint URL and the count of entries in the default tree.

3. A string following the format mentioned in § 4.3.5.2 is returned on the standard output and will be parsed by the staging daemon.
4.3.5.5 The dataset management utilities for ALICE

As we have seen, PROOF datasets are used by the dataset stager to determine which files to download. Since PROOF datasets are directly accessible by end users, while the dataset stager itself is not, creating particular datasets is a way for the user to communicate which files to transfer to the local analysis facility.

For the ALICE use case, PROOF datasets are entirely created starting from AliEn File Catalog entries, which are returned by issuing a find command from within the AliEn shell. A set of utilities has been developed to create PROOF datasets from entries of the AliEn File Catalog. Since afdsmgrd considers only files which are marked as non-staged and non-corrupted, such utilities also provide the user with a function that allows to “reset” file flags to trigger the download.

This set of utilities is available as one single ROOT macro (afdsutil.C) to enable on the PROOF client. On AAFs it is also available as a special PARfile. Available since late 2010, the dataset management utilities have been obsoleted in late 2012 by the introduction of the TDataSetManagerAliEn class which makes the AliEn File Catalog directly accessible by PROOF without the need to create intermediate datasets (§ 4.3.6).

4.3.5.6 Alternative usage of the staging daemon

The dataset staging daemon is greatly flexible as it allows for specifying a custom command to retrieve the final data. Such custom command usually includes, as seen in § 4.3.5.4, a call to the ROOT executable to verify whether the file can be opened and the tree can be actually read.

Since the command can do arbitrary things, it can be also used to perform preliminary data filtering and reduction, in cases where local users do not need all the available tree branches or events and there is little storage available.

Such approach is currently used at Subatech in Nantes (FR) on the SAF: it is not automatically enabled for each file, but instead you would stage “virtual” file names to trigger the appropriate filtering.

If we consider for example the following actual AOD file:

```
alien:///alice/data/2013/LHC13d/000195725/ESDs/muon_pass2/AOD134/0073/AliAOD.root
```

we can request for staging the following virtual file:

```
alien:///alice/data/2013/LHC13d/000195725/ESDs/muon_pass2/AOD134/0073/AliAOD.FILTER_AODMUONWITHTRACKLETS.root
```

This file does not actually exist, but the copy macro parses the file name, downloads the plain AliAOD.root and saves locally the same AOD filtered using a
The AOD filtering method by means of the dataset stager has been developed by Laurent Aphecetche and deployed by Jean-Michel Barbet.

4.3.6 A PROOF interface to the AliEn File Catalog

The PROOF standard datasets model (§ 4.3.3.1) has a series of limitations primarily due to their static nature and the attempt to use them to reflect the status of the current local storage.

Starting from the principle that all major experiments have large scale File Catalogs, a modern dataset manager should act as a mere interface to access it through PROOF.

For what concerns the presence of files on the local storages, there is no need to save statically which files are staged inside a dataset: all modern storage systems, including XRootD\textsuperscript{[38]}, feature mechanisms to perform fast lookups. A new dataset manager should leverage such features properly.

In the ALICE experiment, as storage in the AAFs is automatically managed and changes continuously (§ 4.3.3), efforts have been made to keep the information provided by static datasets up to date. Unfortunately, the number of datasets used in CAF only trespasses 5000, and the synchronization process is extremely slow and resource consuming.

Datasets are also scanned by the dataset stager, whose process is tremendously slowed down by the presence of thousands of inactive datasets.

To overcome these problems in ALICE, the new TDataSetManagerAliEn class has been created: it is a new dataset manager which acts as an intermediate layer between PROOF datasets and the AliEn file catalog.

Dataset names do not represent any longer a static list of files: instead, it represents a query string to the AliEn file catalog that creates a dataset dynamically.

Locality information is also filled on the fly by contacting the local file server: for instance, in case a XRootD pool of disks is used, fresh online information along with the exact host (endpoint) where each file is located is provided dynamically in a reasonable amount of time.

The main local URL to contact is retrieved in one of the two following methods:

- In case the storage is a dedicated one, we will assume that its namespace is the same as the AliEn File Catalog’s: that is, files are stored locally with a directory structure identical to the AliEn path. In this case, a simple substitution is made to guess the local file name.

- In case we are reusing a Tier’s storage, each local file name is called a Physical File Name (PFN), which does not clearly point to the corresponding
AliEn file name or the Logical File Name (LFN), as it is constituted by a hexadecimal unique identifier. In this case, the AliEn whereis query is performed on each desired file to retrieve the endpoint URL, which is returned only if the file is effectively available on the local Tier storage.

Both file catalog queries and locality information are cached on ROOT files: cache is shared between users and its expiration time is configurable. Being it a mere cache, it is volatile by definition, and cannot therefore be used for triggering staging requests. To overcome this problem, a separate and more straightforward method for issuing dataset staging requests has also been provided (§ 4.3.6.2).

Another advantage of this model is that, having no static dataset repository, it can be used with PROOF-Lite as well.

How to configure PROOF and PROOF-Lite to use this dataset model is explained in § C.3.

### 4.3.6.1 Dataset queries

The new dataset manager is backwards-compatible with the legacy interface: each time users wish to process or obtain a dataset, instead of specifying a string containing a dataset name you will specify a query string to the file catalog.

Query strings are *semantic*, meaning that they represent data based on information rather than on the effective path. Three different query string formats are available. For official data:

```
Data;Period=<LHCPERIOD>;Variant=[ESDs|AODXXX];Run=<RUNLIST>;Pass=<PASS>
```

For official Monte Carlo:

```
Sim;Period=<LHCPERIOD>;Variant=[ESDs|AODXXX];Run=<RUNLIST>
```

Raw queries to the AliEn catalog are also possible with a syntax that resembles the `alien_find` command:

```
Find;BasePath=<BASEPATH>;FileName=<FILENAME>
```

An exhaustive user’s guide to AliEn dataset queries is available[1]. PROOF datasets are generated dynamically and cached. It is worth noting that an additional parameter can be specified to the query to select the data access mode:

```
Mode=[local|remote|cache]
```

where *local* is the default. The meaning of the values is:
• local is used to tell PROOF to analyze only those files having a local copy available;
• remote passes PROOF only the remote URL: files will be directly taken remotely by querying the AliEn file catalog;
• cache generates local URLs for all files, even those not present on the local storage: this assumes the presence of something like the XRootD vMSS system described in § 4.3.3, that automatically downloads files upon request.

4.3.6.2 Issuing staging requests

The dataset stager uses PROOF datasets on files to know what files to download and to signal the download status. With the AliEn datasets model, there are no local files anymore, apart from the cache.

Staging requests are now issued with a very straightforward method. The user wanting to stage a certain dataset simply types:

```c
1 gProof->RequestStagingDataSet("QueryString");
```

where the query string has the format described in § 4.3.6.1. Under the hood, the method:

• queries the file catalog;
• creates a file containing the dataset in the “old” format;
• stores it in a “dataset requests” directory.

Similarly, the download status can be monitored with:

```c
1 TProof->ShowStagingStatusDataSet("QueryString", ["opts"]);
```

where filtering options are possible as well[1]. Requests can also be cancelled:

```c
1 gProof->CancelStagingDataSet("QueryString");
```

The dataset manager has a special configuration option (see § C.2.1) to deal with such special repository: once a dataset staging has been completed, the corresponding file is deleted. Since we are not using static datasets anymore, there is no need to keep the dataset files: all the scalability problems of the staging daemon are therefore simply solved, as the number of dataset files in the “requests” directory is always low, thanks to this automatic cleanup.
4.3.7 The first virtual Torino Analysis Facility for ALICE

As PROOF got endorsed by the ALICE collaboration as an official computing model complementary to the Grid\cite{44}, we became interested in having a PROOF-based interactive infrastructure in the Torino’s Tier-2. Due to the modest availability of resources, in 2008 we developed a model where every physical node runs two virtual machines: one being the Grid node, while the other is a PROOF worker.

A result of this work was a small prototype we called the “Virtual Analysis Facility”\cite{15}. Despite the roughness of such a dedicated solution, with no cloud controller involved, the prototype allowed us to prove the feasibility of virtualization in a small computing centre performing HPC, both from the management and from the performances perspective, where the latter was clearly demonstrated by a series of specific benchmarks.

Later on, when the AAF model was born in 2010, some analysis facilities started to be deployed in various ALICE computing centres. We wanted to make use of the experience previously acquired both in PROOF and in virtualization to provide an AAF-compliant solution which was also sustainable for our computing centre. Open source cloud controllers such as OpenNebula\cite{75} started to become popular and stable, providing a more generic and richer solution with respect to our original prototype.

Our intention of having a PROOF solution was the opportunity to migrate the whole computing infrastructure to a virtualized one, whose challenges, implementation and outcomes are fully covered in § 2. Our first use case was to bring the original prototype to the production stage by using more modern tools, and the first iteration of our cloud shared resources only between the Grid and PROOF.

We call our first production and AAF-compliant Virtual Analysis Facility the Torino Analysis Facility (TAF): as we will see in the following sections, we have decided to retain the compatibility with other analysis facilities, allowing ALICE users to switch to us by just changing the PROOF hostname, without relinquishing the advantages of a dynamic and virtualized deployment. To achieve the result, the TAF has some differences in the configuration with respect to the AAF model which remain transparent to the end user.

4.3.7.1 Overview

The TAF is accessible to all ALICE users as it uses the same LDAP and certificate authentication model covered in § 4.3.2. To connect to the TAF, users have to specify the hostname of the PROOF master:

```
TProof::Open( "username@pmaster.to.infn.it" );
```
The TAF is technically an ordinary PROOF analysis facility, with:

- one head node running a system-wide xproofd daemon forking into PROOF masters: such head node is a physical node;
- a variable number of virtual machines, each one running a system-wide xproofd daemon forking into PROOF workers.

Even if it does not make use of PROOF on Demand (PoD), which would make it incompatible with the AAF specifications, our deployment is dynamic, i.e. we can change on the fly the number of PROOF workers without restarting the xproofd daemons.

This can be in practice achieved by starting and shutting down dedicated TAF virtual machines: the boot and the shutdown procedures automatically invoke a series of scripts, as described in § 4.3.7.4, adding a form of dynamicity in our “static” deployment (§ 4.2.1).

It must be noted that, since the TAF is an independent implementation of the AAF specifications, the installation procedure and script for standard AAFs[^4] do not apply to it. Custom TAF management tools are fully covered in § D.

### 4.3.7.2 Storage and datasets model

Since TAF workers are disposable virtual machines, data is not distributed on them: instead, a separate and dedicated storage is used. The storage is composed of:

- a storage array, serving three logical units of 17 TB each;
- a single frontend server, aggregating the space into a virtual volume of 51 TB.

The storage array is connected to the server using fiber channel, and the server is connected to the local Tier-2 network, hosting also all Grid nodes and PROOF nodes, through a 10 GbE interface. The connection uses DM-Multipath[^36] for greater performances and reliability.

The three volumes are seen by the storage server as separate block devices: they are formatted separately with three different XFS filesystems. Disk aggregation is performed by GlusterFS: the three partitions are bricks of the same volume. GlusterFS is not configured, in this case, to perform any replication or striping, as data redundancy is already performed at RAID level by the storage server.

Each client mounts the GlusterFS volume via FUSE so that PROOF can access data by simply opening files from the filesystem tree (like local files), as ROOT does not have any direct GlusterFS interface.
We benchmarked the GlusterFS storage using a TSelector with an empty event loop, run under PROOF several times with an increasing number of workers. The minimal increase we used is 6 workers, corresponding to the number of cores of each PROOF virtual machine. The purpose of this test is to measure how GlusterFS scales with several concurrent file accesses in a real environment: the local network infrastructure used by PROOF and GlusterFS is not dedicated, as it is for the storage array.

Collected data is reported in Table 4.2 and plotted in Figure 4.4: results show how the throughput scales almost linearly with the number of accesses. The peak is reached with 84 workers, which is therefore the optimal number of workers to use on TAF when doing I/O bound analysis. After that value, where we reach the maximum of 600 MB/s, a plateau is reached. That maximum throughput rate is the one obtained by performing a read test on the storage frontend server: neither the network infrastructure, nor the network interface (10 GbE) constitute a bottleneck for our setup, which results optimally balanced.

GlusterFS has been preferred over xrootd as it has advanced data management features, as seen in § 2.4.4.1: since it is a mountable filesystem, all the ordinary filesystem operations are possible, as if all the files were local. Moreover, GlusterFS makes very easy to add, replace and rebalance “bricks” as online operations, i.e. without stopping the storage server. A XRootD interface serving the GlusterFS-mounted filesystem is also provided for compatibility: no aggrega-
<table>
<thead>
<tr>
<th>PROOF Workers</th>
<th>Speed [MB/s]</th>
<th>Per worker [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>216</td>
<td>7.2</td>
</tr>
<tr>
<td>36</td>
<td>281</td>
<td>7.8</td>
</tr>
<tr>
<td>42</td>
<td>315</td>
<td>7.5</td>
</tr>
<tr>
<td>48</td>
<td>355</td>
<td>7.4</td>
</tr>
<tr>
<td>54</td>
<td>416</td>
<td>7.7</td>
</tr>
<tr>
<td>60</td>
<td>473</td>
<td>7.9</td>
</tr>
<tr>
<td>66</td>
<td>511</td>
<td>7.7</td>
</tr>
<tr>
<td>72</td>
<td>535</td>
<td>7.4</td>
</tr>
<tr>
<td>78</td>
<td>573</td>
<td>7.3</td>
</tr>
<tr>
<td>84</td>
<td>598</td>
<td>7.1</td>
</tr>
<tr>
<td>96</td>
<td>562</td>
<td>5.9</td>
</tr>
<tr>
<td>108</td>
<td>560</td>
<td>5.2</td>
</tr>
<tr>
<td>126</td>
<td>563</td>
<td>4.5</td>
</tr>
</tbody>
</table>

**Table 4.2:** Results collected while measuring the throughput of the GlusterFS TAF storage with multiple concurrent PROOF accesses. Test performed via an I/O bound analysis running an empty event loop on a tree with GlusterFS volume mounted via FUSE on each PROOF node.
tion or caching features of XRootD are used.

We repeated our previous benchmark fetching data from the XRootD mounted GlusterFS partition: no differences at all were shown, primarily because in our setup (GlusterFS aggregating three bricks from the same node) we do not take advantage of any redirection feature of either GlusterFS or XRootD.

Despite all the other AAFs use XRootD as protocol to access data, the user does not have to care about such discrepancy: as AAF specifications suggest, input data is specified through a dataset name. Such datasets, as we have seen in § 4.3.3.1, are mere list of files: datasets with the same name across different AAFs are guaranteed to point to the same data, while physically retrieving it from different sources.

Given the small amount of users, we have decided to not activate any cache-like feature on our storage, such as the vMSS of XRootD (§ 4.3.3): this means that if a user attempts to retrieve a file which is not there, it simply fails. In practice we treat our dedicated storage as a custodial storage, where the sysadmin has full and exclusive control on what is stored there.

Data is still transferred to the local storage via the afdsmgrd dataset stager (§ 4.3.5), however users do not have permissions to trigger data staging via datasets.

The drawback of a dedicated storage model is that it is not directly possible to use exactly the same storage used for the Tier-2 Grid site: files with a meaningful Logical File Name (LFN) in the AliEn File Catalog are stored on local sites with hexadecimal strings as filenames that correspond to their “unique identifier” in the catalog. Each of these entries is called a Physical File Name (PFN).

In order to map LFNs to PFNs it is necessary to query the catalog, which constitutes a significant bottleneck. The problem is twofold.

- **LFN → PFN.** If we know that the files we wish to process are already stored on our reference Tier-2 site, we can query the catalog once to map all the LFNs to PFNs, provided that the files are available, and cache the results for further queries. Such lightweight solution has been implemented in the PROOF interface to the AliEn File Catalog, as extensively described in § 4.3.6.

- **PFN → LFN.** If we wish to list the available files on our reference Tier-2 storage by their LFNs we have to perform reverse queries to the catalog. While the first problem is relatively simple to work around, the second might pose serious scalability problems, as the LFN directory structure has to be “rebuilt” (and possibly cached and kept up-to-date) before the content can be naturally browsed. Sensible solutions for this problem are under discussion, as we will see in § 5.3.4.
4.3.7.3 Software distribution model

AAFs used AliEn Packman as software distribution method (§ 4.3.1.2), prior to CernVM-FS. Since in our Grid Tier-2 we already had a shared software partition exported through the Lustre filesystem, we have decided to rely on that one for what concerns AAF software.

Notably, ALICE software was kept up to date on that partition by means of AliEn Packman itself, for Grid purposes. In practice, the TAF uses a central Packman repository shared with the Grid worker nodes instead of separated Packman installations for each PROOF node.

It must be noted that relying on an external and site-specific software repository is not compliant to the Infrastructure as a Service (IaaS) model: in other words, we cannot simply use our PROOF virtual machines and run them on any cloud. The IaaS model will be embraced by the Virtual Analysis Facility, as we will see in § 4.4, which started as an evolution of the TAF model.

When the CernVM-FS software repository for ALICE was still experimental, we have started using it first on the TAF, then on Grid nodes as well. The TAF was the first AAF that migrated to CernVM-FS (February 2013).

4.3.7.4 Dynamically changing PROOF resources

As we have seen, TAF has a “semi-static” PROOF configuration: it appears like a static and dedicated PROOF cluster, yet it is possible to add and remove PROOF nodes without manually intervening on the configuration and without restarting the xproofd daemons.

This is possible via the PROOF resources configuration file, commonly called proof.conf. This file is read by PROOF every time a new session is opened. In a standard and static PROOF deployment the file is created by the system administrator: in our deployment, the file is automatically created and updated by a mechanism that relies on SSH and a special script. The procedure is illustrated in detail in § D.2.

The technique of managing the proof.conf file with external tools to provide dynamic resources is the same one used by PROOF on Demand (PoD).

4.3.7.5 Virtual machine images and Tier-2 integration

As we said, the first version of the Torino Analysis Facility (TAF) is based on a real host used as PROOF master, whereas PROOF workers are virtual machines.

The creation of the TAF was done right after the migration of the Grid site. Many points in common between PROOF and Grid virtual worker nodes were identified, such as:
same shared Grid software repository used (a Lustre volume, § 4.3.7.3)

same base operating system required from software for binary compatibility (namely: a RHEL 5-based operating system)

after the CernVM-FS migration, same cache and proxy configuration;

same resource requirements in terms of GB of RAM per core, and disk space;

same list of valid CAs and CRLs for Grid authentication;

Tier-2 specific configurations deployed via our Puppet service (§ 2.4.1.2) in both cases.

For those reasons we have decided to opt for maintaining a single virtual machine image for both cases, and to specialize it to either a Grid node or a PROOF node by means of the OpenNebula contextualization mechanism.

This approach is hybrid between realizing two special PROOF and Grid images, and maintaining a single minimal operating system installation with all the configuration performed through contextualization. As we debated in § 3.1.1.4, saving two different images would double the maintenance and update work, while the RPM packages for a Grid node are so many that installing all of them during contextualization raises the boot time up to 20 minutes and the chances of something failing during the process are higher.

Our base virtual machines for the “old” TAF are SLC 5 64 bit minimal installations with a certain generation of the Grid packages installed using a standard procedure[56]. The node supports PBS/TORQUE as well, which is specific of our infrastructure.

The OpenNebula contextualization procedure is specific as well. In our case, it consists of running a Bash script which substantially adds to the virtual machine a series of SSH key fingerprints of known infrastructure hosts for security reasons. Eventually, the node asks to join the PROOF cluster and a hook to remove it during a graceful shutdown is inserted in the init.d scripts (§ 4.3.7.4, § D.2).

Contextualization stanzas are described in detail in § D.6.

Even if the TAF configuration procedure has been packetized in order to be easily exported and replicated on other sites, the final configuration is tightly integrated with the rest of the Tier-2 infrastructure, constituting an example of datacenter virtualization as opposed to Infrastructure as a Service (IaaS). However, it allowed us to prepare the ground for the IaaS-compliant Virtual Analysis Facility, as we will see in § 4.4.
4.3.8 Spawning and scaling the TAF

Since all the configuration and the machinery to associate virtual machines to the PROOF cluster are embedded, scaling the “old” TAF is as easy as starting and stopping virtual machines from the OpenNebula interface.

An OpenNebula virtual machine template is registered: it contains all proper references to invoke the contextualization. Since everything is modularized, even a system administrator with no experience on PROOF can scale up and down the PROOF cluster when necessary: the OpenNebula Sunstone web interface provides a graphical “knob” that anybody can turn upon need even when the PROOF expert is unable to intervene. Screenshots showing how to start and stop TAF virtual machines are displayed in Figure 4.5.

![Virtual Machines](image)

(a) Starting new TAF nodes via the registered OpenNebula template. The number of new nodes can be specified.

![Virtual Machines](image)

(b) TAF can be scaled down by shutting down its nodes.

**Figure 4.5:** Scaling up and down the “old” TAF can be done easily through the graphical interface of OpenNebula: no PROOF expert is needed for the task.

It must be noted that the “old” TAF does not have the possibility to scale up and down by interacting directly with the cloud controller to start and shutdown
virtual machines: every operation must be performed manually. Automatic scalability is instead offered by the elastiq daemon which is part of the Virtual Analysis Facility, as we will see in § 4.4.7.

4.4 The Virtual Analysis Facility

Clouds are often considered as an evolution to the Grid model when it comes to abstracting resources from the corresponding physical hardware and geographical location. Compared to the Grid, however, clouds have much less requirements in terms of implementation and they expose much more lightweight interfaces.

Both the Grid and clouds are meant to support a broad variety of use cases. To achieve such genericity, the Grid is designed as a very complex system in its implementation: every use case is configured to use a subset of the available functionalities.

The layer of resources abstraction on which the clouds are based is hardware virtualization: this allows clouds to expose very simple interfaces to launch, monitor and control virtual machines, which end users can make as complex as they wish. In other words, clouds achieve genericity by entirely moving specific implementation problems to the configuration of the virtual machines, and not the cloud infrastructure itself.

As we have seen, the “old” Torino Analysis Facility (TAF) (§ 4.3.7) was born primarily to make easy for a system administration to temporarily convert some hardware resources dedicated to the Grid to PROOF: the implementation of such approach led to a solution which cannot be easily exported to many computing centres—although it is a solution based on virtualization, it cannot be considered “cloud computing”.

Based on the experience acquired on virtualization and PROOF thanks to the TAF, we have decided to develop the Virtual Analysis Facility (VAF) to tackle the complexity of deploying a PROOF cluster with an Infrastructure as a Service (IaaS) approach: following the cloud terminology of abstracting “everything” into a service, the Virtual Analysis Facility is an entire cluster of zero-configuration virtual machines dedicated to PROOF analysis. The Virtual Analysis Facility is completely self-contained, meaning that it can run on every cloud, public or private, which allows the contextualization of virtual machines. Moreover, its configuration is not specific to either ALICE or other experiments, making the Virtual Analysis Facility a complete and generic “analysis cluster in a box”.

In the following sections we will cover all the conceptual and implementation aspects of the Virtual Analysis Facility. We start with an overview of the non-trivial challenges we have to face when building a stack of distributed applications running on the cloud, defining the concept of “cloud awareness” (§ 4.4.1).
overview of the components is then presented (§ 4.4.2), and in particular the CernVM ecosystem (§ 4.4.3) and other external components (§ 4.4.4).

How the Virtual Analysis Facility seamlessly uses Grid authentication credentials for its users is explained in § 4.4.5. Within the context of the Virtual Analysis Facility, a new important feature has been added to PROOF: the dynamic addition of workers, illustrated in § 4.4.6. The Virtual Analysis Facility is elastic, capable of scaling up and down automatically: this is obtained through a specific daemon called elastiq (§ 4.4.7). User interaction with the Virtual Analysis Facility is covered in § 4.4.8.

Finally, performance considerations showing the benefits of using PROOF and the quick deployment of Virtual Analysis Facility virtual machines are presented in § 4.4.9.

4.4.1 Cloud awareness

The main problem of running an infrastructure on the so-called “cloud” is that we have no clear definition of what the cloud, or a specific cloud, is. Such uncertainty usually covers two aspects.

- **Resources are diverse.** In the cloud, we request for some “resources”, but we cannot ever be sure whether they correspond to real ones or not.

  Since Amazon EC2, clouds provide different “resource profiles”, each defining a combination of number of CPUs, amount of RAM memory and size of an attached (“ephemeral”) storage. Profiles have semantic names like “m1.large”, however profiles with the same name on different clouds might differ. In practice, a profile name does give you only coarse grained information on how your virtual machine will be made.

  Moreover, the assigned resources may not correspond exactly to their underlying physical resources. In fact, in some cases and on some clouds it might happen that resources are overcommitted, meaning for instance that the some of the CPUs assigned to my virtual machine might be shared with some other virtual machine.

- **Virtual machines are volatile.** When requesting a virtual machine, we cannot know when it will be actually ready to be used: in some cases it might happen that your request is enqueued and satisfied as soon as possible, while in other cases a request fails immediately and you will have to reattempt the submission manually.

  The biggest problem of availability is that virtual machines run on a certain cloud which is not under the submitter’s control: this means that failures might happen, killing my virtual machine. Moreover, some “dynamic”
clouds might define a maximum hard lifetime for virtual machines, abruptly shutting them down when over time quota.

In other words, clouds are potentially troubled environments, where things can go wrong, and sometimes in a very catastrophic way. This problem in fact is not cloud-specific and it exists within the Grid as well: in the cloud we have virtual machines, whereas in the Grid we had “jobs”.

The clouds however do not only provide a higher level of abstraction to the resources, but an abstraction in the administrative domains as well. While Grid sites are internally managed by the scientific community, there exist a variety of public clouds on which we would share our resources with industry and private users. While a specific application problem can be solved by “tweaking” some configuration parameters of certain Grid sites, when we are on the cloud we are forced by a rigid separation of the administrative domains to solve the application problem at application level.

Concerning the two big uncertainties of cloud computing, we must build our “application” (which in our case is an entire Infrastructure as a Service) in a way that:

- it reacts to failures as smoothly as possible;
- a failure of part of the system will never turn into a failure of the whole system;
- it dynamically scales according to the quality and amount of continuously changing resources.

In practice, our application should somewhat accept failures and underperforming resources as a natural part of the cloud, and work around them as much as possible, as opposed to trying to fix them. We call an application which satisfies such conditions a cloud aware application.

More specifically, for what concerns HEP computing, our IaaS applications are essentially virtual clusters with one head node and several identically configured computing nodes. We have already said that our virtual machines become available gradually and might perform very differently: the essential way of tackling the problem of cloud awareness in our case is to separate computing resources from the data they analyze.

4.4.1.1 PROOF is cloud aware

The Grid (as all data driven batch analysis) is not cloud aware, as its workflow consists in:

1. pre-split the input dataset to a pre-determined number of workers (“jobs”);
2. run them;
3. when the last one has finished, collect data and merge the results.

This way of assigning the workflow is called push scheduling: even if input data is evenly distributed among the workers, they may not finish processing at the same time.

We already know that the PROOF computing model features a pull scheduler instead (the “adaptive packetizer”, § 4.1.4): data is assigned to the workers as long as the analysis is ongoing. No pre-assignment of data to the workers is the first requirement of a cloud aware data driven application: from this perspective, PROOF seems to constitute a good choice for performing cloud computing in HEP.

In the context of cloud awareness, PROOF missed one feature that has been added as part of this work: the possibility to add new workers while an analysis is ongoing, also called the dynamic workers addition. This new feature will be covered in detail in § 4.4.6.

It must be noted that what makes PROOF a cloud aware application is that it is an interactive computing model, where the interactivity is between the master and the workers (§ 4.1.2): it is such continuous communication that makes possible the adaptive assignment of data and the dynamic addition of workers.

The whole Virtual Analysis Facility setup aims to be a cloud aware application as much as possible: in addition to the PROOF foundations, in the next section we will see how all the ancillary components were also deliberately chosen, configured and developed to contribute to the cloud awareness.

### 4.4.2 Components

The Virtual Analysis Facility is a IaaS cluster of virtual machines. Its main components are:

- ROOT and PROOF as an analysis framework and computing model;
- the CernVM ecosystem for the cloud deployment;
- PROOF on Demand (PoD) and HTCondor for scheduling users and preventing the issues of a static PROOF deployment (§ 4.2.1);
- the new elastiq daemon communicating to an EC2 interface to provide automatic and transparent scalability of the VAF cluster;
- an interface for authentication and authorization allowing Grid users to connect via SSH using only their Grid credentials, called sshcertauth;
• a client providing the shell and workers environment based on specific experiments frameworks.

All these components will be described in detail in the following sections.

Despite the large number of components and their intrinsic complexity, none of them requires manual configuration. In practice, a Virtual Analysis Facility cluster is constituted by a set of unmodified µCernVM virtual machines specialized and configured through contextualization. After the boot, such virtual machines do not require any manual login to perform extra configuration steps: instead, the VAF can be used right out of the box.

4.4.3 The CernVM ecosystem

The CernVM ecosystem is constituted by a set of tools and components specifically designed to tackle the issues of running on demand virtual machines on cloud environments.

The base component is a virtual machine image called CernVM, whose latest version is CernVM 3, also known as µCernVM.

The other components used in the context of the Virtual Analysis Facility are an online web interface to create contexts for CernVM, called CernVM Online, and finally the CernVM Filesystem (known as CernVM-FS), a mountable filesystem downloading and caching files on demand.

4.4.3.1 µCernVM: an on-demand operating system

µCernVM\cite{19} is an operating system based on Scientific Linux CERN 6 specifically designed for running as a virtual machine. Unlike many cloud-oriented operating systems, µCernVM is not a “just-enough”, stripped down operating system containing a minimal set of components: instead, it is a fully fledged installation of SLC 6\cite{101}.

In spite of that, the base µCernVM image weights less than 20 MiB in size: as a matter of fact, the base image contains only a minimal set of components required to boot the virtual machine, while the whole root operating system is mounted remotely. We can therefore say that µCernVM is an “on-demand” operating system: the small size of the base image makes the deployment very easy in various network and cloud conditions.

The root filesystem itself is based on AUFS\cite{12} which is a union filesystem, i.e. a filesystem composed by overlaying different filesystems which will act and appear as one. In particular, AUFS is used to combine:

• a read-only remote filesystem called CernVM-FS: the remote mountpoint contains a full installation of µCernVM (more details on CernVM-FS in
a writable local filesystem used to store all the differences to the read-only mountpoint.

A schema of such overlay is represented in Figure 4.6.

Figure 4.6: Layered architecture of \( \mu \text{CernVM} \): through the AUFS kernel module, a writable partition is overlayed to the root filesystem on CernVM-FS. Since it is not possible to save a modified version of the image, all configuration can only occur through contextualization.

Source: Jakob Blomer.

There exist several remote mountpoints for the root filesystem of \( \mu \text{CernVM} \): each one constitutes a snapshot of a certain installation of RPMs. When \( \mu \text{CernVM} \) is booted for the first time, the most recent snapshot is chosen, and throughout reboots the same snapshot is kept.

Being the whole operating system available only on a remote mount point, with \( \mu \text{CernVM} \) there is no possibility to save a customized base image with your preconfigured appliance: instead, every customization must be performed by means of contextualization.

Thanks to the contextualization mechanism and consistent root filesystem snapshots, the preferred way of upgrading the operating system consistently is to simply delete the virtual machine and create a new one with the same contextualization applied.

Being a full SLC 6 operating system, \( \mu \text{CernVM} \) features the Yum package manager: it is however not recommended to upgrade the system by means of \texttt{yum upgrade} as this would defeat the purpose of having an on demand operating system.

Every snapshot is practically an installation of a certain set of RPM packages with their dependencies consistently resolved. The resolution of dependencies
is performed deterministically by a custom tool which resolves the problem as an Integer Linear Programming (ILP) minimization problem, where the optimal set of solutions represents the consistent list of packages to install. The reason why a custom tool has been written is that the standard Yum tool does not resolve package dependencies in a deterministic way, but it uses a shallow heuristics: this means that the final set of packages installed if using Yum differs if we change the order of their installation.

Each and every snapshot is meant to be kept forever available on the CernVM servers: this aspect and the dependency consistency are used to tackle the problem of HEP long-term data preservation\[^{64}\]. This means that, in principle, in some years from now it will be possible to run some “ancient” analysis tool under exactly its original environment.

Since the Virtual Analysis Facility is based on \(\mu\)CernVM, the same considerations on long-term preservation apply: a parallel analysis run on the VAF today will be able theoretically to run in parallel in many years from now by reproducing the same environment as today.

**Contextualization in \(\mu\)CernVM.** The Virtual Analysis Facility was the first public project based on \(\mu\)CernVM: since its development started before the first stable release of \(\mu\)CernVM, the development of the Virtual Analysis Facility is a testbed of many \(\mu\)CernVM features and helped in fixing several bugs and problems.

A major contribution to \(\mu\)CernVM was a partial rewrite of its contextualization mechanism, inherited from previous versions of CernVM.

\(\mu\)CernVM supports contextualization in the amiconfig\[^{47}\] format, which consists in a single text file divided into sections, where parameters are simple key-value pairs. Such single text file can be generated, as we will see in § 4.4.3.3, by means of the CernVM Online interface.

While the context file is interpreted by the amiconfig tool, the way to retrieve such file varies according to the cloud controller used. The part of contextualization used to retrieve the amiconfig file has been completely rewritten and it supports the following contextualization mechanisms.

- **CernVM Online.** This contextualization occurs interactively after boot. The context script is retrieved from the CernVM Online web site.

- **EC2.** A special external HTTP server, called the metadata server, is configured in a way that it is visible only from the virtual machine. It runs on a special link-local IPv4 address (namely, 169.254.169.254): the context script is downloaded from such metadata server.

- **CloudStack.** As for EC2, the method is based on a metadata server as well. Its address however is the IP address of the DHCP server which provided
the address for the virtual machine.

- **OpenNebula.** As we have already seen for the “Virtual Routers” in § 3.2.2.2, OpenNebula burns a virtual CD-ROM attached to the virtual machine containing a single shell script with the defined variables. The context script in this case is converted into a long Base64 string, and assigned to the variable EC2_USER_DATA.

The meta-contextualization script first tries to determine the context source, then it retrieves the plain text context script to a known location, and eventually the amiconfig tool is invoked.

Differently from its predecessors, µCernVM also supports contextualization through CloudInit\(^\text{[29]}\).

### 4.4.3.2 CernVM-FS

The CernVM Filesystem\(^{[3]}\), also known as CernVM-FS or CVMFS, is a remote filesystem mountable via FUSE using HTTP as transport layer and heavily relying on different levels of caching.

CernVM-FS was born to address the problem of efficient experiments software deployment on CernVM virtual machines. Since CernVM (prior to µCernVM, or CernVM 3) was supposed to be a self-contained operating system specially designed for running HEP applications, the software from the experiments had three major problems:

- several versions of the experiment’s framework are available at the same time, each one taking several GiB of disk space: it is impossible to create a special virtual machine image containing a single one of such versions;

- experiments have, in some cases, daily release cycles: deployed software must be available as soon as possible to the end user;

- since a CernVM image is supposed to run on many clouds, the hypothesis of mounting a software repository internal to a certain computing centre is unfeasible.

In other words, a remote filesystem designed to deploy all that cannot fit into CernVM, which however gives the impression that everything is inside, needed to be developed. An alternative solution such as Andrew File System (AFS) was evaluated, but it posed many problems of integration: it requires special kernel support, it has no clear control on the cached files and it has a series of inefficiencies due to the fact that it is a read-write filesystem, which is a feature we do not need.

CernVM-FS was developed with the following relevant features.
• HTTP-based transport layer. Relying on an existing protocol means that a simple web server can act as a CernVM-FS server. More importantly, a wide range of caching proxies exist, Squid being the most well known. Having a site cache dramatically improves

• Based on immutable objects. Every file in the CernVM filesystem is an “object” addressed by its checksum: the URL used to actually download the file does not contain the full path, but a reference to the checksum. Therefore, files in different directories but with the same content point to the same URL, allowing for a more efficient caching.

• Different levels of caching. Different levels of Squid proxies can be configured: and even in cases where no Squid proxy is used, CernVM-FS data is downloaded only once and cached inside the virtual machine.

• FUSE-mountable. The CernVM filesystem is mountable and appears like a local read-only filesystem. Since it is based on FUSE, code maintenance is easier and does not need updates whenever something in the Linux kernel changes.

• Distributed directory catalogs. The file hierarchy is saved in small Sqlite3 files, which are never processed by the servers. Such files store the hierarchy up to a certain depth. Filesystem lookups occur by downloading the Sqlite3 catalogs on demand and processing them on the client.

CernVM-FS is also available as external packages, and it is not needed to use CernVM to benefit from it. In fact, CernVM-FS is currently the official method of deploying software for all LHC experiments, and the CernVM-FS packages must be installed on all Grid machines supporting such experiments.

The µCernVM operating system is entirely based on CernVM-FS: as we have seen, the whole root filesystem is a CernVM-FS filesystem, which appears to be writable thanks to an overlay.

4.4.3.3 CernVM Online

CernVM Online[^27] is a web application[^27] providing a graphical interface to create and store CernVM contexts.

Contexts are divided in modules, each one responsible for a specific section in the amiconfig context script, handled by different amiconfig plugins.

The CernVM Online interface is accessible with a CERN login, but any user with an email address can create an account. Contexts are “public” by definition, even if a user by default can only see her contexts. There is also an experimental “context marketplace” where users can share their contexts with other users:
the concept is similar to the “virtual machines marketplace”, but it is specific to CernVM. Since as we have already pointed out contextualization is the only way of customizing a \( \mu \)CernVM image, the combination of the single \( \mu \)CernVM image and a context fully defines a virtual machine.

Contexts that might contain sensitive information (such as passwords) can be encrypted with a secret key: they will be stored in an encrypted format on the server as well.

Contexts created through CernVM online can be passed to the virtual machines in two different ways. Using a special button, it is possible to retrieve the “rendered context”, which is a text file in amiconfig format. Such text file will be passed as “user-data” (see § 3.2.3.3) to the CernVM virtual machine.

It is not however always possible to supply a user-data to a virtual machine directly prior to the boot. CernVM Online gives the option to “pair” a running virtual machine to an existing context using a paradigm derived to pairing a Bluetooth device to a computer. The workflow is the following:

1. user uses the “pair” option on the CernVM Online interface, and picks a certain context;
2. a “pairing PIN” is displayed on the browser, like the one in Figure 4.7;
3. the user enters a dash followed by the pin at the login prompt of the already running virtual machine;
4. as the virtual machine correctly downloads the context from the CernVM Online website, the PIN screen disappears.

Figure 4.7: The pairing PIN screen of the CernVM Online interface: by typing it at the login screen of the running virtual machine, it will be contextualized accordingly.
Simplified interface for the VAF: abstract contexts. During the development of the Virtual Analysis Facility, some contributions were made to the CernVM Online interface. In particular, the concept of “abstract contexts” has been introduced. Normally, the user creates a virtual machine context by configuring a full and complete list of options from the CernVM Online interface.

In some cases, however, like for the Virtual Analysis Facility, one would like either to prevent users from modifying some options, or to present them an easier interface, which is no longer generic but tailored to the specific use case.

Abstract contexts are a way of choosing which plugins to enable, which ones to display and to define a specific HTML interface for choosing the options. Abstract contexts can be created by some admin users, which are presented the interface shown in Figure 4.8(a). Created abstract contexts are private by default: they can be published through the contexts list shown in Figure 4.8(b)

Whenever an abstract context is published, users will have the option to either create a new generic context, or a virtual machine context based on a certain abstract one. As we will see in § 4.4.7.6, the Virtual Analysis Facility is created by means of a simple interface representing an abstract context.

4.4.4 External components

The Virtual Analysis Facility is essentially a cluster of µCernVM virtual machines, one head node plus several computing nodes, configured as a HTCondor cluster, on top of which we submit and run PROOF on Demand jobs. The configuration is generic enough to support the submission of any kind of jobs, and not only PoD jobs: the VAF can be therefore used simply as a ready to go HTCondor cluster, even though the supported use case remains PoD.

The presence of HTCondor and PoD are however concealed to the eyes of the user, who, as we will see, is provided with a simple command line interface which hides the technicalities of the implementation (§ 4.4.8).

Instead of developing a custom solution to run PROOF on the Virtual Analysis Facility, we have decided to use two external components such as HTCondor and PoD: the rationale of this choice is described in the following sections, along with some technicalities and caveats specific to cloud computing environments. We will also see how both HTCondor and PoD are cloud aware components, following the prescriptions already mentioned in § 4.4.1: in particular, we will see how the usage of a Workload Management System to deploy PROOF adds for free the potential for automatically scaling the number of virtual machines based on the usage.
Abstract context template
Here you can define a new abstract context.

General

Plugins to enable

Plugins to display

Custom HTML for visualization

(a) Interface to create abstract contexts, showing the HTML editor with code coloring.

Your abstract context definitions

<table>
<thead>
<tr>
<th>Name</th>
<th>Public</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elastic Virtual Analysis Facility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elastic Virtual Analysis Facility v6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) List of abstract contexts: by ticking the checkbox, they will be published to everybody, making possible to create a specific context based on an existing abstract one.

Figure 4.8: CernVM Online abstract contexts creation, listing and publishing. The interface to manage abstract contexts is available to administrators only.
4.4.4.1 HTCondor

HTCondor\textsuperscript{[115]} is a popular Workload Management System undergoing active development since 1988 and currently used also by many Grid sites. µCernVM provides, at the time of writing (early 2014), HTCondor v8.0.1 preinstalled. HTCondor on µCernVM is not active by default: it can be activated and configured via the CernVM Online interface which features a special section dedicated to it.

The HTCondor architecture is composed by many daemons having different roles and running on different machines: its modular architecture allows for complex multi-tiered setups providing great scalability over WANs. The HTCondor setup we feature in the Virtual Analysis Facility is simply constituted by a head node and many identical computing nodes.

The head node is called the HTCondor central manager, and it is the only machine in the pool running the following two daemons:

- the collector, continuously collecting the global cluster status;
- the negotiator, matching waiting jobs to the appropriate slots based on the information retrieved from the collector.

Other two daemons run on every machine, including the central manager: schedd, tracking every job run by the machine, and the startd, deciding how many “resources” (or job slots) are available on the machine where it is running. They communicate with the collector on the central manager. A schema of how the daemons interoperate is presented in Figure 4.9, while a more complete description of the HTCondor architecture is available on the HTCondor website\textsuperscript{[85]}.

During our first test runs on some clouds of the HTCondor cluster provided by the Virtual Analysis Facility, we found out that HTCondor by default requires every machine being part of the cluster to have a properly registered Fully Qualified Domain Name. Unfortunately, on some clouds (notably, CERN’s OpenStack deployments by means of the EC2 interface), performing a reverse lookup from the current IP address leads to no result or a wrong response, due to some peculiar CERN DNS setup. It was therefore necessary to configure HTCondor in a way that it does not attempt to use hostnames, using always the IP addresses instead. Details of such configuration are available in § E.1.

Since the necessity to use HTCondor with IP addresses only is important outside the scope of the Virtual Analysis Facility, some modifications to the HTCondor amiconfig plugin and the corresponding CernVM Online HTCondor configuration module were made: it is now possible to generate a consistent HTCondor configuration tailored to this use case by selecting “Use IP addresses: Yes” from the web interface.

From the same interface, we also added the possibility to choose an appropriate port range used by HTCondor daemons for every communication, since
Auto-registration of nodes. A peculiar feature of the HTCondor cluster architecture is that it makes no prior assumption on the amount of resources available on the cluster—in particular, there is no “static list” of computing nodes. The way HTCondor is configured in our setup uses a “shared secret” authentication: when a new computing node becomes available, the started communicates its availability to the central manager’s collector, authenticating using the shared secret.

In other words, the computing nodes in a HTCondor cluster autoregister to the list of available nodes as soon as they start, with no other intervention needed. Likewise, if for some reason a node becomes unavailable, the central manager’s collector removes it from the list of available nodes after some time.
4.4 – The Virtual Analysis Facility

Figure 4.10: Screenshot of the HTCondor module from the CernVM Online web interface allowing for a simple and graphical configuration of a HTCondor cluster.

(self-removal). These two features combined together make a virtual cluster of HTCondor nodes cloud aware and thus appropriate for the Virtual Analysis Facility, which is designed to be very resilient when virtual machines appear and disappear over “turbulent” clouds.

The interesting fact of HTCondor’s intrinsic dynamicity is that it allows for an opportunistic resources exploitation model. Another similar WMS was used in an early prototype of the Virtual Analysis Facility: TORQUE\textsuperscript{[127]}, which however has shown not to deal very resiliently with dynamic addition and removal of resources. TORQUE relies on a nodes list which needs to be updated either manually, or automatically with the help of external tools. As we have already seen, the Torino’s Grid Tier-2 migrated to the private cloud infrastructure uses TORQUE with some hacks to add and remove nodes without editing the configuration files manually (§ 3.1.2). Other more advanced solutions for dynamically deploying TORQUE clusters on the cloud exist\textsuperscript{[20]}.

**Fair users scheduling.** While discussing static PROOF deployments (§ 4.2.1) we argued that PROOF alone does not perform any form of users scheduling. On our self-contained Virtual Analysis Facility we delegate the scheduling task to HTCondor, that features efficient users scheduling mechanisms.

Even if the jobs submitted are of a single and very simple type (PoD agents), users scheduling becomes an important point in case the Virtual Analysis Facility is used by several users at the same time, as it prevents from assigning busy resources, and because proper HTCondor algorithms allow every user to get a decent share of resources on the long run.
Waiting queue monitoring. An interesting side effect of having a batch system managing PROOF slots is that, as any batch system does, a list of enqueued jobs waiting to be executed is maintained. Under normal conditions, jobs are executed immediately after enough resources become available: whenever there are PoD jobs waiting in the queue for too long, it might mean that there are not enough free resources to process the requests.

If we are able to monitor the queue to detect such condition, we might be able to react appropriately by attaching extra resources to our system—for instance, by requesting additional Virtual Analysis Facility virtual machines. Such virtual machines will attach to the HTCondor pool automatically as soon as they are ready, given the ability of HTCondor nodes to autoregister to the central manager.

The principles of monitoring the waiting queue is extensively exploited by a tool written specifically for the Virtual Analysis Facility, called elastiq and covered in § 4.4.7: as we will see, elastiq provides automatic elasticity for the Virtual Analysis Facility by turning virtual machines on and off based on the load.

4.4.4.2 PROOF on Demand

As we have seen in § 4.2.4, PROOF on Demand (PoD) takes care of deploying personal PROOF clusters on a variety of non-dedicated environments, such as existing batch systems. PoD is even capable of running PROOF opportunistically, while having only SSH access to the remote machines.

In the Virtual Analysis Facility setup we have decided to use PoD over batch systems instead of using the much simpler SSH option: in the previous chapter, we have already seen how using HTCondor provides us with fair shares and queue monitoring, two features that we would not have so easily with a simple SSH. Moreover, HTCondor running PoD jobs ensures an improved job control and termination in case of problems.

Thus, having a dedicated batch system provides clear advantages and robustness, though it might look like an overkill. Being the Virtual Analysis Facility a ready-to-go IaaS setup, it must be noted that all the complexities of configuring a HTCondor cluster are entirely hidden: since we are providing a preconfigured system that works out of the box, we can afford to use a more complete approach rather than pod-ssh.

Within the context of the Virtual Analysis Facility, we have started deploying PoD on CernVM-FS, in the SFT repository. Our deployed versions are for CernVM, SLC and all RHEL-based operating systems. They can be found under:

\[ \text{/cvmfs/sft.cern.ch/lcg/external/PoD/} \]

The version of PoD we deploy contains a series of modifications to avoid as much as possible the usage of DNS reverse lookups, since we have already noted
that in some clouds we are not provided with consistent and reliable FQDN information. We have directly modified the original codebase by forking the PoD project on GitHub\cite{46}. Details on the new configuration options are presented in § E.2.

4.4.5 Authentication: sshcertauth

One of the problems of deploying the Virtual Analysis Facility is giving users an easy and secure way to access it, possibly by accepting their existing credentials.

From the user’s perspective, the Virtual Analysis Facility works like a Grid user interface: the user does not need any special software to be installed on her computer, except a standard SSH client. Once logged in to the Virtual Analysis Facility head node, all the software needed to run the analysis will be available from there, inside a certified environment.

The approach of using the Virtual Analysis Facility is different from the AAFs, where the user needed ROOT and PROOF installed on their computer: the extremely lightweight approach is supposed to:

- save users the time needed to configure and compile experiment’s software;
- keep software versions and updates under the control of the VAF developers, instead of asking users to perform any manual upgrade.

Even though the CernVM Online interface gives the possibility to create manually a set of users along with their passwords, authenticating SSH users using their some of their existing credentials is a convenient option for VAFs deployed by administrators for many users of the same organization.

Since the Virtual Analysis Facility is aimed to HEP users, who already have a Grid X.509 certificate and key pair used to access their Grid middleware, we created a server-side tool called sshcertauth which allows them to connect to the Virtual Analysis Facility with a small set of requirements which is likely to be already satisfied by everybody:

- a SSH client;
- a web browser;
- Grid credentials both on the filesystem and in the web browser.

In the following paragraphs we will see how such authentication method works, and how the server-side tools are configured.
4.4.5.1 Rationale

The user’s Grid “private key”, corresponding to its public key and a X.509 certificate, is a RSA key, normally stored on the filesystem in the following location:

```
~/.globus/userkey.pem
```

The key is stored in the PEM format. The most common way of performing passwordless SSH connections is to use a keypair, where:

- the public key is stored on the server in a list of authorized keys;
- the private key is kept secret on the client.

To connect to the server from the client, the user supplies her private key through the `-i` switch of the `ssh` command. By chance, RSA Grid private keys stored in PEM format are directly understood by a standard OpenSSH client, meaning that it would be theoretically possible to connect to a certain remote host by simply typing:

```
ssh -i ~/.globus/userkey.pem user@host.domain
```

where we supply the Grid’s private key to the SSH client. There are two problems behind this approach:

1. the server does not have any knowledge of which Grid users’ public keys to authorize;
2. the server does not know at all how to map a certain private key to a specific username.

SSH authentication is trivial with respect to Grid authentication: the first is simply based on checking if the supplied private key corresponds to a preregistered public key, while the latter uses an additional component (the user’s certificate) to check whether she comes from a list of trusted organizations—i.e., it is checked whether her certificate is valid and has been issued by a trusted Certification Authority (CA).

There exist a special SSH client/server called GSI-OpenSSH\[51\] (also known as gsssh) capable of using a proxy certificate created from a Grid certificate to authenticate users. However, GSI-OpenSSH needs to be present on the user’s client machine, and its installation is not trivial at all, as it depends on several Grid components.

In practice, we need an approach that works more or less like GSI-OpenSSH, but without requiring any special program to be installed on the user’s client.
All Grid users are likely to have their Grid credentials installed in the browser as well, as many Grid services are accessible through HTTPS certificate authentication. While performing HTTPS authentication, the client sends the user’s certificate as well, which contains a copy of the public part of the user’s key: this means that we could in principle use a HTTPS web server to authenticate users and supply to the SSH server the extracted public key to authorize, which was sent by the user herself when connecting to the website.

It is in practice possible to implement a two-step user authentication that works as follows.

1. The user connects via HTTPS to a certain website with certificate authentication. Transparently, her certificate and public key are sent out. If authentication succeeds, user’s certificate is checked and mapped to a certain user name, and the extracted public key is added to the list of authorized keys for that user name. The web interface communicates to the user her username for the SSH connection.

2. The user can now connect via SSH using the supplied username and the Grid private key: after obtaining the HTTPS authorization, the SSH connection can be performed many times.

While HTTPS certificate authentication can be performed by many known web servers, our sshcertauth is the PHP web application that runs behind the scenes to map certificates to users, extract and validate public keys.

### 4.4.5.2 Components

The sshcertauth web application must interact with the user through a web interface, and with the operating system to add and remove authorized public keys. Two distinct modules are used for that.

- The web application is a PHP script. It interacts with the user through a generated HTML interface, and with the web server for retrieving authentication information via environment variables. The PHP application runs with the privileges of the web server, that are usually very restricted.

The application is executed by accessing through a particular web page, and the web server providing it is responsible of performing the HTTPS certificate authentication. Purpose of the PHP script will be to try to map the user’s subject to a Unix username, and to authorize her public key.

- The other component, called “keys keeper”, is a shell script manipulating a central public keys repository in a safe way—that is, it takes care of locks
for avoiding writing the same system file at the same time, and it silently ignores public keys not added by sshcertauth.

In order to access some system directories, the script needs superuser privileges. The security model adopted uses `sudo` to grant permissions, meaning that the script should be appointed to escalate privileges by a manual addition into the sudoers file.

4.4.5.3 The web interface

As we have seen, the web interface is managed by a lightweight web application. It is the web server providing the web page taking care of the first phase of authentication: configuration hints for an Apache2 web server are provided in § E.3.1.

When the user connects to a certain HTTPS address, the browser usually asks her which personal certificate to present (see for example Figure 4.11(a)). At this point, the sshcertauth web application has not been called yet: it is all managed by the SSL engine of the specific web server we use.

Once the certificate has been presented, the web server validates it. The validation procedure is the standard X.509 authentication, which includes the following steps.

- The certificate is checked for validity: certificates not yet valid or expired ones are rejected.
- Certificate must have been issued by an authorized Certification Authority: valid CAs are usually configurable in the configuration of the web server.
- Depth of the certificate chain is also checked, and must not be deeper than a configured value. Usually a depth of 2 is sufficient, meaning that the user’s certificate has been issued by a CA, which is directly present in the list of valid CAs. A depth of 3 means that the CA present in the list has issued a certificate that in turn issued the user’s certificate, and so on.
- It is checked whether the certificate is included in the CA’s Certificate Revocation List (CRL): if it is, certificate is rejected.

In case the certificate is accepted, the web server runs the sshcertauth PHP web application. We must configure the web server in a way that it sends extended SSL authentication information to our sshcertauth application. We will need in particular:

- the full user’s certificate, containing the public key;
- the certificate’s subject for mapping it to a certain user name.
The public key is the piece of information we will need to authorize SSH connections. Such key must be extracted from the certificate, and saved in the appropriate sshd-specific format: a single line ASCII encoded string, which makes possible to use a single file to store multiple authorized public keys.

The sshcertauth application relies on openssl to perform the extraction and conversion. The procedure is detailed in § E.3.3.

The application then checks if it is possible to map the certificate’s subject to a Unix user name. In case the mapping is successful, the “keys keeper” application is called to add the public key to the list of valid ones for that user. Users mapping is based on plugins, and it is described in § 4.4.5.4, while the “keys keeper” application is described in § 4.4.5.5.

SSH public keys have a format which allows the addition of arbitrary text comments at the end of the key. We use such field to append a special string containing the “expiration date” of the added key. The string starts with the text “Valid until:”, like the following example:

```
ssh-rsa AAAA...0tsC7 Valid until: Feb 05 2014 05:36:36 +0100
```

This string is important for the keys keeper application, because it identifies which keys were added by sshcertauth (other keys are simply ignored), and when to eliminate them from the authorized keys list.

On a successful authentication, the user is presented with a summary web page like the one in Figure 4.11(b). The web page tells user the information she needs to know to connect via SSH:

- the assigned user name;
- the expiration date of the authorized key.

### 4.4.5.4 Users mapping plugins

As we have mentioned, a plugins architecture is used to map user’s certificate subjects to Unix usernames. Even if the web server’s authentication succeeds, because the certificate is considered valid, the mapping might be unsuccessful. While the first authentication step is managed by the web server, the second step is controlled by sshcertauth.

Each mapping plugin takes as input the certificate’s subject string, such as:

```
/DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=dberzano/
```

and upon success return a Unix username and the authentication validity.
(a) The web server requires a certificate to log in: the browser asks you which one to supply. In some cases, more than one choice is possible.

(b) User has been successfully authenticated, and mapped to a certain Unix user: information on the expiration of the authentication is shown.

**Figure 4.11:** Virtual Analysis Facility certificate authentication handled by sshcertauth, using a web browser as client. This step temporarily authorizes users to connect via SSH.
Two authentication plugins are provided by default: the ALICE LDAP plugin, returning actual ALICE usernames for the users part of the ALICE VO, and a generic “pool accounts” plugin, assigning predefined pool accounts in a round-robin fashion to all users having a valid Grid certificate.

**The ALICE LDAP plugin.** This plugin queries the ALICE LDAP server with the user’s subject to return the corresponding username. Details on such LDAP server are available in § C.1. This plugin actually checks if a user is part of the ALICE VO: when connecting with a valid Grid certificate which is not registered, authentication will fail.

**The pool accounts plugin.** Every user with a certificate belonging to a recognized CA will be able to connect: no VO check is performed. Dummy pool accounts are assigned to each subject: mappings are maintained in order to reassign the same pool user to the same certificate subject.

Since many instances of sshcertauth might try to access the same mappings file at the same time, access has been made thread-safe by means of `flock()`.

This plugin has configuration parameters detailed in § E.3.5.

**Other plugins.** It is foreseen to implement more additional plugins:

- a plugin to manually mapping certain subjects to users;
- specific plugins for other LHC experiments.

The procedure to write a custom plugin is explained in § E.3.4.

### 4.4.5.5 The keys keeper

The public keys management part is separated from the web interface for security reasons. The keys keeper is an application not meant to be used manually, which must be invoked with superuser privileges: its purpose is to add an authorized public key for a certain user, and to remove expired ones.

SSH public keys are usually stored, in the default sshd configuration, in a single file which resides in the home directory of each user:

```bash
~/.ssh/authorized_keys
```

The expected configuration for the keys keeper however does not allow users to manage their own public keys. Public keys for all users are stored in a directory managed by the root user: such directory contains many files, one for each
user, each one containing one or more authorized public keys. Details on the configuration options are available in § E.3.2.

When the web interface has authenticated the user, it invokes the keys keeper to add its public key to the list with a certain expiration date. The keys keeper checks whether the same key already exists in the public keys database: if it does, it just updates the expiration date, instead of duplicating entries. If it does not exist, the full key is simply added. If the file for a certain user does not exist yet, it is created with proper permissions: the file must belong to the root user, and it can be written by root only.

We have already mentioned that the authorization for connecting to SSH is temporary: this implies a background mechanism that periodically examines all the stored keys, and removes the expired ones. The second operation of the keys keeper is exactly that.

- All the key files are checked for expired keys: only keys containing the “Valid until:” comment string are considered, as we have seen in § 4.4.5.3. The string indicates that the key has been stored by sshcertauth: other keys are considered as manually added, and they will be left there.

- If a key has expired, it is eliminated from the file.

- If a certain user file is left with no keys, the file is eliminated.

Such periodic purge of old keys can be triggered by periodically invoking the script through crontab.

The keys keeper synchronizes multiple accesses to the same files by locking the files. Since there is no shell interface to acquire actual locks, a “lock directory” is created for every file. The reason why a lock directory is used instead of a lock file is that the mkdir command uses a syscall which, on most local Unix filesystems, is guaranteed to be atomic, preventing other instances to bypass the lock while it is being created\[^{69}\].

Directory creation is atomic on all ext filesystems and XFS, and it is not atomic on NFS and other network filesystems.

The advantage of the mkdir command is that it fails if the directory already exists: if we had decided to use the touch command for using files instead, we would have had to check for the lock file existence separately, making the whole operation non-atomic.

### 4.4.5.6 Restricting access to some users

Since the sshcertauth authentication mechanism is a mere wrapper that relies on solid system tools, users access can be controlled, for instance, by manipulating the configuration of sshd. If we are in a situation where we use some mapping
plugin that returns a specific username for a certain user, we can specify a list of allowed ones in the sshd_config file like this:

```
AllowUsers user1 user2... userN
```

### 4.4.6 PROOF dynamic workers

As we have already mentioned in § 4.2.3, when a PROOF session is opened, all the workers available in that moment are initialized and considered for processing: it was not possible for newly available workers to join and offload a currently running analysis.

Although the lack of this feature was never a problem, due to the static nature of most PROOF deployments, with the spread of PoD-based dynamic and opportunistic deployments it brings real advantages, improving both the user’s workflow, as we will see in § 4.4.6.2) and the overall time-to-results, as shown by a detailed calculation reported in § 4.4.9.1.

When PoD discovers the existence of new workers, the information must be propagated to allow using the new resources when scheduling processing: the full chain of propagation is explained in § 4.4.6.1.

Being the PROOF connection layer based on XRootD, the xproofd component required little modification (§ 4.4.6.3): XRootD’s network architecture already allows the master to accept new connections from new xproofd servers.

Most of the work has been implemented in the PROOF master component, *i.e.* the ROOT session spawned by the xproofd daemon running on the master: as we will see in § 4.4.6.4, the PROOF master must retrieve the list of workers from xproofd, and it must initialize the new workers with a consistent environment.

When the PROOF master has individuated new workers, it must communicate them to the packetizer currently in use. The process of making the packetizers interact with the dynamic addition of workers is illustrated in § 4.4.6.5.

Finally, a description of the capabilities of PROOF to deal with failures in existing workers is covered in § 4.4.6.6.

#### 4.4.6.1 Chain of propagation of the new workers

PROOF on Demand is used to allow the usage of PROOF without the need for static deployments. In particular, PoD permits running PROOF over a Workload Management System: this is exactly the setup provided by the Virtual Analysis Facility (§ 4.4.4.2).

When submitting jobs on a Workload Management System they become available gradually: if such jobs are PoD workers, they will connect back to the PoD master and added to the list of available workers for PoD.
Thanks to the dynamic workers, there is a “chain of propagation” of the new workers information that starts from PoD and goes up to the packetizer.

1. **PoD master server.** It writes the new information to the so-called `proof.conf` file, *i.e.* the file where PROOF will read the available workers. The technique of updating this file to make the information available to PROOF is also described in § 4.3.7.4.

2. **xproofd.** The xproofd daemon, independently from PoD, already supports the automatic registration of new xproofd servers. This aspect is important: the `proof.conf` file is *not* read by xproofd, but by the PROOF master, and xproofd does not need it to bookkeep the list of available nodes.

3. **The PROOF master.** Opening a new PROOF session is equivalent to connecting to a new PROOF master. The PROOF master reads the `proof.conf` file at startup, and with the “dynamic workers” option it also polls for it periodically. The PROOF master does not read the file directly: instead, it asks the xproofd daemon to provide the file. It must be again noted how, despite the xproofd daemon provides the `proof.conf` file, the daemon itself does *not* read it for maintaining the connections.

4. **The packetizer.** The packetizer is the PROOF scheduler: in order to work, it must know what are the available resources. The packetizer does *not* share the list of resources with the PROOF master: instead, it keeps an independent list internally, which must then be synchronized. With dynamic workers, a generic interface for adding workers to the packetizer is used, where specific implementations depend on the packetizer itself.

Prior to the introduction of the dynamic workers, the chain of propagation stopped right after xproofd: neither the PROOF master, nor the packetizer, got to know the existence of the new workers.

### 4.4.6.2 User’s workflow

The two most important use cases where the dynamic workers addition turns out to be useful are:

- the Grid: the ATLAS experiment is testing a PoD plugin for the PanDA middleware allowing the submission of PROOF workers as Grid jobs;
- the Virtual Analysis Facility:
amount of time. Usually, not all of the available workers will be available, but the user never knows exactly how many workers she will get and when to stop waiting.

For what concerns the Virtual Analysis Facility, workers are run by virtual machines: the availability of the requested workers often depends on the deployment time of the newly requested virtual machines. Also in this case it is uncertain how many workers will be actually made available to the user, and when they will be available.

Due to these uncertainties, the user's workflow, prior to the introduction of the dynamic workers, was the following.

1. User requests a certain number of workers.
2. User monitors the number of workers obtained until satisfied.
3. A PROOF session is opened, and the analysis is started with the workers obtained so far.
4. If during the analysis new workers become available, they will not be used: instead, they will be available from the next PROOF connection.

To give an idea of the usual wait time, a plot showing the ramp-up of the requested resources on real Grid sites is shown in the context of pull and push schedulers comparison in § 4.4.9.1, in Figure 4.14. From that data we have calculated how much time is required to get a certain number of workers, having requested 100 workers: results are shown in Table 4.3.

<table>
<thead>
<tr>
<th>Grid site</th>
<th>Time for N workers [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td>CERN</td>
<td>133.0</td>
</tr>
<tr>
<td>CNAF</td>
<td>138.6</td>
</tr>
<tr>
<td>Roma 1</td>
<td>806.4</td>
</tr>
<tr>
<td>Napoli</td>
<td>520.2</td>
</tr>
<tr>
<td>Milano</td>
<td>868.9</td>
</tr>
</tbody>
</table>

Table 4.3: Time, in seconds, required to get respectively 25, 50 and 75 PROOF workers after requesting 100 workers. The table shows how even to get only half of the workers the wait time is greater than 5 minutes, while in the time span of one hour some sites never assign at least 75 workers: the large waiting time and the uncertainty are enough to justify the usage of the PROOF dynamic workers. Reference data is shown in Figure 4.14.

From the results it becomes evident the advantage of using the dynamic workers feature: depending on the Grid site you end up in, the amount of time re-
quired to get even only half of the requested workers varies from around 5 minutes of a “big” site (CERN, in this case) to more than half an hour for a small site (like the Napoli and Milano sites from our data).

With the dynamic workers, users do not need to care how much resources they will get and when: the workflow changes in a way that the analysis can be started when the first worker becomes available, while others will seamlessly join the existing process. Compared to the static situation, new workers are available immediately, and not only after restarting the analysis: the user’s time spent in waiting a “certain” number of workers is now used in computing. A calculation on the time saved by actively using such waiting period is presented in § 4.4.9.1.

4.4.6.3 Updating workers list in xproofd

The xproofd daemon, as we have seen, is in charge of reading the proof.conf resources file and returning the parsed results to the PROOF master.

This function, which is part of the xproofd scheduler module, has been designed to correctly return the workers the first time it is invoked, and to return nothing upon subsequent calls, with an error code communicating that the PROOF master should use the workers already assigned.

A modification has been made to the function in order to always return the full list of workers when requested, in case dynamic workers are on. In order to avoid problems and incompatibilities, it is sufficient to turn the dynamic workers off to restore the former behavior.

There is no further modification performed to the xproofd daemon: as we have already noted, the XRootD architecture upon which xproofd is based already provides dynamic connections and disconnections of xproofd servers, and the proof.conf file is not used by xproofd, but only parsed and served to PROOF.

4.4.6.4 Configuring workers from the PROOF master

The PROOF master server is the part that required consistent modifications. The modifications were added in a way that, in case any problem occurs, they can be simply turned off by disabling the dynamic workers: in other words, they were kept separated as much as possible, instead of being convoluted with the rest of the code.

While the problem of replicating a consistent environment via library loading and packages enabling has been pretty simple to deal with, some work has been done to find the correct way of synchronously updating the list of nodes. Moreover, some ancillary work has been done to fix the wrong behavior of some functions, which never showed up before setting up the dynamic workers feature.
**Preemptive main messages collection loop.** The PROOF server's architecture is based on sending and receiving messages. As we have seen, the transport layer is handled by xproofd, which manages the message queue.

The two functions in charge of sending and receiving messages are:

- `TProof::Broadcast()`, enqueuing a message;
- `TProof::Collect()`, processing the received message from the queue.

Both functions support sending and receiving from a certain list or class of workers only, if desired. For instance, during processing, the `Collect()` function only expects “next packet” requests from the list of “active” workers, *i.e.* workers actually involved in the analysis.

While the `Broadcast()` function is used to send a specific message, it is not possible to choose which message to process while retrieving it with `Collect()`: the next one available is processed accordingly. This means that if you forgot to process the response for a certain message, such response will be enqueued, and any unrelated call to `Collect()` will retrieve it before anything else.

The `Collect()` function does not run in background, to avoid thread safety issues with many other ROOT components: instead, it is called every time a `Broadcast()` issues a request that implies a response from the remote server. A `Collect()` function launched while no messages are supposed to be sent by an active worker would block PROOF indefinitely.

Due to the synchronous architecture of such messaging model, polling for new workers cannot be done in a separate thread: instead, a current operation must be interrupted at certain points to periodically yield control to the workers poller.

The majority of PROOF messages require a single `Collect()` that exits immediately. The PROOF processing main loop is one “big” `Collect()` call which lasts until the end of the analysis: since we would like to add workers while such operation is in progress, the `Collect()` function has been made “preemptive”, interrupting its selection periodically to look for new workers.

In case new workers are found, they are initialized together and in parallel. Once again, due to the synchronous nature of PROOF’s loops, while initialization of the new is performed, other operations are put on hold for a moment. The reason is that the initialization implies a series of `Broadcast()` and `Collect()` directed to the new workers only, and we don’t want the main process loop to intercept initialization messages that should be processed separately. In any case, the cost in terms of performances is small.

**Polling for new workers.** Polling for new workers from the PROOF server implies querying the xproofd daemon: we have already seen that it has been mod-
ified in order to always return the list of workers, even when § 4.4.6.3 it has already been sent, in case we are in dynamic workers mode.

Each worker is uniquely identified by its ordinal number: it is a numeric identifier in the form:

\[
\text{<master\_ordinal>.<worker\_ordinal>}
\]

The <master\_ordinal> identifies the master of reference in a multi-tier architecture. In practice, since we are always referring to a single-tiered architecture, our ordinals will always be in the form:

\[
0.<\text{worker\_ordinal}>
\]

with one degree of freedom only. The <worker\_ordinal> is a progressive number.

Ordinals are assigned by the xproofd daemon when reading the resources file. While polling for new workers, the full list of workers currently active is returned: the PROOF server checks if some of the workers are new by using the ordinal as unique identifier.

If new workers are found, they are initialized properly. The initialization phase implies other Collect() calls, but we are currently working inside another main Collect(): to avoid infinite recursion, inner Collect() calls are told not to poll for new workers.

**Asymmetries in broadcasting and collecting messages.** While using PROOF in dynamic workers mode, we found a series of small asymmetries in the way some initialization messages were sent and collected: in some cases, while we correctly performed Broadcast() on the specified set of workers only, the function Collect() happened on all active workers.

In the dynamic workers context, while initializing new workers, we keep messages requesting new packets from other workers on hold. Initialization happens inside a main loop represented by an outer big Collect(), and we want all the inner calls to the same functions to:

- be directed to the new workers only;
- not polling themselves for new workers to avoid infinite recursion.

If we directed the Collect() calls to all the active workers, we would resume packet processing as those requests are the next in the queue, and never initialize the new workers.

Asymmetries in send and receive were never spotted before, as in the standard “static” PROOF no deferred initialization of workers ever occurred.
Deferred initialization of workers. In the dynamic workers mode, initialization occurs when the analysis is started, and not right after opening the connection to the master. This means that PROOF starts in the so-called “sequential mode”, and “goes parallel” afterwards.

The initialization process consists in starting distributed ROOT sessions whose environment is consistent. The following operations are executed to prepare the environment of the new workers: each operation is executed in parallel on all workers before proceeding to the next one.

- **Activation.** Sockets for the new workers are added in the list of the “active” sockets.
- **Set working directory.** A working directory for each new worker is prepared. The directory is local to each worker.
- **Find workers on the same filesystem.** If more than one worker is created on the same node, or if several workers share the same filesystem (which can also be remote), this step groups them. This information will be used to send packages (PARfiles) only once for all workers sharing the same filesystem.
- **Upload and enable packages.** The list of packages currently enabled is retrieved: such packages are uploaded and enabled on the new workers.
- **Load macros.** Current macros are loaded on the new workers.
- **Add library and include paths.** Some paths to look for libraries and header files are set identically on all workers based on the settings of the PROOF master.
- **Send selector.** The current TSelector, i.e. the class responsible of analyzing data, is finally sent to all new workers by the PROOF player.

### 4.4.6.5 Adding workers to the packetizer

PROOF’s scheduler, the packetizer, is the last element in the chain that needs to know the new workers, as it may maintain an independent list of workers instead of sharing it with the PROOF server, in order to store additional information.

Every packetizer inherits from the TVirtualPacketizer base class, defining the base structure. In particular, they all implement the GetNextPacket() function, triggering an assignment of the workload.

The virtual interface has been adapted in order to include a new function, used to communicate new workers to the packetizer: the AddWorkers() function. The PROOF player, managing the event loop, will invoke such function after sending the current selector to the new workers.
The \texttt{AddWorkers()} function has been implemented in the “unit packetizer”, for non data-driven analyses, and in the adaptive packetizer already illustrated in § 4.1.4. Other packetizers not implementing the function will just continue using only the existing workers and ignore the new ones, without failing abruptly.

### 4.4.6.6 Removal of active workers

PROOF components do not have functions to “remove” workers from the list: instead, they all have (including the packetizer) a function to mark existing workers as “bad”, in order to skip them.

It might happen that, due to bad user code, some analyses crash, leaving some workers down. Such workers are marked as bad, and PROOF continues the analysis using the other workers.

Data processed by the removed worker is lost, and it will not be reprocessed. This is a precise design decision: in the majority of cases, workers fail because certain input data has triggered an unwanted behavior from bad user code. Reassigning such data to other workers might lead to a chain reaction of workers failing for the same reason.

Instead, debug information is reported to help the user fixing the error that led to the crash.

### 4.4.7 Elasticity embedded in IaaS: elastiq

A central component of the Virtual Analysis Facility is a daemon, running on the head node, capable of automatically scaling up and down the cluster based on the current load. The daemon is written in Python and it is called elastiq, as in “elastic queue”. The source code is available on GitHub\cite{elastiq}.

elastiq works in a way that it stays completely invisible to the eyes of the user (§ 4.4.7.1): it interacts with a Workload Management System on one side, and with the cloud controller on the other (§ 4.4.7.2).

Since the daemon is critical to the Virtual Analysis Facility, it has been designed in order to be as robust as possible: the daemon’s robustness comes in great part from its simplicity (§ 4.4.7.3). Since the daemon is capable of scaling up (§ 4.4.7.4) and down (§ 4.4.7.5) the virtual cluster by starting and shutting down virtual machines, by defining proper minimum quotas it can be used to deploy an entire virtual cluster by just running the head node: elastiq will take care of the rest (§ 4.4.7.6). How the daemon deals with problems due to the cloud infrastructure is described in § 4.4.7.7.

At the end of the section we will present a brief discussion on different elastic clusters competing for the same resources (§ 4.4.7.8), and a list of future development and integration plans (§ 4.4.7.9).
4.4.7.1 Rationale: user transparency

The primary reason why we developed the elastiq daemon is to elastically scale a cluster without any user awareness or intervention.

In the context of the Virtual Analysis Facility, PROOF on Demand (PoD) is used to run PROOF dynamically: instead of using the simple pod-ssh we have decided to use the classic PoD over batch systems (§ 4.4.4.2), and we have configured the Virtual Analysis Facility to be essentially a HTCondor cluster (§ 4.4.4.1). Apart from the benefit of scheduling different users, the presence of a Workload Management System allows us to:

- keep track of the amount of user’s requests by looking at the queue;
- monitor the cluster’s status to see how many jobs are running where.

In principle we would like to keep the user’s workflow as simple as possible, and at the same time we would like to avoid the waste of resources. With the elastiq daemon, we can do both: when properly configured, the user simply submits PoD jobs. If there are not enough available resources, elastiq notices that there are jobs waiting for some time, and requests new virtual machines. All the user does is submitting PoD jobs, and new virtual machines are started in the background: thanks to the auto-registration of nodes of HTCondor (§ 4.4.4.1), as soon as the new nodes are up they start fetching the waiting jobs.

When the user finishes working, she stops her pod-server and the booked resources are freed. Thanks to the dynamic architecture of PoD, if the user forgets to turn off the pod-server, it does that automatically after some time of inactivity. elastiq spots the inactive virtual machines, i.e. virtual machines not running any job, and it shuts them down. Again, thanks to the HTCondor architecture, it is not needed to remove the node from the list of HTCondor resources, as it will do the cleanup automatically after some time.

The user does not even know the number of virtual machines, and it does not interact directly either with them, or with the cloud controller: the only thing that the user sees is a job submission system, where its resources are provided by disposable, “unimportant” virtual machines, in a model commonly referred to as “cattle computing”.

It must be noted that, despite the elastiq daemon has been developed within the context of the Virtual Analysis Facility, its purpose is absolutely generic and not related to either PoD or PROOF: elastiq does not look into the type of the jobs, it just reacts when there are generic jobs waiting.

It does not even need to run inside a virtual machine, or from the head node: in fact, elastiq can run on any machine which has access to the status of the HTCondor cluster, and which can communicate with the cloud controller.
elastiq ignores nodes in the cluster which are not virtual machines: this means that it is in principle possible to have a physical HTCondor cluster, made of some static nodes, with a virtual elastic extension on the cloud, capable of absorbing peak times. HTCondor and elastiq together work, in this case, as a way of bridging physical nodes and virtual nodes—all of this without the user ever noticing or dealing with the complications of the deployment.

4.4.7.2 Cloud communication layer

In order to launch, shut down and list virtual machines, elastiq needs to communicate with the cloud. Currently, there are many cloud controllers, each one having its specific cloud API: it is however complicated to support each one of them.

There are solutions like Deltacloud\textsuperscript{111} that aim to provide an uniform frontend communicating with multiple cloud backends, basically allowing to write an application which supports the Deltacloud API and that can be migrated to multiple clouds by just changing the backend.

We have decided not to use Deltacloud for two main reasons.

- Deltacloud is not a library: instead, it is a small daemon that interacts with your application using a RESTful API. This means having two different daemons (elastiq and Deltacloud) running for a very simple task.

- Deltacloud aims to provide a common interface for a very large number of features and API calls, while what we need to do is in fact extremely simple: deploy, shut down and list. Although very complete, using Deltacloud would be an overkill for our use case.

We have defined a minimal set of requirements for finding an appropriate library to communicate with clouds: such requirements are parameters that must be embedded in the node running elastiq—with the Virtual Analysis Facility we will do that through the contextualization of the head node.

- User credentials and API endpoint. All clouds expose a RESTful API having a certain endpoint URL. Credentials are usually a pair constituted by a username and a password.

- Reference to the image. We are using \( \mu \)CernVM virtual machines (§ 4.4.3.1), which must be stored in the image repository of our cloud. The “reference” to the image is a cloud-dependent identifier pointing to the correct image of \( \mu \)CernVM. We will use such reference to know which virtual machine to start on demand.
4.4 – The Virtual Analysis Facility

- **Flavour**: All clouds have a certain number of profiles defining the amount of resources (RAM, disk, CPU) to assign to a virtual machine, which go under the name of “flavours” (§ 3.2.3.2). When we request a new virtual machine, we will need to tell its flavour to the cloud controller.

- **Contextualization**: Since μCernVM virtual machines are entirely configured via contextualization, we need a contextualization file for the new virtual machines, and a cloud API that supports contextualization.

We have taken into consideration the libcloud\(^{[67]}\) library as an alternative to Deltacloud. Unlike Deltacloud, libcloud is a Python library, which we can easily import into elastiq. The libcloud interface supports many backends, however it is not possible to provide the contextualization in a generic way for every one of them.

We have noticed that all many popular clouds support, sometimes out of the box, an implementation of the EC2 API, along with their one. In our tests we have used the following clouds:

- **Amazon**: EC2 is the native interface;
- **OpenStack**: has an EC2 backend;
- **CloudStack**: has an EC2 backend;
- **OpenNebula**: it is shipped with an external daemon, called econe-server, responsible of translating EC2 requests to OCA—we have configured it in Torino’s Tier-2, see § 3.2.3.

Moreover, Deltacloud can be configured to support the EC2 interface as frontend to any backend, for those clouds not supporting EC2 directly.

The EC2 interface is simple and popular, and it is considered a de-facto standard. Given that, we have decided to use the EC2 protocol for elastiq: the solid boto\(^{[21]}\) Python library is the interface of our choice for accessing EC2 services. Configuration details are provided in § E.4.

### 4.4.7.3 Design: robustness and simplicity

elastiq is a critical component to the Virtual Analysis Facility, as it is responsible for resources provisioning: it is therefore very important that the daemon is stable and robust.

Since elastiq is meant to manage relatively small clusters (up to a hundred of virtual machines is a reasonable limit) and to do very little operations, Python was the language that we have chosen: since there are no performance critical parts
in elastiq, the simplicity and amount of libraries of Python has been preferred to the performances of a compiled language. elastiq requires Python 2.7 to run.

The key feature of elastiq is that it is completely stateless and fault tolerant. Let’s see in detail what this means.

First of all, elastiq does not keep the state of the cluster in any internal structure: the state of the queue is given by the condor_q command, and the state of the cluster by the condor_status command. Since there is already HTCondor maintaining a stateful overview of the infrastructure, it is not needed to replicate such functionalities.

Likewise, the state of the virtual machines is maintained by the cloud controller, queried via the EC2 api (§ 4.4.7.2). elastiq performs a query to HTCondor and another EC2 query to find which nodes are virtual, and which virtual machines are HTCondor nodes.

The major consequence of a stateless design is that elastiq can be restarted without the fear of losing track of how many and which virtual machines were automatically launched: in other words, elastiq does not keep the correspondence between HTCondor nodes and EC2 virtual machines, but it finds it out whenever it’s needed.

elastiq is fault tolerant in a sense that it “tolerates” faults and errors: if a query to HTCondor or to EC2 fails, it will appropriately retry without taking any other action. When a new virtual machine is requested, elastiq does not keep track of the virtual machine request and does not wait for it to be up: elastiq forgets about the query as soon as it has performed it. From that point on, when the virtual machine is up, HTCondor will take proper actions to register the node and offload the cluster: if the virtual machine for some reason fails or it is never executed, elastiq will not know. Instead, it will simply notice that there are still jobs waiting in the queue and will issue a new virtual machine request accordingly.

As we will see in § 4.4.7.4, the number of virtual machines requested is kept in memory for some time to avoid issuing multiple requests for the same waiting jobs: however elastiq does not keep track of every single request, only the overall number of virtual machines requested is kept for the time window of the expected deployment time.

**The event queue.** elastiq has a main loop periodically executing actions in a certain order. The loop is constituted by an event queue where each operation is supposed to run at a certain time: after the operation has been executed, it is removed by the event queue.

At the beginning, only two operations are inserted in the event queue: a query to the HTCondor queue, and a query to the HTCondor cluster status. Each of those two operations reschedules itself to the queue after a certain period.

If two or more events are to be executed at the same time, they are executed
in their insertion order, as in a FIFO list.

The timed event queue is useful to execute many different periodic operations with different periods each: in § 4.4.7.4 we will see how it is used to give elastiq an estimate on how requested virtual machines are supposed to be up and running.

### 4.4.7.4 Scale up workflow

In this section we describe the behaviour of elastiq when scaling up the cluster. The following four factors are taken into account to decide how to scale up the cluster:

- a configurable *minimum quota* of virtual machines that must be running, even when there is no job waiting;
- a configurable threshold of jobs waiting for at least a configurable time in the HTCondor queue;
- the number of virtual machines “allegedly” running: that is, the number of virtual machines requested which are most likely being deployed or booting but which are not yet available;
- a configurable *maximum quota* of running virtual machines.

elastiq periodically checks if the minimum quota of virtual machines is satisfied, disregarding the number of waiting jobs. If not, it launches a certain number of virtual machines accordingly. The minimum quota can be used for two reasons:

- deploy an entire cluster by deploying only the head node, as we will see in § 4.4.7.6;
- improve the promptness of the cluster by guaranteeing that there is always a minimum amount of resources which are not dynamically handled and which are immediately available.

The main idea of elastiq is that it launches virtual machines when there are jobs waiting. elastiq checks the HTCondor queue periodically, and takes note of the number of jobs waiting to be executed. If the number of jobs waiting is above a certain threshold for more than a certain number of seconds, elastiq does the following actions.

- Given in the configuration a parameter representing the “job capacity” of each virtual machine (*i.e.* the number of cores, since our jobs are all monocore), it calculates how many virtual machines satisfy the request. The number is rounded up.
• By taking into account the number of virtual machines “allegedly” running plus the number of total virtual machines actually running (also the ones not part of the HTCondor cluster), it honors the maximum quota. The reason why the maximum quota considers “other” virtual machines is that we would like to prevent misconfigured HTCondor virtual nodes to start and overflow the cloud without ever joining the HTCondor cluster.

• New virtual machines are started: there is a retry if a request fails.

• For each virtual machine requested, the number of virtual machines “allegedly” running is increased: using the expected deployment time given in the configuration, an event per virtual machine is added to the event queue to decrease that number after a time equal to the expected deployment time.

The mechanism of taking into account the number of allegedly running virtual machines has been introduced to avoid the following situation:

• there are some jobs waiting;

• new virtual machines are started;

• the “waiting threshold” is less than the deployment time: new virtual machines are requested before waiting for the others to come up;

• too many virtual machines are finally running;

• some of them are idle and they get shut down after a while.

In practice by taking into account the number of virtual machines requested we can prevent useless boot and shut down cycles by maintaining an essentially stateless nature.

It must be noted that the maximum quota is a “soft” quota: each tenant on the cloud has proper “hard” quotas, but if a certain tenant wants to run other virtual machines along the Virtual Analysis Facility it is possible to limit it via the “soft” maximum quota.

A simplified schema on how the “scale up” works is represented on the right side of Figure 4.12.

4.4.7.5 Scale down workflow

elastiq periodically checks the status of the EC2 virtual machines belonging to the HTCondor cluster to find idle ones, i.e. virtual machines with no HTCondor jobs running. Only HTCondor nodes which are virtual machines are considered.
4.4 – THE VIRTUAL ANALYSIS FACILITY

Figure 4.12: Simplified schema showing how elastiq works: on the left, the queue is monitored and new virtual machines are started upon waiting jobs. On the right, idle virtual machines are turned off.

If a virtual HTCondor node has been idle for a certain configurable amount of time, a shutdown request is issued to the cloud controller via EC2. Minimum quota is taken into account during this operation.

The workflow is the following.

- HTCondor virtual machines are checked: a list of those which have been idle for some time, i.e. a “shutdown list”, is prepared.

- Minimum quota is honored by removing some virtual machines from the shutdown list, if necessary.

- Virtual machines are removed in random order.

The reason why we remove virtual machines in a random order, rather than in HTCondor or EC2 order, is that we want all virtual machines at some point to be replaced by new ones. This method is useful, for instance, for automatic rolling updates: since every time we create a new μCernVM virtual machine it automatically gets the most recent snapshot of the root filesystem from CernVM-FS (§ 4.4.3.1), in a dynamic environment we will end up with worker nodes which are always reasonably fresh and updated.

A simplified version of the “scale down” schema is represented in Figure 4.12.

4.4.7.6 IaaS/PaaS deployment in one click

In the simplest topology of a IaaS cluster, we deal with a single head node and multiple cluster workers. The elastiq model completely changes the deployment workflow for whoever launches the Virtual Analysis Facility, or, to be more generic, any μCernVM cluster with HTCondor and elastiq.

In this model it is not the user, but elastiq, taking care of deploying the cluster workers. The user’s workflow is then reduced to:
• creating a head node context in § 4.4.3.3;

• typing appropriate EC2 credentials into the abstract context for the Virtual Analysis Facility on CernVM Online: Figure 4.13.

![Figure 4.13: CernVM Online interface to create a Virtual Analysis Facility based on HTCondor: by typing the appropriate EC2 credentials, elastiq takes care of deploying the worker nodes, while the user only runs the head node instance.](image)

We will see in § 4.4.7.9 how we are planning to simplify furthermore the CernVM Online web interface with a dedicated section for creating dynamic clusters.

Strictly speaking, we are deploying a cluster which exposes an interface making essentially transparent the architecture from the tasks of either:

• submitting batch jobs;

• submitting PoD workers (see § 4.4.8).

The user never needs to see, in principle, what lies behind the batch system (or PoD) interface. Since this architecture effectively allows users to:
• create an application based on a certain framework;
• run it in parallel on a virtual cluster;

this model is closer to Platform as a Service (PaaS) than to IaaS, since it provides
the user a clustered and distributed platform to run her usual physics analysis.

It must be noted that the Virtual Analysis Facility also supports static deployments, which are identical to the dynamic ones but without the elastiq daemon: in such deployments users have to take care themselves of creating and destroying
the workers.

Due to the easiness of creation of an entire cluster in a few clicks, the presence
of elastiq in the deployment of the Virtual Analysis Facility is an actual implementa-
tion of the computing model commonly referred to as “Tier-3 (or Analysis Facility) in a box”, where each user can have her personal analysis cluster on demand, which is automatically shut down when unused.

4.4.7.7 Fault tolerance

The elastiq daemon deals smoothly with two types of deployment errors that
might occur:
• virtual machines going to “error” state;
• virtual machines not joining the cluster in a timely fashion.

The first type of error occurs when the cloud infrastructure has problems that
either prevent a virtual machine to be deployed, or when the cloud controller
loses track of the virtual machine (for instance if the hypervisor running it goes
down abruptly). In particular, on some cloud infrastructures (like CERN’s Open-
Stack) requesting too many virtual machines in a certain time window might lead
to the error state for some of them.

The daemon keeps track of the instance IDs of the virtual machines that it
launched: if one of the owned virtual machines is in error state, elastiq terminates
it (even if the virtual machine is down it must be cleaned up in order to free a
quota slot) and requests for a new one. The list of owned instance IDs is saved to
a state file, re-read if elastiq needs to be restarted for some reason.

In the second case, elastiq checks if a virtual machine has already joined the
batch cluster within a configured time span. If it has not joined the cluster yet,
elastiq assumes that a problem has occurred, and the virtual machine is termi-
nated. In this case, a replacement virtual machine is not requested: this is be-
cause a “late” virtual machine is supposedly due to a misconfiguration and not
an infrastructure glitch (for instance it might be stuck at the bootloader phase).
In any case, the “scale up” mechanism (§ 4.4.7.4) will request a new virtual ma-
chine, if needed, as soon as possible.
4.4.7.8 Elasticity

The computing model we are proposing abstracts the concept of the Grid to an higher level including the virtual machines: since our common physics use cases are still a large job split into many independent subjobs (even in the case of PoD, where the only difference is that they are not independent anymore), the idea of monitoring a batch system to automatically deploy virtual machines seamlessly integrates with common workflows.

With appropriate tools, if we run Grid pilot jobs on the batch system instead of ordinary jobs, we can even have to some extent a disposable and elastic Grid site on demand.

We claim that our computing model is an “elastic cloud”. This essentially means that:

- there is a layer of abstraction for the resources allowing to run virtual machines;
- the number of resources at our disposal is large enough;
- more importantly, there is a sustained rate of virtual machines shutting down.

The last point is the most important: an elastic model is forced to keep new virtual machines waiting in a queue until some “virtual machine slots” are freed by other dying virtual machines. In other words, in such computing model we must guarantee that resources leave space for other resources periodically, or again: an elastic IaaS cluster works well in a large enough environment where it competes with many similar other elastic clusters.

In the computing centre in Torino an hybrid model has been implemented: although the Grid site (§ 3.1) and many virtual farms (§ 3.2) are “anelastic” farms made of long-living virtual machines, there is some space reserved to the elastic use cases.

There are similar tools providing elastic deployment of virtual machines controlled by job submission on a HTCondor cluster: in particular, there is Cloud Scheduler\textsuperscript{[11]}. Cloud Scheduler is more complex than elastiq, and generally the setups have a single node of submission for many different use cases: Cloud Scheduler monitors the queue and reads the specifications of HTCondor jobs to understand whether new virtual machines need to be deployed or existing ones can be reused. elastiq differs from Cloud Scheduler as it addresses a simpler use case where the mechanism giving elasticity to the infrastructure is \textit{internal} to the infrastructure itself.
4.4.7.9 Future work

Although the elastiq daemon has been conceived around HTCondor, it is not difficult to make it more generic: the next WMS we are going to support is Work Queue\cite{work_queue}, especially for dynamic on demand Quality Assurance (QA) validator infrastructures based on Makeflow\cite{makeflow}, capable of reusing opportunistically and through virtual machines some farms when not used, such as High Level Trigger (HLT) farms, which are idle outside data taking periods.

Currently using the CernVM Online (§ 4.4.3.3) interface it is possible to create, as we have seen, a head node with appropriate EC2 credentials that will take care of starting the other nodes. We have seen as creating such special head node is eventually equivalent to create an entire cluster, thanks to elastiq. We however find it counterintuitive, especially given the presence of a specific (now under development) interface for creating clusters: the idea is to create a special head node taking care of deploying:

- fixed services (such as a CernVM-FS Squid proxy);
- scalable services (such as worker nodes, by means of elastiq).

The idea is to find a model of convergence between CernVM Online and the concept of CernVM Cloud\cite{cernvm_cloud} using a single web interface (CernVM Online) and managing the scalability of small clusters directly with elastiq. In practice, the combination of a new CernVM Online interface and a boosted elastiq capable of running fixed services are planned to converge towards CernVM Cloud.

4.4.8 The VAF client

As we have seen in § 4.4.5, users access the Virtual Analysis Facility by means of SSH: this means that no particular software should be installed on the user’s computer, because everything that is needed is directly available on the Virtual Analysis Facility.

After logging in to the Virtual Analysis Facility, a helper client is available to control it. The client is essentially a wrapper around some PoD commands that takes care of setting a consistent shell environment on the client session, the master server and each worker.

In § 4.4.8.1 we describe how complex environment settings are kept consistent, and how such complexity is hidden to the user. Initialization of remote workers environment also includes the distribution of Grid credentials on every computing node: this is covered in § 4.4.8.2. How PoD commands are wrapped in order to add environment settings transparently is described in § 4.4.8.3.
4.4.8.1 Modular environment settings

The VAF client takes care of setting:

- the “local” environment, i.e. the environment of the client session;
- the “remote” environment, i.e. the environment of the master and the workers;
- a “common” environment, loaded both locally and remotely.

Local and remote environment scopes are defined by several small scripts, loaded in an orderly fashion. Each environment script is searched, in order, in the following two directories:

- first, they are searched in the VAF client installation directory;
- if they are not found there, they are searched in the user’s home directory, under ~/.vaf.

In case a script is not found in either directory, it is simply skipped.

This mechanism allows the administrator to embed complex experiment specific settings into hidden system-wide scripts, leaving only simple options available to the end user. For instance, it is possible to leave a single file configurable by the user containing only the version of the software framework to use, and keep all the complexity of dependency resolution and export of paths and library paths to the hidden, system-wide scripts.

Currently, the VAF client is shipped with embedded support for ALICE and CMS environments. For ALICE, the user has only one configuration file containing a single line, to set the AliRoot version:

```
export VafAliRootVersion='v5-05-38-AN'
```

The system-wide ALICE scripts will take care of creating the AliEn token and setting the appropriate ROOT version. ALICE uses Environment Modules\[113\] for dependency definition and resolution.

CMS environment settings are managed by SCRAM: as for ALICE, Virtual Analysis Facility user has a single configuration file where she specifies the desired version of CMSSW:

```
export VafCmsswVersion='CMSSW_6_2_6'
```

SCRAM will be used in the background to manage software dependencies. After setting the appropriate variables, the user simply enters a configured environment by typing:
4.4 – The Virtual Analysis Facility

A detailed explanation on how the environment files are loaded in order is available in § E.5.1.

### 4.4.8.2 Distributing Grid credentials

When executing a PROOF analysis on the Virtual Analysis Facility, it might happen that the tasks running on the workers need, for some reason, to access Grid services requiring some form of authentication: the most common reason is accessing files from a Grid storage requiring authentication.

Grid authentication implies the presence of a user certificate and private key. Such credentials are not directly used to access the services: instead, they are used to issue a short-living certificate with its own private key called a *proxy certificate*.

In general, when performing distributed computing, it is unsafe to distribute the original credentials on all computing nodes. Mechanisms such as *tokens* or proxy certificates are a safer way to go: even if a proxy certificate gets compromised, its short duration limits potential harms.

A proxy certificate can be created directly on the Virtual Analysis Facility head node using the Grid middleware tools provided by your experiment. The certificate is commonly stored in a temporary directory, in a file named:

```
/tmp/x509_up<UID>
```

where `<UID>` is your current user id. In principle, PROOF supports certificate authentication, and it is capable of distributing proxies on all computing nodes: this form of authentication is the one used by AAFs and it is described in § C.1.

However, in our simple PoD setup, we do not need any form of authentication, being each PoD (and therefore PROOF) session private to each user: for this reason, we use the environment script to distribute proxy credentials instead of using PROOF.

The VAF client sets the remote environment by concatenating all the “common” and “remote” environment scripts into a single remote environment. It is also possible to include a *payload* that will be embedded in the remote environment. The payload is generated by a configurable executable on the standard output.

For distributing the Grid credentials, the payload executable is a single-line script:

```
#!/bin/bash

echo "GridProxyBase64=$( cat /tmp/x509up_u$UID | gzip -9 | base64 | tr -d '\n')"
```
This one-liner:

- reads the proxy;
- compresses it;
- converts it to a single-line base64;
- generates a shell script line containing the base64-encoded, compressed version of the proxy in an environment variable.

A sample (truncated) output would be:

```
GridProxyBase64=H4sIAGwP+lICAyvOLyhKzEksSMxLE1LJiUBecmJuWJxfklJPpp...
```

The rest of the remote environment script will reverse the operations: the variable will be decoded, uncompressed and finally written back to the remote /tmp/x509up_<UID>.

### 4.4.8.3 Simplified PoD commands

Once entered in the Virtual Analysis Facility environment via the `vaf-enter` command (§ 4.4.8.1), the user can start a PoD server with:

```
vafctl start
```

This is a wrapper to the `pod-server` command with proper environment and VAF-specific settings. Once the PoD server is started, new workers can be requested with:

```
vafreq N
```

where N is the desired number of workers. Behind the scenes, the `pod-submit` command is used, configured with the HTCondor plugin.

The number of active workers can be seen by running the `vafcount` command, which continuously updates its status until interrupted manually:

```
> vafcount
Updating every 5 seconds. Press Ctrl-C to stop monitoring...
[20140211-155241] 2
[20140211-155246] 6
...
```
4.4 – The Virtual Analysis Facility

There is also the vafwait command, used to wait until a certain number of workers is ready, before executing the next command. For instance, if we would like to start our analysis only when there are at least 5 workers available, we would do something like:

```
vafwait 5 && root -l 'MyAnalysis.C+'
```

Once the session is finished, the PoD server can be stopped like this:

```
vafctl stop
```

All the ordinary PoD and HTCondor commands can still be used as well, but the VAF client conceals all the potentially complicated aspects by providing a straightforward interface with a few commands only.

4.4.9 Performance considerations and measurements

The Virtual Analysis Facility is composed by a variety of components with complex interactions among them. To give an idea of the performances and benefits of the Virtual Analysis Facility, we have conducted two different types of tests, involving the different components.

The first class of tests focuses on PROOF and PoD as the base for a pull scheduling architecture for HEP analysis. The tests were not conducted on a virtualized and dedicated environment, instead PoD was run on the Grid, in order to compare, on the same production resources, the behaviour of standard Grid jobs and the same analysis run as a PROOF task. Measurements and calculations are reported in § 4.4.9.1.

The second class of tests aims to give an estimate of the time required to deploy a virtual machine on different cloud infrastructures. Results of these tests are reported in § 4.4.9.2.

4.4.9.1 Efficiency of PROOF compared to batch models

We have extensively seen how PROOF constitutes a full-fledged pull scheduling model, thanks to the presence of the adaptive packetizer (§ 4.1.4) and the recently added support for dynamically added workers (§ 4.4.6), realized within the context of the Virtual Analysis Facility.

Since PoD allows to use batch systems opportunistically for PROOF analyses, and since both HEP batch analyses and PROOF tasks rely on event-based parallelism, it is possible to compare the same analysis task on the same batch system both in pure batch mode and with PROOF and PoD: the purpose is to measure the actual time to results.
Tests were conducted using the PoD PanDA plugin, currently experimental, used by ATLAS for using PROOF on the Grid\cite{136}. All submissions have been maintained local for network constraints to each Grid site, meaning that it is currently not possible to run a single PROOF analysis with PanDA on multiple Grid sites at the same time, although a solution is underway.

Tests were run in the Italian ATLAS Grid sites, plus the CERN site: they consisted in the submission, via PoD PanDA, of 100 worker requests. We have measured the ramp-up of the available workers over time; data has been collected by analyzing the output of the command:

```
pod-info -l
```

which returns the startup time of each PoD worker. It must be noted that:

- the time required for a pilot job to start a PoD agent was not taken into consideration: it has been never observed, however, to be other than negligible;
- all our time measurements start as soon as the first worker is available;
- the number and types of Grid jobs running on the sites at the time of tests inevitably have an influence on the rate of finishing jobs: the test conditions cannot be under our direct control;
- Grid sites may apply fair share user quotas which might limit the total number of obtained workers, in spite of the actual free resources.

Results were collected for up to one hour: in the context of interactive analyses longer wait times are unfeasible. Ramp-ups are shown in Figure 4.14: the displayed data, which is the starting point of our forthcoming considerations, has been collected within the context of a collaboration with ATLAS\cite{136}.

For the following analytical calculation based on the actual data shown in Figure 4.14 we want to estimate the maximum possible speedup attainable by running the same analysis on the same data and the same computing resources using PoD and PROOF instead of launching independent batch jobs. Since the analysis is the same, the serialized computing time (i.e. the time it would take to run the whole analysis as a single job) will be the same in both cases: the speedup we are talking about refers to the time to results, which we define as the time elapsed from the start of the first job to the collection of the results. Our simplified model does not take into account the time required for merging several partial results.

We start by defining an analytical model for the ramp-up functions in Figure 4.14: they all show a similar behaviour, including a saturation of the total
number of obtained workers over time, which is always less than the number of requested workers. The displayed curves can be fitted using the following parametric function:

\[ n(t) = \frac{p_0 t}{1 + p_1 t} \]  

(4.1)

giving the number of workers \( n \) available at a given time \( t \), where the assumption here is that the first worker is obtained at \( t = 0 \).

The function is valid for positive values of \( p_0 \) and \( p_1 \). The parameter \( p_0 \) is the rate of finishing jobs, assumed constant: the number of workers available is proportional to \( p_0 \). The submission rate is assumed constant as well. Scheduling policies and site capacity are convoluted in the parameter \( p_1 \). Therefore, a couple of those parameters is characteristic to a certain Grid site. An asymptote is found in (4.1):

\[ \lim_{t \to \infty} n(t) = \frac{P_0}{P_1} \]  

(4.2)

describing the maximum number of jobs that a single user can obtain on the site on a single submission, and reproducing the saturation behaviour observed in measurements. Fit results yield parameter pairs for each Grid site, reported with their error in table Table 4.4: since data in Figure 4.14 does not subtract the time required to obtain the first worker, the fit function contains a third offset parameter which can be ignored for the purposes of our dissertation. The complete
function used for fitting is:

\[ n(t) = \frac{p_0(t + p_2)}{1 + p_1(t + p_2)} \] (4.3)

<table>
<thead>
<tr>
<th>Grid site</th>
<th>( p_0 )</th>
<th>( p_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERN</td>
<td>0.28250 ± 0.01440</td>
<td>0.00258 ± 0.00016</td>
</tr>
<tr>
<td>CNAF</td>
<td>0.25351 ± 0.01776</td>
<td>0.00271 ± 0.00023</td>
</tr>
<tr>
<td>Roma 1</td>
<td>0.04272 ± 0.00202</td>
<td>0.00030 ± 0.00004</td>
</tr>
<tr>
<td>Napoli</td>
<td>0.07610 ± 0.00458</td>
<td>0.00084 ± 0.00008</td>
</tr>
<tr>
<td>Milano</td>
<td>0.03842 ± 0.00227</td>
<td>0.00036 ± 0.00005</td>
</tr>
</tbody>
</table>

**Table 4.4:** Parameters of (4.1) for the considered Grid sites, obtained by fitting collected data represented in Figure 4.14.

By inverting (4.1) we can find the time \( t \) required to have \( n \) running jobs:

\[ t(n) = \frac{n}{p_0 - p_1 n} \] (4.4)

Assuming that the initialization time of each worker similar in the Grid and PROOF case, we can ignore it for our comparison purposes. A PROOF analysis using dynamic workers (§ 4.4.6) and the adaptive packetizer (§ 4.4.6.5) is considered a pull scheduling model: workers are never idle and they spend their full analysis time in processing. In this case, a PROOF analysis completing its execution in a time \( t' \) has used a serialized computing time given by the integral:

\[ T(t') = \int_0^{t'} n(t)dt = \frac{p_0}{p_1} \left( p_1 t' - \log(1 + p_1 t') \right) \] (4.5)

The PROOF analysis with dynamic workers can run on any number of workers, this is shown by the fact that the result does not contain a dependency from \( n \) anymore.

For what concerns the pure push case, *i.e.* standard Grid batch submission, if our reference analysis needs a serialized time \( T \) and it is divided in \( n' \) jobs, it will be completed when the very latest job completes, *i.e.:

\[ t' = t(n') + \frac{T}{n'} = \frac{n'}{p_0 - p_1 n'} + \frac{T}{n'} \] (4.6)

where the time \( t' \) is given by the time required to get the \( n' \)-th worker, plus the time required for executing a single job, assumed to be even for all jobs \( (T/n') \).
In this case we still have a dependency on $n'$: in a batch model we must decide exactly how many workers will do the job before submission. In order to eliminate this dependency, we assume that $n'$ is chosen in order to minimize the time to results $t'$. By finding the zeroes of the derivative of (4.6), we obtain an analytical expression for $n'$ depending on the sole serialized processing time $T$:

$$n'(T) = \frac{p_0 \sqrt{T}}{\sqrt{p_0} + p_1 \sqrt{T}}$$

We now substitute (4.7) into (4.6) to finally find the “best” time to results attainable with push scheduling as a function of the serialized time $T$:

$$t'_\text{push}(T) = \frac{2\sqrt{p_0 T} + p_1 T}{p_0}$$

We can now plot (4.8) with the numerically inverted (4.5) on the same plot to compare the results; for our purposes, we have chosen $p_0$ and $p_1$ from the CERN site, obtaining what we can see in Figure 4.15(a).

Within the considered range (a serialized computing time of up to 10 days), the plot shows that a pull scheduling architecture such as PROOF exploits resources in a smarter way, leading to results faster. In particular, the pull/push ratio as a function of the serialized time is computed for the CERN site in Figure 4.15(b), and some values are also reported in Table 4.5.

<table>
<thead>
<tr>
<th>Serial time [days]</th>
<th>Push time [hr]</th>
<th>Pull time [hr]</th>
<th>Pull speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0:19</td>
<td>0:13</td>
<td>28%</td>
</tr>
<tr>
<td>1</td>
<td>0:31</td>
<td>0:23</td>
<td>26%</td>
</tr>
<tr>
<td>2</td>
<td>0:52</td>
<td>0:40</td>
<td>25%</td>
</tr>
<tr>
<td>5</td>
<td>1:50</td>
<td>1:26</td>
<td>21%</td>
</tr>
<tr>
<td>10</td>
<td>3:17</td>
<td>2:42</td>
<td>18%</td>
</tr>
</tbody>
</table>

Table 4.5: Some values taken from Figure 4.15(a) and Figure 4.15(b) to compare the time required to obtain the results on the CERN Grid site using either batch jobs or pull jobs like PROOF with PoD. The percentage speedup of pull scheduling compared to push scheduling is also reported.

From the results it appears that pull scheduling modules like PROOF are very efficient for short analyses (PROOF was in fact primarily meant for this type of analyses): however, even in the case of longer analyses (10 days), the speedup obtained is still considerable (18%), justifying the usage of a computing model alternative to the Grid.
Figure 4.15: Push and pull scheduling time to results compared as a function of the serialized computing time which characterizes an analysis. Parameters from the CERN site have been used for such comparisons. Time curves are shown separately, and their ratio showing the relative speedup of pull scheduling is also presented.
The two curves in Figure 4.15(a) are asymptotically equivalent, substantially meaning that for “long enough” analyses there is no real benefit from using a pull scheduling model: end user analyses do not fall into the category and might in fact get a real benefit.

The equivalence for large values of $T$ can be easily demonstrated. The expression for pull scheduling (4.5) for $T \to \infty$ (which implies $t' \to \infty$) reduces to:

$$ T(t') \approx \frac{p_0}{p_1} t' $$

which are identical.

### 4.4.9.2 Startup latency of new virtual nodes

The startup latency of a Virtual Analysis Facility virtual machine, i.e. the time elapsed between the virtual machine is requested and the PoD workers assigned to run there effectively join the PROOF cluster, is a tricky parameter to predict, due to the number and diversity of the operations involved.

Let’s suppose that we have a new Virtual Analysis Facility with only the head node instantiated, and therefore zero worker virtual machines. We proceed by requesting PROOF workers through the VAF client (§ 4.4.8), then when the workers are available we collect the data by using, as for the Grid case (§ 4.4.9.1), the command:

```
1 pod-info -l
```

The latency we will measure will be the convolution of several background factors: a list of the most significant ones follow.

- The minimum waiting time threshold configured in elastiq (see § 4.4.7 and § E.4): if elastiq just checked the queue right before submitting requests, we would have to wait up to that time before it actually requests new virtual machines.
- Availability of free resources: the cloud might not have enough capacity to quickly satisfy the incoming requests.
- Network speed: since our images are based on the $\mu$CernVM loader, the operating system is entirely downloaded on demand, as we have seen in
§ 4.4.3.1. Network or caching proxy inefficiencies inevitably lead to slow startup times.

- HTCondor, PoD and xproofd auto-registration times: those times are usually very low, but large overall times might also indicate a problem in one of such components.

Without an accurate profiling involving each of the relevant components at different levels it is difficult to individuate which component is responsible of a large delay: as we will see, from our measurements in two different conditions we obtain very fast deployment times, indicating that the components are working well together and suggesting, for the moment, no further inspection for latencies. Thanks to the usage of \( \mu \)CernVM, there are certainly some factors that we know they cannot negatively impact the deployment time.

- The \( \mu \)CernVM kernel is not the stock SLC 6 kernel, which is meant for physical machines: being a kernel used only on virtual machines, it was possible to strip it down to a minimum, greatly reducing the impact of loading unneeded modules\(^{[19]} \).

- There is almost zero transfer time required for copying the less than 20 MB base image from the image repository to the destination hypervisor: deployment of images is definitely not an issue, being the primary reason why \( \mu \)CernVM has been created.

- In the presence of properly configured proxy caching services, the impact of downloading a whole operating system from an external network is seen only on the first virtual machine booting: all the others will be able to rely on data cached by the proxy server and retrieved entirely from the internal network.

We have run our tests on two different network infrastructures of different sizes. The first infrastructure tested is the private cloud in Torino based on OpenNebula that has been set up as part of the work for this thesis, and extensively covered in § 2. The other infrastructure, of a larger size, is the CERN Agile Infrastructure, based on OpenStack. For both cases:

- we made sure that enough resources were available from the cloud infrastructure;

- we pre-ran a \( \mu \)CernVM boot in order to allow for the local Squid proxy to precache boot data;

- we set the elastiq wait threshold to less than 10 seconds.
The reason behind such conditions was that we wanted to measure essentially the performances of the µCernVM boot process and the auto-registration promptness of the dynamic services while ruling out as much as possible external factors. In both cases we have requested a number of PROOF workers sufficient to trigger a request for 10 new virtual machines.

Results, shown in Figure 4.16 and reported in Figure 4.16, report the average boot of such 10 virtual machines and the standard error. Numbers are quite surprising: a new virtual machine is ready to join the PROOF cluster from zero in less than 5 minutes, which is a perfectly acceptable time if we consider that, after the boot, virtual machines can be reused several times before transiting to idle and being garbage collected.

The results obtained at CERN are better because of a more performant network infrastructure, while the network at the Torino computing centre has to sustain more relative load since it provides connectivity for many services extraneous to the cloud (§ 2.4.3).

![Figure 4.16: Startup latency of Virtual Analysis Facility virtual machines on two different cloud infrastructures.](image)

<table>
<thead>
<tr>
<th>Cloud</th>
<th>Latency [m:ss]</th>
<th>Stddev [m:ss]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERN OpenStack</td>
<td>3:57</td>
<td>0:07</td>
</tr>
<tr>
<td>Torino OpenNebula</td>
<td>4:36</td>
<td>0:03</td>
</tr>
</tbody>
</table>

Table 4.6: Startup latency of Virtual Analysis Facility virtual machines on two different cloud infrastructures: data has been collected by means of the `pod-info -l` command.

It is interesting to note how there was a significant improvement since our
first measurements, which had yielded 6 min 15 s at CERN and 5 min 51 s in Torino\textsuperscript{[18]}; this is because many components that were installed during the Virtual Analysis Facility contextualization procedure are now served directly as part of \(\mu\text{CernVM}\) over CernVM-FS, and only the necessary parts are downloaded on demand.
Chapter 5

Outcomes and perspectives

5.1 Contributions and results

The scope of this work covers many aspects of cloud computing applied to HEP, and in particular:

- methodologies and issues of the migration of a computing infrastructure to a cloud provider;
- the study and effective solutions of a multitenancy model in a relatively small computing centre;
- the realization of a scalable virtual analysis cluster with a great focus on usability and portability.

Such objectives were reached through a constant involvement with different communities and the direct participation to several HEP projects: in particular, we will sum up the contributions to the Italian Institute for Nuclear Physics (INFN) (§ 5.1.1) and the LHC ALICE experiment (§ 5.1.2) communities, as well as the active participation to the development of PROOF (§ 5.1.3) and CernVM (§ 5.1.4).

5.1.1 INFN

Within the context of the computing centre of INFN Torino, the major outcome has been performing the conversion of the computing infrastructure into a private cloud (§ 2). The efforts behind this work involved a delicate redesign of many parts of the infrastructure without the interruption of the provided services.

This work completely changed the resource provisioning model offered by the computing centre, as well as the maintenance paradigm. The computing centre
is no longer a mere Grid Tier-2: instead, it is a private cloud controlled by OpenNebula (§ 2.3), where resources are available in the form of virtual machines.

Many of the existing tools were reused and updated to facilitate the maintenance of the infrastructure from third parties: the number of hypervisors currently running since the first conversion in late 2011 has doubled as of today (February 2014), and all of them were installed using procedures well known to the computing centre’s personnel (§ 2.4.1). The management of the physical infrastructure, due to the uniformity of the physical machines to install (all of them are now KVM hypervisors), has greatly improved.

A well-defined management model for maintaining virtual machine base images has been defined during the years, and it is currently consolidated: to avoid the proliferation of virtual machine images, most of the virtual services are currently configured through contextualization (§ 3.1.1) as much as possible. This model had a great impact on the security update policies of the computing centre: after the migration of the Tier-2 (services and worker nodes) to the cloud infrastructure (§ 3.1), services are updated more often, as the operation scales more rapidly (a single virtual machine upgrade replicated many times) and can be done by gradually replacing running services, with no interruption (§ 2.2.4.2).

Many management operations were automatized by writing Ruby and Bash scripts based on the OpenNebula interface: the scripts are easy to use and extensively documented (§ A).

After running the first “golden” services directly managed by the computing centre’s personnel—the Grid (§ 3.1), the “old” TAF (§ 4.3.7 and the new Virtual Analysis Facility (§ 4.4), an effort has been done to define a resource provisioning model to allow external users to run their own virtual machines. Our multitenancy model is based on the concept of “virtual farms” (§ 3.2), where each user is given a quota and a sandbox with an “elastic” public IP address: the sandbox prevents the user from exposing her virtual farms and our infrastructure at risk, while the presence of one distinct IP per virtual farm helps tracking down suspect activities. The user procedure is very simple and well documented\[^{[58]}\], and the infrastructure is being rolled out to users in these days.

Torino’s computing centre is the first Tier-2 of INFN having migrated entirely to a cloud-based model: the only computing centre having a production cloud is the Tier-1 at CNAF, Bologna, that used a custom approach to the cloud\[^{[100]}\]. Notwithstanding the proclivity of HEP computing experts to develop custom solutions, we have decided to stay mainstream and to follow closely the evolution of industry standards: our approach proved to be successful, as we could count on support from a broader community of experts also outside HEP. Torino’s private cloud is now considered an important example that other INFN computing centres are starting to follow, and as we will see soon in § 5.2, the migration has been the starting point of a national project for distributed analysis on the cloud.
5.1 – Contributions and results

5.1.2 ALICE

A great part of this work has been dedicated to supporting analysis with PROOF in the ALICE experiment. In particular, significant contributions have been done to the definition of the data model, data moving policies and dataset management of ALICE Analysis Facility (§ 4.3).

Within the context of ALICE, the dataset stager daemon has been developed (§ 4.3.5): the daemon is currently in production in every AAF since 2010, and it has proven to be stable. Since 2011 its source code is part of ROOT.

For the ALICE Analysis Facility a series of utilities for creating and managing PROOF datasets have been developed (§ 4.3.5.5): such utilities are available as a PROOF package that can be enabled on every AAF. An interface to access the AliEn File Catalog as PROOF datasets has been developed as an evolution of the dataset model (§ 4.3.6), which finally allows to access ALICE files from PROOF by querying the catalog instead of maintaining separate lists. The interface is one of the solutions made to overcome the limitations of the current AAF data model (§ 4.3.4), and it is the default method of accessing files on the Virtual Analysis Facility for ALICE (§ 4.4).

The Virtual Analysis Facility, although generic by definition, has ALICE in mind as first potential customer, and all the first VAF PROOF tests were performed by running ALICE use cases.

5.1.3 PROOF

Apart from the already mentioned dataset stager (§ 4.3.5), several contributions were made to PROOF including the ones concerning the computing model and supporting utilities such as PROOF on Demand (PoD).

Previous PROOF deployments have been carefully analyzed: the majority of PROOF deployments was made of statically dedicated computing nodes, and several issues were identified in such setups (§ 4.2.2). We already knew that PoD (§ 4.2.4) allowed to run PROOF with zero installation on top of non-dedicated batch resources: we have extended this idea to the clouds by creating the Virtual Analysis Facility (§ 4.4), a virtual analysis cluster which comes configured out of the box, specifically thought for running PROOF analyses.

PROOF was the best starting point for the Virtual Analysis Facility on the cloud, as it already featured a very efficient pull scheduler capable of smoothing out performance differences between the computing nodes, which is a very useful feature when running on heterogeneous virtual machines. By concentrating on making PROOF more “cloud aware” (§ 4.4.1.1), we have added one impor-
tant and missing feature: the ability to dynamically add workers to PROOF while an analysis is running (§ 4.4.6), which finally eliminates the need to wait until “some” workers are ready before starting the analysis. This feature has been proven to be particularly useful when PROOF workers are obtained via PoD, either in the form of virtual machines or Grid jobs, where workers always become available gradually.

By doing an analytical comparison based on actual measurements we have also shown how, on the same set of Grid resources, running the same analysis code with a pull scheduling model (with dynamic workers) such as PROOF leads to results faster than a plain batch analysis § 4.4.9.1.

In order to make the Virtual Analysis Facility support as much as possible heterogeneous clouds, we have contributed to the development of PoD adding some features and fixing some connectivity problems (§ 4.4.4.2).

5.1.4 CernVM

The realization of the Virtual Analysis Facility for PROOF analysis was possible due to a synergy with the development of CernVM: the Virtual Analysis Facility is, in fact, based on µCernVM (§ 4.4.3.1), and the web interface that allows users to run it easily is CernVM Online (§ 4.4.3.3).

The relevant contribution to µCernVM was harmonizing the support of different contextualization mechanisms in a single script, and in particular the addition of the support for OpenNebula contextualization (§ 4.4.3.3). Moreover, since the Virtual Analysis Facility was the first use case for µCernVM, we could discover a series of minor issues, whose fixes helped leading to the release of a stable version of µCernVM in early 2014.

The Virtual Analysis Facility itself is completely configured as a particular contextualization of µCernVM: to do so, we have contributed to the development of CernVM Online by adding a simplified configuration interface for the VAF. Some ancillary work had benefits outside the scope of the Virtual Analysis Facility: support for µCernVM configuration options has been added to CernVM Online as part of this work.

5.2 Italian research project of national interest

As part of the work covered in this thesis, in 2011 a project proposal has been submitted to the Italian Department of Education (MIUR) in collaboration with 11 Italian universities (including the University of Torino) and the INFN LNL (Laboratori Nazionai di Legnaro). The project aimed to design and make operative an infrastructure of computing for HEP experiments at national level, based on
PROOF and cloud computing technologies.

The project successfully obtained fundings for three years, becoming a “Research Project of National Interest”: the keystone of the project is the adoption of the Virtual Analysis Facility which naturally meets the project’s expectations. As a side effect, many of the universities involved in the project are finally starting to migrate their physical infrastructures to the cloud, in order to support the Virtual Analysis Facility and future computing models exclusively based on the cloud.

In January 2014 the universities and INFN computing centres that took part in the project finally started testing and adopting the Virtual Analysis Facility as a cross-experiment HEP computing model for the cloud.

5.3 Perspectives and future work

In the following sections we will describe the future plans for the private cloud in Torino (§ 5.3.1), the Virtual Analysis Facility and its use cases (§ 5.3.2), PROOF and its integration with ROOT (§ 5.3.3) and the adoption of the developed tools in the ALICE experiment (§ 5.3.4).

5.3.1 The private cloud in Torino

The private cloud set up in Torino is currently stable and in production: as we have opened it recently to external users, the main task would be to support such users and get feedback from them. Standards have been taken into great consideration: private cloud users access it using the EC2 interface (§ 3.2.3), while the OpenNebula native interface has not been exposed. Since EC2 is widely supported by many cloud controllers, our setup allows us to test possible alternatives to OpenNebula, and, in case they better suit our cases, replace the cloud controller while keeping exposing the same EC2 API.

Currently, for managing all the virtual machines related to the Grid (worker nodes, § 3.1.1 and services, § 3.1.5) we still use the OpenNebula native interface and virtual machine templates: for a more uniform management, the first step is to use exclusively the EC2 interface for managing all virtual machines.

Since currently we are keeping a separate OpenNebula user’s database, a great improvement will be delegating the authentication to the national INFN identity provider or to other external entities: the outcome will be that users will not need to have new access credentials for the cloud.

It is also foreseen to provide users with a web interface to manage their virtual machines. Currently we use OpenNebula Sunstone (§ 2.3.2) for our internal purposes, but it is believed to be not simple enough for our use case.
Due to the successful experiment of instantiating a single virtual farm spanning on two different clouds (§ 3.2.2.4), continuing the collaboration with other raising small cloud communities is important to find common standards of interoperability.

5.3.2 The Virtual Analysis Facility

The Virtual Analysis Facility is gathering its first users especially thanks to the national project involving it (§ 5.2), and it is helping to define an analysis model based on cloud computing for HEP. As the number of users is increasing rapidly, the feedback is expected to increase likewise.

The Virtual Analysis Facility was originally designed to run PROOF: as we have seen, however, it is only a batch system which scales elastically, independently from the submitted jobs. For reaching a wider community, our work plan is to regard the Virtual Analysis Facility as one of the main components of the CernVM ecosystem, and to support the Work Queue scheduler as an alternative to HTCondor, as we have already discussed in § 4.4.7.9.

Even though the Virtual Analysis Facility is already capable of supporting use cases different from PROOF, more work is needed on the user interfaces (CernVM Online in particular, § 4.4.3.3) to finally establish it as the preferred way of managing an automatically scalable virtual cluster by hiding the configuration complexities.

5.3.3 PROOF and integration with ROOT

The PROOF packetizer (§ 4.1.4) already deals well with the non-uniformity of cloud resources. It is also capable to take into consideration data distribution to minimize network transfers. A change in the packetizer to automatically rank multiple data sources while an analysis is running is needed.

In general, following the new development directions of PROOF, its code should be completely revised, by eliminating all the deployment parts which are already managed well by PoD: starting from the codebase of PROOF-Lite would be a good start.

Currently, the management of event loops in ROOT is clumsy: writing an analysis in the form of a TSelector should in principle provide the level of abstraction necessary to run the same code locally, on a batch system or on PROOF. However, modifications for PROOF are still needed: a more uniform and transparent way to separate the analysis code from the event loop is needed.

Many parts of PROOF are for the moment serial, and pose scalability problems when the number of connected PROOF workers is very high: for instance, the packetizer operates serially, and all PROOF connections are managed by a single
serial loop, as we have seen in detail while describing the internals of dynamic workers addition (§ 4.4.6.4). It would be therefore interesting to inspect lock-free techniques for dynamic data assignment which will improve the scalability of PROOF, and use a new communication model based on the parallel processing of network messages.

5.3.4 The ALICE experiment

The ALICE Analysis Facility model has made using PROOF for analysis popular in the ALICE collaboration: however, as we have extensively seen, the AAF model has several maintenance problems—essentially because it requires continuous manual interventions to fix data inconsistencies (§ 4.3.4) and to restart locked machines (§ 4.2.2).

The Virtual Analysis Facility has already in place all the configurations needed for ALICE (for instance, the authentication, § 4.4.5.4); moreover, a direct interface to get rid of the datasets to directly query and cache the AliEn File Catalog has been developed (§ 4.3.6), and it became the preferred model to access data from the Virtual Analysis Facility. Despite the VAF model represents an evolution to the AAF model, a convergence has to be found to make it reach the whole ALICE community, starting from the experiences of early adopters in the Italian community (§ 5.2).

As we have seen in § 4.3.7.2, accessing the AliEn File Catalog constitutes a big bottleneck which has a significant impact on the slowdown especially when accessing small files. A solution for caching the site URLs returned by the Catalog has been developed § 4.3.6, whereas solving the problem of listing a local storage content (that involves a reverse query to the Catalog) can pose serious scalability problems, especially when considering that all the preexisting files should be indexed.

Although knowing the storage’s content is a general problem, we can think of limiting the indexing only to the files we freshly mirror for reading them with the Virtual Analysis Facility. A possible solution, currently under discussion, would be to keep two separated local namespaces: the legacy namespace, effectively containing the files along with their “complicated” PFNs, and namespace that replicates the AliEn Catalog’s structure, where each entry is stored as a symbolic link to a file in the PFN namespace. While the first namespace will be exported to the Grid nodes as usual, the new namespace would be exported to the Virtual Analysis Facility nodes only. When using XRootD to export the symbolic links we can even think of relying on its Virtual Mass Storage System (vMSS) capabilities (see § 4.3.3) to refresh broken links. This system would constitute an effective implementation of distributed AliEn catalog caches.
Appendix A

Cloud management utilities

A series of tools has been written to help managing our private cloud infrastructure and its integration with the Grid site. These tools are either Ruby or Bash scripts.

Every communication with OpenNebula occurs through the Ruby OpenNebula Cloud API (OCA)\(^83\) bindings.

A.1 Cloud environment for users: cloud-enter

The `cloud-enter` shell script prompts user for the OpenNebula username and password, and sets the environment for the usage of both the OpenNebula tools and the euca2ools\(^40\) communicating with the EC2 interface.

```
1  > cloud-enter theclouduser
2  Password for cloud user "theclouduser": ***
3  Authenticating...OK
4
5  EC2 commands start with euca-*, OpenNebula commands start with one*
6  Use [Tab] to complete.
7  Type exit to return to your normal shell.
8
9  cloud@inftnt user: theclouduser
10  >
```

A.2 Syncing cached datastores: one-cache-sync.rb

This is the tool used for keeping OpenNebula cached datastores in sync. It must be run as a privileged user. Synchronization is bidirectional: images present only
on a certain hypervisors are transferred to the master; then, all images present on the "cached" datastores are transferred on all hosts.

The scripts considers only the datastores whose name begins with cached_.

Image transfer occurs by means of the rsync command, wrapped by a modified version of the scpWave.py script: the purpose of this script is to speed up file transfer, by making an host that already received the file being the source for other hosts, creating a transfer swarm similar to the torrent model. Our modification consisted in using rsync instead of scp for further speedups.

If an image has been removed from the datastore, it is also removed from every hypervisor.

The command runs in “dry run” mode by default, not performing any change. Synchronizing everything occurs with a very simple command, and the last output line clearly indicates whether there were errors or not:

```
> one-cache-sync.rb --no-dry-run
...
All OK: looks like there were no errors
```

### A.3 Extended virtual machines info: onevm-addr.rb

This script has the same functionalities of the onevm list command, but it provides extra information on all the IP and FQDN addresses with which a virtual machine is known. Sample output (shortened):

```
> cloud-enter proof
...
> onevm-addr.rb

<table>
<thead>
<tr>
<th>VMID</th>
<th>User</th>
<th>...</th>
<th>LCM</th>
<th>Hypervisor</th>
<th>Leased addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>4339</td>
<td>proof</td>
<td>...</td>
<td>runn</td>
<td>one-kvm-34</td>
<td>192.168.6.199, t2-vprf-199</td>
</tr>
<tr>
<td>4340</td>
<td>proof</td>
<td>...</td>
<td>runn</td>
<td>one-kvm-04</td>
<td>192.168.6.200, t2-vprf-200</td>
</tr>
<tr>
<td>4359</td>
<td>proof</td>
<td>...</td>
<td>runn</td>
<td>one-kvm-13</td>
<td>192.168.6.215, t2-vprf-215</td>
</tr>
</tbody>
</table>
```

### A.4 Connecting to serial consoles: onevm-console.rb

To provide a way of accessing a virtual guest when all other methods fail, most of our base images are configured to expose a virtual serial console. This is achieved by configuring the virtual machine properly, and by adding to the OpenNebula template a RAW section that will be passed as is to KVM:
**A.5 – Draining virtual worker nodes: onevm-wn-drain.rb**

This command interacts with PBS and OpenNebula to gracefully shut down virtual worker nodes: nodes are shut down only when there are no more jobs running on them. It works as follows.

- First of all, nodes are put in “offline” state in PBS: this means that they will not accept any new job.

- Mapping between OpenNebula virtual machine IDs and PBS worker nodes occurs internally.

- The main loop checks the PBS nodes status to retrieve the number of running jobs. The script is capable of parsing the XML output returned by `pbsinfo -x`, which is usually run remotely via SSH.

- If the node has zero jobs, the virtual machine is sent the shutdown signal. Moreover, the node is removed from PBS.

- The main loop continues until there are worker nodes to shut down.

The script has three configurable parameters:

- `n, --names wn1,wn2,...`: list of nodes to drain

---

```bash
raw_text
# Virtual serial console, added manually as raw XML data
RAW = [
    TYPE = kvm,
    DATA = "<devices>
    <serial type='pty'><target port='0'/></serial>
    <console type='pty'><target port='0'/></console>
    </devices>"
]
```

To connect to the serial console, one must normally ssh to the hypervisor, finding the virtual machine id with `virsh list` and finally connecting using `virsh`. The `onevm-console.rb` script makes this workflow automatic. From the cloud head node:

```bash
onevm-console.rb -r <ONE_VM_ID>
```

The optional `-r` switch keeps retrying in case it fails: with this switch, the command can be issued when the virtual guest has not booted yet, in order to connect as soon as the boot starts and inspect boot messages.
LOUD MANAGEMENT UTILITIES

- \texttt{-t, --thresholdTHR}: number of consecutive failures before stop monitoring a node (default: 3)

- \texttt{-s, --sleepSEC}: seconds to sleep between loops (default: 60)

An usage example that wraps it around \texttt{screen} to allow detaching the session:

```
> screen -dmS drain onevm-wn-drain.rb -n t2-vwn-45 -t 10 -s 60
```

```
> screen -r drain

[20140103-112144] Setting node t2-vwn-45 (t2-vwn-45.to.infn.it) offline

[20140103-112145] === Getting nodes statuses ===

[20140103-112146] Node t2-vwn-45 (t2-vwn-45.to.infn.it) with VMID 4126 has 3 jobs(s) and 0 error(s)

[20140103-112146] Sleeping 60 seconds...

[20140103-112146] Node t2-vwn-45 (t2-vwn-45.to.infn.it) with VMID 4126 has 0 jobs(s) and 0 error(s)

[20140103-134502] No more jobs on node t2-vwn-45

[20140103-134502] >> Removing node t2-vwn-45.to.infn.it from the queues...

[20140103-134509] >> OpenNebula VM 4126 shut down

=== Final report ===
Nodes drained successfully: t2-vwn-45
```

A.6 Creating a virtual network: onevnet-new.rb

This helper script creates both a virtual network template for OpenNebula and a series of commands to run to configure the corresponding "virtual" entries in Cobbler: this is done because Cobbler manages the database of the Tier-2 network, keeping in sync the DHCP and DNS databases.

With a proper switch it is possible to “commit” changes by adding the created network and running the Cobbler commands. The list of command line options follow for reference.

- \texttt{-r, --rangeIP}
  IP mask (default: 192.168.0.0)

- \texttt{-m, --macMAC}
  base MAC address (default: 02:00:00:00:00:00)

- \texttt{-i, --[no-]isolation}
  isolates network, \textit{i.e.}: in OpenNebula (default: false)
-p, `--cobbler-profile` PROFILE
   associated Cobbler fake profile (default: 00-AfakeProf)

-d, `--domain` DOMAIN
   DNS domain for added hosts

-b, `--base-name` BASENAME
   host base name

-n, `--net-name` NETNAME
   network name

-c, `--[no-]cobbler`
   invokes cobbler (default: false)

-o, `--[no-]onevnet`
   invokes onevnet (default: true)

-[-no-]commit
   invokes commands as opposed to printing to screen (default: false)

-s, `--skip-addr` 0,255,bcast
   skips IP addresses ending with .0, .255 and broadcast addresses

### A.7 Creating a user sandbox: onevrouter-generator.rb

This script creates a new user sandbox as described in § 3.2.5 by performing all the required OpenNebula operations. The main task is to instantiate a Virtual Router. The list of command line options follow.

-u, `--username` USERNAME
   username to create

--public-ip PUB_IP
   public IP address

--priv-ip PRIV_IP
   private IP mask

-c, `--[no-]commit`
   commit changes (default: don't commit)

Example of usage:
We are going to create a User with her own VRouter, Pub and Prv network with the following parameters:

* User name : newuser
* Public IP : 193.205.66.123
* Public Hostname : cloud-gw-123.to.infn.it
* Private IP range : 172.16.123.0..172.16.123.255

NOTE: template files will be created for the network and VRouter, but no change will be committed to OpenNebula.

Is this OK? Type "yes, it is":

After answering the question, the operation proceeds. In this case, templates will be created and proper instructions will be printed, while when using the --commit option, the Virtual Router is instantiated and user, virtual networks and ACLs are committed to OpenNebula.

A.7.1 OpenNebula template of the Virtual Router

The configuration file that follows is the OpenNebula virtual machine template we use for each one of our Virtual Routers. The CONTEXT section will result in the creation of a context.sh file put in the root directory of the contextualization CD-ROM containing all the variables defined there.

The contextualization script init.sh will interpret the variables and configure the Virtual Router for the defined virtual private network (sandbox-Prv in this example). Note that the Virtual Router has also a second network interface on the public network: the virtual network VRouter-Pub defines a pool of public addresses available for Virtual Routers.

This is the template used by the onevrouter-generator.rb script to create new Virtual Routers.

```ruby
NAME = "sandbox-VRouter"
# Memory in MiB (not MB)
MEMORY = 150
```
# Number of virtual and real CPUs
VCPU = 1
CPU = 1

# VRouter is a 32-bit VM
OS = [ ARCH = "i686" ]

# CACHE=none is important for live migration
DISK = [
  IMAGE = "OpenWrt-KVM-trunk-build23",
  DRIVER = raw,
  CACHE = none,
  TARGET = vda ]

# Private network
NIC = [
  NETWORK = "sandbox-Prv",
  IP = "172.16.123.254",
  MODEL = virtio ]

# Public network
NIC = [
  NETWORK = "VRouter-Pub",
  IP = "193.205.66.123",
  HOSTNAME = "cloud-gw-123.to.infn.it",
  MODEL = virtio ]

# Contextualization
CONTEXT = [
  FILES = "/path/init.sh /path/key.pub /path/update-elastic-ip.sh",
  DNS1 = "$NETWORK[ DNS1, NETWORK = \"VRouter-Pub\"]",
  DNS2 = "$NETWORK[ DNS2, NETWORK = \"VRouter-Pub\"]",
  GATEWAY = "$NETWORK[ GATEWAY, NETWORK = \"VRouter-Pub\"]",
  WAN_IP = "$NIC[ IP, NETWORK = \"VRouter-Pub\"]",
  WAN_MASK = "$NETWORK[ NETWORK_MASK, NETWORK = \"VRouter-Pub\"]",
  FQDN = "$NIC[ HOSTNAME, NETWORK = \"VRouter-Pub\"]",
  LAN_IP = "$NIC[ IP, NETWORK = \"sandbox-Prv\"]",
  LAN_MAC = "$NIC[ MAC, NETWORK = \"sandbox-Prv\"]",
  LAN_MASK = "$NETWORK[ NETWORK_MASK, NETWORK = \"sandbox-Prv\"]",
LAN_DOMAIN = "\$NETWORK[DOMAIN_NAME, NETWORK = \"sandbox-Prv\" ]",
HOST_PREFIX = "\$NETWORK[HOST_PREFIX, NETWORK = \"sandbox-Prv\" ]",
TARGET = "vdb"

# Pin to the Services cluster: needed for the private network
REQUIREMENTS = "CLUSTER = \"Services\"

# Striping policy
RANK = -MEM_USAGE

# Used by virsh to attach a VNC console accessible only from the hypervisor,
# that is 127.0.0.1. Port is chosen automatically (should be 5900 + vmid).
GRAPHICS = [ LISTEN=0.0.0.0, TYPE=vnc ]

# Virtual serial console, added manually as raw XML data
RAW = [ TYPE = kvm,
       DATA = "<devices>
               <serial type='pty'><target port='0'/></serial>
               <console type='pty'><target port='0'/></console>
               </devices> " ]

A.8 Sending commands to multiple hosts: xpdsh.rb

This Ruby script is a wrapper around the pdsh utility[^86], which allows for sending commands to multiple hosts. xpdsh.rb, unless otherwise specified, has the following improvements:

- uses an environment variable to determine how to get the list of hosts: for instance, on the cloud head node the variable is set to return the cloud hypervisors with the onehost command;
- a preliminary check is performed to exclude hosts which do not appear as being available: this is to avoid failures when giving each command, as it happens with plain pdsh;
- it has a shortcut to retrieve the hostkeys in one go.
A.9 Customising the OpenWRT image: make-openwrt.sh

As illustrated in § 3.2.2.1, after building our custom Virtual Router configuration, the produced image needs further patches.

Instead of interfering with the OpenWRT build process, we decided to adopt an approach that makes any update of the OpenWRT codebase easier and more consistent: we inject our modifications directly into the produced image by manipulating it with guestfish\[^{53}\]. A list of the patches follows.

- A more modern web interface for LuCI is selected.
- The serial console is enabled in `/etc/inittab`.
- An initial boot script is added to fetch and interpret OpenNebula configuration from a virtual CD-ROM.

The build script also saves the image directly to the OpenNebula image repository automatically incrementing a “build number”: the resulting image will be visible with a name similar to:

```
OpenWrt-KVM-trunk-build23
```
Appendix B

Automatic addition and removal of Grid nodes

Some scripts have been created to manage automatic addition and removal of Grid nodes to the PBS/TORQUE queue and to the Puppet master. A description of each script follows.

B.1 Add a new node: create-node

As the name suggests, this script, installed on the TORQUE central manager, is used to add a new node to the list. The way of running this script is to associate a SSH public key to it from an unprivileged user:

1. `command="/path/to/create-node --remote" ssh-rsa AAAA...`

then the new remote virtual node, when booting, will invoke, with an appropriate private key:

1. `grep -c bogomips /proc/cpuinfo | ssh user@torquemaster -i userkey.pem`

which sends the number of cores of the new node via stdin to the create-node script. The script then retrieves the remote host name from the SSH_CLIENT environment variable and deletes a possibly stale node with the same name:

1. `qmgr -c "delete node <WORKERNODE>"

and re-adds it in "offline" mode with the number of cores read from stdin:

1. `qmgr -c "create node <WORKERNODE> state=offline,np=<NUM_CORES>,
2. properties=lcgpro,ntype=cluster"`
Please note that properties=lcgpro is a specific setting of our TORQUE installation.

The unprivileged user on the server must have proper TORQUE permissions to manipulate the list of nodes. To grant such permissions, from root:

```
qmgr -c 'set server managers += unpriv_user@master.host.com'
```

Another operation performed is the addition of the SSH public key of the host to the list of known ones on the central manager. To do so, the ssh-hosts-keys script, described later on in § B.3, is invoked. Since it requires root permissions, a special line must be added in the sudoers file. If the current unprivileged user name is “qmanager”:

```
Defaults!/full/path/to/ssh-hosts-keys !requiretty
qmanager ALL = NOPASSWD: /full/path/to/ssh-hosts-keys
```

### B.2 Delete or set offline a node: delete-offline-node

With the same SSH key mechanism used by the create-node script (see § B.1), it is possible to invoke the delete-offline-node script, which puts a certain node offline or it deletes it.

The node is not obtained via SSH environment variables though: it is read through stdin. So, to delete or put offline a certain node, you would do, with the appropriate SSH private key:

```
echo <WORKERNODE> | ssh user@torquemaster -i userkey.pem
```

There is a single script performing either the “delete” or the “set offline” operation. It is a multi-purpose script that has to be symlinked to select the proper operation:

```
ln -nfs delete-offline-node delete-node
ln -nfs delete-offline-node offline-node
```

When deleting a node, the following two operations are performed in order:

- node is deleted from TORQUE;
- the ssh-hosts-keys script is invoked to remove its key from the known ones (see § B.3).

When setting a node offline, the following command is invoked:

```
pbsnodes -o <WORKERNODE>
```
B.3 Update list of hostkeys: ssh-hosts-keys

In our configuration of TORQUE, hostbased SSH authentication is required for sending job data to the nodes, as no NFS shared directory is set. For this to work, two files need to be kept up to date on the central TORQUE master:

- `/etc/ssh/ssh_known_hosts`: the list of known SSH hosts with their public keys;
- `/etc/ssh/hosts.equiv`: the list of hosts that will be allowed to connect using hostbased authentication and their keys already in the list of known ones.

The ssh-hosts-keys script takes care of updating both lists automatically by reading the list of nodes from TORQUE. It also features a lock mechanism to prevent multiple runs of ssh-hosts-keys modifying the same files at the same time.

The script has three operational modes:

- in delete mode, it simply deletes the specified hostname from both of the aforementioned files;
- in update mode, the list of nodes is read from TORQUE and new nodes are added accordingly to the lists;
- in delete-update mode, the delete and update operations are executed in sequence.

The syntax is the following:

```
ssh-hosts-keys [delete|update|delete-update] [WORKERNODE]
```

where the `<WORKERNODE>` name is optional if running in update mode. The script must be run as root.

B.4 Delete node from Puppet: puppetca-clean-caller

This script is a wrapper to a single operation:

```
puppetca --clean <WORKERNODE>
```

and it must be executed from the Puppet master with proper privileges. As for the other management scripts, it is normally invoked via SSH from a remote worker node, with an appropriate private key: on the Puppet master, the corresponding public key is associated to the script.
The script automatically determines the `<WORKERNODE>` name from environment variables set by SSH, and it removes the certificate of a node with the same name, if it exists, from the list of issued certificates.

For security reasons, the script is associated to a public key of a special dummy user, which has the power of calling this script only via sudo. This means that the public key is associated to the command with sudo prepended:

```
command="sudo -u puppet /path/to/puppetca-clean-caller" ssh-rsa AAA...
```

where “puppet” is the name of the Puppet master authorized user. In the sudoers file, to allow running the script non-interactively and to preserve SSH environment variables, we must add lines similar to the following:

```
Defaults:puppetremote env_keep = "SSH_CLIENT SSH_CONNECTION"
puppetremote ALL = (puppet) NOPASSWD: /path/to/puppetca-clean-caller
```

where “puppetremote” is the dummy unprivileged user having its public key associated to the clean command. Note that the puppetca-clean-caller script must not be owned by the user “puppetremote” for security reasons: in such case, sudo would refuse to run it.
Appendix C

ALICE Analysis Facilities
configuration details

C.1 Configuring authentication and authorization

PROOF is configured to use the GSI authentication mechanism in AAFs. The following lines enable security and the GSI protocol, which is also configured to create delegated proxies on every PROOF node (-dlgpxy:2):

```
# The security library
xpd.seclib libXrdSec.so

# GSI protocol with proxy creation
xpd.sec.protocol gsi -cr1:0 -dlgpxy:2
-certdir:/path/to/valid/certauths -d:0
-cert:/path/to/hostcert.pem
-key:/path/to/hostkey.pem
-exppxy:<workdir>/<user>/.creds/x509_u<uid>
```

where variables in angled brackets are written literally and will be substituted properly by xproofd at runtime.

Mapping of certificate subjects to LDAP users is configured with the following lines:

```
# Users LDAP mapping
sec.protparm gsi -gmapfun:libXrdSecgsiGMAPLDAP.so
sec.protparm gsi -gmapfunparams:/path/to/ldap.conf
sec.protparm gsi -gridmap:/path/to/grid-mapfile
```
Those configuration lines point to two external files: an LDAP configuration file, containing all the parameters needed to contact the LDAP server:

```
srv: ldap://aliendb06a.cern.ch:8389
base: ou=People,o=alice,dc=cern,dc=ch
attr: uid
```

and a Grid “mapfile”, used to manually map certain certificate subjects to specific user names, primarily for administrative reasons:

```
"/C=IT/O=INFN/OU=Host/L=Torino/CN=pmaster.to.infn.it" admin,dberzano
```

In the above example, a host certificate subject is mapped either to the user `admin` or `dberzano`, depending on the name specified while connecting from the client.

Apart from the configuration lines needed directly by the `xproofd` daemon, many authentication variables are needed by each user’s PROOF master spawned by `xproofd`. The `xproofd` configuration file allows for passing variables in the environment of the spawned PROOF master by using the `xpd.putenv` directive, or in a configuration file specific for the PROOF master by using the `xpd.putrc` directive. The following variables are needed (variables in angled brackets will be substituted by PROOF):

```
xpd.putenv XrdSecGSIUSERPROXY=<workdir>/<user>/.creds/x509_u<uid>
xpd.putenv X509_USER_PROXY=<workdir>/<user>/.creds/x509_u<uid>
xpd.putenv XrdSecSSLUSERCERT=<workdir>/<user>/.creds/x509_u<uid>
xpd.putenv XrdSecSSLUSERKEY=<workdir>/<user>/.creds/x509_u<uid>
xpd.putenv XrdSecSSLCADIR=/path/to/valid/certauths
xpd.putenv XrdSecGSICADIR=/path/to/valid/certauths
xpd.putenv XrdSecGSISRVNAMES=fqdn.of.head.node.com
xpd.putenv alien_API_USER=<user>
xpd.putrc XSec.GSI.DelegProxy 2
```

### C.1.1 Users LDAP in RHEL 5

In order to use AliEn user names as local Unix user names, PROOF nodes are configured to use the AliEn LDAP as additional users database. In RHEL 5-compatible systems, there are two relevant configuration files:

- `/etc/ldap.conf` for LDAP parameters and mapping between remote entry fields and local user’s fields;
- `/etc/nsswitch.conf` to set up multiple Linux users databases and their priority (a local one and the LDAP).
The /etc/ldap.conf used for AAFs follows.

1. `suffix "ou=People,o=alice,dc=cern,dc=ch"`
2. `uri ldap://aliendb06a.cern.ch:8389/`
3. `timelimit 30`
4. `bind_timelimit 30`
5. `pam_filter objectclass=posixAccount`
6. `pam_login_attribute uid`
7. `pam_member_attribute memberuid`
8. `pam_password exop`
9. `nss_base_passwd ou=People,o=alice,dc=cern,dc=ch`
10. `nss_override_attribute_value loginShell /bin/bash`
11. `nss_override_attribute_value userPassword x`
12. `nss_override_attribute_value gidNumber 1395`
13. `nss_map_attribute uidNumber CCID`
14. `nss_reconnect_tries 4`  # number of times to double the sleep time
15. `nss_reconnect_sleeptime 1`  # initial sleep value
16. `nss_reconnect_maxsleeptime 16`  # max sleep value to cap at
17. `nss_reconnect_maxcontries 2`  # how many tries before sleeping

The /etc/nsswitch.conf file needs only one modification:

```bash
passwd: files ldap
```

meaning that the users database (passwd) is formed by a local file (i.e., the standard /etc/passwd) and the LDAP server configured in /etc/ldap.conf. The first database has priority: if a user is not found locally, then the database is queried.

In order to speed up LDAP queries, it is common practice to have the name service caching daemon (nscd) running. To do so:

```bash
chkconfig nscd on
service nscd start
```

The nscd caches DNS requests as well. To flush the cache:

```bash
service nscd reload
```

The bare `restart` function will not flush the cache.
C.1.2 Users LDAP in RHEL 6

Handling of LDAP users database changed dramatically from RHEL 5 to 6. In RHEL 6, the *system security services daemon (sssd)* is used. The configuration file `/etc/sssd.conf` for AAFs follows.

```conf
[sssd]
config_file_version = 2
services = nss, pam
domains = default

[nss]
filter_users = root,ldap,named,avahi,haldaemon,dbus,radiusd,news,nscl
override_shell = /bin/bash
override_homedir = /home/%u
#override_gid = 99

[pam]

[domain/default]
ldap_tls_reqcert = never
auth_provider = ldap
ldap_schema = rfc2307bis
ldap_search_base = ou=People,o=alice,dc=cern,dc=ch
ldap_group_member = uniquemember
id_provider = ldap
ldap_id_use_start_tls = False
ldap_uri = ldap://aliendb06a.cern.ch:8389/
cache_credentials = True
ldap_tls_cacertdir = /etc/openldap/cacerts
entry_cache_timeout = 600
ldap_network_timeout = 3
ldap_access_filter = (objectclass=posixaccount)
ldap_user_uid_number = CCID

To enable sssd:

```
authconfig --enablesssd --enablesssdauth --enableslocauthorize --update
service sssd restart
```

The `/etc/nsswitch.conf` needs proper modification as well: in RHEL 5 we added the `ldap` parameter to the `passwd` line, whereas here we add `sss`:
Since sssd has its own caching mechanism, we need to disable users and groups caching from nscd. In the `/etc/nscd.conf` file:

```
1   enable-cache passwd no
2   enable-cache group no
```

then the daemon needs to be restarted.

### C.2 Dataset stager (afdsmgrd) configuration

The configuration of the dataset stager (afdsmgrd) stays in one file, which can also be shared with PROOF or xrootd, as all the unknown configuration directives found are simply ignored. A list of the configuration directives understood by afdsmgrd follows.

#### C.2.1 Base configuration

```
set VARIABLE=value
```

Every occurrence of `$VARIABLE` will be substituted with its value in the rest of the configuration file. You can have multiple `set` statements.

```
xpd.stagereqrepo [dir:]directory
```

This directive is shared with PROOF: `directory` is the full path to the dataset repository. *Defaults to empty*: without this directive the daemon is not operative.

The `dir:` prefix is optional.

```
dsmgrd.purgenoopds true|false
```

Set it to `true` (*default is false*) to remove a dataset when no file to stage is found. If no file to stage is found, but corrupted files exist, the dataset is kept to signal failures. Used in combination with `xpd.stagereqrepo` makes it “disposable”: only the datasets effectively needed for signaling the staging status will be kept, improving scalability and stability.

```
dsmgrd.urlregex regex subst
```

Each source URL present in the datasets will be matched to `regex` and substituted to `subst`. `regex` supports grouping using parentheses, and groups can be referenced in order using the dollar sign with a number ($1 for instance) in `subst`. 
Matching and substitution for multiple URL schemas are supported by using in addition directives `dsmgrd.urlregex` up to `dsmgrd.urlregex4` which have the same syntax of this one.

Example of URL translation via regular expression. Configuration line:

```
1  dsmgrd.urlregex alien://(.*$) root://xrd.cern.ch/$1
```

Source URL:

```
1  alien://alice/data/2012/LHC12b/000178209/ESDs/pass1/
2  12000178209061.17/AliESDs.root
```

Resulting URL:

```
1  root://xrd.cern.ch/alice/data/2012/LHC12b/000178209/ESDs/pass1/
2  12000178209061.17/AliESDs.root
```

dsmgrd.sleepsecs secs

Seconds to sleep between each loop. The dataset stager checks at each loop the status of the managed transfers. Defaults to 30 seconds.

dsmgrd.scandseveryloops n

Every n loops, the dataset repository is checked for newly incoming staging requests. Defaults to 10.

dsmgrd.parallelxfrs n

Number of concurrent transfers. Defaults to 8.

dsmgrd.stagecmd shell_command

Command to run in order to stage each file. It might be whatever you want (e.g., executable or shell script). If you add `$URLTOSTAGE` and/or `$TREENAME` in the `shell_command`, they'll be substituted respectively with the destination URL and the default ROOT tree name in the file (as specified in the dataset staging request from ROOT). An example:

```
1  dsmgrd.stagecmd /path/to/afdsmgrd-xrd-stage-verify.sh
2  "$URLTOSTAGE" "$TREENAME"
```

Return value of the command is ignored: standard output is considered, as explained in § 4.3.5.2. Defaults to `/bin/false`. 
**C.2 – Dataset Stager (afdsmgrd) configuration**

### C.2.1 dsmgrd.cmdtimeoutsecs *secs*

Timeout on staging command, expressed in seconds: after this timeout, the command is considered failed and it is killed (in first place with SIGSTOP, then if it is unresponsive with SIGKILL). Defaults to 0 (no timeout).

### C.2.2 dsmgrd.corruptafterfails n

Set this to a number above zero to tell the daemon to mark files as corrupted after a certain number of either download or verification failures. A value of 0 (default) tells the daemon to retry forever.

### C.2.2 MonALISA monitoring plugin

The Dataset Stager supports generic monitoring plugins. The only plugin distributed with the stager is the MonALISA monitoring plugin.

**dsmgrd.notifyplugin /path/to/libafdsmgrd_notify_apmon.so**

Set it to the path of the MonALISA plugin shared object. By default, notification plugin is disabled.

**dsmgrd.apmonurl apmon://apmon.cern.ch**

This variable tells the ApMon notification plugin how to contact one or more MonALISA server(s) to activate monitoring via ApMon. It supports two kinds of URLs:

- http[s]://host/path/configuration_file.conf (points to a remote file where to fetch the list of servers from);
- apmon://[:password]monalisahost[:8884] (a single server to contact directly).

If the variable is not set, yet the plugin is loaded, MonALISA monitoring is inhibited until a valid configuration variable is provided.

**dsmgrd.apmonprefix MY::CLUSTER::PREFIX**

Since MonALISA organizes information in clusters and hosts, here you can specify what to use as cluster prefix for monitoring datasets information and daemon status. If this variable is not set, MonALISA monitoring is inhibited. Please note that the suffix _datasets or _status is appended for each of the two types of monitoring.
### C.2.3 A sample configuration file

```plaintext
1. `xpd.stagereqrepo /aaf/var/proof/datasets`
2. `dsmgrd.purgenoopds true`
3. `dsmgrd.urlregex alien://(.*)$ /storage$1`
4. `dsmgrd.sleepsecs 20`
5. `dsmgrd.scandseveryloops 30`
6. `dsmgrd.parallelxfrs 10`
7. `dsmgrd.stagecmd /aaf/bin/af-xrddm-verify.sh “$URLTOSTAGE” “$TREENAME”`
8. `dsmgrd.cmdtimeoutsecs 3600`
9. `dsmgrd.corruptafterfails 0`
```

### C.3 Enabling PROOF AliEn datasets: TDataSetManager-AliEn

TDataSetManagerAliEn is a PROOF dataset manager (§ 4.3.6) acting as an interface to the AliEn file catalog. In order to enable it, the `xpd.datasetsrc` directive must be specified in the PROOF configuration file:

```plaintext
1. `xpd.datasetsrc alien`
2. `cache:/path/to/dataset/cache`
3. `urltemplate:http://myserver:1234/data<path>`
4. `cacheexpiresecs:86400`
```

**alien**

Tells PROOF that the dataset manager is the AliEn interface (as opposed to file).

**cache**

Specify a path on the local filesystem of the host running user's PROOF master.

This path is not a URL but just a local path. Moreover, the path must be visible from the host that will run each user's master, since a separate dataset manager instance is created per user.

If the cache directory does not exist, it is created, if possible, with open permissions (rwxrwxrwx). On a production environment it is advisable to create the cache directory manually beforehand with the same permissions.

**urltemplate**

Template used for translating between an `alien://` URL and the local storage's URL.
<path> is written literally and will be substituted with the full AliEn path without the protocol.

An example on how URL translation works. Template URL:

```
root://alice-caf.cern.ch/<path>
```

Source URL:

```
alien:///alice/data/2012/LHC12b/000178209/ESDs/pass1/
12000178209061.17/AliESDs.root
```

Resulting URL:

```
root://alice-caf.cern.ch/alice/data/2012/LHC12b/000178209/ESDs/pass1/12000178209061.17/AliESDs.root
```

cacheexpiresecs

Number of seconds before cached information is considered expired and refetched (e.g., 86400 for one day).

C.3.1 AliEn datasets with PROOF-Lite

Unlike TDataSetManagerFile, the AliEn dataset manager can be used with PROOF-Lite.

By default, PROOF-Lite creates on the client session (which acts as a master as well) a file-based dataset manager. To enable the AliEn dataset manager in a PROOF-Lite session, run:

```
gEnv->SetValue("Proof.DataSetManager",
  "alien cache:/path/to/dataset/cache "
  "urltemplate:root://alice-caf.cern.ch/<path> "
  "cacheexpiresecs:86400");
TProof::Open("");
```

where the parameters meaning has been described in § C.3. The environment must be set before opening the PROOF-Lite session.
Appendix D

TAF management scripts

The “old” TAF (§ 4.3.7) is an Analysis Facility for ALICE exposing a standard AAF interface, but featuring different management procedures. The tools used to manage the TAF are illustrated in the next sections.

D.1 Monitoring: af-monalisa.pl

MonALISA\[^{28}\] is the monitoring tool used by ALICE. MonALISA is also used to monitor various parameters of all AAFs: some summary information is also available on a web page\[^{78}\].

The default MonALISA client is not available on the old TAF: a new and simpler client has been written in Perl to send monitoring data in an AAF-compatible format. The Perl client is configured to be invoked every minute by crontab and it uses the ApMon Perl bindings to send data to MonALISA.

The configuration file is a Perl script itself. A list of the configuration variables follows.

$masterHost
real hostname of the PROOF master

$proofConnectAlias
hostname of the PROOF master as seen from the outside

$apMonClusterPrefix
MonALISA organizes monitoring information in clusters: this is the cluster name to assign to the current AAF

$apMonHostPort
send monitoring data to host and port
The configuration file for TAF follows.

```
our $masterHost = "pmaster.to.infn.it";
our $proofConnectAlias = $masterHost;
our $apMonClusterPrefix = "PROOF::TAF::STORAGE";
our $apMonHostPort = "193.206.184.58:8884";
our $apStatus = "Virtually stable!";
our $apRootVer = "v5-34-13";
our $storagePath = "/storage";
```

The command can also be run manually. While sending out information, it will also display it on the standard output:

```
> /path/to/af-monalisa.pl
Jan 22 19:24:32 ApMon[INFO]: starting bg processes
the following additional options:
  Number of workers: [0]
  Number of sessions: [0]
  Status: [Virtually stable!]
  PROOF up: [1]
  ROOT version: [v5-34-13]
  xrootd version: [v3.0.4_dbg]
  Total space [Mb]: [51488716]
  Free space [Mb]: [596806]
```

While some information is set through the aforementioned variables, other information is retrieved automatically in real time:

- number of PROOF workers assigned per user;
- number of users connected (“sessions”);
- xproofd up or down;
- xrootd version.
D.2 Nodes addition and removal: af-proof-nodes.sh

The af-proof-nodes.sh script manages dynamic addition and removal of PROOF workers. It can be used in three different ways.

1. The script is invoked automatically when a new node is started or when an existing node is shutting down gracefully. The node also attempts to re-add itself periodically in cases where it was previously marked as down for some reason.

2. On the PROOF head node, it is periodically invoked as a cron job to check the health status of the nodes known by PROOF. In case some nodes cannot be reached, they are eliminated from the PROOF configuration.

3. It is also possible to invoke the script manually: this is mostly done to list nodes.

PROOF, as we have seen, is composed by xproofd daemons forking into ROOT sessions. Such daemons are based on the XRootD protocol: as long as the xproofd master daemon is running, every new xproofd worker will autoregister and deregister.

Whenever a PROOF session is started, the PROOF master is created first, which is a ROOT session. One of the first tasks is to scan the resources file (commonly called proof.conf), a file containing a single master line plus several worker lines, like this:

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master 188.184.135.182</td>
</tr>
<tr>
<td>2</td>
<td>worker dberzano@128.142.202.66 port=21001</td>
</tr>
<tr>
<td>3</td>
<td>worker dberzano@128.142.202.66 port=21002</td>
</tr>
<tr>
<td>4</td>
<td>worker dberzano@128.142.200.45 port=21001</td>
</tr>
<tr>
<td>5</td>
<td>worker dberzano@128.142.200.45 port=21002</td>
</tr>
</tbody>
</table>

Since the proof.conf file is read each time a new PROOF session is started, we can add worker nodes to PROOF without restarting xproofd daemons by having an external mechanism which updates this file—which is exactly what this file does.

The script resides on the PROOF master, and it is run automatically when performing SSH to it using a certain private key. The master has the corresponding public key associated to the command:

```
command="/path/to/af-proof-nodes.sh --remote" ssh-rsa AAAA...
```
Since the key is locked to one command, it is not even possible to specify additional switches: the \texttt{--remote} mode makes the script listening for commands on the standard input.

A node adds itself by issuing:

\begin{verbatim}
  echo add \texttt{<N\_WORKERS>} | ssh user@master -i userkey.pem
\end{verbatim}

which redirects the output of `echo` to the standard input of the script. The script detects the IP address and host name of the PROOF worker via SSH environment variables (namely, the `SSH\_CLIENT` variable). The `add` subcommand:

- adds the host’s SSH key fingerprint to the list of known hosts;
- adds the host to the `proof.conf` file with `<N\_WORKERS>` workers.

The script is safe against re-additions, meaning that a node is never added twice. When a node shuts down gracefully, it sends the following command:

\begin{verbatim}
  echo delete | ssh user@master -i userkey.pem
\end{verbatim}

which removes the node accordingly from the known hosts and the list of PROOF workers.

The script with the \texttt{--cleanup} option is executed periodically to check if some of the PROOF workers are down. To do so, `netcat` is used:

\begin{verbatim}
  for p in 22 1093 ; do nc -z proof-worker 1093 ; done
\end{verbatim}

The command attempts to create a TCP connection to the specified host and ports (namely, SSH and \texttt{xproofd} ports). If one of the connection attempts fails, the host is removed from `proof.conf`.

The script can be also used manually, mainly to list the available PROOF workers.

\section*{D.3 Creating ALICE PARfiles: \texttt{af-proof-packages.sh}}

PROOF supports special packages, called PARfiles (from PROOF Archive), to enable custom functionalities by shipping and loading extra code.

PARfiles must be present on the master, which takes care of supplying them to all workers. Users can upload their own custom PARfiles to the master if they want. In case of sysadmin-defined PARfiles, they must be placed to the workers manually.

A PROOF session is normally a bare ROOT session: to enable the AliRoot framework, the preferred method is to supply a PARfile which do not contain the
actual ALICE libraries (which are instead available on CernVM-FS), but the sole lines required to enable them.

The old TAF has a base AliRoot PARfile, which is copied many times, one for each AliRoot version available on CernVM-FS. Each PARfile will know by its name which is the correct version to enable.

Two scripts (a main script in Bash plus a helper script in Ruby) have been created to perform the following operations.

- Cross-check the list of available AliRoot versions from CernVM-FS.
- Remove PARfiles pointing to non-existing AliRoot versions.
- Create packages for the new ones.
- Send the new list of packages to the PROOF master and all workers.

Conveniently, the script has been placed in crontab to synchronize the list automatically. No system administrator’s intervention is needed when a new AliRoot version is released, unlike on other AAFs.

D.4 Synchronizing PROOF configuration: af-sync.sh

The first “old” TAF implementation relied on the Tier-2 Puppet (see § 2.4.1.2) to keep the PROOF configuration in sync between all PROOF workers.

To make the TAF independent from Puppet, we created the af-sync.sh script that deploys new TAF configuration to the PROOF nodes whenever it’s needed.

The TAF configuration as we have seen is composed by a single file controlling mostly the path prefixes. It is usually not necessary to change such configuration in production, but it is very useful for debug purposes.

D.5 Restarting PROOF on failures: init.d/proof

A special PROOF startup script has been created to start xproofd daemons and monitoring their stability. When starting PROOF, the script ensures the presence of special directories and proper permissions.

When PROOF starts, a lockfile is created in a standard place /var/lock/subsys/proof. The command:

```bash
service proof restart-if-down
```
checks if such file exists. If it does and PROOF is down, it restarts it.

The command is invoked every minute in the root’s crontab, and dramatically mitigates downtimes in case xproofd goes down for some reason.

### D.6 Contextualization

The “old” TAF receives contextualization data with the OpenNebula standard procedure, which consists in burning a virtual ISO image containing contextualization variables in a shell script (context.sh) plus all the files we want to include.

We contextualize by running a script (init.sh). The script is retrieved and executed by a special hook placed in the image’s /etc/rc.local, usually empty, which is the very last script being called during the boot:

```bash
export CONTEXT_DIR=/mnt/context
export CONTEXT_DEV=$(ls -1 /dev/vd? | tail -n1)
mkdir -p $CONTEXT_DIR
mount -t iso9660 $CONTEXT_DEV $CONTEXT_DIR > /dev/null 2>&1
if [ -f $CONTEXT_DIR/init.sh ]; then
    # CONTEXT_DIR and CONTEXT_DEV are available to the context script
    . $CONTEXT_DIR/init.sh
fi
umount $CONTEXT_DIR > /dev/null 2>&1
rm -rf $CONTEXT_DIR
```

The `init.sh` script performs a series of operations, divided in “stanzas”, each one represented by a Bash function.

**ConfigLocalFs**

Using LVM, the extra empty block device passed to the virtual machine as “ephemeral disk” is partitioned into a large working directory for PROOF sessions and the swap space. Partitions are also added to the /etc/fstab and filesystems are mounted.

**ConfigSharedSw**

The remote Lustre volume containing experiments software is mounted.

**ConfigSshHostKeys**

Removes this host’s SSH keys: this is usually overlooked when saving base images, but for security reasons each host must have different keys.

**ConfigSshKnownHosts**

Fingerprints of SSH public keys of the infrastructure services are placed in
the list of known hosts. This is for security reasons: the virtual machine
knows which services to trust when doing automatic SSH. In this case, au-
tomatic SSH is performed when self-adding the node to the PROOF cluster.

ConfigPuppet
Host is automatically added to Puppet. Puppet is then run once manually
and synchronously to retrieve possible configuration updates, then started
as a daemon. More details are in § 2.4.1.2

ConfigProofConf
Host self-adds to the PROOF cluster (§ D.2), then a cron job repeating the
self addition periodically is inserted. As we have said, the mechanism is
safe against double additions.

ConfigDeContext
A hook in the shutdown process is placed in order to remove the host from
the list of PROOF nodes.

ConfigCernVmFs
Updated CernVM-FS packages are installed. This is required since the base
virtual machine is CentOS 5, and not CernVM.

ConfigSharedData
The GlusterFS volume containing data (§ 4.3.7.2) is mounted.

ConfigDisableYum
Yum automatic updates are disabled. This is to ensure that the virtual ma-
chine represents as much as possible a certain "snapshot" of packages. Up-
dates are done by saving a new virtual machine image.
Appendix E

The Virtual Analysis Facility

Technical aspects of the configuration of the Virtual Analysis Facility are covered in this part, which also provides with proper scripts and examples.

E.1 HTCondor configuration

As we have seen in § 4.4.4.1, some default HTCondor configuration options needed to be changes in order to support running in an environment where we can rely only on the IP addresses to identify a node.

In general, HTCondor heavily relies on hostnames for a series of operations. In particular, domain names are used to establish whether a user is allowed to submit a job from a certain host.

E.1.1 No UID verification based on hostname

CONDOR_HOST = <CollectorIp>
This parameter represents the address of the host running the collector. Normally the FQDN is used: we fill this field automatically using the IP address of the central manager to prevent resolution problems.

UID_DOMAIN = *
HTCondor uses domain names to understand under which user to run jobs.
If the user has submitted a job from a host whose domain name is the one specified by this parameter, then its user ID is kept on the running machines. Elsewhere, the jobs will be run as user “nobody”. Setting this parameter to “*” preserves the original user ID in every case, without checking the domain name.

TRUST_UID_DOMAIN = True
The domain name of the submitting machine is trusted: no reverse lookup is performed to check it.

**SOFT_UID_DOMAIN = True**

When running jobs under a certain user ID, such ID is checked in the passwd database to see if it corresponds to an actual user. Unfortunately only the local /etc/passwd file is checked: however, when using for instance the Virtual Analysis Facility's LDAP plugin for authentication, actual users are not present in such file. This option is turned on to prevent jobs to fail if the submitter's user ID is not present in the local passwd database.

### E.1.2 Resilient connections and ports

**UPDATE_COLLECTOR_WITH_TCP = True**

Normally, HTCondor's collector uses UDP for network connections. Since on the cloud we cannot be sure about the network's quality or isolation, we are using TCP because it is more reliable. This is the usual setting when running HTCondor on WANs.

**COLLECTOR_SOCKET_CACHE_SIZE = 1000**

By default, the collector does not cache any connection (value is set to 0), meaning that the collector opens a TCP connection every time it needs to communicate. This number is the maximum number of collector's connection kept open in order to reuse them. This setting is very important to improve the efficiency of TCP updates, while UDP updates are connectionless and do not need it.

**HIGHPORT = 42000 and LOWPORT = 41000**

These settings extend the port range used by HTCondor to a maximum of 1000 ports: when using TCP for every update, the default settings do not provide a range which is wide enough, leading many connections to fail.

### E.1.3 No DNS lookup

The following options are set to turn off every DNS lookup in HTCondor. It must be noted that DNS lookups occur also while reading configuration files, so those options must be set in the first configuration file read by HTCondor, i.e. /etc/condor/condor_config. If the NO_DNS option is set in any other file (like the condor_config.local file or any file in config.d), it will be simply ignored.

**NO_DNS = True**

Turns off almost every DNS lookup in HTCondor. Cluster nodes are auto-
matically assigned a name based on their IP address and a manually chosen domain name, instead of doing any reverse lookup.

\[
\text{NETWORK\_INTERFACE} = \langle \text{CurrentIpAddress} \rangle
\]

This parameter must be set to the IP address used for HTCondor communications of the current host. It should not be mandatory to set this parameter: however, if not set, HTCondor will consume all the available system’s file descriptors, due to a bug\textsuperscript{[55]}. 

\[
\text{DEFAULT\_DOMAIN\_NAME} = \text{the-virtual-af}
\]

This parameter is set to a dummy domain name: it is mostly used to assign a name to every job slot when \texttt{NO\_DNS} is set to \texttt{True}. Since no reverse lookups are made, and all authentications based on DNS lookups are turned off, any dummy name would fit.

### E.2 PROOF on Demand patches and configuration

PROOF on Demand (PoD) needed some modifications to not use FQDNs on unreliable networks. The original code has been forked and patched, and the modifications were made public\textsuperscript{[46]}.

When creating the list of available workers for PROOF (the so-called proof.conf file), PoD uses the following configuration option:

\[
\texttt{[server]}
\begin{align*}
\texttt{proof\_cfg\_entry\_pattern} &= \texttt{worker \%user\%}\%host\% \texttt{port=\%port%}
\end{align*}
\]

where the pattern specifies to use hostnames to create worker entries. The resulting proof.conf will look like:

\[
\begin{align*}
\text{master vaf-master.cern.ch} \\
\text{worker username@vaf-node-01.cern.ch port=21001} \\
\text{worker username@vaf-node-01.cern.ch port=21001} \\
\text{worker username@vaf-node-02.cern.ch port=21001} \\
\text{worker username@vaf-node-03.cern.ch port=21001}
\end{align*}
\]

However, PoD had no option to allow using IP addresses in place of hostnames. In our patch we have added the \%ipv4\% pattern variable, plus the possibility to specify a pattern for the master line (proof\_cfg\_master\_pattern).

Moreover, since PoD attempts to connect to PROOF workers using hostnames, we have also added an option to tell PoD to use IP addresses instead. The new option is called use\_ip\_instead\_of\_host. The new appropriate configuration section looks like:
which results in something like:

<table>
<thead>
<tr>
<th>Line</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master 128.142.243.234</td>
</tr>
<tr>
<td>2</td>
<td>worker username@128.142.135.36 port=21001</td>
</tr>
<tr>
<td>3</td>
<td>worker username@128.142.202.36 port=21001</td>
</tr>
<tr>
<td>4</td>
<td>worker username@128.142.201.252 port=21001</td>
</tr>
<tr>
<td>5</td>
<td>worker username@128.142.202.76 port=21001</td>
</tr>
</tbody>
</table>

In our configuration, since we assume that all hosts can communicate to each other, we turn off PoD’s automatic packet forwarding feature in favour of using direct connections. By default, PoD autodetects if the remote PROOF daemons are directly accessible from the master. If not, it falls back to packet forwarding. By turning off the option we are suggesting PoD that, if a connection fails, it is not because of firewall issues but because of a faulty worker, and no attempt to revert to packet forwarding should be performed.

### E.3 Configuration of sshcertauth

This section provides some technical detail on the configuration of sshcertauth (§ 4.4.5) and all the components related to it—including, in particular, the web server and sshd. The source code of sshcertauth is available on GitHub.

#### E.3.1 Configuring Apache 2 for SSL authentication

In the architecture of sshcertauth, the first authentication step is performed by a properly configured web server.

The web server must be configured in a way that:

- it validates the user’s certificate;
- it provides the sshcertauth PHP script with proper SSL information.

The following variables must be configured.

- **SSLCertificateFile** and **SSLCertificateKeyFile**
  - Paths to the server’s certificate and private key, in PEM format.
SSLCACertificatePath
Path to the directory containing several PEM files, one for each authorized CA. Expected filenames are in the form 0123cdef.N, where the first part is the certificate's hash obtained via:

```bash
openssl x509 -in certificate.pem -noout -hash
```

and $N$ is a number.

SSLVerifyDepth
How many levels of issuers to check before deciding that the certificate is invalid. For example, if set to 0, it accepts only self-signed certificates; 1 means certificates directly issued by a known valid CA, and so on.

SSLVerifyClient require
Requires SSL valid authentication in the proper <Directory/> section hosting the sshcertauth PHP script.

SSLOptions +StdEnvVars +ExportCertData
Apache2 sets some environment variables available to the CGI applications. With +StdEnvVars, SSL authentication variables are sent as well: those include the variable storing the user certificate’s subject, which we need for mapping the certificate to a Unix user name. The +ExportCertData option exports the whole certificate in PEM format: we need it for extracting the public key and storing it in the list of SSH authorized keys.

A minimal example containing only the relevant configuration options follows.

```bash
# Certificate and key for this host
SSLCertificateFile /etc/grid-security/hostcert.pem
SSLCertificateKeyFile /etc/grid-security/hostkey.pem

# Certificates of the authorized CAs
SSLCACertificatePath /etc/grid-security/cadir

# Maximum certificate depth
SSLVerifyDepth 10

<Directory /var/www/html/auth>
  # Require SSL auth
```
Currently, in the Virtual Analysis Facility (VAF) setup, the following two directories on CernVM-FS is used to retrieve the list of valid Grid certification authorities. Either from ATLAS:

```
/cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase/etc/ grid-security/certificates
```

or from ALICE:

```
/cvmfs/alice.cern.ch/x86_64-2.6-gnu-4.1.2/Packages/ AliEn/<ALIEN_VERSION>/api/share/certificates
```

where <ALIEN_VERSION> is substituted with the latest AliEn version found.

### E.3.2 Keys keeper and sshd

The “keys keeper” is a shell (Bash) script used to manage the list of valid SSH public keys: its functionalities and features are covered in § 4.4.5.5.

The script is meant to be run as a privileged user using the following syntax:

```
keys_keeper.sh [addkey|expiry] [-k|--keydir <dir>] [-u|--user <user>] [-v|--verbose]
```

There are two working modes:

**addkey**

Adds the given key (from stdin) to the current user’s authorized keys. User is passed with the `--user` argument. If the current key is already present, its expiration date is updated. If not, the key is appended.

**expiry**

Scans the keys directory in order to remove expired keys. Those keys are removed from the files, and if a file has no more keys inside, it is removed. Keys without a valid expiration date signature are not managed and left alone: it is therefore possible to manually add custom keys without the fear of having them removed by error.
The script assumes that sshd is configured to look for authorized public keys for all users in a single directory: such directory contains many files, one for each user, whose file name is the user name. Every file supports multiple keys: when removing the last key from a file, the file itself will be removed, instead of being left empty.

In case the script is not run as root user, it attempts to escalate privileges using the `sudo` command. This means that, for non-interactive usage, an appropriate entry should be placed in the `/etc/sudoers` file to allow it to run as root without any password prompt. The sudo command should also be instructed to run safely in the background, without an actual tty attached (by default sudo refuses to run anything without a tty). To do the trick, the two following lines are used:

```
Defaults!/path/to/keys_keeper.sh !requiretty
apache ALL=(ALL) NOPASSWD: /path/to/keys_keeper.sh
```

The parameter `--keydir` is mandatory, but if not given an autodetect from the PHP configuration file (`conf.php` in the same folder of this script) is performed. For this to work, the command-line interface for PHP must be installed.

No status output on the current terminal is produced by default: output will be handled by the system logger (which will end up in `/var/log/messages` on most systems). If the verbose option is enabled, messages will be printed on stderr as well.

The script exits with an error state of zero on success, nonzero on failure.

**E.3.3 Extracting a public key to the SSH format**

The sshcertauth application is called by the web server with an appropriate environment. Such environment includes a variable containing the whole client’s certificate in a multi-line PEM format. The associated variable can be accessed in PHP with:

```
$_SERVER['SSL_CLIENT_CERT']
```

The `openssl` command is used first to extract the public key from the certificate: output will be in PEM format as well. This is the command that performs the conversion, expecting the certificate to come from the standard input:

```
openssl x509 -noout -pubkey
```

Then, openssl is invoked once again to decode the public key; input is always expected to come from stdin:

```
openssl rsa -pubin -noout -text
```
Output from the chain of commands is a human-readable description of the modulus and exponent of the RSA key, representing the public parts of the keypair. A sample output follows.

```
1 Modulus (2048 bit):
  08:ac:0d:81:48:80:d0:2e:66:eb:49:60:00:f5:24:
  c0:bb
20 Exponent: 65537 (0x10001)
```

The sshcertauth PHP application parses the modulus and the exponent, and assembles them into a single-line string representing the public key which follows a well-defined encoding\[141\]. This openssl-based solution has been chosen in favor of a “library” solution based on phpseclib\[87\] because of some constraints in the PHP version used in CernVM.

### E.3.4 Writing a users mapping plugin

A users mapping plugin is a PHP module containing a single function. The function takes the user subject as input, and stores in variables passed as references a valid Unix user name and the authentication validity in seconds.

The function must be called `authGetUser` and it has the following signature:
The last parameter, $errMsg$, is an array where the function is expected to append any error message. The function itself is considered successful when it returns true, unsuccessful otherwise.

In the configuration file it is also possible to specify a maximum validity of the authentication: whatever the value returned by the plugin to $validitySecs$ is, sshcertauth will always honour the maximum specified in the configuration file.

How to select the appropriate authentication plugin is explained in § E.3.6.

### E.3.5 Pool accounts mapping plugin configuration

The pool accounts plugin has some configuration options to determine the format of the numbered user names, the file where to save the mapping between subjects and pool accounts, and for how long it should keep the mapping.

- $mapFile$
  - mapping database writable by the webserver’s user

- $mapValiditySecs$
  - mapping validity, pruned after that many seconds

- $mapUserFormat$
  - format string for user account name: must contain a printf-like %u, which will be substituted with a pool account ID

- $mapIdLow$
  - lower boundary for pool ID

- $mapIdHigh$
  - upper boundary for pool ID

Sample configuration options for the pool accounts plugin are provided: in this example, pool accounts will be from pool000 to pool020. Once a user has been authenticated, within two days from the last authentication she will get the same pool account: this is the “mapping validity”.

```php
function authGetUser($certSubject, $userName, $validitySecs, $errMsg)
```

```ruby
$mapFile = '/tmp/x509-mapfile';
$mapValiditySecs = 172800; // 2 days
$mapIdLow = 0;
```
Such configuration options go in the only configuration file of sshcertauth, as explained in § E.3.6.

### E.3.6 Configuration file for sshcertauth

The following options can be configured in sshcertauth.

- **$suggestedCmd (optional)**
  - When authenticated, the user is suggested to type the command specified by this string to authenticate to the Virtual Analysis Facility. By default the string is empty, meaning that no command is suggested. The string can contain special variables that are properly substituted.

- **$sshPort (optional)**
  - Port where the SSH server is listening to. Used by the $suggestedCmd option to suggest via HTML the command that the user must use to connect.

- **$maxValiditySecs**
  - Maximum authentication validity, in seconds. The value returned by the authentication plugin is chopped to this value if greater.

- **$pluginUser**
  - Name of the user’s plugin. The specified file name, with the .php extension appended, will be loaded from the plugins/user directory.

- **$opensslBin**
  - Path to the openssl binary: by default, it is searched in the $PATH.

- **$sshKeyDir**
  - Directory containing the public keys, one per user, as explained in § E.3.3.
E.4 elastiq

The elastiq daemon (§ 4.4.7) is the daemon embedded in the Virtual Analysis Facility to provide elastic scalability of a virtual cluster. In the following sections we will go through the shell environment needed by elastiq, and its configuration file.

E.4.1 Environment and setup

The elastiq daemon is a single Python script requiring at least Python 2.7 to work. Unfortunately the default Python version shipped with SLC 6 (and therefore µCernVM) is Python 2.6.

On CernVM-FS, under the sft.cern.ch repository, there is the desired version of Python available, but no boto\[^{21}\] is installed. Currently, the contextualization procedure of the Virtual Analysis Facility manually installs boto with an environment where the non-default Python 2.7 is enabled.

To enable it, the /etc/init.d/elastiq script will load an environment, found in /etc/sysconfig/elastiq. The environment looks as follows:

```
#!/bin/bash

# elastiq -- by Dario Berzano <dario.berzano@cern.ch>
#
# See: http://github.com/dberzano/elastiq
#
# Environment for the elastiq daemon. This sample configuration file
# uses a configuration from CVMFS where available.
#
if [ -d '/cvmfs/sft.cern.ch' ] ; then
  # GCC
  source /cvmfs/sft.cern.ch/lcg/external/gcc/4.7.2/
  x86_64-slc6-gcc47-opt/setup.sh '' # empty arg needed!
  # Python 2.7
  export PythonPrefix=/cvmfs/sft.cern.ch/lcg/external/Python/2.7.3/
  x86_64-slc6-gcc47-opt
  export PATH="$PythonPrefix/bin:$PATH"
  export LD_LIBRARY_PATH="$PythonPrefix/lib:$LD_LIBRARY_PATH"
```
# Boto
export PyBotoPrefix='/var/lib/condor/boto'
export PATH="$PyBotoPrefix/bin:$PATH"
export LD_LIBRARY_PATH="$PyBotoPrefix/lib:$LD_LIBRARY_PATH"
export PYTHONPATH="$PyBotoPrefix/lib/python2.7/
                       site-packages:$PYTHONPATH"

# elastiq variables

# elastiqUser='condor'
export elastiqLogFileDir=$( cd "$PWD"/../var/log ; pwd )

fi

E.4.2 Configuration

The configuration file of elastiq is a ini file divided into four sections: let's see them in detail.

E.4.2.1 Sections

elastiq. Those options control the event loop of elastiq and the thresholds.

sleep_s
Time between two runs of the event loop.

cHECK_QUEUE_EVERY_s
Check interval for the HTCondor queue.

cHECK_VMS_EVERY_s
Check interval for the HTCondor virtual machines.

waiting_jobs_threshold
Minimum number of waiting jobs triggering a virtual machine execution.

waiting_jobs_time_s
Minimum number for the number of waiting jobs to stay above the threshold before new virtual machines are requested.
n_jobs_per_vm
   Number of jobs per each new virtual machine: this depends on the flavour, and it is used to estimate the number of “allegedly running” virtual machines (§ 4.4.7.4).

idle_for_time_s
   Virtual workers must be idle for that number of consecutive seconds before they are shut down.

estimated_vm_deploy_time_s
   Average time required for a new virtual machine to join the HTCondor cluster since it is requested. Used to estimate the “allegedly running” virtual machines and the virtual machines that “timed out” because they have not joined the cluster within that time.

check_vms_in_error_every_s
   Virtual machines launched by elastiq that went to the EC2 “error” state are identified and reinstated every that many seconds.

batch_plugin
   The batch system to monitor: for the moment only htcondor is supported.

**debug.**  Control debug options. For the moment it is possible to turn on the “dry run” for the deploy and shutting down EC2 commands.

dry_run_shutdown_vms
   If set to nonzero, virtual machines are not actually shut down, but a message is printed on the log.

dry_run_boot_vms
   If set to nonzero, virtual machines are not actually booted, but a message is printed on the log.

**quota.**  Minimum and maximum “soft” quotas.

min_quota
   Minimum number of virtual machines. Used to deploy always a certain number of workers, even when there is no load on the queue.

max_quota
   Maximum number of virtual machines running. Used when different virtual machines of different types run in the same tenancy.
**ec2.** This section contains all the parameters to interact with the EC2 interface, including the credentials and the features of the new virtual machines.

**api_url**
The API URL of EC2. It can be HTTPS, but the certificate is checked against a list of valid ones: if the endpoint certificate is invalid (mostly because it has been issued by an unknown CA), every EC2 command will fail.

**aws_access_key_id**
The ID identifying an EC2 user.

**aws_secret_access_key**
The password associated to the ID.

**image_id**
The AMI ID of the image of μCernVM.

**api_version**
This parameter is optional and it is used to force elastiq to use a specific version of the EC2 API: it is useful on some CloudStack deployments where the EC2 server is very strict in terms of compatibility.

**key_name**
Name of the root public key to authorize. Essentially used for after-boot management.

**flavour**
Flavour of the new virtual machines.

**user_data_b64**
The full user-data text, used for contextualization, converted to a single-line base64 string. This is very important, as it is the file that tells how the new virtual machines should be. In the Virtual Analysis Facility setup, it is automatically taken by the head node’s user-data with a couple of substitutions to tell the contextualization to configure a worker instead of a master node.

### E.4.2.2 Sample configuration file

```plaintext
[elastiq]
sleep_s = 5
check_queue_every_s = 15
check_vms_every_s = 45
```
The VAF client

The VAF client is a wrapper script around PoD providing easier access to some Virtual Analysis Facility functionalities (§ 4.4.8).

E.5.1 Loading the configuration files

The VAF client is constituted by the vaf-enter script, installed on the Virtual Analysis Facility and available in the $PATH.

As we have seen, the script sets both the “local” and the “remote” environment through a set of scripts. “Local” means the host running vaf-enter; “remote” is the host effectively running the PROOF master. If we are not using pod-remote, local and remote hosts are the same.

The general idea is that there is a central local.conf and remote.conf configuration script, with “experiment” settings, plus a series of scripts loaded
before or after the main ones. In addition, there are also “common” scripts, loaded both locally and remotely. As we have seen in § 4.4.8.2, the remote environment also includes a “payload” mostly used to transfer Grid proxy credentials to the worker nodes.

To clarify, let’s see the order in which environment scripts are loaded. For the local environment:

- common.before
- local.before
- local.conf, the “main” local configuration script
- PoD_env.sh, the PoD configuration file found in \$POD_LOCATION
- common.after
- local.after

For what concerns the remote environment:

- payload: it is *executed*, then its *output* (not its content) is loaded
- common.before
- remote.before
- remote.conf
- common.after
- remote.after

Scripts are searched in either of the following paths:

- a system-wide path: <dir_of_vaf-enter>/../etc
- ~/.vaf

where the system-wide scripts have precedence. If some experiment-specific settings are installed system-wide, user is prevented from overriding them.

If an environment file is not available it is silently skipped, except for the PoD_env.sh file that must be loaded to enable PoD.
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAF</td>
<td>ALICE Analysis Facility</td>
</tr>
<tr>
<td>ACL</td>
<td>Access Control List</td>
</tr>
<tr>
<td>AFS</td>
<td>Andrew File System</td>
</tr>
<tr>
<td>ALICE</td>
<td>A Large Ion Collider Experiment</td>
</tr>
<tr>
<td>AOD</td>
<td>Analysis Object Data</td>
</tr>
<tr>
<td>ATLAS</td>
<td>A Large Toroidal ApparatuS</td>
</tr>
<tr>
<td>BDII</td>
<td>Berkeley Database Information Index</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer Assisted Detection</td>
</tr>
<tr>
<td>CAF</td>
<td>CERN Analysis Facility</td>
</tr>
<tr>
<td>CA</td>
<td>Certification Authority</td>
</tr>
<tr>
<td>CERN</td>
<td>European Organization for Nuclear Research</td>
</tr>
<tr>
<td>CE</td>
<td>Computing Element</td>
</tr>
<tr>
<td>CMS</td>
<td>Compact Muon Solenoid</td>
</tr>
<tr>
<td>COW</td>
<td>Copy on Write</td>
</tr>
<tr>
<td>CRL</td>
<td>Certificate Revocation List</td>
</tr>
<tr>
<td>CT</td>
<td>Computed Tomography</td>
</tr>
<tr>
<td>DHCP</td>
<td>Dynamic Host Configuration Protocol</td>
</tr>
<tr>
<td>DICOM</td>
<td>Digital Imaging and COmmunications in Medicine</td>
</tr>
<tr>
<td>DNAT</td>
<td>Destination NAT</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name Server</td>
</tr>
<tr>
<td>EGI</td>
<td>European Grid Infrastructure</td>
</tr>
<tr>
<td>ESD</td>
<td>Event Summary Data</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FQDN</td>
<td>Fully Qualified Domain Name</td>
</tr>
<tr>
<td>FUSE</td>
<td>Filesystem in Userspace</td>
</tr>
<tr>
<td>GSI</td>
<td>Grid Security Infrastructure</td>
</tr>
<tr>
<td>HA</td>
<td>High Availability</td>
</tr>
<tr>
<td>HEP</td>
<td>High-Energy Physics</td>
</tr>
<tr>
<td>HLT</td>
<td>High Level Trigger</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>HTC</td>
<td>High Throughput Computing</td>
</tr>
<tr>
<td>ILP</td>
<td>Integer Linear Programming</td>
</tr>
<tr>
<td>INFN</td>
<td>Istituto Nazionale di Fisica Nucleare</td>
</tr>
<tr>
<td>IP</td>
<td>Interaction Point</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>JBOD</td>
<td>Just a Bunch of Disks</td>
</tr>
<tr>
<td>JRAF</td>
<td>JINR-Russia Analysis Facility, located in Dubna, Russia</td>
</tr>
<tr>
<td>KIAF</td>
<td>KISTI Analysis Facility, located in Daejeon, South Korea</td>
</tr>
<tr>
<td>KSM</td>
<td>Kernel SamePage Merging</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based Virtual Machine</td>
</tr>
<tr>
<td>LAF</td>
<td>Lyon Analysis Facility</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LDAP</td>
<td>Lightweight Directory Access Protocol</td>
</tr>
<tr>
<td>LFN</td>
<td>Logical File Name</td>
</tr>
<tr>
<td>LHCb</td>
<td>Large Hadron Collider Beauty</td>
</tr>
<tr>
<td>LHC</td>
<td>Large Hadron Collider</td>
</tr>
<tr>
<td>LS1</td>
<td>Long Shutdown 1</td>
</tr>
<tr>
<td>LVM</td>
<td>Logical Volume Manager</td>
</tr>
<tr>
<td>LV</td>
<td>Logical Volume</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access Control</td>
</tr>
<tr>
<td>MDS</td>
<td>Meta Data Server in the Lustre Filesystem</td>
</tr>
<tr>
<td>MIUR</td>
<td>Ministero dell’Istruzione, Università e Ricerca</td>
</tr>
<tr>
<td>MonALISA</td>
<td>MONitoring Agents using a Large Integrated Services Architecture</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System</td>
</tr>
<tr>
<td>OCA</td>
<td>OpenNebula Cloud API</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>OSS</td>
<td>Object Storage Server in the Lustre Filesystem</td>
</tr>
<tr>
<td>PEM</td>
<td>Privacy-enhanced Electronic Mail</td>
</tr>
<tr>
<td>PFN</td>
<td>Physical File Name</td>
</tr>
<tr>
<td>PROOF</td>
<td>Parallel ROOT Facility</td>
</tr>
<tr>
<td>PXE</td>
<td>Preboot Execution Environment</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a Service</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>PanDA</td>
<td>Production ANd Distributed Analysis, the ATLAS Grid middleware</td>
</tr>
<tr>
<td>PoD</td>
<td>PROOF on Demand</td>
</tr>
<tr>
<td>PoD</td>
<td>PROOF on Demand</td>
</tr>
<tr>
<td>QA</td>
<td>Quality Assurance</td>
</tr>
<tr>
<td>QGP</td>
<td>Quark Gluon Plasma</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RAID</td>
<td>Redundant Array of Inexpensive Disks</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>RSS</td>
<td>Resident Set Size</td>
</tr>
<tr>
<td>SAF</td>
<td>Subatech Analysis Facility, located in Nantes, France</td>
</tr>
<tr>
<td>SAN</td>
<td>Storage Area Network</td>
</tr>
<tr>
<td>SAS</td>
<td>Serial Attached SCSI</td>
</tr>
<tr>
<td>SATA</td>
<td>Serial Advanced Technology Attachment</td>
</tr>
<tr>
<td>SCRAM</td>
<td>Source Configuration, Release And Management, the CMS development environment</td>
</tr>
<tr>
<td>SDN</td>
<td>Software Defined Network</td>
</tr>
<tr>
<td>SE</td>
<td>Storage Element</td>
</tr>
<tr>
<td>SKAF</td>
<td>Slovakia Košice Analysis Facility</td>
</tr>
<tr>
<td>SRM</td>
<td>Storage Resource Manager</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Sockets Layer</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>TAF</td>
<td>Torino Analysis Facility</td>
</tr>
<tr>
<td>TCP</td>
<td>Transfer Control Protocol</td>
</tr>
<tr>
<td>TM</td>
<td>Transfer Manager</td>
</tr>
<tr>
<td>TPC</td>
<td>Time Projection Chamber</td>
</tr>
<tr>
<td>VAF</td>
<td>Virtual Analysis Facility</td>
</tr>
<tr>
<td>VMM</td>
<td>Virtual Machine Monitor</td>
</tr>
<tr>
<td>VNC</td>
<td>Virtual Network Computing</td>
</tr>
<tr>
<td>VO</td>
<td>Virtual Organization</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide-area Network</td>
</tr>
<tr>
<td>WIDEN</td>
<td>Web-based Image and Diagnosis Exchange Network</td>
</tr>
<tr>
<td>WLCG</td>
<td>Worldwide LHC Computing Grid</td>
</tr>
<tr>
<td>WMS</td>
<td>Workload Management System</td>
</tr>
<tr>
<td>vMSS</td>
<td>Virtual Mass Storage System</td>
</tr>
</tbody>
</table>
References


[38] Dorigo, A., Elmer, P., Furano, F., and Hanushevsky, A. XROOTD-A Highly scalable architecture for data access. WSEAS Transactions on Computers 1, 4.3 (2005).


[46] GitHub fork of PROOF on Demand. https://github.com/dberzano/PoD.


[51] GSI-enabled OpenSSH. http://grid.ncsa.illinois.edu/ssh/.


[99] **Salomoni, D.** Batch-like clouds. Workshop gruppo Cloud INFN—https://agenda.infn.it/materialDisplay.py?contribId=11&materialId=slides&confId=6920.


[109] **The CINT C Interpreter built into ROOT.** http://root.cern.ch/drupal/content/cint.

REFERENCES


Acknowledgements

First and foremost I would like to thank all of my supervisors: Massimo Masera, Stefano Bagnasco and Gerardo Ganis. I am grateful for the liberty and trust you gave me during the last years, and I hope that they eventually paid off. I reckon a Ph.D. is more than programming and writing papers: thank you for your mentorship and for the responsibility it took.

A big “thank you!” goes to the ALICE Collaboration, which has been more like a family to me (cousin rivalry and big lunches included): thanks to everyone I have known by e-mail before meeting them in person, thanks for the trust you give me by using my scripts, and thanks for giving me space on the support mailing lists.

Thanks to the ALICE working groups at the University and INFN in Torino that supported and tested the Virtual Analysis Facility way long before we released it publicly. Thanks in particular to our long time users: Stefania Bufalino, Elena Botta and Roberta Arnaldi.

Thanks to all the personnel of the Computing Center at INFN Torino. The project we have started there was originally meant to be the realisation of a mere cloud prototype, and it expanded into something much bigger. This happened, I like to believe, for a fortunate combination of human factors: the initiative and skills of Riccardo Brunetti, with whom I have shared the office, and whose company I am longing for; the cautious (or, maybe, “fake conservative”) spirit of Stefano Lusso, excellent system administrator; and the patience of Stefano Bagnasco (thanks for the second time), who picked the right time for advertising our work. Thanks to Sara Vallero who joined the team recently and managed to face a difficult learning curve by bringing significant contributions. Last but not least, thanks to Matteo Concas for the help he gave us both during his thesis work and the summer we spent together at CERN.

Thanks to all the CERN PH-SFT group that hosted be during the most critical period of my Ph.D.: it is impressive how such a small group manages to give
a fundamental support to all LHC experiments. Thanks in particular to all the members and contributors of the CernVM team: Gerardo Ganis, Predrag Bunčic, Jakob Blomer, Ioannis Charalampidis.

A very special thanks goes to René Meusel and George Lestaris, the best office mates I have ever had: thanks for all the support, for hearing my rants, for the good advice, for the long discussions on everything; and thanks in particular for maintaining such an unpretentious attitude (“I am not an expert on anything!”), which really does not make justice to your skills.

Thanks to Anar Manafov for bringing us PoD and for supporting my work on the Virtual Analysis Facility: it is funny how long time ago the VAF was meant to be an alternative to PoD, and it ended up constituting one of its core elements.

Thanks to Francesco Prelz for being a kind man and for his immense curiosity on almost everything.

Thanks to Piergiorgio Cerello for the opportunity to collaborate with him and his fascinating projects; it requires guts to take bold initiatives in a world where many people love to sit and complain.

During my path to getting a Ph.D. position I had the privilege to spend my time with some good persons that I have never had the chance to thank. First off, thanks to all the people I have worked with and met at Subatech, Nantes: in particular Diego Stocco, Guillaume Batigne and Gines Martinez. Special thanks to Laurent Aphecetche for his support and ideas on the data model for PROOF and to Jean-Michel Barbet because I have never met a system administrator that meticulous ever since.

Thanks also to the people at LPC, Clermont-Ferrand, and in particular Pascal Dupieux and Xavier Lopez, that have done more than they possibly could to make me feel at ease.

Many things changed in my life during the last years, and a lot of time has been spent doing my Ph.D. Thanks to my wonderful and smart wife Miriam for understanding this and sharing my company with my work. Most importantly, thanks for all the decisions we have been taking together since almost three years, especially the toughest ones, and for making me want to call “home” every place we will go.

Thanks to my family for the support: since some years I even have more persons I can count on and whose example matters to me.
Lastly, thanks to my friends: the ones that I have not seen in a while, the ones that are lost ("So long, and thanks for all the fish"), and the new ones. I owe a debt of gratitude especially to Jacopo and Laura for accepting to be part of the beginning of my pursuit of happiness, whatever it means and whatever it takes.