Heterogeneous Distributed Calibration Framework for the High Level Trigger in ALICE

Sebastian Robert Bablok

Thesis for the degree of Philosophiae Doctor (PhD)
at the University of Bergen

October 30, 2008

Abstract

This thesis focusses on the development of a distributed and re-configurable online calibration environment for the **ALICE** (**A** **L**arge **I**on **C**ollider **E**xperiment) **HLT** (**H**igh **L**evel **T**rigger). ALICE[1] is one of the four major experiments at the new accelerator ring (Large Hadron Collider - LHC) at CERN in Geneva (CH). The experiment is laid out to investigate the properties of strongly interacting matter created in heavy ion collisions at ultra-relativistic energies. It is assumed that the universe consisted of such a state of matter shortly after the Big Bang. The HLT is a sub-system of ALICE, focussing on online event reconstruction and analysis, event selection and data reduction. These tasks are executed on a large computing farm close to the experiment. To achieve high accuracy in the analysis, calibration is a major issue for the software components running in the HLT.

The thesis describes the interfaces and components developed for the HLT calibration framework. These interfaces cover the data exchange with the other ALICE sub-systems connected to the HLT. Their purpose is to provide calibration and condition settings to the HLT and to send freshly produced calibration objects to the destined targets. The exchange of data also requires synchronisation with the other sub-systems and within HLT itself. The interfaces (described in chapter 4.3, 4.4 and 4.5) and their interplay (depicted in chapter 5), as well as the FED-API (Front-End-Device-API) (specified in chapter 3) have been the main effort during this PhD project. How well they meet the imposed requirements is presented in the summary and outlook in chapter 6.

The outline of this thesis is the following: In the first chapter a very brief overview of the physics goal of the experiment is given. ALICE with its detectors and systems is presented in chapter two. Chapter three focusses on the FED-API, which deals with the connection of the ALICE DCS (Detector Control System) to the Front-End-Electronics (FEE) of the different detectors. The FED-API is (re-)used for the HLT as well. The different parts constituting the HLT calibration framework, – interfaces and their related components – are described in detail in chapter four. They are the main topic of the thesis. Especially the heterogeneity of the connected systems and the diversity of the used mechanisms are major aspects. Chapter five joins the different constituents and displays their interaction and synchronisation procedures. Moreover, examples for the usage of the HLT calibration framework and benchmarks for dedicated

[1] The ALICE Collaboration consists of more than 1000 members from 109 Institutes in 31 Countries (cf. `http://aliceinfo.cern.ch/Public/en/Chapter3/Chap3Collaboration-en.html`).

interfaces are presented. Finally, the results are summarised and an outlook on further developments and enhancements is given in chapter six. The appendix covers the usage of the presented interfaces. The calibration framework for the ALICE HLT, as it is described here, refers to the state of Autumn 2008.

Introduksjon

Denne doktoravhandlingen baserer seg på utviklingen av et fordelt og rekonfigurerbart online kalibreringsrammeverk for **ALICE** (**A** **L**arge **I**on **C**ollider **E**xperiment) **HLT** (**H**igh **L**evel **T**rigger). ALICE[2] er et av fire hovedeksperimenter ved den nye akseleratorringen (LHC - Large Hadron Collider) på CERN i Geneve (CH). Eksperimentet har til hensikt å studere egenskapene ved materie produsert i kollisjoner mellom tunge ioner i ultra-relativistiske energiområder. Man antar at universet var i en slik tilstand i de tidligste fasene etter Big Bang. HLT er et subsystem av ALICE som fokuserer på online event rekonstruksjon og analyse, event seleksjon og reduksjon av data fra ALICE. Dette arbeidet utføres ved en stor datamaskinfarm lokalisert nær eksperimentet. For å oppnå høy nøyaktighet i analysen er kalibreringen av komponentene som utfører HLTs oppgaver særlig viktig.

Avhandlingen beskriver grensesnittet og komponentene utviklet for HLT kalibreringsrammeverket. De dekker utveksling av data mellom HLT og de tilkoblede ALICE subsystemene. Deres oppgave er å utføre påkrevd kalibrerings- og betingelsesinnstillinger i HLT, samt å forsyne nylig produserte kalibreringsobjekter til bestemmelsesstedene. Datautvekslingen krever også synkronisering med de andre subsystemene og innen selve HLT. Grensesnittene (beskrives i kapittel 4.3, 4.4 og 4.5) og deres samspill (beskrives i kapittel 5), dessuten FED-APIen (Front-End-Device-API) (spesifisert i kapittel 3) har vært et hovedfokus i denne avhandlingen. Hvor godt de oppfyller nødvendige betingelser presenteres i kapittel 6.

I første kapittel gis en kort oversikt over fysikken bak ALICE eksperimentene. I kapittel 2 blir ALICEs detektorer og systemer presentert. Kapittel 3 fokuserer på FED-APIen som håndterer sammenkoblingen av ALICE DCS (Detektor Kontroll System) og Front-End-Elektronikk (FEE) på de forskjellige detektorene. FED-APIen er igjen brukt i HLT. Avhandlingens hovedtema, grensesnittene og de tilhørende komponentene som utgjør HLTs kalibreringsrammeverk, beskrives detaljert i kapittel 4. Spesielt er heterogeniteten av de sammenkoblede systemene og diversiteten i de anvendte mekanismene satt i fokus. I kapittel 5 bringes de ulike elementene sammen for å belyse deres samvirke og synkroniseringsprosedyrer. Utover dette presenteres her eksempler på anvendelsen av HLT kaliberingsrammeverket og målte referanseverdier. Avslutningsvis, i kapittel 6, foretas en oppsummering av resultatene, samt en fremtidsrettet vurdering av potensiell videreutvikling og forbedring. Appendikset gir en brukerveiled-

[2] ALICE Samarbeidet: 31 land, 109 institutt og mer enn 1000 medlemmer (se `http://aliceinfo.cern.ch/Public/en/Chapter3/Chap3Collaboration-en.html`).

ning for de presenterte grensesnittene. Kalibreringsrammeverket for ALICE HLT som beskrives her, gjelder status per høsten 2008.

Conventions used in this PhD thesis

This document uses several common typographical conventions.

To emphasise source code or documentation of the tools and applications, they are written in a `monospaced font`.

In this thesis several UML (Unified Modelling Language) diagrams are presented. The design and preparation of these diagrams has been guided by the following book describing UML 2.0:
M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins: ***UML 2 glasklar***, Carl Hanser Verlag, München (Germany), ISBN: 3-446-22575-7, 2004

If at the beginning something looks impossible, start with small steps ...

"Beginne damit, das
Nötige
zu tun.
Dann tue das
Mögliche
und plötzlich tust du das
***Unmögliche**."*

Franz von Assisi (zugeschrieben)

"Start by doing what is
necessary,
then do what is
possible,
and suddenly you are doing the
impossible."

ascribed to St. Francis of Assisi

Acknowledgements

In the development of the interfaces described within this thesis, I enjoyed very good cooperation and fruitful exchange of ideas with people of the other ALICE systems. I would like to say thank you to the involved persons from the ALICE DCS, the ALICE ECS and the ALICE Offline group. In addition I want to state my regards to the other people participating in the development of the FED-API.

And, of course, I experienced a perfect collaboration in the whole ALICE HLT group. Representatively, I would like to name here Prof. Dr. Volker Lindenstruth and Dr. Timm M. Steinbeck (the complete HLT collaboration is listed in the appendix C).

Moreover, I am thankful to the Norwegian Research Council (NFR) for supporting this project financially and for the chance of getting experiences abroad.

With this PhD thesis, I finally complete my student life. Along this road many people have accompanied me – during study, work and private life – family, friends, teachers, fellow students and colleagues (often taking more than one role):

My friends in Germany, who held fast the bonds of friendship. They and others showed me that distance and friendship are not necessary contradictory: Kerstin & Marcus Lohr, Frederic Hoffman, Anna Th. Cibis, Christian Kofler, Michael Schaaf, Larissa Bollinger, Eric S. Conner and Stephan Reichel. With joy I remember your phone calls, letters, post cards and visits. A special greeting goes to Cornelia "Conny" Ebitsch, who convinced me to *"start doing the necessary"*. Danke!

In my new home, Bergen, I have felt (and feel) welcome as well. Not only in science and work aspects, but also in social life activities the Nuclear Physics Group at the University of Bergen has made (and makes) my living here most comfortable. In the order they stepped into my life, these are: Ketil Røed, Matthias Richter, Are S. Martinsen, Kenneth Aamodt, Dag T. Larsen, Dr. Johan Alme, Prof. Dr. Joakim Nystrand, Prof. Dr. Håvard Helstrup, Prof. Dr. Kristin F. Hetland, Dr. Jens I. Jørdre, Dr. Hongyan Yang (*Xie Xie*, especially for the exotic taste discoveries in the Chinese cuisine), Gaute Øvrebekk (it has been (and is) a great time being office mates), Øystein Djuvsland, Kyrre Skjerdal, Camilla H. Stokkevåg (Tusen takk for alle de hyggelige og fine kaffepausene, og all norsk-hjelpen), Dominik Fehlker, Øystein S. Haaland, Dr. Boris Wagner, Hege Erdal, Kristian Ytre-Hauge, Henrik Qvigstad, Dana Huang and Lijiao Liu. With quite a few of you I share the predicate friends, even after this relatively short time, and I feel honoured by it.

My very grateful thanks go to Dr. Kalliopi "Kelly" Kanaki, for showing a lot of patience in explaining to me the ALICE related physics aspects, for the Mediterranean flair she brought in our group, for quite some delicious recipes and mainly for the unappreciative work of proof reading this thesis and for giving a wide array of useful suggestions for improvements: *ευχαριστώ πολύ*.

Furthermore, I also like to acknowledge Prof. Dr. Kjetil Ullaland for his very friendly and straightforward character, his never ending source of new ideas, the jokes during lunch time and for taking the job of being my co-supervisor. Mange takk!

There are three important persons without them I would have never started working on a PhD:

– Dr. Randolf Straky, without him I would have never considered the idea of taking a PhD; – Mein Freund, mögen unsere Abende weiterhin so ergiebig *"inspired"* sein.

– Prof. Dr. Ralf Keidel, who brought me to the ALICE project and supported me during my computer science studies and my Diploma thesis. Above all, he encouraged me in moving to Bergen and taking the PhD-stipend. His support has not stopped after leaving to Bergen – You deserve my honest thankfulness.

– Prof. Dr. Dieter Röhrich, who offered me the opportunity of working in this thrilling field of science and writing a PhD-thesis on the presented topic. His relaxed and kind manner and his ability of explaining complex physics properties in an easy and well understandable way has helped me a lot in finding my way through the relevant physics topics. Moreover, his style has made me interested in a subject, that does not actually belong to the roots of my studies. In permitting me to travel to the relevant conferences in India, China and Canada, he allowed me to contribute to the corresponding community and to experience other countries's culture as well. I always enjoyed the amicable atmosphere of him and his wife Dr. Bianca Ross, which reaches far beyond work life – You both have taken a major part in me feeling at home here. I will always perceive myself being connected to you in deep gratitude.

Finally, a deep and everlasting gratefulness I feel for my family. I thank my parents Brigitte and Wolfgang Bablok (I have always felt the assurance of your fatherly guiding hand), who supported and encouraged me to all intents and purposes. Your advice reached and helped me all over the world. Further, I wish to point out that it is perfectly wonderful having an elder brother: Frank Bablok. He and his family – my sister-in-law Ursula, my niece Sabine and my nephew Michael – have been at the ready whenever needed for backup and assistance or for a cheering up.

Bergen in October 2008

Sebastian R. Bablok

Contents

List of Figures

List of Tables

Chapter 1

Physics motivation

The Standard Model (SM) describes the properties of elementary particles constituting the universe as we see it today and the fundamental forces[1] that govern their interactions. It has also implications to our current understanding of the birth of our universe, the Big Bang, as well as astrophysical observations. Part of the SM is the theory of strong interactions between elementary particles, i.e. quarks. Gluons are mediating this fundamental force between the quarks. The theory of Quantum-Chromo-Dynamics (QCD) describes this behaviour.

Quarks are strongly interacting particles and bound by gluons inside hadrons[2]. Their characteristics are described by QCD. Two properties of QCD are asymptotic freedom and confinement. Asymptotic freedom states that interactions between quarks become weaker at smaller distances and increase the more the quarks are apart. This prevents us from observing individual quarks. Confinement means that the force between quarks does not diminish as they are separated, resulting in an infinite amount of energy needed to separate two quarks. In Lattice-QCD it is predicted that hadrons, set under conditions of high temperature and density, undergo a phase transition from a state of hadronic constituents to a plasma of unbound quarks and gluons, the Quark-Gluon-Plasma (QGP).

One way to reach high temperature and densities is by colliding heavy ions at ultra-relativistic energies, thus studying the primordial conditions of the Big Bang. Several accelerators with a large range of beam energies have been and will be used in laboratories all over the world, like CERN (SPS, LHC), GSI (SIS, FAIR) and BNL (RHIC). These different accelerators can access different parts of the phase diagram shown in figure 1.1.

The QGP is created in the initial state of heavy-ion collisions. It cannot be observed directly, its life time in the experiments is too short[3]. A set of observables have been identified as indicators for the QGP. These observables are jet quenching, collective flow patterns and yields of heavy quarks [1] [2].

[1] Three of the four known fundamental forces are included in the SM, gravity is excluded.

[2] Hadrons are built from quarks. Most common are the baryons to which the protons and neutrons with three quarks each belong.

[3] At the LHC the lifetime of the created QGP is expected to be $4 - 10\ fm/c$ [1].

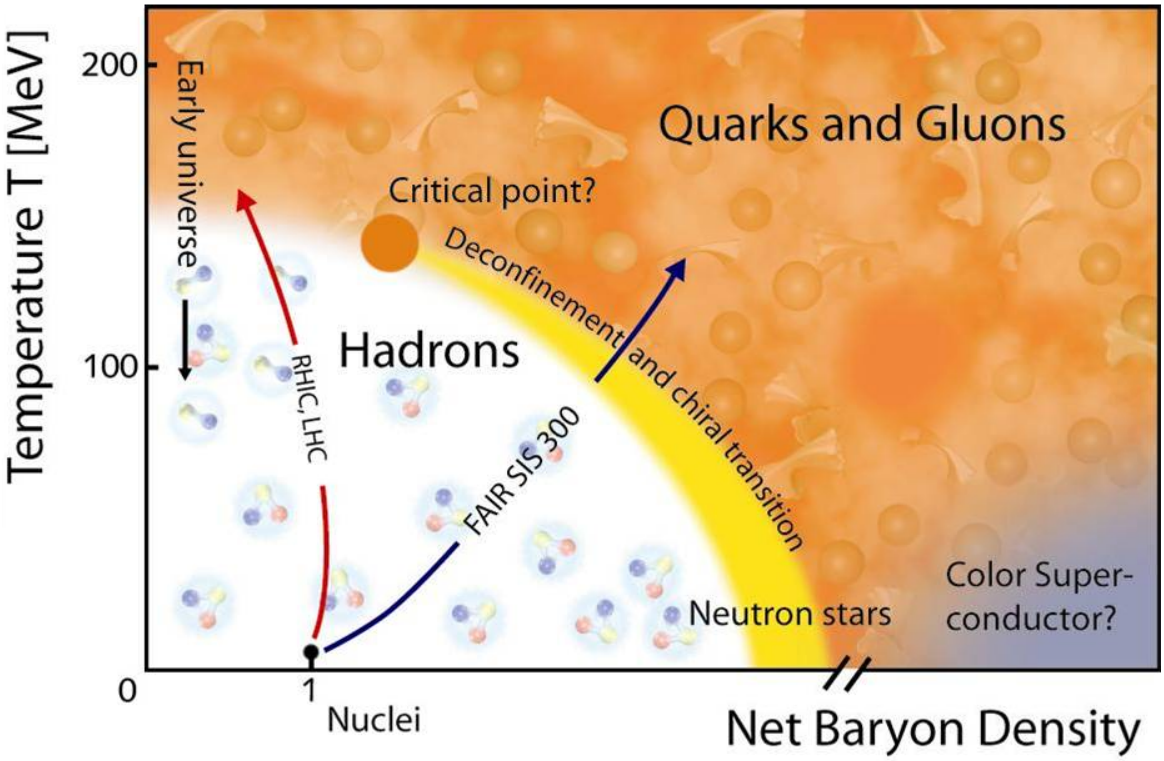


Figure 1.1: Phase diagram of strongly interacting matter. Different accelerators access different regions of the diagram (in temperature and net baryon density). The LHC aims for a region, where the state of the early universe can be studied.

Chapter 2

ALICE at the LHC

2.1 The LHC

The Large Hadron Collider (LHC) is the world's largest accelerator ring for hadrons at the moment. It is built in an underground tunnel at CERN in Geneva and has a circumference of 27000 m. The tunnel is at an average depth of 100 m. Bunches of particles circulate inside two beam pipes in opposite directions. In order to avoid beam - gas interactions there is ultra high vacuum inside the pipes. A sketch of the LHC ring is displayed in figure 2.1.

Superconducting magnets are used for bending and focussing of the beam. They are cooled down to 1.9 K by liquid helium. In sum the LHC uses 9593 magnets (154 dipole magnets are used for bending the beam), which create a B-field of about 8.33 T.

The ring is designed for acceleration of protons (p) and heavy ions (mainly lead (Pb), but other ions are foreseen as well). The hadrons inside the ring travel at close to speed of light (99.9999991 %). Collisions of $\mathrm{p}+\mathrm{p}$ will reach an energy of 14 TeV in the centre of mass system. The design luminosity of the LHC in proton beams is $\mathscr{L}= \mathit{10^{34}}\ \mathit{cm^{-2}s^{-1}}$. In heavy ion collisions an energy of 5.5 ATeV (centre-of-mass energy per nucleon pair) is reached[1]. A luminosity of $\mathscr{L}= \mathit{10^{27}cm^{-2}s^{-1}}$ is expected for the Pb beam in the LHC.

The hadrons inside the accelerator ring are circulated in bunches[2]. To collide the hadrons in the centre of the experiments the two beams are aimed at each other by dedicated focussing magnets around the experiment locations [3]. The LHC operation has started in September 2008, first collisions are expected for spring 2009.

2.2 ALICE – an overview

ALICE stands for ***A*** ***L***arge ***I***on ***C***ollider ***E***xperiment. It is one of the four large experiments of the LHC. The other three experiments are ATLAS (*A* *T*oroidal *L*HC *A*pparatu*S*), CMS (*C*ompact *M*uon *S*olenoid) and LHCb (*L*arge *H*adron *C*ollider

[1] This sums up to 1150 TeV as total energy in $\mathrm{Pb}+\mathrm{Pb}$ collisions.

[2] It is planned to have 1.1×10^{11} protons per bunch.

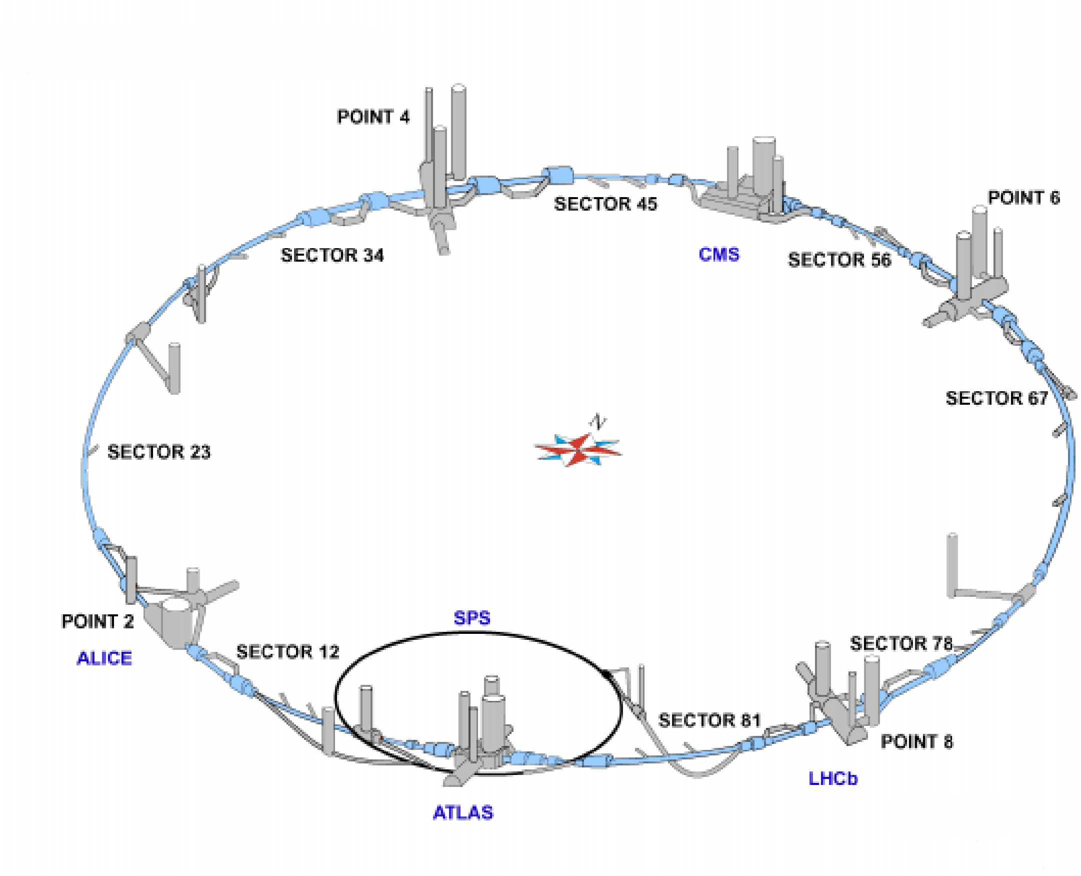


Figure 2.1: The figure displays the LHC ring with the location of the four major experiments. Hadrons are injected into the LHC in both directions by the SPS (Super Proton Synchrotron), where they are pre-accelerated.

*b*eauty). Two smaller experiments are also built at the LHC: TOTEM (*TOT*al *E*lastic and diffractive cross section *M*easurement) and LHCf (*L*arge *H*adron *C*ollider *f*orward).

For ALICE the LHC will provide a $\mathrm{p}+\mathrm{p}$ collision rate of about 200 kHz[3]. The $\mathrm{Pb}+\mathrm{Pb}$ interaction rate is 8 kHz. While the other three major experiments are designed for $\mathrm{p}+\mathrm{p}$ collisions, ALICE mainly focuses on heavy ion collisions, where the QGP shall be investigated. But $\mathrm{p}+\mathrm{p}$ and $\mathrm{p}+\mathrm{A}$ collisions must be measured and analysed as well. Thereby ALICE is able to handle the above mentioned interaction rates. The rate of $\leq$ 200 kHz in $\mathrm{p}+\mathrm{p}$ collisions is limited by the pile up of events in the Time Projection Chamber (TPC) barrel. It is able to cope with the highest particle multiplicities anticipated for $\mathrm{Pb}+\mathrm{Pb}$ collisions in the LHC. The first estimate had been $dN_{ch}/dy \approx 8000$, but latest results from RHIC indicate a lower multiplicity of $dN_{ch}/dy \leq 3000$ at the LHC.

[3] The LHC achieves an interaction rate of 40 MHz in $\mathrm{p}+\mathrm{p}$, which is used in ATLAS and CMS. For ALICE the interaction rate is reduced by temporarily defocusing the beam in order to allow ALICE handling the pile up in the Time Projection Chamber (TPC).

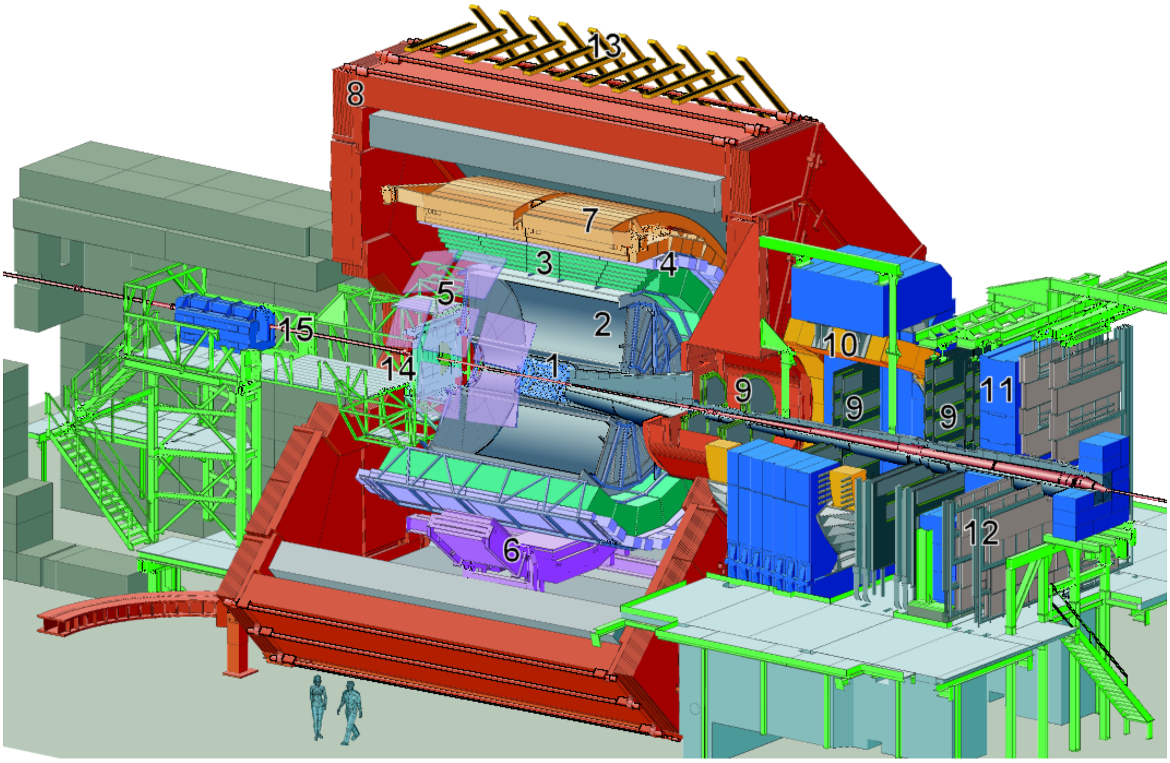


Figure 2.2: The overview presents the assembly of ALICE detectors:
1) ITS with its detectors SPD, SDD and SSD sits around the interaction point in the centre of ALICE;
2) TPC barrel;
3) TRD supermodules;
4) TOF supermodules;
5) HMPID, which covers only a small region on one half upper location inside the ALICE solenoid magnet;
6) PHOS is sited below the interaction point;
7) EMCal will be locate opposite azimuthal to PHOS and covers a larger area;
8) L3 Solenoid magnet surrounds all major barrel detectors;
9) Muon Tracker Chambers (5 chambers);
10) dipole magnet for the measurement of the muon momenta;
11) Muon Filter, a massive iron block;
12) Muon Trigger Chambers (2 chambers);
13) ACORDE on top of the L3 magnet for cosmic ray detection;
14) Forward Detectors including FMD, PMD, V0 and T0;
15) beam pipe of the LHC.
The ZDC is not in the figure. It is located 116 m on both sides of the interaction point between the beam pipe inside the LHC tunnel.

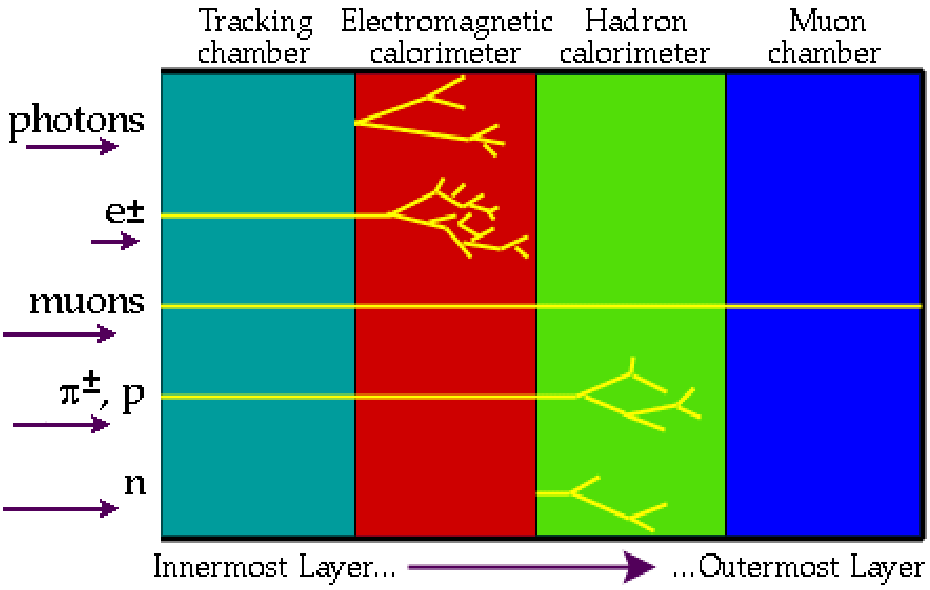


Figure 2.3: For the detection of different particles different techniques are applied. The presented sequence of techniques is typical for experiments in high energy physics.

For particle identification ALICE uses a broad variety of different techniques (a general sketch is shown in 2.3) [2] [4] [5]:

- ionisation energy loss,
- transition radiation,
- Time-Of-Flight,
- Cherenkov radiation,
- electromagnetic calorimetry.

ALICE is designed to have the interaction point surrounded by a solenoidal magnet, which includes all major tracking and particle identification barrel detectors. The magnet is reused from the old L3 experiment of the LEP (Large Electron-Positron Collider) ring and produces a modest magnetic field of 0.5 T. The field is required to measure the momentum of charged particles inside the ALICE barrel.

ALICE has overall dimensions of 26 m × 16 m × 16 m (given by the L3 magnet and the DiMuon arm) and weights approximately 10000 t. Figure 2.2 sketches the setup of ALICE and its main detectors [1] [6] [7] [8].

2.3 The ALICE detectors

The following sections describe briefly the ALICE detectors, their location and acceptance and detection methods. The official ALICE coordinate system is a right-handed

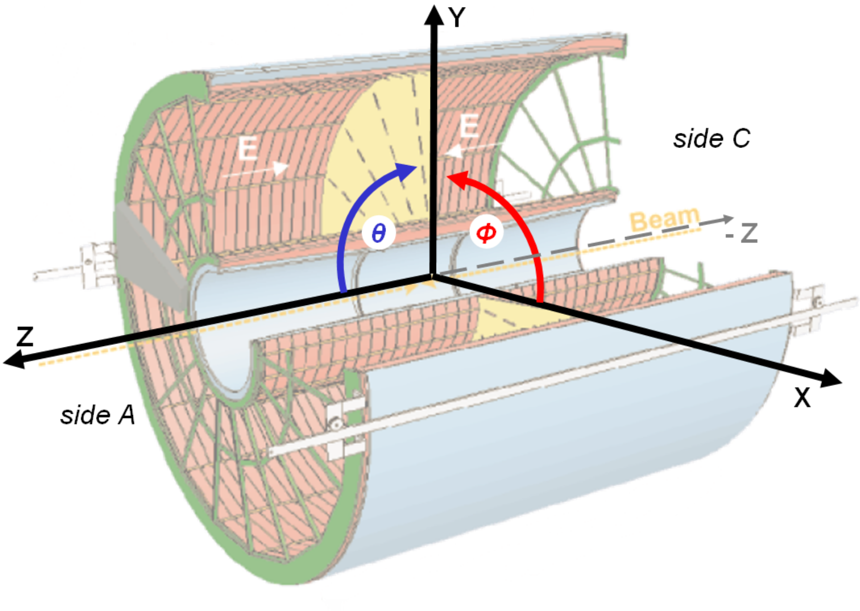


Figure 2.4: For the description of the location and coverage of the detectors in ALICE the angles θ and ϕ are used. The azimuthal angle ϕ is located in the **X**-**Y** plane, the polar angle θ in the **Z**-**Y** plane. The positive **X** axis is pointing to the centre of the LHC ring.

orthogonal Cartesian system. The axes X, Y and Z have their origin at the nominal interaction point $(X,\ Y,\ Z)\ =\ (0,\ 0,\ 0)$.

The azimuthal angle ϕ is counted from positive X to the Y axis and the polar angle θ from positive Z to the Y axis. The scheme is visualised in figure 2.4 [9]. For the acceptance pseudo-rapidity (η) is used, which can be calculated from a given θ angle by the formula: $\eta = -ln\left[tan\left(\frac{\theta}{2}\right)\right]$ [4].

2.3.1 ITS

The Inner Tracking System (ITS) is closest to the interaction point. The beam pipes of the LHC enter the ITS from both sides and the beams are supposed to cross each other for collisions in the very centre of its cylindrical geometric structure. The ITS cylinder has a diameter of $\approx$ 90 cm, where the inner 6 cm are taken by the beam pipe. It contains three different detectors: Silicon Pixel Detector (SPD), Silicon Drift Detector (SDD) and Silicon Strip Detector (SSD). Each detector consists of two layers of silicon detectors. The inner most is the SPD. It is designed for detection of primary vertices, as well as secondary tracks from weak decays of strange, charm and beauty particles. Second comes the SDD with layer three and four. Finally the SSD with silicon detector layer five and six builds up the outer part of the ITS. Drawings about the position and the layout of the ITS are shown in figures 2.5 and 2.6.

As the name ITS already indicates, their main purpose is the reconstruction of the primary and secondary vertices. The ITS has a pseudo-rapidity acceptance of $|\eta| < 0.9$, for the most inner pixel layer the coverage is $|\eta| < 1.98$. The outermost layer of

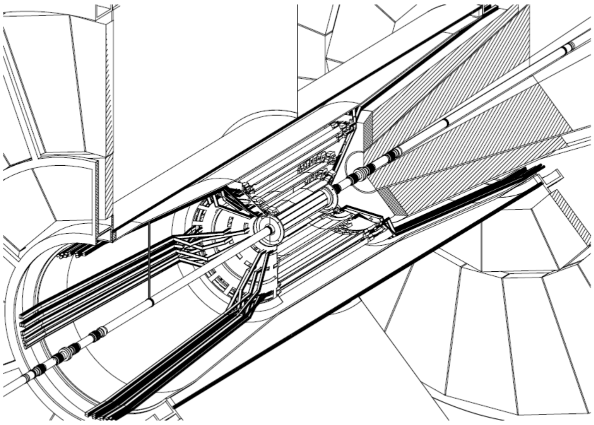


Figure 2.5: With its three silicon vertex detectors (SPD, SDD and SSD) the ITS sits centrally inside the TPC barrel around the beam pipe and the interaction point.

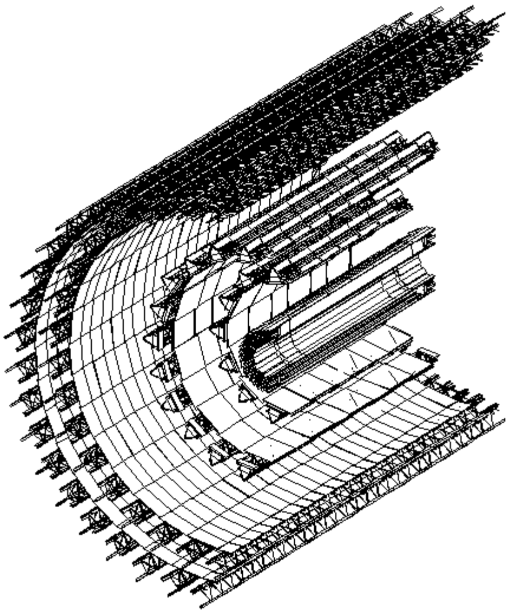


Figure 2.6: Overview of the ITS detectors: The inner two layers belong to the Silicon Pixel Detector, surrounded by the two layers of the Silicon Drift Detector. The outer two layers constitute the Silicon Strip Detector.

the SSD is essential for matching the tracks to the ones reconstructed by the TPC, the surrounding detector. ITS is able to detect simultaneously up to 15000 tracks. With the outer four layers the ITS can perform a first particle identification of low momenta particles (< 100 MeV/c) via dE/dx measurements as well [1] [7] [8] [10].

2.3.2 TPC

The ITS is surrounded by the TPC. With its cylindric barrel (inner radius (r_{in}): $\approx$ 90 cm, outer radius (r_{out}): $\approx$ 250 cm, length along the beam axis: 500 cm) the TPC is the largest tracking detector in ALICE for charged particles. It consists of a large field cage, which is filled with a high purity gas mixture of $\mathrm{Ne/CO_2/N_2}$ (90/10/5). Charged particles crossing the TPC ionise the gas along their path. Due to a high voltage of 100 kV, which is applied to the field cage along the beam direction, the electrons from the ionisation drift towards the end plates of the TPC[4], where the signal is amplified and collected at about 280 000 readout pads on each side. These pads are connected to Front-End-Cards (FEC), which are responsible for the actual readout of the detector data. The procedure of measuring tracks of charged particles inside the TPC is sketched in figure 2.7.

Each side of the TPC (end plates) is divided into 18 trapezoidal sectors. The sector

[4] The maximum drift time in the TPC is about 90 μs.

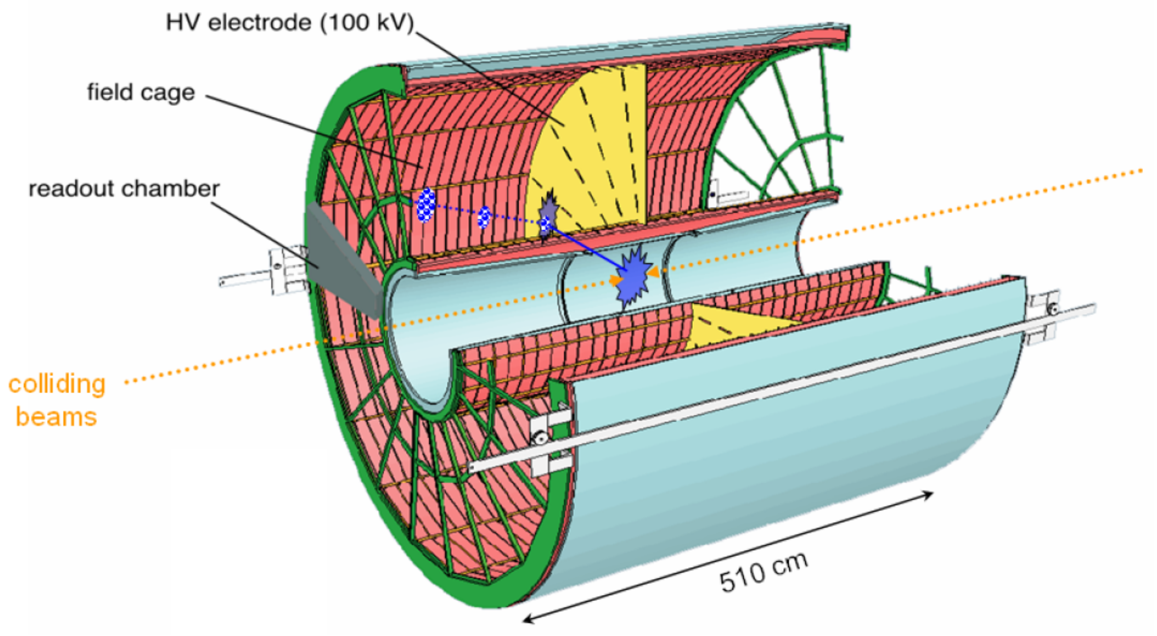


Figure 2.7: This figure of the TPC displays how the track of a charged particle is measured in the TPC central barrel. The electrostatic field along the beam line forces the electrons produced from ionisation to drift towards the readout chambers (end plates).

covers 2 Multi Wire Proportional Chambers (MWPC), the Inner ReadOut chamber (IROC)) and the Outer ReadOut Chamber (OROC). The IROC subdivided into 2 partitions, the OROC into 4 partitions. A ReadOut Control Unit (RCU) controls each partition and is connected to 18 - 25 FECs. For configuration and monitoring of these electronics, the RCU hosts a dedicated card, the DCS board. The latter one plays an important role in the later described TPC Front-End-Electronics (FEE) control chain, which is described in chapter 3. The recorded event data is sent over optical fibres to the Data Acquisition System (DAQ) by a so called Source-Interface-Unit (SIU), which is hosted by the RCU as well.

The TPC has an acceptance of $|\eta| < 0.9$ for the full track length and for reduced track length of $|\eta| < 1.5$. It is able to cope with a central collision rate of 200 Hz. The TPC allows also for identification of particles via $\mathrm{d}E/\mathrm{d}x$ measurements [1] [7] [8] [11].

2.3.3 TRD

The Transition Radiation Detector (TRD) provides full azimuthal coverage over more than the total length of the TPC barrel. This corresponds to a pseudo-rapidity acceptance of $|\eta| < 0.84$. Its main purpose is electron identification with momenta of higher than 1 GeV/c. In addition it acts as a fast trigger for charged particles with high momentum.

The TRD consists of 18 supermodules, which match the outline of the 18 TPC

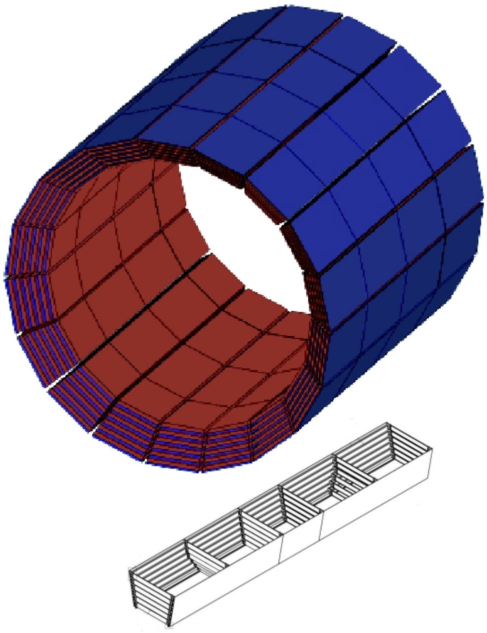


Figure 2.8: The TRD consists of 18 supermodules (lower part of the figure), located around the TPC barrel and matching the outline of the 18 sectors of the TPC.

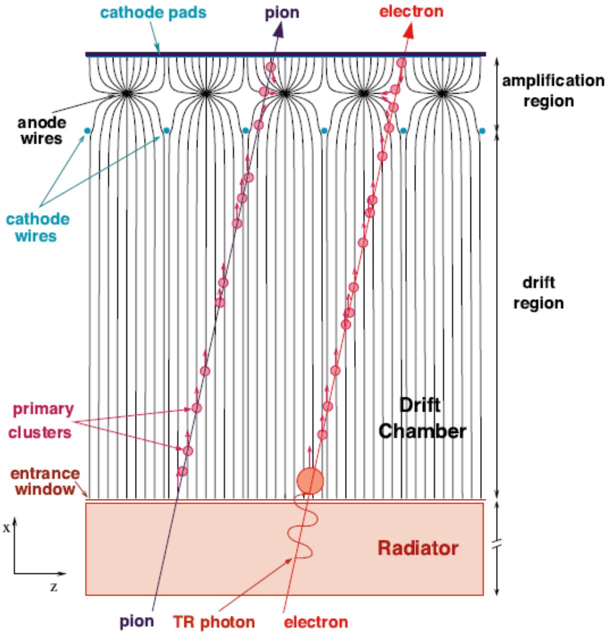


Figure 2.9: The working principle of the TRD: electrons emit photons while passing through a radiator. These photons are measured via the ionisation of the gas in the drift region.

sectors. Each supermodule has 5 stacks with 6 layers each. The layers consist of a carbon fibre laminated Rohacell / polypropylene fibre sandwich radiator, a drift section and a multi-wire proportional chamber for the readout. The TRD is filled with a gas mixture of Xe/CO$_2$ (85/15). The readout electronics are mounted directly on the detector back panel and controlled via dedicated DCS boards (see description of the DCS board on page 29). The TRD structure is shown in figure 2.8.

Electrons passing through the radiator emit photon radiation. These photons ionise the gas in the drift section. The resulting electrons as well as electrons from ionisation due to collisions are measured in the MWPC as shown in figure 2.9. This results in a clear electron - pion separation [1] [7] [8] [12].

2.3.4 TOF

The Time-Of-Flight (TOF) detector is located around the TRD supermodules. Like the TRD it has 18 supermodules each divided into 5 segments. Therefore it covers nearly the same pseudo-rapidity region ($|\eta| < 0.9$). It is designed for particle identification in the intermediate momentum range (up to 2.5 GeV/c for pions and kaons and 4 GeV/c for protons). The TOF in ALICE is a gaseous detector with Multi-gap Resistive-Plate Chambers (MRPC), filled with $C_2H_2F_4/i-C_4H_{10}/SF_6$ (90/ 5/ 5) [1] [7] [8] [13].

2.3.5 PHOS

The PHOton Spectrometer (PHOS) is an electromagnetic calorimeter with a limited coverage area at midrapidity. It consists of five modules with 56 × 64 channels for detection of photons. Each channel is built of a lead-tungstate ($PbWO_4$) crystal with the dimensions of 22 × 22 × 180 mm^3, an Avalanche Photo-Diode (APD) and related readout electronics. The acceptance of the PHOS covers 0.24 of a unit in pseudorapidity ($|\eta| < 0.12$) and 100^o in the azimuthal angle at the bottom of ALICE, 460 cm away from the interaction point. It spreads over an area of ≈ 8 m^2. The layout of the five PHOS modules is shown in figure 2.10. Since PHOS uses similar FEE like the TPC, its control and monitor system is also similar (see section 3.2 for more details).

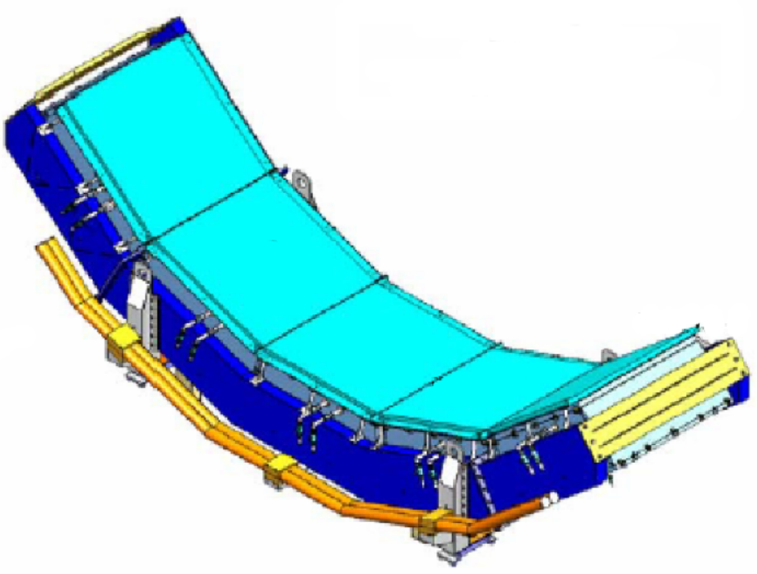


Figure 2.10: Layout of the five modules of the PHOS detector with the CPV on top.

The PHOS detector is designed to detect photons in the range from 0.5 to 100 GeV. Photons entering the $PbWO_4$ crystals initiate an electro-magnetic shower. The resulting scintillation light is measured by APDs attached to each crystal.

There are plans to add a Charged Particle Veto (CPV) detector on top of each module. The CPV consists of MWPCs with a cathode pad readout. It will be used to separate the measurements of charged and neutral particles in the PHOS module. The CPV is counted as a separate detector [1] [7] [8] [14].

2.3.6 HMPID

The High Momentum Particle Identification Detector (HMPID) is built of seven modules (1.5 × 1.5 m^2) located in a single-arm array at the two o'clock position in the ALICE barrel, 4.8 m away from the interaction point. It has an acceptance of 5 % of the central barrel phase space ($|\eta| < 0.6$, $1.2^o < \phi < 58.8^o$). Figure 2.11 shows the layout of the seven HMPID modules.

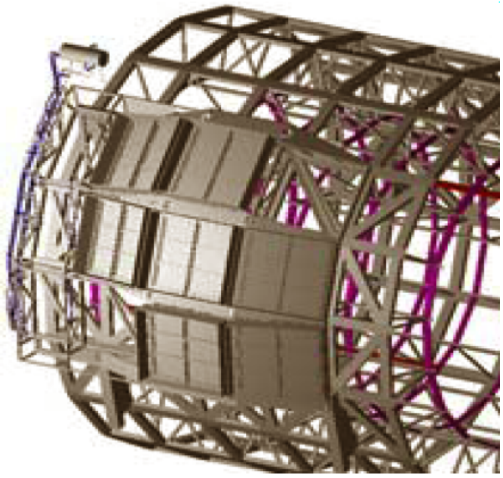


Figure 2.11: The seven modules of the HMPID are mounted on the space frame at two o'clock position inside the ALICE barrel.

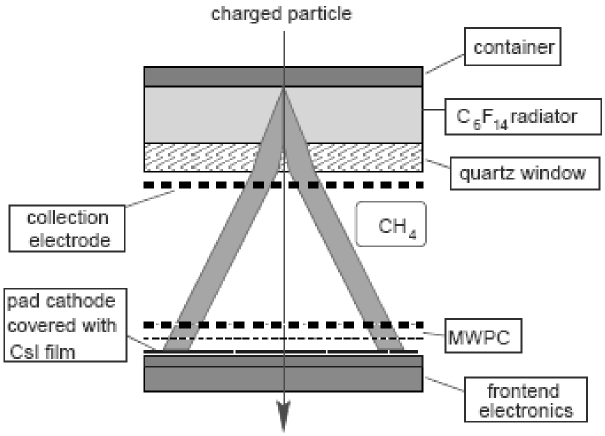


Figure 2.12: HMPID working principle: A charged particle with high momentum crossing the radiator emits Cherenkov photons, which are measured in the MWPC via photocathodes.

The HMPID modules consist of a liquid radiator (C_6F_{14} (Perfluorohexane)), a gas chamber with CH_4, a MWPC and solid CsI (Caesium Iodide) photocathodes. The HMPID identifies hadrons with high momentum ($p_t > 1$ GeV/c). Its detection mechanism is based on proximity-focusing Ring Imaging Cherenkov (RICH) counters. When a fast particle traverses a thin layer of liquid C_6F_{14} it emits Cherenkov photons. These photons are detected in the MWPC by the photocathodes, see figure 2.12 [1] [7] [8] [15].

2.3.7 DiMUON

The forward muon arm, also called DiMuon spectrometer, is a complex assembly of five Tracking and two Trigger Stations combined with Muon absorbers and filters and a large dipole magnet. The Muon Chambers cover an acceptance of $-2.5 < \eta < -4.0$; the Tracking Stations are positioned between 5.36 and 14.22 m away from the interaction vertex, the Trigger Stations between 16.12 and 17.12 m. The dipole magnet is placed at a distance of 7 m and provides a B-field of 0.67 T. Front absorbers are used to protect the spectrometer from photons and hadrons coming from the interaction point. To shield it from particles produced in the beam pipe as well, a conical absorber tube with an outer diameter of 60 cm is installed around the beam pipe. The layout of the DiMuon spectrometer is presented in figure 2.13.

The DiMuon spectrometer uses cathode pad chambers for the Tracking Stations. Each chamber is made of two cathode planes, while two chambers form one Tracking Station. The Trigger Stations are made of four Resistive Plate Chambers collected in two stations.

The DiMuon spectrometer is designed to measure high momentum muon pairs ($p_t > 4\ GeV/c$) coming from particle decays with heavy quark content [1] [8] [16] [17].

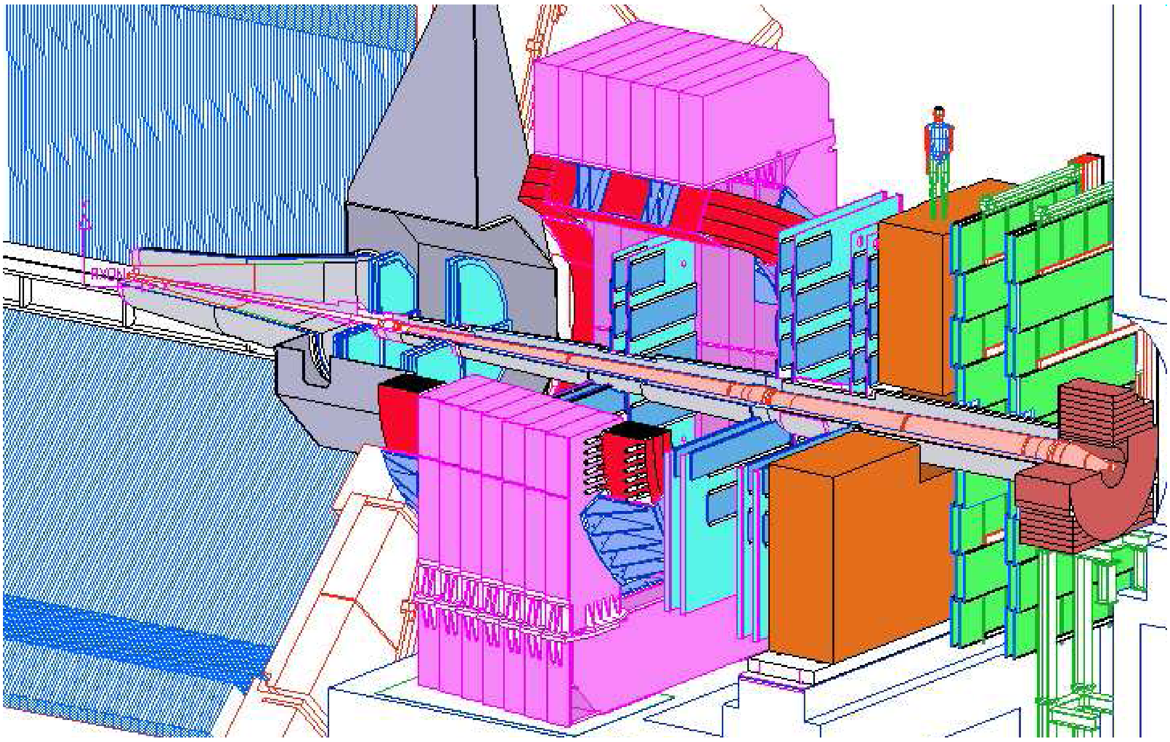


Figure 2.13: Layout of the DiMuon spectrometer. Beginning from the interaction point: first the absorbers in grey, the Tracking Chambers in light blue, the dipole magnet with its coil in red and pink, the massive iron block of the Muon filter in orange and the Trigger Chambers in green.

2.3.8 Other detectors

A set of several smaller detectors complete the setup of ALICE. Most of them are located close around the beam line outside of the ITS detectors: the Forward Multiplicity Detector (FMD), the Photon Multiplicity Detector (PMD), the Vertex 0 Detector (V0) and the Time 0 Detector (T0). These detectors are used for global event characterisation and for triggering.

The FMD is made of five silicon-strip ring counters located on both side of the interaction point ($-3.4 < \eta < -1.7$ and $1.7 < \eta < 5.0$). It provides ALICE with information of charged particle multiplicity [7] [8] [18].

The PMD is located opposite to the Muon arm at $2.3 < \eta < 3.7$. It is built as a honeycomb wire chamber measuring event-by-event photon multiplicity and their spatial distribution [7] [8] [19].

The V0 is built of two arrays of scintillator counters to provide trigger information. One of these arrays sits on each side of the interaction point close to the beam line [7] [8] [18].

The T0 consists of two arrays of Cherenkov counters, asymmetrically located on both sides of the interaction point. It delivers fast timing and trigger information as well as a t_0 time reference for TOF [7] [8] [18].

Two other detectors are located outside the L3 magnet: the ALICE COsmic Ray

DEtector (ACORDE) and the Zero Degree Calorimeter (ZDC). ACORDE is built from an array of plastic scintillators sitting on top of the L3 magnet ($|\eta| < 1.3$, $|\phi| < 60^o$). Its main purpose is triggering on cosmic rays entering the ALICE barrel. The ZDC is located far away from the central barrel: 116 m on both sides of the interaction point between the beam pipes. The ZDC is used for centrality determination by measuring the spectator nucleons[5] in Pb + Pb collisions [7] [8] [20].

In the future (2010) ALICE will be upgraded with an ElectroMagnetic Calorimeter (EMCal). It covers a larger area than PHOS but has a smaller granularity and lower resolution. The detector is located in the upper left corner of the ALICE barrel, 4.36 m away from the interaction point ($|\eta| < 0.7$, $80^o < \phi < 187^o$). For the FEE it will use a similar setup like the PHOS [1] [7] [8] [21].

2.4 The ALICE offline / online systems

ALICE has five online systems for readout, controlling, configuring and monitoring the different detectors: the Experiment Control System (ECS), the Data Acquisition (DAQ), the High Level Trigger (HLT), the Detector Control System (DCS) and the Trigger (TRG). The analysis software framework AliRoot and the ALICE GRID storage and computing facility (AliEn - ALICE Environment) are in the Offline system.

2.4.1 Offline

The ALICE Offline project covers the computing model, GRID activities and related tasks developed for the analysis environment of ALICE. The computing model comprises data definitions, the analysis software and the used framework, as well as the visualisation of events. To verify the results of the analysis software simulation packages together with event generators have been included. In addition ALICE Offline takes care of the interface to the ALICE GRID and their access via AliEn. The different building blocks are visualised in figure 2.14.

The analysis framework in ALICE is called AliRoot (ALICE Root). It is based on ROOT, an object-oriented software framework written in C++. ROOT comes with a C++ - interpreter (CINT) for on-the-fly interpretation of source code, Run Time Type Information (RTTI) and automatic generated code documentation. Nearly all classes are inherited from `TObject`, the "master" base class in ROOT. This feature has been adopted from languages like Java or Smalltalk. ROOT provides a broad set of utilities, containers and visualisation components, like graphs or histograms. There are ports to all major platforms, running under Unix, Linux, Windows or Mac OS[6] [22] [23].

Components for analysing detector data can be started by macros in the AliRoot environment[7]. Additionally, AliRoot provides a tool for event monitoring and visualisation: ALICE Event Visualisation Environment (AliEve). AliEve uses OpenGL for

[5] Non-interacting nucleons in a heavy ion collision are called spectator nucleons.

[6] The ROOT repository can be acquired from http://root.cern.ch/ .

[7] The AliRoot environment is an enhancement of the ROOT environment, taking the facilities like

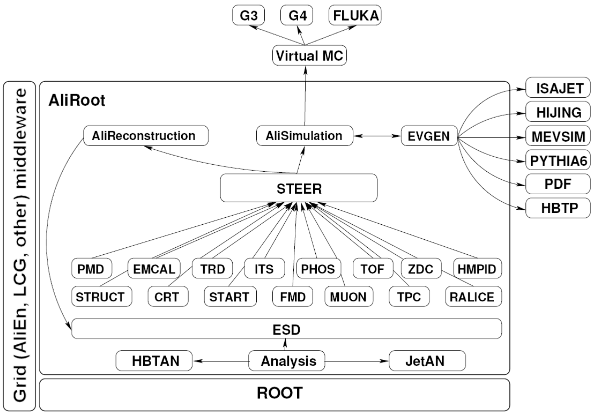


Figure 2.14: The different parts of the ALICE computing model: the analysis framework (Root and AliRoot), analysis software of the different detectors, simulation software (G3, G4, FLUKA) and event generator (EVGEN), data definitions like Event Summary Data (ESD) and the GRID access for the computing environment (AliEn).

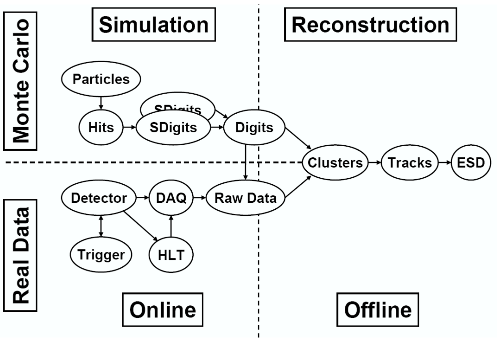


Figure 2.15: This figure sketches the comparison in the reconstruction of real data and Monte Carlo simulations in Online (HLT) and Offline. In both cases finally reconstructed events are stored in ESDs.

3D visualisation of events inside the detector models of ALICE. Furthermore it can display histograms and browse through ROOT structures.

In order to have the analysis components tested in advance, AliRoot provides interfaces to several particle simulation packages:

- **FLUKA**[8]: Fully integrated Monte Carlo[9] simulation package for particle transportation and interactions in nuclear matter. It has been written in FORTRAN [25].
- **GEANT3**[10] (G3): Monte Carlo simulation package for elementary particles in matter written in FORTRAN [26].
- **Geant4** (G4): Port of the FORTRAN package of G3 to C++ with an object-oriented design [27].

In order to have a common data structure ALICE Offline has defined the format for raw event data shipped by the FEEs of the detectors. In addition the structures for event fragments, subevents and complete events are specified. They are filled

the CINT, RTTI and the standard ROOT utility libraries and extending them with the libraries of the AliRoot classes.

[8] FLUKA stands for "FLUktuierende KAskade".

[9] Monte Carlo simulations are a set of classes for simulations with repeated random sampling. These classes have in common: a defined set of inputs, inputs randomly chosen from the domain (with deterministic computation) and aggregation of the individual computation results to the final result [24].

[10] GEANT stands for "GEometry ANd Tracking".

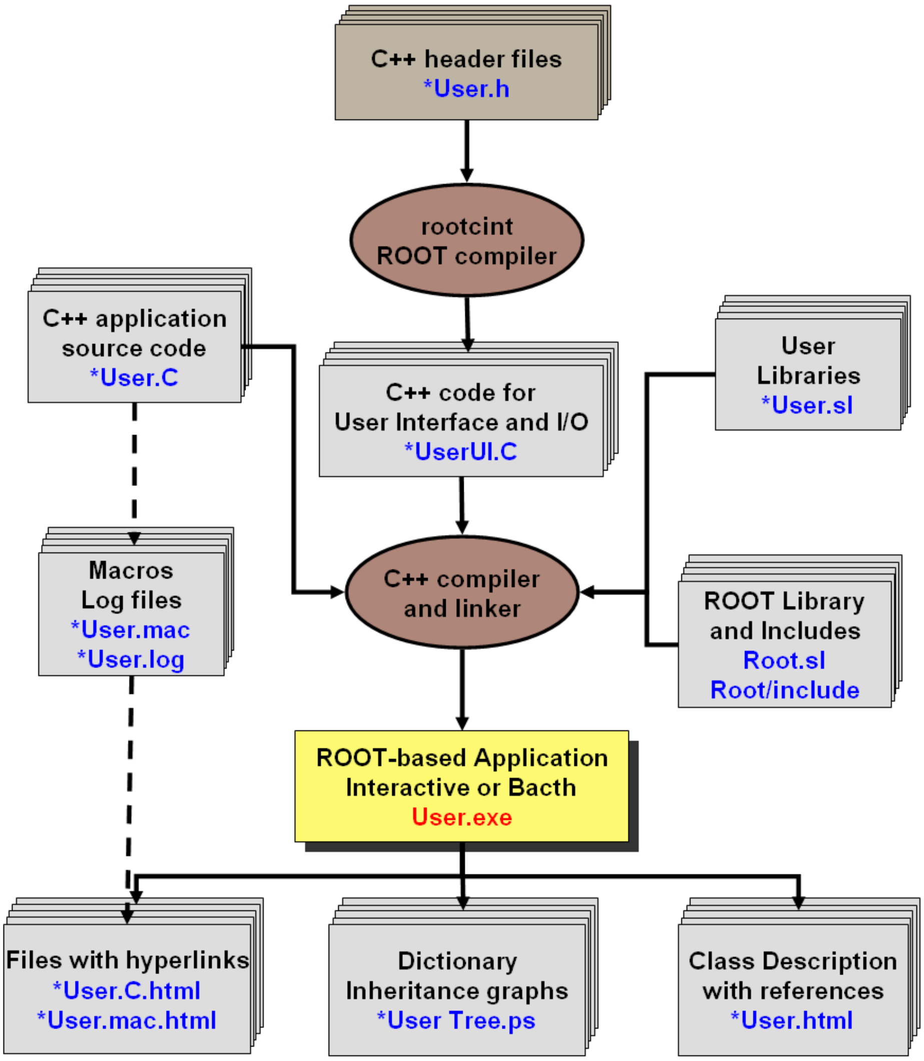


Figure 2.16: Sketch of how code is processed in the ROOT using the different tools in the ROOT environment.

during the different steps of the analysis process. A special protocol, the Common Data Header (CDH), is used to include meta data like event IDs, trigger messages or Regions-of-Interests (RoI). After reconstruction the results of the analysis are stored in Event Summary Data (ESD) blocks.

ALICE Offline also coordinates the GRID activities in ALICE. This includes the tasks of organising the GRID resources gathered for ALICE analysis and data storage[11], as well as enabling easy access to it. These GRID resources are provided by connected computing centres. They are organised and categorised[12] in the so called Tiers:

[11] The overall organisation of the various GRID resources for all LHC experiments are handled by the LCG (LHC Computing GRID) project [28].

[12] The classification uses the MONARC model (Models of Networked Analysis at Regional Centres for LHC Experiments) [29].

- **Tier 0**: CERN computing centre, used for computing and mass storage of raw data. Its tasks cover calibration- and alignment processes and a first reconstruction. The Tier 0 is connected with 10 GB/s to the ALICE DAQ to store the recorded data. (Additionally CERN will also host a Tier 1 and a Tier 2 centre.)
- **Tier 1**: Large computing centres (outside CERN). They provide a computing environment for a subsequent reconstruction as well as mass storage facilities. The stored data are copies of the data hosted in the Tier 0 at CERN. In addition they store the processed data from Tier 1 and Tier 2 centres.
- **Tier 2**: Smaller computing centres, which are only used to process GRID jobs, but do not necessarily provide storage facilities. They create and reconstruct simulated data and perform end users analysis.

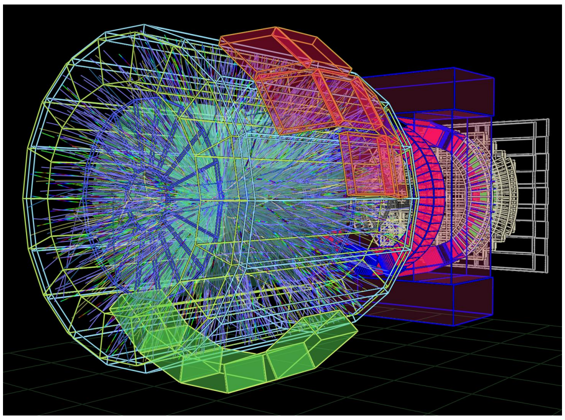


Figure 2.17: The pictures shows the AliRoot representation of a simulated $\mathrm{Pb}+\mathrm{Pb}$ event in ALICE.

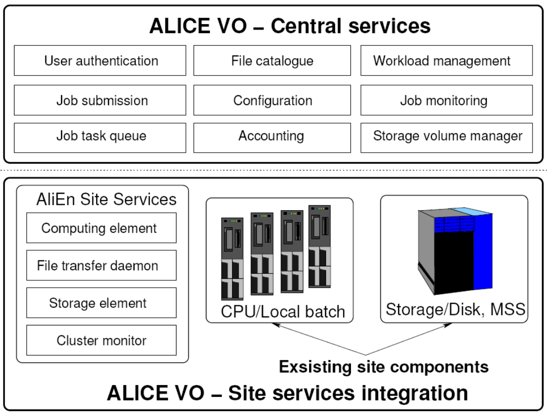


Figure 2.18: The sketch displays the AliEn components and their deployment in the ALICE VO (Virtual Organisation).

The developed middleware for allowing ALICE users to access the GRID facilities is called AliEn. It is designed to enable an easy access to the GRID and to hide the underlying complexity and heterogeneity to the users. Open source components of web services, common network protocols and distributed agents constitute the base of AliEn. User authentication is achieved by personal GRID certificates, which have to be registered in the ALICE VO (Virtual Organisation). Data in AliEn are organised in special AliEn FCs (File Catalogues). The AliEn FC is a virtual file system, which does not own the files but stores only the association between Logical File Names (LFN) and Physical File Names (PFN) on a mass storage system[13]. It features easy replication and caching of the corresponding files on different GRID sites [30] [31] [32].

In addition, the ALICE GRID hosts the Offline Conditions DataBase (OCDB), sometimes also referred to as Conditions DataBase (CDB). This database contains all calibration- and alignment settings produced for ALICE. The OCDB is accessed

[13] ALICE uses the CERN Advanced STORage manager (CASTOR) as mass storage system (`http://castor.web.cern.ch/castor/`).

by the AliCDB access classes in AliRoot using AliEn for the GRID requests. New calibration objects are entered after each run by the Offline Shuttle [33]. More details about the OCDB and the Offline Shuttle are given in the description of the interfaces between HLT and Offline in section 4.4.

All the different GRID activities on the net can be monitored using MonALISA (Monitoring Agents using a Large Integrated Service Architecture), a GRID tool for observing GRID sites and tasks [34] [35].

2.4.2 Experiment Control System – ECS

During a run all ALICE detectors, as well as all online systems of ALICE are steered by the ECS. Well defined states and transition commands are used to facilitate the control. Therefore every detector and each online system has implemented Finite State Machines (FSM), which can be plugged into the ECS system and allow ECS to take over control. A more detailed description of the FSM connections can be found in section 4.3 describing the HLT-ECS interface. In addition ALICE Offline is connected to ECS to get notified about the runs, especially to start the Offline Shuttle after each run (see section 4.4.1 on page 70). Figure 2.19 displays the connections of the ALICE systems and the event data flow of the experiment.

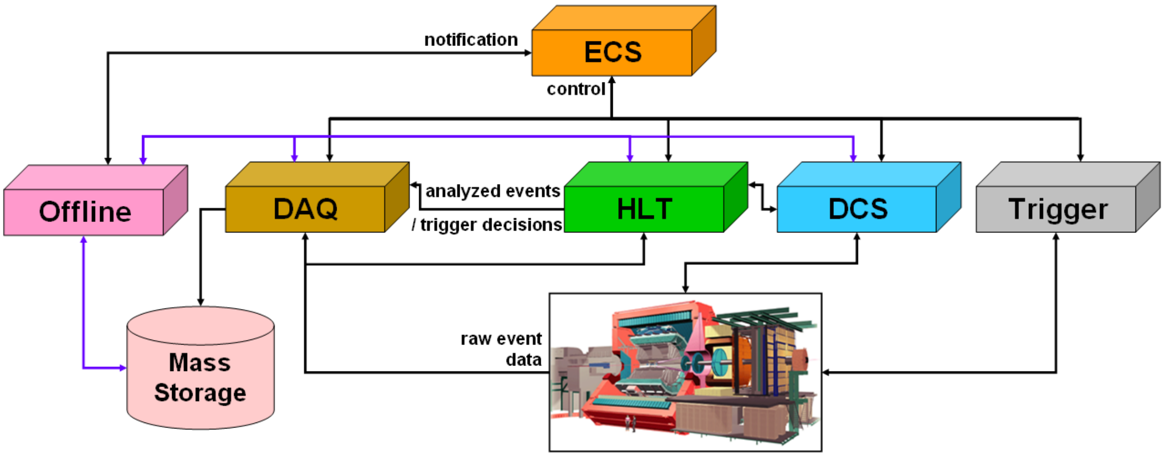


Figure 2.19: The ECS controls all ALICE online systems and includes a connection to Offline for notification about runs. The online systems (DAQ, HLT, DCS and Trigger) are connected to the detectors of ALICE. DAQ and HLT receive raw event data from the experiment. While HLT analyses the events online and provides trigger decisions to DAQ, DAQ builds events and transfers the data to permanent mass storage. Offline retrieves data from and provides data to the online systems DAQ, DCS and HLT.

Activity Domains, which are defined by their different tasks in ALICE, are steered by the online systems. An example of an Activity Domain can be the components responsible for the configuration of the FEEs of a detector; these components are hosted by DCS. ECS coordinates the different domains in a hierarchical control structure, and compiles them to partition(s). A partition is a set of detectors and their required parts of the online systems needed for configuring and taking experiment data. In that sense it is the largest entity to be steered by an operator. The largest partition consists of all detectors together forming the global partition. Partitioning enables to separate the control of different detectors and to steer them independently. This allows to have a subset of detectors taking data, while others are in calibration mode. Each online system can be run autonomously, even without ECS, but for data taking, they have to be connected and synchronised via ECS.

The ECS also provides the human interface to the operators of the different partitions and the global run operator. In addition ECS integrates ALICE in the LHC control setup. Thereby it retrieves general beam information for the experiment and synchronises ALICE with the accelerator control [7] [8].

Operationally the ECS works as an independent system but in ALICE it is physically part of the DAQ system - running on dedicated nodes in the DAQ cluster.

2.4.3 Trigger – TRG

The main task of the ALICE Trigger (TRG) is selecting events. It has three different stages of trigger signals without taking the HLT into account, which provides a final level of triggers after the read out of participating detectors. These three stages are called Level 0 (L0), Level 1 (L1) and Level 2 (L2) trigger decisions. To conclude on a trigger the Trigger system receives input from several Trigger detectors in ALICE. The detectors participating in the trigger depend on the experiment type ($\mathrm{p} + \mathrm{p}$ or $\mathrm{Pb} + \mathrm{Pb}$), the chosen physics observables and the trigger classes. The L0 trigger is sent to the detectors after 1.2 μs, the L1 signal arrives after 6.5 μs. The last trigger from TRG takes past-future protection into account and therefore arrives at the end of this interval, which is after 88 μs. The past-future protection is used to prevent corruption of readout data due to pile-ups in certain detectors, especially the TPC, which is the slowest detector in ALICE[14].

The trigger input from the participating detectors is collected and synchronised in the Central Trigger Processor (CTP), implemented as VME (Virtual Machine Environment) modules. The CTP sends out its trigger signals to the Local Trigger Units (LTU) of each detector. The LTUs relay the signals then to the FEE of the corresponding detectors via the Timing, Trigger and Control (TTC) broadcast system, the network that connects the TRG unidirectional with the detectors. The TTC is also used to synchronise the TRG with the LHC clock.

In addition a link from DAQ notifies TRG about buffers running full on the DAQ side. Custom made hardware, the so called BusyBoxes in the case of the TPC, FMD and PHOS, send detector busy information using a dedicated notification protocol [36] [37] [38]. In that case TRG has to reduce the trigger rate to enable the detectors and the DAQ to handle the event data rate [7] [8]. The tasks of the TRG in the online data flow are visualised in figure 2.20 [39].

[14] The sensitive window in the TPC is ≈ 88 μs. It is directly related to the maximum drift time in the detector [8].

2.4.4 Data Acquisition – DAQ

When the detectors receive the trigger signal to read out their FEE, they send their raw event data via Detector Data Links (DDL) to DAQ. A DDL consists of a Source Interface Unit (SIU) on the sender side, a pair of optical fibres to transport the data and a Destination Interface Unit (DIU) on the receiver side. It can transfer data in both directions with a rate of 200 MB/s. The DIUs are mounted on the so called DAQ - ReadOut Receiver Cards (D-RORC) in dedicated computers of the DAQ cluster. In addition they make exact copies of the data and send them via SIUs and DDLs to the HLT for online processing. To return back processed data HLT uses the same mechanism: SIUs on HLT - ReadOut Receiver Cards (H-RORC) send data via DDLs to DIUs on D-RORCs on the DAQ side. There the HLT data are included in the normal storage path of the DAQ system as well. In that perspective the HLT is handled as a separate detector with the exception that additionally trigger decisions from HLT are evaluated and applied to the storage policy.

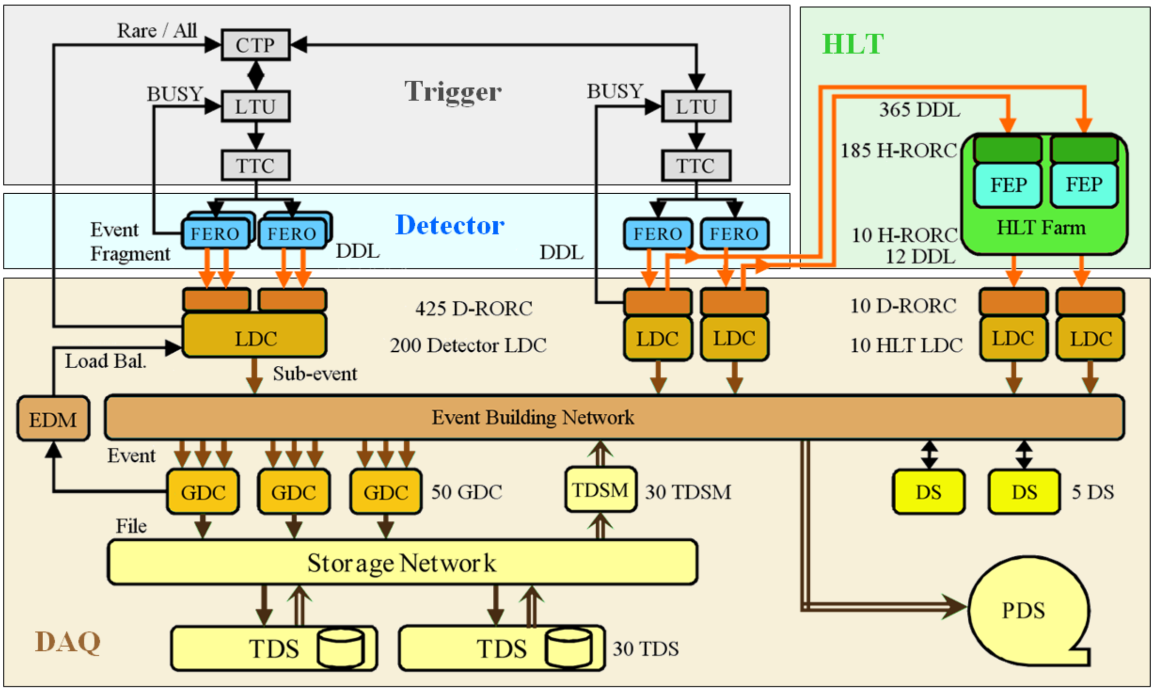


Figure 2.20: The figure depicts the online data flow of event data and their storage procedure by DAQ. The readout of the detector FEEs (Front-End Read-Out – FERO) is driven by the Trigger system: The CTP sends trigger decisions to the LTUs of each detector; this signal is then relayed to the FEE of the detectors via the TTCs. After read out the data are sent to DAQ and a copy is transferred to the HLT. After event building in the DAQ farm, the data are stored to mass storage. [39]

The DAQ system consists of Local Data Concentrators (LDC), nodes in the DAQ cluster which collect the data from the detectors, and Global Data Concentrators (GDC). The LDCs host the D-RORCs and each LDC can handle more than one D-RORC. They have to be able to handle several times 200 MB/s. The data are relayed to the Event Builder on the GDCs via the Event Building Network, a standard Gigabit ethernet connection using TCP/IP. The Event Builder merges the data from the different subevents. The merging of events can be done in parallel on the Event Building Network, several events at the same time. The complete events are stored to CASTOR, the mass storage system used in ALICE. The ALICE CASTOR files can be accessed later for offline analysis via AliEn on the ALICE GRID.

Detector	p+p	Pb+Pb
ITS Pixel		0.14
ITS Drift	0.0018	1.5
ITS Strips		0.16
TPC	2.45	75.9
TRD	0.0111	8.0
TOF		0.18
PHOS		0.02
HMPID		0.12
MUON		0.15
PMD		0.12
Trigger		0.12
Total	2.500	86.5

Table 2.1: The event size per detector for minimum-bias $\mathrm{p+p}$ and central $\mathrm{Pb+Pb}$ events in MB [7].

In the process of storing, the events are first saved to a Transient Data Storage (TDS). Later on they are moved to the Permanent Data Storage (PDS) in the CERN computing centre. The DAQ is able to store 1.25 GB/s to mass storage. The data flow in the DAQ and the integration to the systems of the HLT, the TRG and the detectors are displayed in figure 2.20.

The LDCs and the GDCs use standard 32 bit Intel PCs running Scientific Linux CERN 4 (SLC4). The storage facility in DAQ consists of 18 arrays of 4-5 TB discs. In the current setup, they are run in RAID (Redundant Array of Inexpensive Disks) level 6, which allows storage of 50 TB. Table 2.1 shows the data taking design parameters and the expected data volume per detector for minimum-bias $\mathrm{p+p}$ and central $\mathrm{Pb+Pb}$ collisions.

In addition dedicated machines provide an interface to the Offline Shuttle for transferring new calibration data, which will be calculated by specific Offline components in the DAQ net [7] [8] [39].

2.4.5 High Level Trigger – HLT

The ALICE HLT is designed to perform event reconstruction, event analysis and calibration calculations online, as well as to allow for a first detector performance monitoring during the run. Results of the analysis are written to ESD blocks. The

HLT provides event selection, chooses RoI within an event and compresses event data. These tasks enable DAQ to cope with the expected event data rate of 25 GB/s[15].

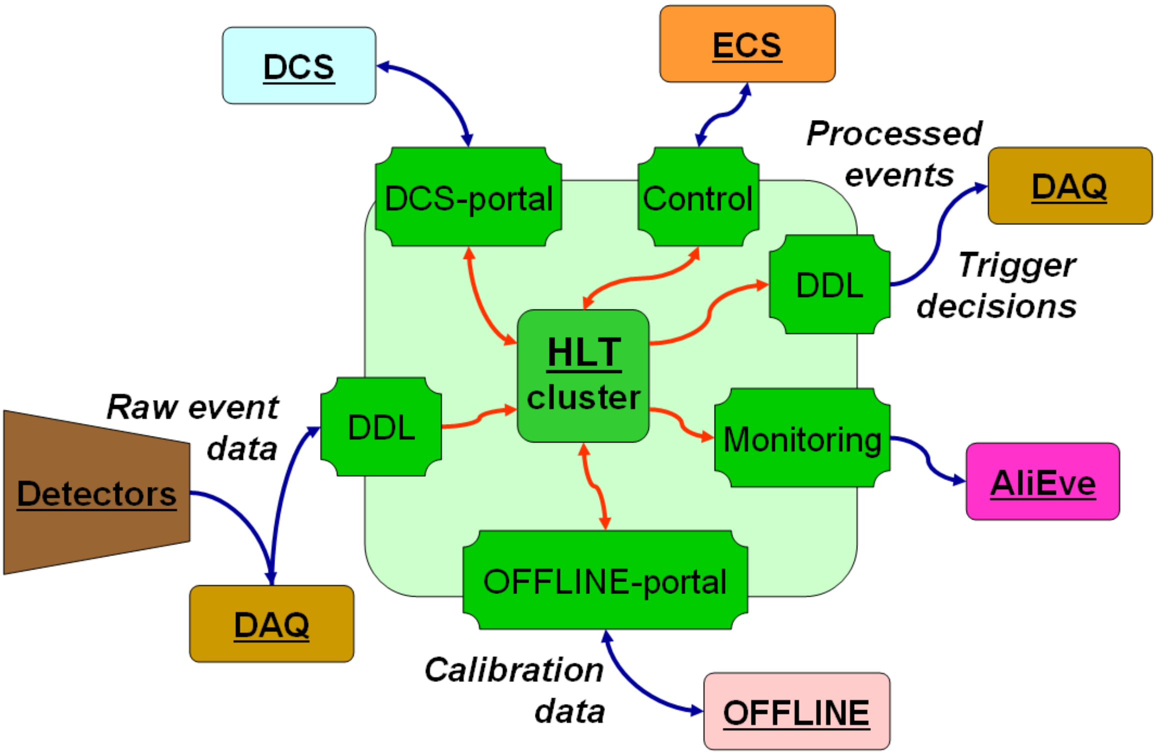


Figure 2.21: The figure sketches the HLT connections to the other systems in ALICE. ECS controls the HLT by a dedicated interface. Raw event data are received from the detectors. After processing the results are transferred to the corresponding systems. Additional input is retrieved from DCS and Offline. Experiment monitoring is enabled by interfacing AliEve.

To accomplish these computationally intensive tasks a large PC farm with interfaces to the various other systems in ALICE has been set up. Raw event data from the FEEs of all major detectors in ALICE are received during a run as direct copies from the DAQ LDCs. Analysis components process the data on the HLT cluster. The results are transferred to the corresponding systems [7] [8] [40].

The connections of the HLT to other systems are sketched in figure 2.21 and are the main focus of this thesis. They are described in the following chapters.

Hardware

The HLT consists of a large computing farm with (in the end) up to 1000 multiprocessor nodes. The nodes consist of off-the-shelf PCs, located in two counting rooms. In the current setup most of the nodes host two AMD Dualcore Opteron 2 GHz CPUs on a dualboard with 8 GB RAM, two Gigabit ethernet connectors and an Infiniband

[15] ALICE DAQ can archive about 1.25 GB/s; therefore the data rate has to be reduced by a factor of 20.

backbone for high throughput transmission. Some of the nodes are already equipped with Intel Quadcore CPUs, a complete upgrade to Quadcore CPUs is foreseen. In size the HLT cluster is comparable to a Tier 1 centre.

Most of the nodes are reserved for processing events. A small fraction is assigned to infrastructure tasks, like gateways, development and maintenance nodes and servers used for cluster monitoring. Two fileservers provide storage facilities and distribute the files over the cluster using AFS (Andrew File System). Each of them has a net capacity of 2 TB on a hardware Raid level 6.

In addition dedicated portal nodes provide interfaces to the ALICE systems. They have two ethernet interfaces going into different subnets: one internal to the HLT cluster and one to the subnet of the connected ALICE system. Dedicated interface applications run on these nodes according to the connected system. The interface nodes exist in redundant setups. Chapter 4 gives a detailed description of these interfaces and applications.

The cluster contains also 87 Front-End-Processor (FEP) nodes, where the raw event data from the FEE of the detectors enter the HLT. The data come as direct copies from the D-RORCs and are received in 185 H-RORCs hosted by the FEPs. The H-RORC is a PCI card with a Virtex-4 FPGA and 4 modules of 32 MB Double Data Rate Synchronous Dynamic Random Access Memory (DDR-SDRAM). It is designed for receiving and preprocessing[16] raw event data. For receiving data up to two DIUs are mounted on the cards and connected to DDL fibres. Transactions between the cards and the FEPs are PCI DMA (Direct Memory Access) based. The H-RORCs are also used for transmitting results back to the DAQ. Therefore the DIUs are replaced by SIUs.

The layout of the analysis task levels is shown in figure 2.22. It matches the ALICE detector structure and the different analysis steps. The raw data are processed first on the FEP nodes including the FPGAs of the H-RORCs. Therefore cluster finding can be done in hardware or software. Track finding or Vertex reconstruction respectively is performed on the next layer. The corresponding analysis components run in parallel on different cluster nodes. Afterwards events are merged across detector borders and globally reconstructed, followed by trigger decisions and event selections. Finally, data compression and reduction are applied, before the results are sent back to DAQ.

The hardware of the HLT cluster and the software applications running on them are monitored by SysMES (System Management for Networked Embedded Systems and Clusters) and Lemon (LHC era monitoring). SysMES is a decentralised operating, rule based monitoring tool for networked targets. It can observe hardware sensors such as CPU temperature measurements, as well as processes on the cluster nodes, including their log files.

The results of the monitoring by SysMES are visualised by Lemon. Lemon is a monitoring system based on the client / server principle, using agents on monitored nodes to send the observation results to a central Measurement Repository. Information is exchanged over ethernet. Through these agents it is also possible to observe

[16] Cluster finder modules for TPC and DiMUON are in development.

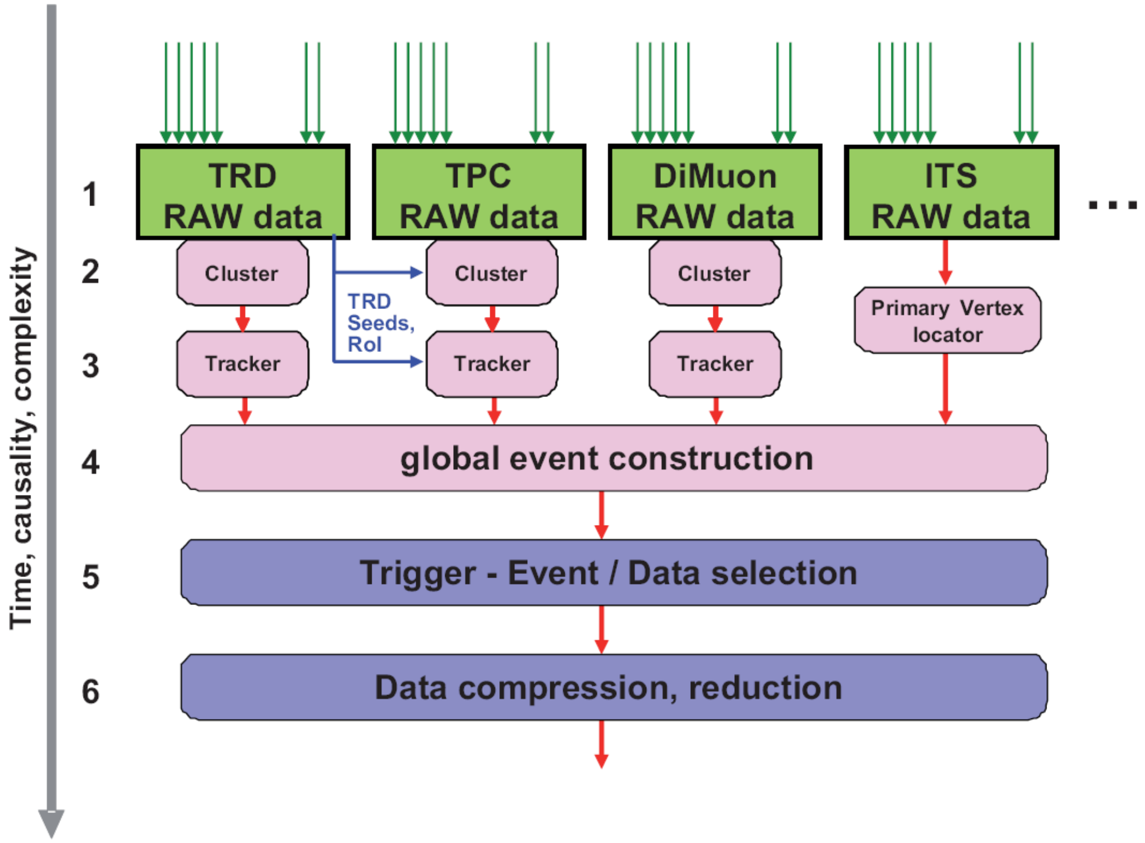


Figure 2.22: The layout of the HLT for analysis task levels. In layer 1 event data is received on the FEP nodes, which are assigned to the different detectors. In layer 2 and 3 cluster finding and tracking are performed, layer 4 merges events globally. Layer 5 extracts trigger information and selects events. In layer 6 data compression and reduction are applied.

remote entities, like the switches and racks of the HLT cluster [8] [40] [41].

Remote administration of the cluster nodes in case of problems is enabled by administration actuators. CHARM[17] (Computer Health And Remote Management) cards on each node fulfil this task allowing full control on the corresponding node [8].

Software - Operating system

HLT runs Ubuntu Linux as operating system on the cluster nodes, currently Ubuntu 6.06 LTS[18] (aka Dapper Drake). The setup uses a 2.6 vanilla kernel[19] with modified configurations like bigphys memory access enabled. Additionally, third party libraries are installed according to the needs of the various software applications of the HLT [8] [40].

[17] The CHARM card is a PCI (Peripheral Component Interconnect) card for cluster monitoring. In the early stages of the ALICE HLT planning the CHARM card was referred to as CIA (Cluster Interface Agent) card [7].

[18] LTS signals *L*ong *T*erm *S*upport versions of Ubuntu.

[19] The kernel is taken from `http://kernel.org/`.

The HLT framework

The software tasks of the HLT, including data transportation, interface applications and analysis components, are steered by a hierarchical system of TaskManagers (TM). On top of it sits a so called RunManager controlling the setup. The RunManager is steered by ECS via the HLT-proxy[20]. It manages underlying Master-, Servant- and Slave-TaskManagers. The Servant- and Slave-TaskManagers start the actual components for analysis and keep track of their execution. In addition they organise and monitor the correct transactions by the data transportation framework PubSub (PublisherSubscriber - see below). The configurations for analysis chains are provided in XML-files, which are prepared before the start of run and then distributed to the assigned nodes. The different TaskManagers are started on all cluster nodes included in the chain. There they start the tasks according to the given configuration. For redundant checks the TaskManagers are interconnected.

A complex system of states handles the internal synchronisation. Communication inside the HLT cluster is achieved by a dedicated InterfaceLibrary, hiding the connection details from the user. More details about the TaskManager system are presented in section 4.3.2.

HLT has developed a generic data transportation framework, which is based on the Publisher-Subscriber principle. It is called HLT PubSub system, and is responsible for the communication between components, transportation of event data and results and also load distribution. It is also able to dynamically re-route the data flow during run time[21]. The system uses a pipelined push architecture. Components can receive data by subscription, process the data and publish the result for the next component in the chain. Thereby components can be pure publisher or pure sink components or a combination of both forming a processing component. Special features like merger, for merging different event streams, thereby combining same events from different streams, scatterer, for splitting an event stream, and gatherer, for collecting streams again, have been implemented by dedicated PubSub components[22]. Communication between different nodes is achieved by special bridging components using the according network protocols for communication, making the data access for analysis components transparent from the source. The different types of PubSub components are sketched in figure 2.23. The PubSub system primarily uses pipes and shared memories in combination with the exchange of data descriptors in the communication, therefore enabling a low overhead processing [8] [40] [42] [43] [44] [45].

The interface applications for exchanging data with the other ALICE systems (ECS, DCS and Offline, including AliEve for online event monitoring) are also regarded as part of the framework, their description follows in chapter 4.

[20] HLT stand-alone tests can be achieved by dedicated user front-ends, see section 4.3.2

[21] Therefore SysMES, the cluster monitoring tool of HLT, can interact with the PubSub framework.

[22] Scatterers and gatherers are mainly used for load balancing, locally on multiprocessor boards and/or among several computing nodes.

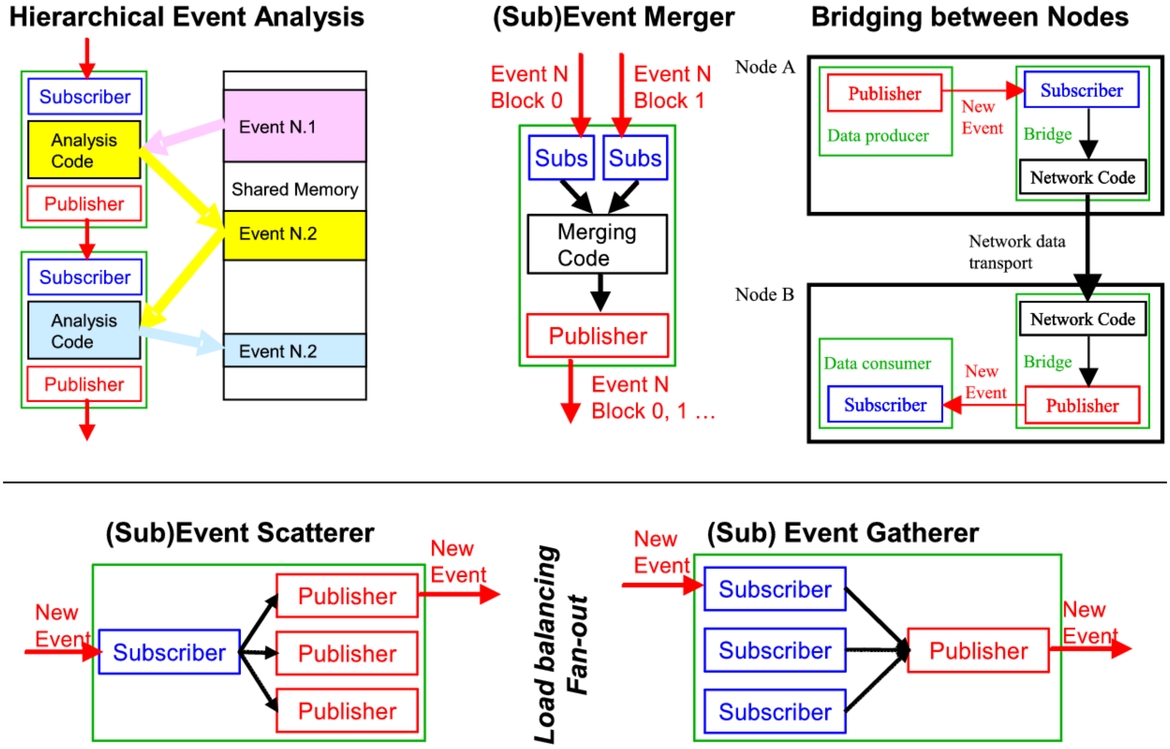


Figure 2.23: The figure displays the different features of the HLT PubSub framework. Components can subscribe to a data stream and publish their results for other components. Data exchange is mainly handled by shared memory. Special merger components can subscribe to more than one data stream and publish the merged results for others. Bridging components allow for data transportation between different nodes. Special scatterer components can split one data stream into several different streams, while gatherer components can collect these streams again into one stream; individual processing of the different streams in between is possible.

AliRoot integration

HLT has implemented an analysis framework[23] separating analysis and data transportation. It acts as middle layer between the PubSub system and the actual analysis components. Since the analysis algorithms are written in AliRoot, it also integrates the Offline framework into the online HLT analysis. Therefore the analysis components can run online using the HLT PubSub system and stand-alone in offline mode. This enables a comparison of the results of the HLT to Offline. A special processing component, the AliRootWrapperSubscriber, acts as adapter and can load different analysis modules written in AliRoot. To have the same module running in AliRoot and HLT PubSub environment, binary compatibility of the libraries is required. This is achieved by an external C interface, implementing all required online functionality for running in a HLT processing chain [8].

[23] The AliRoot integration has been developed by Matthias Richter (IFT – University of Bergen, Norway) and Dr. Timm M. Steinbeck (KIP – University of Heidelberg, Germany).

Analysis Components

Analysis components for the different detectors participating in the HLT have been developed. These components are written in AliRoot and tested with simulated data generated by the corresponding AliRoot module.

For the TPC a fast online Cluster Finder and Tracker have been developed. The Cluster Finder uses 3D space points for reconstruction, the Tracker combines the clusters into tracks. The tracking is performed at sector level, when crossing more than one sector the tracks are merged. The expected event size for the TPC is up to 75 MB. This leads to 15 GB/s after zero suppression. For an efficient storage of the TPC data, the HLT applies a compression algorithm to the event data using online pattern recognition. The reduced event size is only ≈ 11 % with a negligible loss in tracking performance [46]. In addition HLT produces new calibration objects for the TPC online, which are shipped to the OCDB using the dedicated interface application (see section 4.4.1).

The TRD reconstruction in HLT uses the algorithms designed for offline analysis. Calibration reference data produced in the TRD chain are shipped to the OCDB.

PHOS in HLT measures timing and energy information of electromagnetic (photon) showers. Therefore algorithms for signal shape and shower reconstruction have been implemented. The peak finder algorithm in the analysis is able to reduce the data amount from 300 MB/s to 20 MB/s after zero suppression, which are then shipped to DAQ. PHOS calibration parameters are relayed to the OCDB.

The DiMuon HLT algorithms perform a partial event reconstruction online, improving the sharpness of the p_t cut. The Cluster Finder and Tracker can handle the expected event rate of 1-2 kHz [8].

2.4.6 Detector Control System – DCS

The DCS is responsible for the operation of the different detectors. In addition it allows for monitoring of the detectors at different granularities. Thereby it takes care for example of detector cooling, supervising of the FEEs, providing power to the various parts of the detectors and synchronising their ramping up and down procedures with the global run control of the experiment. The configuration of the FEEs can also be done by DCS, using a dedicated database: the DCS Configuration DB. Furthermore the DCS connects to the CERN Safety System, handles the logging done by the detectors, takes care of the alarm and error handling and provides interlocks to the different device units of the detectors.

The ALICE DCS is built as a tree. Its hierarchy control structure distinguishes between two different types of elements in the tree. Nodes, that have a child or children are called Control Units (CU). They can be seen as the branches and are modelled by Finite State Machines (FSM). These FSMs reflect the states of a certain device or a group of devices. Device Units (DU) form the leaves of the tree. They act as adapter between the commands received via the FSM and the interface to the lower level components. Via these commands the associated devices are actually controlled

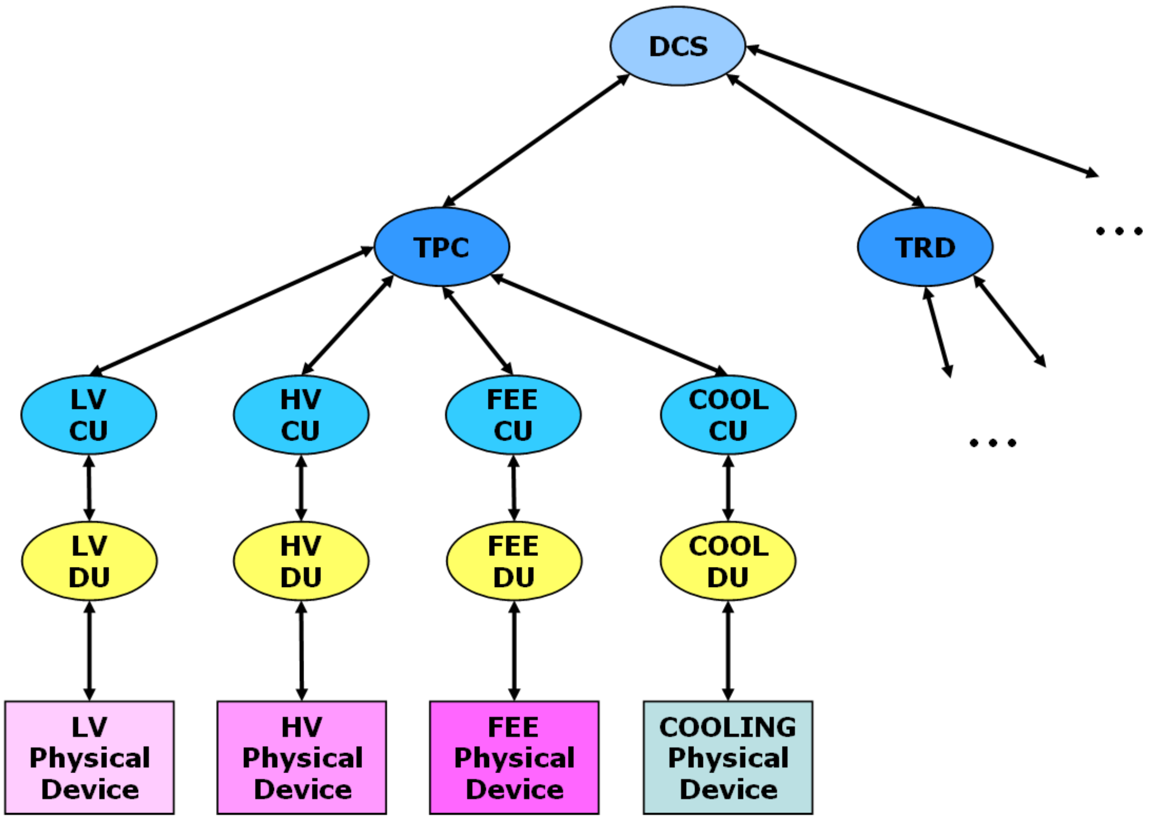


Figure 2.24: The hierarchical structure of the DCS looks like a tree with Control Units (CU) as branches and Device Units (DU) as leaves. The services of the different detectors like High voltage (HV), Low Voltage (LV), Cooling (Cool) are controlled by the CUs implemented in software; the DUs are applying the chosen setting to the actual hardware.

and monitored. While the CUs consist of software, which will run in the DCS counting room, the DUs can be either hardware (like sensors, power supplies or cooling plants) or software / firmware running on a PC or PLC (Programmable Logic Controller). Partitioning has been a major issue in the development of the DCS to grant local operators to control certain systems or dedicated device units. The tree structure of the DCS is presented in figure 2.24.

To integrate each detector into the DCS, the detector experts have developed their control structure in a hierarchical manner up to the SCADA (Supervisory Control And Data Acquisition) system, which connects to the global DCS. In ALICE DCS, PVSS-II (Prozess Visualisierungs - und Steuerungs - System II) has been chosen as SCADA system [7]. The connection of the FEEs of the TPC, TRD, PHOS and FMD into the control of the DCS is described in more detail in section 3.2. These detectors have in common, that they all use the Front-End-Electronics Communication (FeeCom) chain in their integration to DCS [47] [48] [49] [50] [51].

Chapter 3

The FED-API in ALICE DCS

Although, this thesis focusses mainly on the HLT, part of the project work has been carried out on DCS components. The FED-API (Front-End-Device - Application Programming Interface), as presented here, has been primarily developed for the communication between the DCS and the FEE (FeeCom chain, FERO (Front-End Read-Out) configuration). But parts of it are used in the interface from HLT to DCS as well (cf. section 4.5.2), therefore becoming an important part of the thesis.

3.1 The DCS board

The FeeCom chain is used by several detectors, controlling their FEE via the custom made DCS board[1]. The DCS board[2] consists of an embedded system with an Altera EPXA1 FPGA including a hard-wired ARM processor, a flash memory based hard disc, 8 MB SRAM Memory, an Ethernet controller[3] for 10 MBit connections and an optical input for the TTC system. The DCS board is located as embedded system on a Readout Control Unit motherboard (RCU[4] – in the cases of the TPC, PHOS and FMD) or on a ReadOut Board (ROB – in the TRD case). It facilitates the DCS to contact the FEE of the corresponding detector. Additionally, the RCUs and ROBs host an SIU to connect to DAQ. Furthermore, the RCUs handle the bus systems, which talk to the Frond-End Cards (FEC)[5] of the according detector. The ROBs have direct contact to their FEEs. Since each RCU / ROB hosts exactly one DCS board,

[1] The list of detectors includes the TPC, TRD, PHOS and FMD; EMCal will join as well, because it will use the same hardware (DCS board, RCU) like PHOS.

[2] The DCS board has been developed at the Kirchhoff Institute for Physic (KIP) – University of Heidelberg, Germany.

[3] In order to operate the DCS board in the magnetic field of ALICE, the transformer coil has been replaced by a small amplifier circuit, which boosts the output of the physical layer chip.

[4] The DCS board - RCU connection has been implemented by Dr. Johan Alme (IFT – University of Bergen, Norway); together with Ketil Røed (Bergen University College, Norway) he has developed the RCU radiation tolerance solution [52].

[5] The FECs host the read out of the actual detector sensors and have to be configured with e.g. pedestals to achieve a proper readout. In FMD the FECs are replaced by so called FMD Digitizer cards, which fulfil the same purpose.

the amount of them used per detector depends on the detector layout. TPC has 216 DCS boards in sum, one for each TPC partition. PHOS uses 4 per detector module, after the full PHOS installation there will be 5 modules. In FMD one board is used per FMD ring, 3 in total. TRD has a DCS board for each chamber layer, and therefore the highest amount of DCS boards with 540.

The DCS boards run on their own in order to react autonomously to critical situations but receive their power by the hosting motherboard. The ARM processor in the FPGA runs a Busybox-Linux based operating system, which has been specially ported for the target system. Software for the DCS board can be compiled by a dedicated cross-compiler designated for the ARM processor (e.g. arm-uclibc-gcc). Application and data files can be either loaded to the flash disc on the board or mounted from a file server via NFS (Network File System). During normal operation, communication with the DCS board from the upper DCS layers is achieved over ethernet connections.

3.2 The FeeCom chain

The FeeCom chain consists mainly of three components: the FedClient which is part of the PVSS panels of the corresponding detector, the InterCommunicationLayer (InterComLayer or ICL) on the control layer and the FeeServers with their ControlEngines (CE) to contact the actual FEE in field layer. The latter one is running on the DCS boards. The ICL consists of several modules. This includes the FedServer module[6], the ApplicationLayer module, which hosts the database access of the ICL and the CommandCoder (CoCo), and the FeeClient module as contact to the FeeServers. The whole chain is visualised in figure 3.1. The FeeCom chain components are classified as detector-independent and -specific parts. The components responsible for the communication between PVSS and the DCS boards have been designed independent of the detector-specific underlying hardware devices. The FedClient as a whole is detector-specific, while on the ICL side the CoCo and on the FeeServer side the CE are detector-specific.

In the FeeCom chain control and configuration commands are issued at the supervisory layer. The FedClient, which is integrated in the PVSS system of the supervisory layer, sends these commands to the FedServer of the ICL. More details about this interface are described in section 3.4. In the ICL additional configuration data can be added to these commands by requesting the DCS Configuration DB[7]. The detector-specific CoCo prepares the data before the FeeClient module of the ICL distributes the commands to the destined FeeServers. Broadcasts and multicasts to several FeeServers are possible. In the case that these commands are meant for the FEE, the FeeServer hands them over to the CE. The CEs are detector-specific modules, which are integral part of the FeeSevers. They communicate via the well defined FeeServer-CE-API.

[6] The FedServer module of the ICL has been implemented by Benjamin Schockert (ZTT – University of Applied Science Worms, Germany).

[7] The DCS Configuration DB is hosted by an Oracle 10g database server. Data exchange with the database is handled via the Oracle C++ Call Interface (OCCI), which allows for an object oriented design in the database client calls.

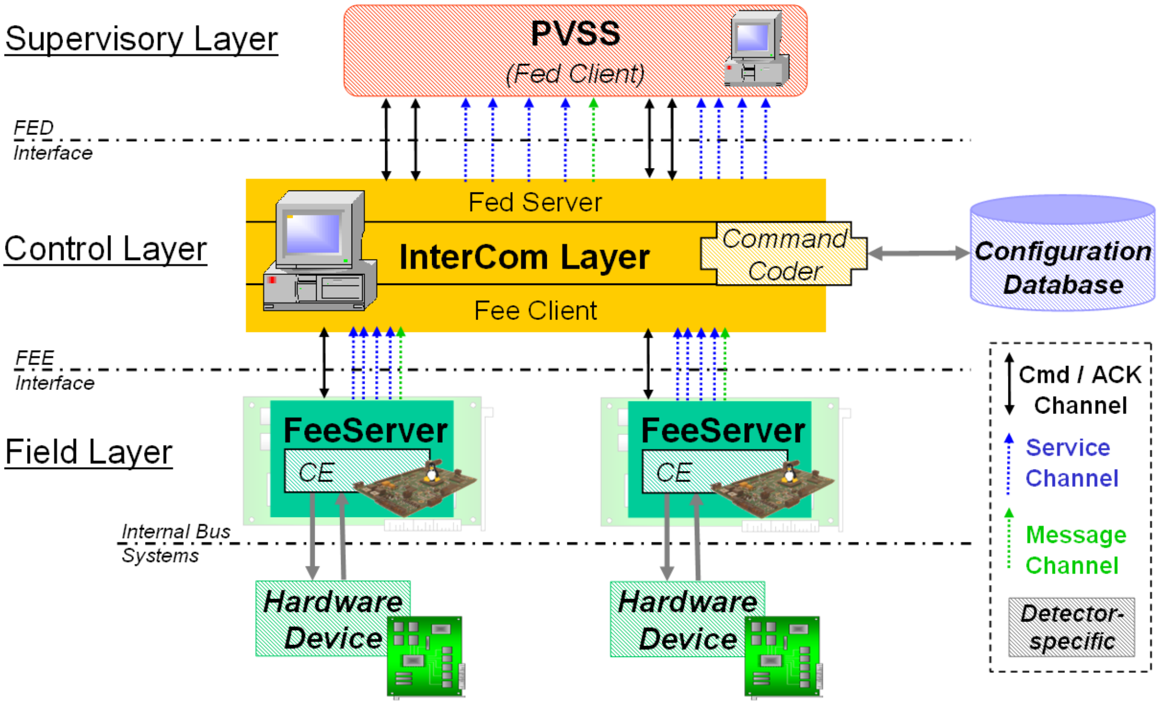


Figure 3.1: The FeeCom chain connects the FEEs of TPC, TRD, PHOS and FMD to the PVSS panels of the according detector. FeeServers on the custom made DCS boards control the underlying hardware devices via detector-specific CEs. The amount of connected FeeServers is equal to the number of DCS boards in the corresponding detector design. Monitored values of the FEEs (like temperatures, voltages, etc.) are delivered to the InterComLayer, where all channels from the FeeServers of one detector are collected. After grouping of these values, they are handed to the PVSS system on the supervisory layer. Control and configuration commands issued from the PVSS are sent to the FedServer of the InterComLayer. Additional configuration data are fetched from the DCS Configuration Database and prepared by the detector-specific CommandCoder, before these commands are distributed to the different FeeServers. The actual execution of commands for the FEE is handled by the CE of the corresponding FeeServer. Acknowledgements (ACK) or error codes are handed back to the PVSS system. In addition the components can send log messages to the supervisory layer.

The CEs know how to contact the underlying hardware devices and execute the given commands. Acknowledgements (ACK) or appropriate error codes for the issued commands are handed back to the FeeClient via separated ACK channels in the FeeServer. Additional result data can be appended to these ACK / error codes. The data of the ACK channel is handed further to the FedClient, as well as to the CoCo for evaluation.

Furthermore, the FeeCom chain allows for monitoring of the detector FEEs. The CE requests information from the hardware devices and translates the answers. Temperatures, voltages, currents, states or similar quantities can be monitored thereby. To reduce the bandwidth for updating these values on the upper layer, a dead band around each monitored value is applied. Only values, which exceed the dead band around their last transmitted value, are sent. These values are collected in the ICL and delivered further to the PVSS panels. Log messages, which are received via dedicated Message channels, are relayed to PVSS panels as well, after applying a filter according to the set log level. To avoid loss of information, all messages are stored to a local log file on the ICL stage.

The communication between FedClient and FedServer, as well as between FeeClient and FeeServer utilises the Distributed Information Management (DIM). The contacts between the CE and the hardware devices of the FEEs are performed via internal bus systems and device drivers [47] [49] [53] [54] [55] [56].

3.3 Distributed Information Management

DIM is a communication framework developed at CERN and widely used among all LHC experiments[8]. Its mechanism is based on the client-server principle. DIM is especially well suited for distributed and heterogeneous environments where it provides a network transparent inter-process communication layer. The communication can facilitate TCP/IP for transportation and control.

A DIM Server provides services (implemented as DIM Service channels) and commands (implemented as DIM Command channels). Both are identified by their names. It is mandatory that these name are unique inside a DIM domain. A DIM domain is defined by a DIM Domain Name Server (DIM_DNS). The DIM_DNS acts as broker between DIM Servers and Clients. At start-up each DIM Server registers its services and commands at the DIM_DNS. DIM Clients request the DIM_DNS[9] for connection details for a given DIM Service or Command. Afterwards the clients can contact the corresponding DIM Server(s) directly. This mechanism is shown in figure 3.2. One DIM Server can handle several DIM Clients. But there can be only one DIM_DNS per DIM domain. The DIM_DNS ensures that the communication channel names are unique among the domain. It keeps an up-to-date dictionary of all servers, services and commands.

[8] DIM has been developed first for the DELPHI (DEtector with Lepton, Photon and Hadron Identification) experiment at CERN.

[9] The host name, where the DIM_DNS is located, is provided to DIM Servers and DIM Clients by the environment variable `DIM_DNS_NODE` .

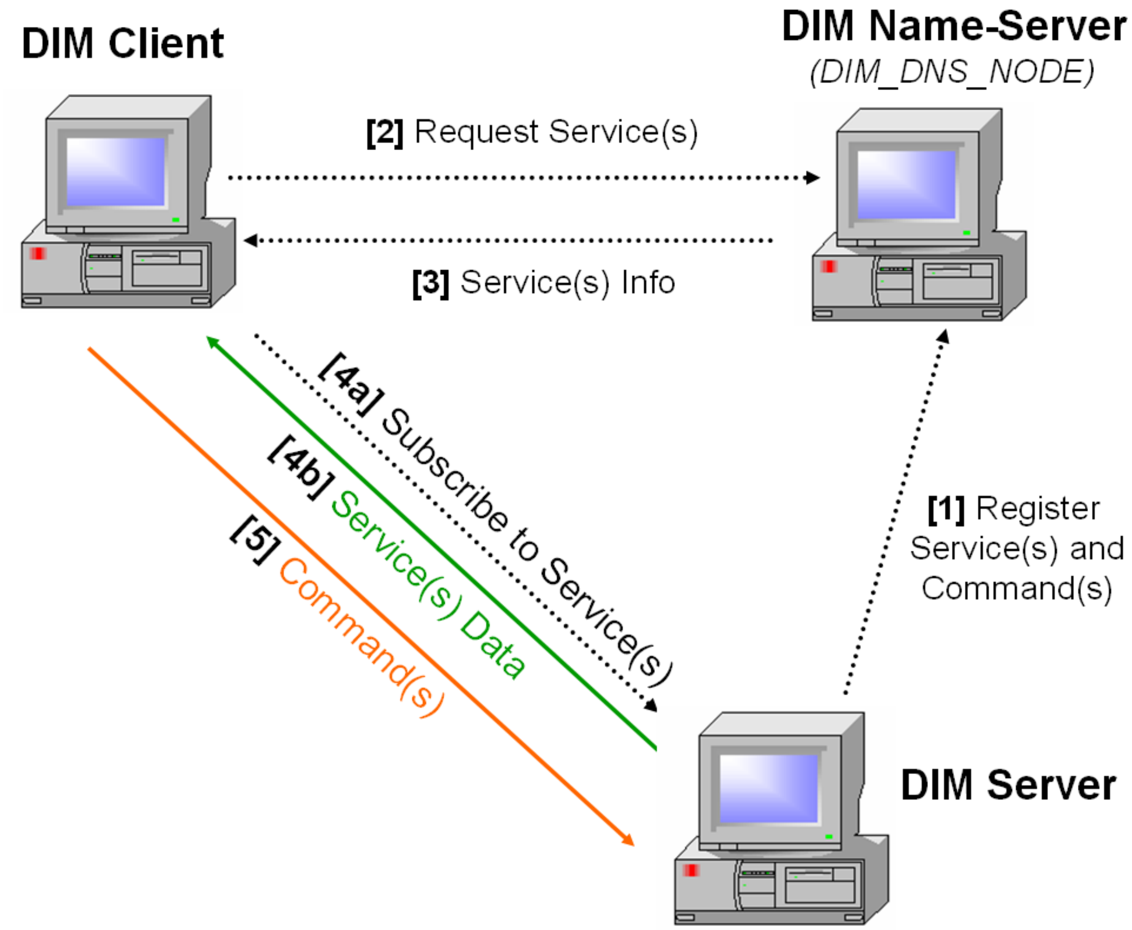


Figure 3.2: The figure displays the role of the DIM_DNS in the connection of DIM Servers and DIM Clients. A DIM Server publishes its services by registering them together with the accepted commands at the central DIM_DNS. A DIM Client can retrieve the connection details for a DIM Service or a DIM Command from the DIM_DNS. Afterwards it is able to contact the DIM Server directly for subscribing to services or for sending commands.

When subscribing to a service the corresponding data are sent to the client. Afterwards the client receives updates on data change or after a given time interval or a mixture of both. Thereby DIM uses a push architecture: the server informs the client about the service data without further requests. This principle is visualised in figure 3.4. The locations of the DIM Services are transparent to the user applications on the client side. The system automatically reconnects after a crash or network failure or a connection break due to migration of a service to a different node / server. The DIM_DNS plays the central role in this feature.

DIM exists in C, C++ and Java implementations and allows for an interface to Fortran. When using the C++ version of DIM, servers can receive the according functionality by inheriting from `DimServer`. `DimService` is the (parent) class for services published by these servers. For commands the class `DimCommand` is used. DIM Clients use (child) classes of `DimInfo` to subscribe to DIM Services, while the client function-

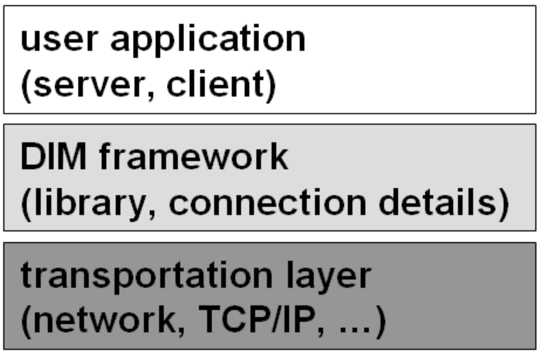


Figure 3.3: The figure depicts the different layers in a DIM application. The connection and data transportation details are handled by the DIM framework and are kept transparent to the user application. The latter one only interacts with the DIM library of its system.

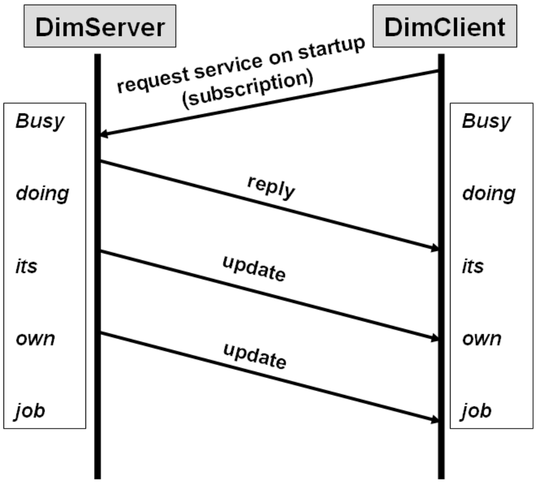


Figure 3.4: The figure displays the push architecture of the DIM framework. DIM Clients have to register only once to a DIM Service, further service updates are triggered by the DIM Server via callback routines on the client side [58].

ality is inherited from `DimClient`. Dedicated DIM modules have been developed for LabView / EPICS (Experimental Physics and Industrial Control System) and PVSS, which allow to integrate DIM Clients and Servers into these SCADA systems[10]. Dedicated browsing tools (DID (Distributed Information Display), DimTree) enable the investigation of a DIM setup. DIM is available on many different platforms like Unix, Windows, Linux, VMS (Virtual Memory System), etc.. Architecture specific details like byte ordering or data alignment are handled by the framework and are transparent to the user applications [58] [57].

3.4 The FED-API – DCS integration

In the view of the DCS the tasks for configuring and controlling the FERO are very similar among the various detectors. Therefore a common interface between the FED and the supervisory system in DCS has been invented: the FED-API[11]. It connects the SCADA system (PVSS-II) of the supervisory layer either with the control layer (ICL) or with the application layer directly located on the field devices. Which setup is applied depends on the detector design. The figure 3.5 shows how the FED-API is applied to the different detector setups in ALICE. In principle they can be categorised

[10] DIM homepage: `http://dim.web.cern.ch/dim/` .

[11] The FED-API has been developed together with Dr. Peter Chochula (CERN) and is based on an idea of Christian Kofler (ZTT – University of Applied Science Worms, Germany).

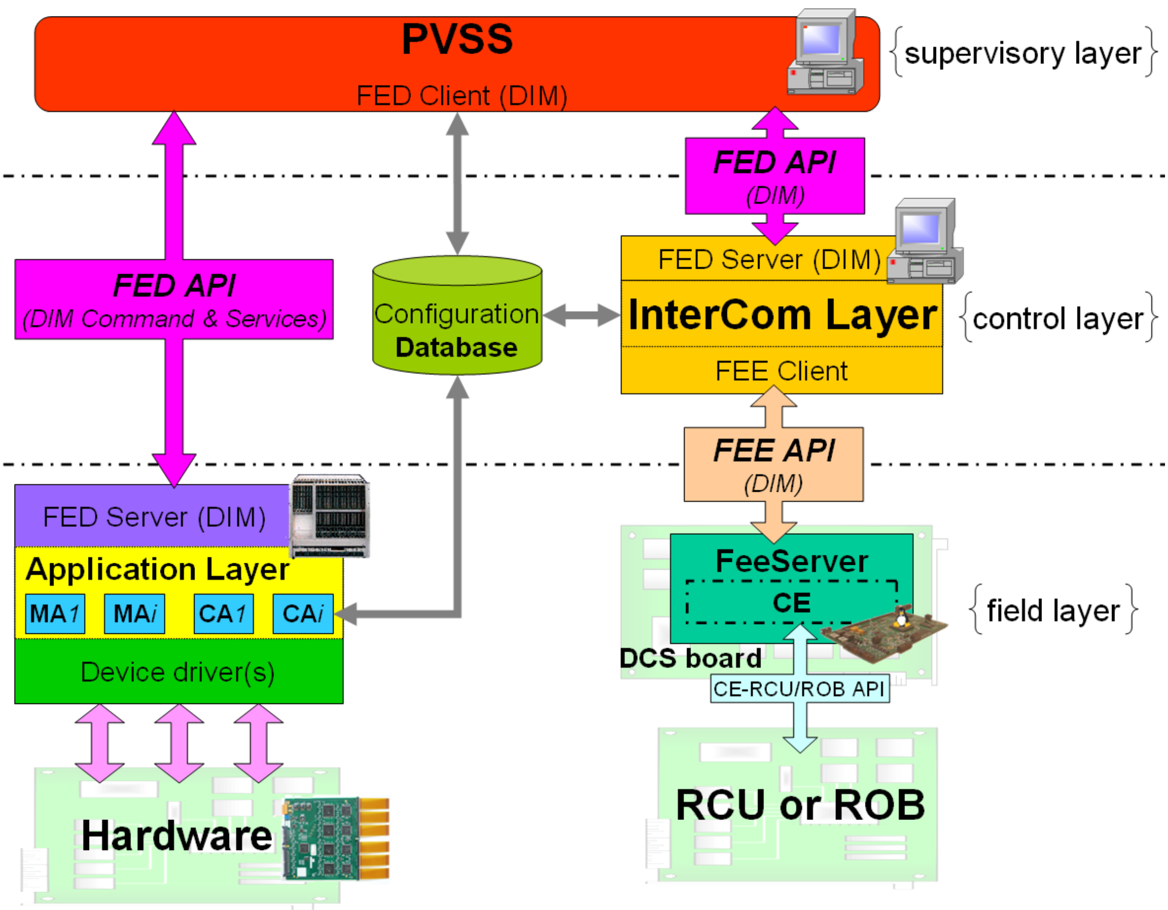


Figure 3.5: The FED-API is a common interface for different detectors between their FERO and the controlling DCS system. It uses DIM as communication framework to transport data for monitoring services and configuration- and control-commands. The figure shows the two types in the setup, where the FED-API is applied. On the supervisory layer PVSS panels are used for communicating with the FED-API. The left side shows a setup where the application layer, hosted by dedicated hardware (e.g. a VME crate or a PLC), sits directly in the field layer. It talks to the electronics on the field devices via device drivers. In this case the monitoring tasks are taken over by Monitoring Agents (MA), the configuration and control are performed by Control Agents (CA). This setup is used for example by the SPD. The right side depicts the FED-API in the FeeCom chain, which is used by TPC, TRD, PHOS and FMD. All these detectors use the InterComLayer, which includes an implementation of the FED-API in its FedServer module. The field devices are addressed by dedicated software (FeeServer) running on the embedded system of the DCS board. Communication between ICL and the DCS board is performed over ethernet. Communication with the control hardware of the field devices (TPC, PHOS, FMD: Readout Control Unit (RCU) - TRD: Read Out Board (ROB)) is achieved via the ControlEngines (CE) of the corresponding FeeServers.

into two different types: detectors using the FeeCom chain and detectors having their control layer directly located on the field devices [59].

This interface has been used as a guideline for the developers of the different detectors implementing their DCS FERO-subsystem. In the implementation DIM is used as communication and transport framework. The usage of the FED-API in the FeeCom chain is described in the following.

The FED-API commands are represented as DIM Command channels. DIM Service channels deliver monitored values and log messages from the FeeCom chain to the supervisory layer. In addition, acknowledgements and respectively error codes for given commands are provided. For certain service channels it is possible to have more than one instance of the defined channel type. Unique channel names guarantee a clear distinction between different instances of a channel type. Since these channels are contacted inside a DIM domain only by their channel name, independent of the hosting DIM Server, they do not even need to be provided by one single FedServer.

3.4.1 FED - Commands

Commands defined in the FED-API are used to send configuration data to the FERO and to execute control commands. Moreover they allow for configuring and controlling components of the FeeCom chain, especially detector-specific FeeServices representing monitored values.

Four different command channels have been defined in the FED-API. They can be divided in two groups: Configuration and Control of the FERO and Configuration and Control of the FeeCom chain. These channels are used to transfer data from the supervisory layer to the control layer (PVSS → FedServer). In detail these channels are the following:

- **Configuring Front-End Read-Out:**
  This DIM Command channel has the name ***ConfigureFERO***. It is intended to configure the FEEs. Therefore the FedClient sends configuration identification tags to the FedServer. The actual configuration data is stored in the DCS Configuration DB and retrieved using the tag(s). After the retrieval the data are prepared by the CoCo and then handed to the ICL as Binary Large OBjects (BLOB) for delivery to the FEEs. This leaves the actual configuration data transparent to the detector-independent modules of the FeeCom chain. If more than one tag is given, the CoCo has to iterate over the list of tags and fetch all corresponding DB entries, before preparing the BLOB. The same applies if a tag returns a ResultSet rather than a single data set.

  In addition, by providing a target name the FedClient tells the ICL for which FeeServer[12] the configuration command has been issued. The target name is transferred by a 20 Byte char array. Shorter target names have to be padded in order to match the defined DIM Command structure representing this channel.

[12] Each FeeServer corresponds to a set of FEEs that can be addressed by one DCS board. In the TPC case it corresponds to a *partition*, in TRD to a *layer*, in PHOS to a *sector* in a module.

The detector experts have agreed on the usage of human readable versions of target names[13]. The usage of the wildcard character "*" in the target name allows for broad- and multicasts[14]. This is achieved by means of the numbering scheme inside a detector structure[15]. For the detectors SPD, TPC and TRD the general structures look like the following:
spd-fed_<sector>_<layer>_<halfstave> [16];
tpc-fee_<side>_<slice>_<partition> ;
trd-fee_<supermodule>_<stack>_<layer> .

Channel name	Channel content	
ConfigureFERO	*target name*	*list of tags*
	`char array [20]`	`int array[]`

Table 3.1: Structure of the **ConfigureFERO** channel: The *target name* for the particular command is transferred by a char array with a fixed size of 20 Bytes, wildcards in the target name are allowed. The *list of tags*, given by an integer array, defines the configuration(s) that ought to be fetched from the DCS Configuration DB and loaded to the FEE(s). An arbitrary number of tags is possible.

- **Control Front-End Read-Out:**
  To sent instructions directly to the FERO, the channel ***ControlFERO*** is used. This command gets its data from the supervisory layer by a char array of arbitrary size, without contacting the DCS Configuration DB in the control layer. The data is then directly sent as BLOB to the specified target(s). For the target name the same usage applies as described above.

Channel name	Channel content	
ControlFERO	*target name*	*data block*
	`char array [20]`	`char array[]`

Table 3.2: Structure of the **ControlFERO** channel: The *target name* for the particular command is transferred by a char array with a fixed size of 20 Bytes, wildcards in the target name are allowed. The *data block*, given by the char array of arbitrary size, is transferred directly to the according FEE(s).

[13] The target name encodes the detector name, as well as its location in ALICE; e.g. "trd-icl" for the TRD InterComLayer, or "tpc-fee_x_y_z" for a certain FeeServer in the TPC (x defines the side, y the slice, z the partition). It is ALICE naming conventions to use lowercase letters.

[14] With a multicast only a defined subset of nodes in a network are addressed [60].

[15] An example for a TPC multicast to all partitions on side A, slice 8 looks like this: tpc-fee_1_8_*; a TRD broadcast like this: trd-fee_*_*_* . In principle there are no restrictions in the combination of the wildcard character usage, although some would not make any sense in the use.

[16] The SPD is not using the FeeCom chain software, although the scheme used in the FED-API applies also for this detector. In principle all detectors in ALICE can be mapped to a three dimensional structure, even though for some one or two of these dimensions are left blank.

- **Configuring FeeCom:**
  The command ***ConfigureFeeCom*** allows for (re-)configuring the detector-independent and -generic[17] components of the FeeCom chain. The command itself is encoded as an ID in an integer number. An optional integer and an optional float value are used as additional command data, e.g. values for update rates or log levels. Their usage and meaning depend on the issued command. For both values at least a *"Don't-Care"* - value has to be given to match the structure of this command channel. At last a target name has to be specified by a char array of arbitrary size. The allowed targets depend on the issued command and can vary from specific FeeCom component names to FeeService[18] names provided by the FeeServer, respectively their CEs. For some, broad- and multicasts might make sense. In that case the same as described earlier applies. Some of the defined commands require the return of result data. If so, these data are handed back by the acknowledge channel as described below. A list of so far defined commands in the FeeCom chain can be found in the appendix A.1.

Channel name	**Channel content**			
ConfigureFeeCom	*CommandID*	*int value*	*float value*	*target name*
	`int`	`int`	`float`	`char array[]`

Table 3.3: Structure of the **ConfigureFeeCom** channel: The desired configuration command is given by an ID number, which is transferred as an integer. An additional value, which can be given either as integer or as float, allows for a more precise setting of the command. Both values are not mandatory in the evaluation, but at least a detector-internally defined *"Don't-Care"* - value has to be set. The *target name* can be of arbitrary size and has to define a module in the FeeCom chain to which the command shall be applied. Wildcards in the target name are allowed.

- **Control FeeCom:**
  The command channel ***ControlFeeCom*** has been introduced to control the FeeCom chain. An ID given by an integer number specifies the corresponding command. An optional integer is used to retrieve additional data for the issued command, e.g. from the DCS Configuration DB. If this tag is not used a *"Don't-Care"* - value should be set. Again, the target name is given by a char array of arbitrary size, and the same applies for the usage of broad- and multicasts, like for the other commands. In the FeeCom chain commands of this channel only correspond to the execution of the FeeSever. They do not return any acknowledgement or result data. The appendix A.2 shows a list of all defined commands in the FeeCom chain.

[17] *"Generic"* in terms of their treatment by the detector-independent part of the FeeCom chain. Although for example the names of monitored services are detector-specific and update rates for monitored values are as well, the update rate for each value is handled in the detector-independent part. The same applies for the dead bands.

[18] A published service with a monitored property of the FEE is called FeeService.

Channel name	Channel content		
ControlFeeCom	*CommandID*	*int tag*	*target name*
	`int`	`int`	`char array[]`

Table 3.4: Structure of the **ControlFeeCom** channel: An integer number defines the issued control command. Additional command data can be defined by an optional integer tag. The *target name* can be of arbitrary size and has to define the module in the FeeCom chain to which the command shall be applied. Wildcards in the target name are allowed.

In the FeeCom chain these channels are implemented as DIM Commands of the FedServer, which is a detector-independent module of the ICL. Depending on the command their further execution or delivery is handled by the ICL itself. In the design for the FeeCom chain it is not foreseen to have more than one instance of each type of these command channels.

3.4.2 FED - Services

DIM Services published by the FedServer cover the sending of data to the supervising layer. In principle an arbitrary amount of DIM Clients could subscribe to these services, but in the ALICE DCS only the FedClient located in the PVSS of the supervisory layer should subscribe to them. The main purpose of these services is to monitor the FEEs and to keep track of the results of issued commands. Monitored values are for example voltages and currents of the FEEs, as well as their configuration states. The service channels for observed values exist for grouped- and single services. A mixture in the usage of these channels is possible and can make sense for defining a priority hierarchy. Grouped service channels have been invented to reduce the amount of channels the FedClient has to subscribe to. Tests with the PVSS have shown that the DIM Client module of PVSS slows down dramatically when handling a large amount of channels. In order to enhance performance, the number of DIM Client modules representing FedClients can be increased. The following list of service channels are defined in the FED-API:

- **Grouped Service channels:**
  The ***Grouped Service channels*** provide monitored FEE properties to the supervisory layer. These can be voltages, temperatures, currents, the states of the FeeServers, or any other monitored property of the FEEs. The grouping policy should be reflected in the channel names, which have "<*Group-name*>_Service" as generic name structure. The respective Fee/Fed-Service defines which type (integer value, float value or char array) of the transmitted values is valid[19]. It is good practice to submit a *"Don't-Care"* - value for the elements of the other types. The name is given in a 256 Bytes char array. In the FeeCom chain a

[19] The char array for a transferred value has been appended at the end of the channel structure, because in the beginning of the FED-API definition, only integer and float values were defined to be Fee/Fed-Services.

value is only updated when it exceeds a dynamically set dead band. The dead band is applied around the last transmitted value of its service[20].

The main idea for grouping services is to reduce the amount of channels the PVSS has to connect to. The amount of grouped service channels depends on the detectors and on the grouping policy chosen by the latter[21].

Channel name	**Channel content**			
<Group-name>_ Service	*int value*	*float value*	*service name*	*char value*
	`int`	`float`	`char[256]`	`char[256]`

Table 3.5: Structure of the **Grouped Service** channel: The actual transferred value is either given by an integer or a float value or a 256 Byte char array. For the not-used-values a *"Don't-Care"* - value should be set. The corresponding Fee/Fed-Service name is given by a 256 Byte char array. The grouping policy for a given Grouped Service channel is indicated in the channel name.

- **Single Service channels:**
  Single Service channels are supposed to provide the supervisory layer with current measurements of observed properties of the FEEs. This includes voltages, currents, temperatures, etc., as well as the current state of the FEEs. The ***Single Service channels*** represent only one value. The corresponding name is given by the channel name, which has the following generic structure: "*<Fee/Fed-ServicesName>*". The actual value is transmitted as an integer, a float or a char array, respective to the type of the value. The elements of the not-used data types should contain a detector defined *"Don't-Care"* - value. Integer and float values are only updated, if their current value exceeds a given dead band around their last transmitted value.

  The amount of used ***Single Service channels*** depends on the detector and the usage of ***Grouped Service channels***.

Channel name	**Channel content**		
<Service-name>	*int value*	*float value*	*char value*
	`int`	`float`	`char array[]`

Table 3.6: Structure of the **Single Service** channel: The actual transferred value is either given by an integer or a float value or a char array of arbitrary size. For the not-used-values a *"Don't-Care"* - value should be set. The corresponding Fee/Fed-Service name is given by the channel name.

[20] Dead bands are only applied to FeeServices represented by integer or float values. Char services do not have any dead bands.

[21] Examples for grouping policies can be 'grouping by FeeServers' or 'grouping-by-observable-property', like temperatures, voltages or states.

- **Acknowledge channel:**
  Normally, commands which are sent to the FEEs, return an acknowledgement (ACK) of their execution or an appropriate error code. This is given by an integer number[22]. In addition, possible result data, depending on the issued command, can be appended. They are transported by a char array of arbitrary size, which can also be empty. If the corresponding afore issued command has been executed by a detector-specific part of the FeeCom chain, the ICL and FedServer handle the result data as BLOB. Time-outs for commands issued to the FeeServers are measured by the ICL. An appropriate error code is generated accordingly if a FeeServer does not answer in time.

  All these information are sent back to the supervisory layer via the ***Acknowledge channel***. The amount of ACK channels depends on the detector, the control design and the handling of the command channels. In the FeeCom chain, the FedServer hosts an ACK channel for each FeeServer plus one for the ICL. Broadcasts to the FeeServers should be answered via the ICL ACK channel. Since there can be several ACK channels a generic structure for the channel name has been defined: "*<Source-name>*_ACK". In the FeeCom chain the *<Source-name>* is given by the FeeServer name respectively the ICL.

Channel name	**Channel content**	
<Source-name>_ ACK	*ACK / error code*	*result data*
	`int`	`char array[]`

Table 3.7: Structure of the **Acknowledge** channel: An *ACK* is represented by a "0", else the appropriate error code is sent as an integer number. Possible additional *result data* is appended in a char array of arbitrary size. The char array can be empty, if no results are provided. The source of this channel is encoded in the channel name.

- **Message channel:**
  The ***Message channel*** is used to send log messages from the lower layers (field - and control layer) to the supervisory layer[23]. Its generic name encodes the detector name / acronym in the channel name: "*<DET>*_MSG". An event type encoded in an integer number represents the severity of the message[24]. The detector, where the event has occurred, is given by a 4 Byte char array. A 256 Byte char array defines the component, which has issued the log message. Another 256 Byte char array is reserved for the description of the log event. Finally a 20 Byte char array shows the timestamp of the log message. The structure of the timestamp is defined as "YYYY-MM-DD hh:mm:ss\0". If the contents of the afore-mentioned char arrays are shorter than the defined size,

[22] A "0" represents an "OK" (ACK); some of the error codes are given by the FeeCom chain, the rest depends on the definitions in the detector-specific components.

[23] In the FeeCom chain, these messages are additionally written to file by the ICL.

[24] In appendix A.3 a list with the possible log types / levels is given.

they have to be padded with "\0"s. The structure of the ***Message channel*** is common among all participating detectors in ALICE DCS and used as well in the communication between FeeServer and ICL.

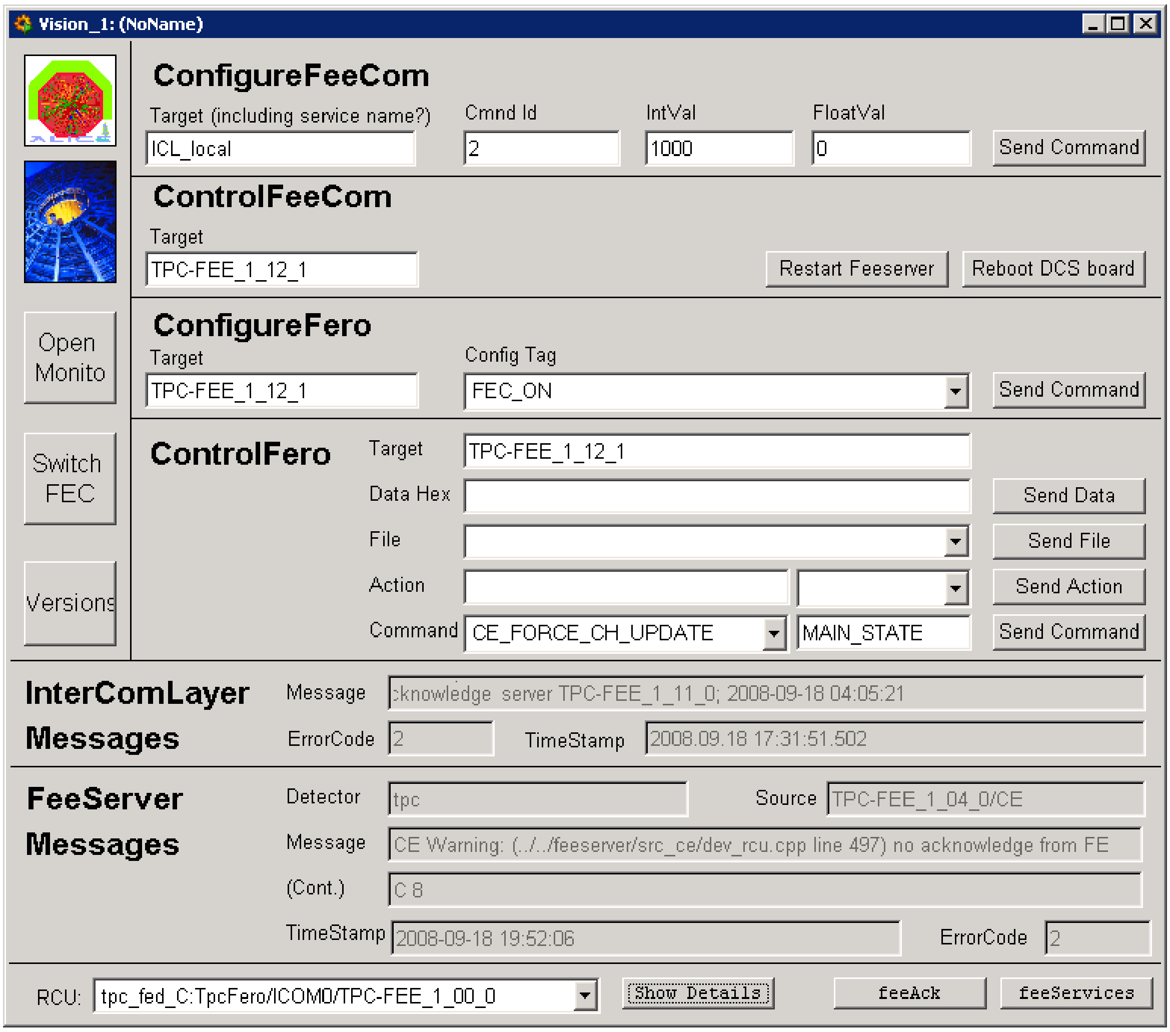
Channel name	**Channel content**				
<DET>_MSG	*logType*	*detector*	*source*	*description*	*date*
	`int`	`char[4]`	`char[256]`	`char[256]`	`char[20]`

Table 3.8: Structure of the **Message** channel: The *logType* (log level) is given by an (unsigned) integer. The corresponding detector is transferred by its acronym in a 4 Byte char array. A finer granularity of the *source* is given by a 256 Byte char array. The actual message is sent by a 256 Byte char array, too. A 20 Byte char array is used for the timestamp of the message.

All the above mentioned channels are implemented in the FedServer module of the ICL. They allow DCS to have full control over the FeeCom chain and their connected FEEs [47] [48] [61] [62]. A display showing the TPC FED-Client on a PVSS panel is given in figure 3.6[25].

The service channel part of the FED-API is also used in the interfaces from the HLT to the DCS. Data that have been calculated online in the HLT and are of interest to the DCS, can be transferred to the DCS system using the FED-API. Therefore the HLT has implemented the above mentioned service channels in its framework. More details about the HLT - DCS connection via the FED-API is described in section 4.5.2.

[25] The PVSS panels for the TPC have been developed by Dr. Christian Lippmann (CERN).



Vision_1: (NoName)

ConfigureFeeCom

Target (including service name?): ICL_local
Cmnd Id: 2
IntVal: 1000
FloatVal: 0

ControlFeeCom

Target: TPC-FEE_1_12_1

ConfigureFero

Target: TPC-FEE_1_12_1
Config Tag: FEC_ON

ControlFero

Target: TPC-FEE_1_12_1
Data Hex:
File:
Action:
Command: CE_FORCE_CH_UPDATE MAIN_STATE

InterComLayer Messages

Message: :knowledge server TPC-FEE_1_11_0; 2008-09-18 04:05:21
ErrorCode: 2
TimeStamp: 2008.09.18 17:31:51.502

FeeServer Messages

Detector: tpc
Source: TPC-FEE_1_04_0/CE
Message: CE Warning: (../../feeserver/src_ce/dev_rcu.cpp line 497) no acknowledge from FE
(Cont.): C 8
TimeStamp: 2008-09-18 19:52:06
ErrorCode: 2

RCU: tpc_fed_C:TpcFero/ICOM0/TPC-FEE_1_00_0

Figure 3.6: The figure shows a screenshot of a PVSS panel displaying the FED-Client implementation for the TPC.

Chapter 4

The HLT interfaces

4.1 Design methodology

Most of the design and implementation of the various ALICE online / offline systems was already done or in progress when the project for developing a calibration framework for the ALICE HLT started. This fact had implications on the choice and on the design of the HLT interfaces, especially on the mechanisms and protocols for communication. The environment constituted itself as very heterogeneous with a mixture of various proprietary and of-the-shelf techniques. A first goal had been to depict the diversity in the setup. In this process UML (Unified Modelling Language) has been used as a tool to visualise the existing design. In a second step the other systems have been interfaced making the various mechanism used inside interoperable. Again UML has been used to display smaller aggregations of the setup. In the following a brief description of the UML notation employed in several figures within this thesis is given[1]. The notation is following the UML 2.0 standard[2] [63].

4.1.1 UML notation overview

UML 2.0 introduces a set of 13 different diagram types with symbol notations according to the needs of their context. Some of the symbols and elements are common among several or all diagram types, certain notations are domain specific. Additional information to classifiers and relations between them can be given by attaching a note to them. *Stereotypes*, which assign an element in the model to a certain group or style are marked by *"«...»"* surrounding the name of the stereotype. Both notations are used in all digram types [63].

[1] The description here is limited to the elements used in the UML diagrams in this thesis, the UML standard includes many more.

[2] The Object Management Group (OMG) is responsible for the specification of UML, the standard for UML 2.0 can be downloaded under: `http://www.uml.org/` .

Notation UML Composite Structure Diagram

The internal structure of a system and the collaboration with other system components can be visualised by UML Composite Structure Diagrams. The notations are the following: Parts and components are represented by boxes. Connectors and ports depict special interaction modules (small boxes attached to the component boxes). Dedicated interfaces can be drawn via the so called *lollipop* representation. The relations (*edges*) between components are given by arrows. The different elements of the notation are presented in figure 4.1.

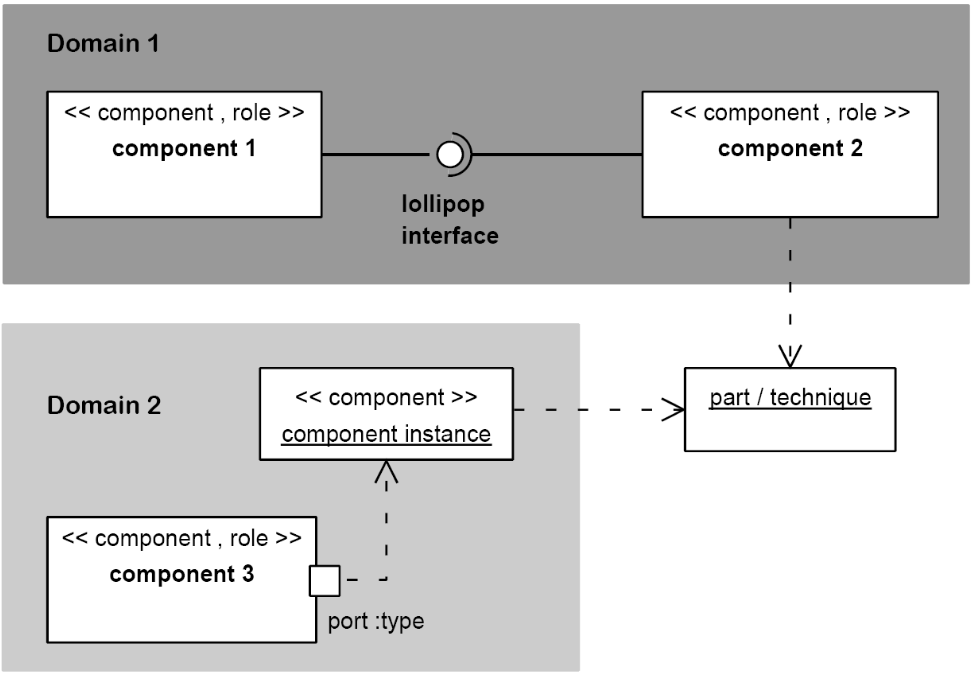


Figure 4.1: The figure shows the different notation elements in a UML Composite Structure Diagram.

Notation UML Class Diagram

UML Class Diagrams can model the design of an application written in object-oriented programming languages. These diagrams display the classes, their inheritance(s) and association(s). Classes are displayed as boxes with the class name written inside. Methods and attributes of the classes can be mentioned in the boxes as well. Inheritance between two classes is signalled by an arrow with a closed but unfilled arrow head. In addition to the normal inheritance notation, the implementation of an interface can be expressed by lollipops, too. The associations between classes can occur in different flavours: simple association, aggregation or composition. They normally express a *"has a"* relation. They are drawn by a simple line or by an arrow with an open arrow head. Multiplicities in the relation are given by numbers at the ends of the connection lines. In addition the UML Class Diagram allows to order the classes into the corresponding packages depicted by symbolic file folders [63]. The notation elements are described in figure 4.2.

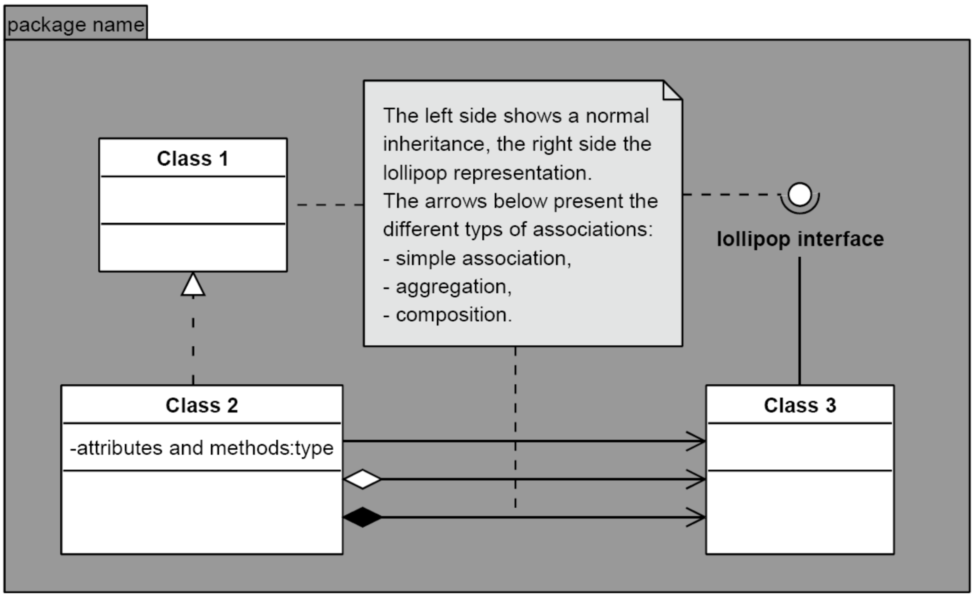

package name
Class 1
The left side shows a normal inheritance, the right side the lollipop representation.
The arrows below present the different typs of associations:
- simple association,
- aggregation,
- composition.
lollipop interface
Class 2
-attributes and methods:type
Class 3

Figure 4.2: The figure shows the different notation elements in a UML Class Diagram.

Notation UML Deployment Diagram

The distribution of the different system components in an environment can be displayed by UML Deployment Diagrams. These diagrams show the deployment of applications and modules on the various nodes in the system. Nodes in a domain are represented by 3-dimensional boxes, special roles of the nodes can be appended to the node description / name. Associations between nodes are given by lines and arrows. Components and applications (classifiers in normal boxes) running on these nodes are assigned by *«deploy»* associations (*deployments*) [63]. Figure 4.3 shows the notation elements of UML Deployment Diagrams.

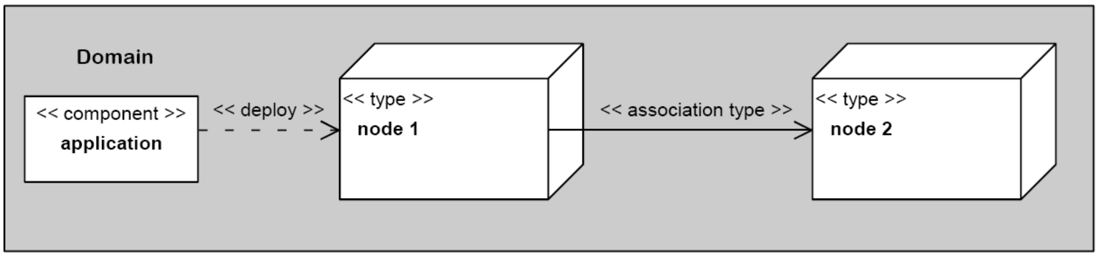


Figure 4.3: The figure shows the different notation elements in a UML Deployment Diagram.

Notation UML Use Case Diagram

The UML Use Case Diagrams visualises use cases of a setup. The different modules are represented by elliptic circles, their associations are given by arrows or lines. Special relations, like *"include"* or *"extend"*, are signalled by attached *«stereotypes»*. Actors interacting with the system are indicated by stick figures, their role in the system is given by the name of the stick figure [63]. The notation elements are presented in figure 4.4.

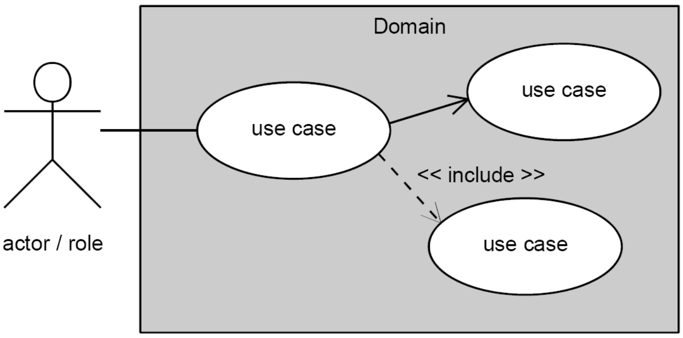


Figure 4.4: The figure shows the different notation elements in a UML Use Case Diagram.

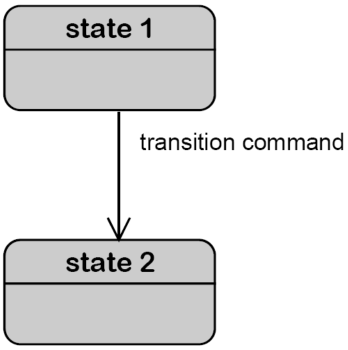


Figure 4.5: The figure shows the different notation elements in a UML State Machine Diagram.

Notation UML State Machine Diagram

The internal states and state transitions of a system are displayed in UML State Machine Diagrams. The different states are represented by boxes with rounded corners, their transitions are given by arrows indicating the transition direction. Transition command names can be added to the arrows [63]. The notation is shown in figure 4.5.

Notation UML Activity Diagram

The order of tasks performed by a component or application can be visualised with UML Activity Diagrams. Normally, these diagrams have one entry point (black circle) and at least one exit point. The main exit point is marked by a black dot in a white circle, exit points of side control flows have an "×" in the white circle. The different activities are represented by rounded boxes, object nodes are given by simple boxes. Incoming and outgoing signals have their own representation (see UML Activity Diagram example in figure 4.6). The flow (*edge*) of actions are signalled by arrows. Branching in the flow is indicated by diamonds, a break or an "exception" in the activity flow can be drawn by a zigzag line. Parallelisation and synchronisation of control flows are presented by black bars with incoming and outgoing arrows. Structured nodes like loops or subcomponents are visualised by big rounded boxes with dashed lines.

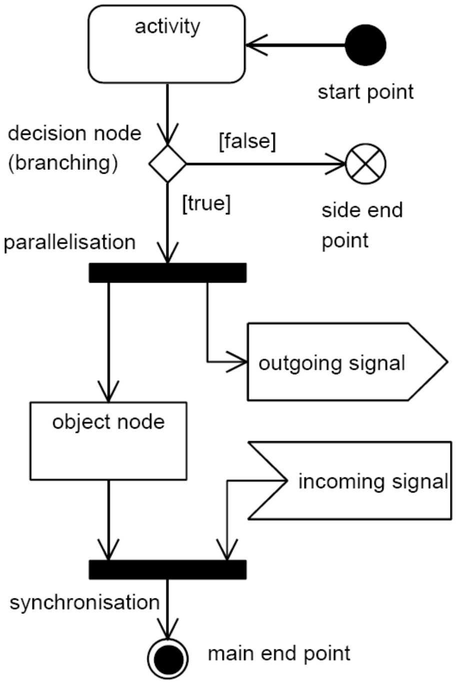


Figure 4.6: The figure shows the different notation elements in a UML Activity Diagram.

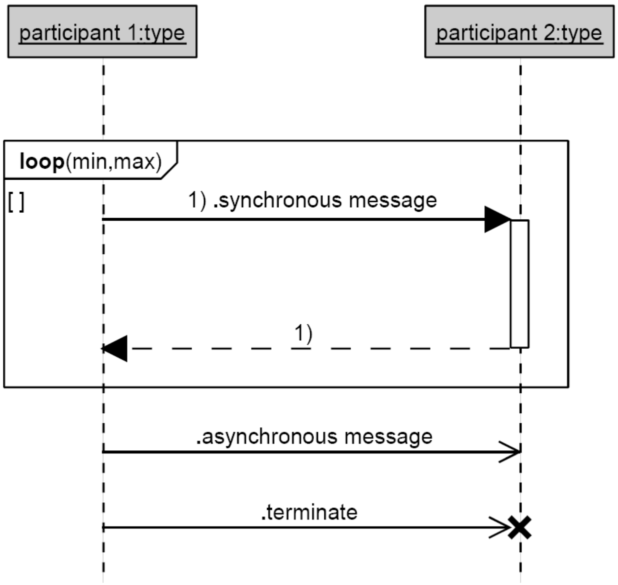


Figure 4.7: The figure shows the different notation elements in a UML Sequence Diagram.

Notation UML Sequence Diagram

The UML Sequence Diagram shows the order of interactions (*messages*) between the participants in a system. A participant is represented by a box with an outgoing *life line*. Messages between the participants are indicated by arrows between the life lines. These messages can be synchronous or asynchronous. Synchronous messages have a closed and filled arrow head and a return message when the corresponding task has finished. The duration of the task is shown by a bar on the life line of the participant. The return message is indicated by an arrow with a dashed line. Asynchronous messages have an open arrow head. The termination of a life line is indicated by an "×"; it can be issued by other participants as well. Control flow instructions (*interaction operators* – loops, branching, etc.) can be incorporated in so called *combined fragments* [63]. These notations are shown in figure 4.7.

Other UML Diagram types

The set of diagrams defined in UML 2.0 comprises 13 different diagram types. In addition to the afore describe ones, UML 2.0 provides Component Diagrams, Package Diagrams, Object Diagrams, Communication Diagrams, Timing Diagrams and Interaction Overview Diagrams for modelling a system [63]. They are just mentioned here for completeness reasons, but they are not used in the further description of the HLT Calibration Framework.

4.2 Interfaces overview

In order to fulfil the tasks mentioned in section 2.4.5 the HLT has redundant interfaces to the various systems in ALICE. These include the ALICE online systems like the ECS, DAQ and DCS, as well as ALICE Offline and the AliEve.

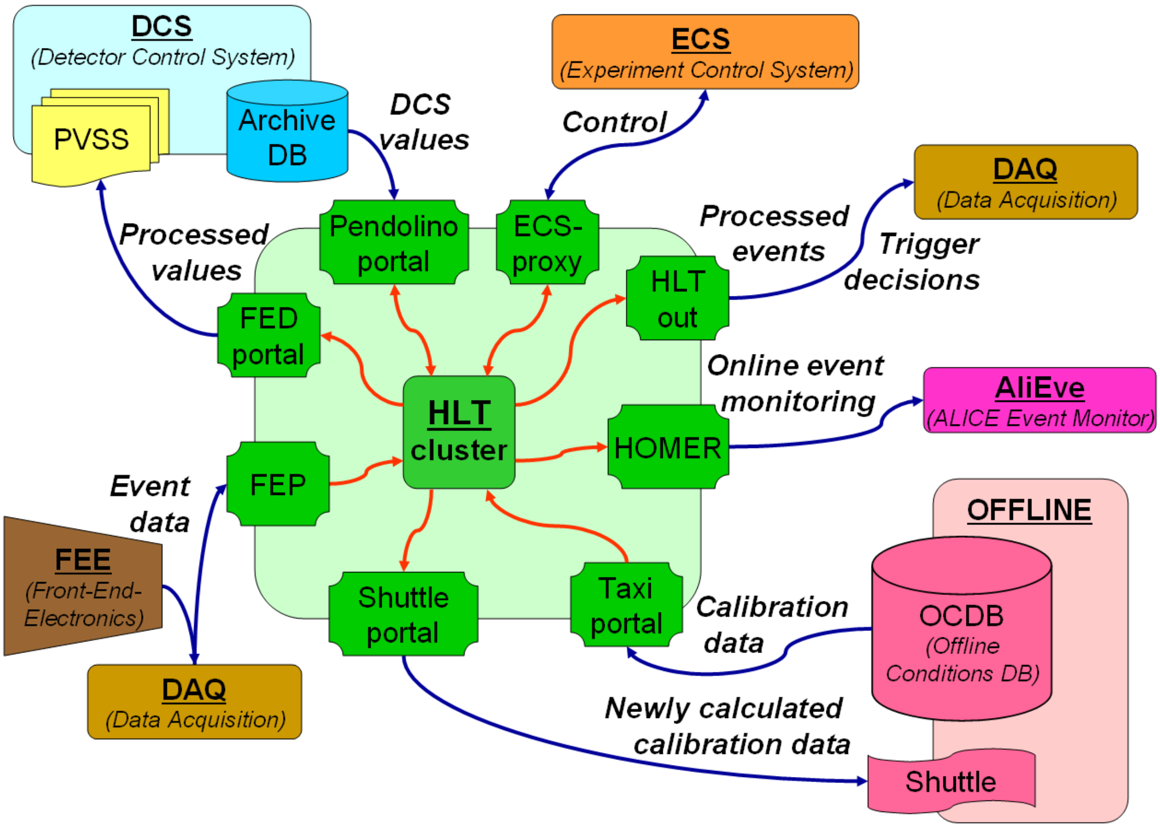


Figure 4.8: The figure shows the different interfaces of the HLT with the other systems in ALICE.

The raw event data are received on HLT side as copies from the DAQ-LDCs and analysed inside the HLT. Afterwards the results, trigger information and compressed event data are sent back to the DAQ nodes for permanent storage. Interfaces to the other systems provide the analysis software of the HLT with required additional input. Calculated results are sent back to the corresponding systems. In the DCS and Offline case the tasks for communication in the two exchange directions (receiving data and sending data) are separated in different applications. DCS values are fetched via the so called Pendolino, while HLT is able to return data back to the DCS over the FED-API. Offline can fetch data from HLT using the Offline Shuttle mechanism and the HLT can retrieve calibration settings from the OCDB in Offline via the HLT Taxi. Online monitoring is achieved by interfacing AliEve. A sketch of these interfaces is shown in figure 4.8, the different involved interfaces are analysed step-by-step in the following sections. They are the main topic of this thesis. Most of the figures in the following descriptions use a colour code for distinguishing between components of the different ALICE systems: green is used for the HLT related parts, orange for the ECS, brown for the DAQ, blue for the DCS, pink for Offline and the AliEve and yellow is used to indicate detector-specific components inside other systems.

The interface applications are hosted by dedicated portal nodes with at least two network interfaces - one for connecting to the corresponding system, and one for collecting or distributing data inside the HLT cluster. The portal nodes are redundant, in case of a failure of one portal the backup node takes over the corresponding task.

Due to the heterogeneity of the ALICE systems different techniques are applied to connect to the different parts. The UML Composite Structure Diagram in figure 4.9 visualises the heterogeneity and complexity of the system and shows the variety of the different techniques used. The employed mechanisms are described in more detail in the following sections. The interfaces to the DAQ and the AliEve are not in the main focus of this thesis, and are mentioned here only very briefly for completeness reasons.

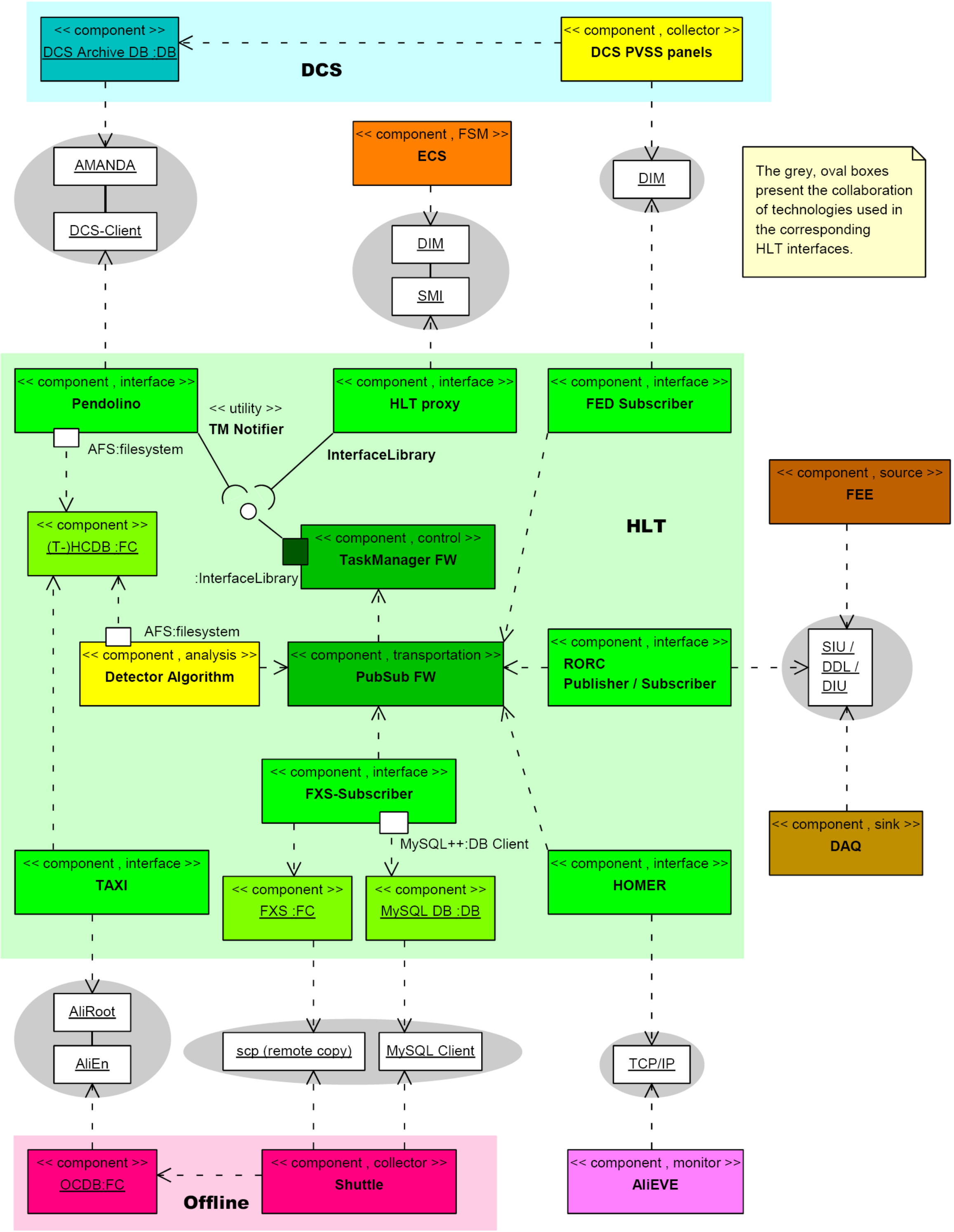


Figure 4.9: This UML Composite Structure Diagram visualises the complexity and heterogeneity of mechanisms used in the different HLT interfaces. The grey circles depict the techniques used in interfacing other ALICE systems. The names and acronyms used in the diagram are explained in the description of the corresponding interfaces.

4.3 ECS

As described in section 2.4 the ECS is responsible for steering and synchronising the different ALICE systems and detectors. The ECS interface integrates the HLT into the global ALICE control and provides the HLT with certain overall run conditions. Two nodes of the HLT cluster are assigned for the connection to ECS. They own a dedicated ethernet connection to the DAQ counting room, which hosts the ECS computers. The high level interface is taken care of by the so called HLT-proxy.

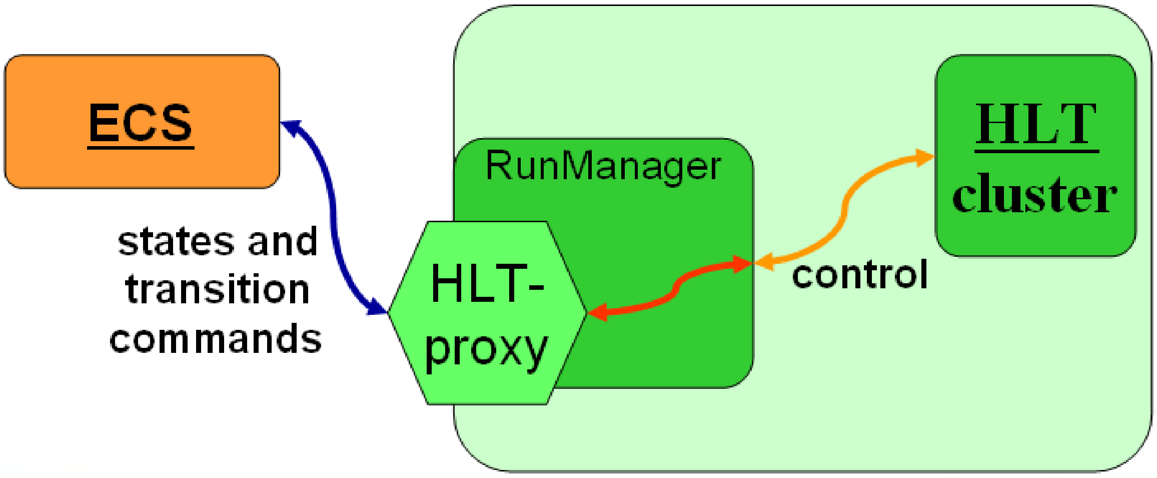


Figure 4.10: Sketch of the ECS - HLT interface. ECS interacts with HLT via states and transition commands (+ additional parameters).

4.3.1 HLT-proxy

During a run ECS controls all online systems in ALICE. Therefore each system like the HLT, DCS, DAQ, TRG and all detectors have implemented Finite State Machines (FSM) as interfaces towards the ECS[3]. From an abstract point of view a FSM consists of a collection of well defined states and transitions between these states.

ALICE has chosen to implement these state machines with the SMI++ (State Machine Interface) framework. SMI++ has been developed for the DELPHI experiment at CERN and the BaBar experiment at SLAC. The framework provides a dedicated language to describe entities in a project and their states, the State Manager Language (SML). Tools and libraries, written in C++, interpret this language and run logic engines for each entity described in the SML file. The connections to the real representations of the entities are done via so called proxies. These proxies inform the logic engines about their current states and accept new transition commands. Each system in an SMI setup can be divided into subsystems. This subcategorisation is called partitioning in SMI and allows to have several instances of a system for different purposes. For data exchange between proxies and logic engines the SMI package

[3] Dr. Franco Carena (CERN), Dr. Sylvain Chapeland (CERN) and Dr. Jean-Claude Marin (CERN) have developed the corresponding modules on the ECS side.

uses DIM as communication framework. The transition commands and state changes are communicated as DIM Command and DIM Services. Therefore a DIM_DNS is required in the setup as well[4] [64] [65] [66].

In ALICE the HLT is represented by one logic engine on the ECS side. This logic engine can be partitioned into several independent HLT entities, if some ALICE detectors are running independently of the global run in the HLT. Every set of detectors performing analysis inside the HLT can be a composite of a partition. In this case the different partitioned logic engines of the HLT represent the part of the HLT which is connected to the corresponding detectors. For each partition a partitioned HLT-proxy has to be started on the HLT side. It is connected to the RunManager controlling the analysis chain of the corresponding detector(s). They communicate as well with their corresponding logic engine(s) on the ECS side. If all detectors are steered by the global run control, the whole HLT acts as a unity.

The deployment of the different modules of the HLT - ECS interface is shown in figure 4.11. On the ECS side there exist the logic engine and the graphical user front-end to steer the whole setup, in addition with the DIM_DNS, which is responsible for the underlying DIM communication. The HLT hosts the corresponding HLT-proxy, the representation of the particular partition of the HLT to the ECS world and the set of RunManager and TaskManagers for the internal control.

The HLT-proxy is written in C++ and acts as a kind of adapter (Adapter Design pattern [67]) between the HLT logic engine on the ECS side (based on SMI++ as control mechanism) and the TaskManager framework for the HLT internal steering (see section 4.3.2). The TaskManager framework is a proprietary development of the HLT collaboration.

For state transitions ECS issues transition commands. The HLT-proxy receives them via DIM channels through the SMI++ framework. After mapping these commands to the syntax of the TaskManager system, the HLT-proxy relays them to the RunManager. For the contact with the RunManager the HLT-proxy uses the InterfaceLibrary of the HLT framework and its internal communication mechanism [42]. The same applies for signalling the current RunManager states in the opposite direction. The proxy polls for this state in regular time intervals. After mapping of the state name to the set used by ECS, it compares the newly received state name with that of the current state. If a change has happened, a check for a valid transition is performed and the ECS is informed. The polling of the states also acts as an intrinsic alive checks of the RunManager.

Well defined transition commands are accepted in each stable state[5]. The HLT enters a transition state, which implicitly changes to the next stable state, when all corresponding tasks for the state transition are done. The different states and their transition commands are described in more detail in section 4.3.2.

The UML State Machine Diagram in figure 4.12 shows the interface defined between HLT and ECS. In this diagram the stable states are shown in the middle column. The

[4] Both, SMI++ and DIM, have been developed by Dr. Clara Gaspar (CERN).

[5] Stable states do not transit automatically to another state without a command, except of a transition to the ERROR state.

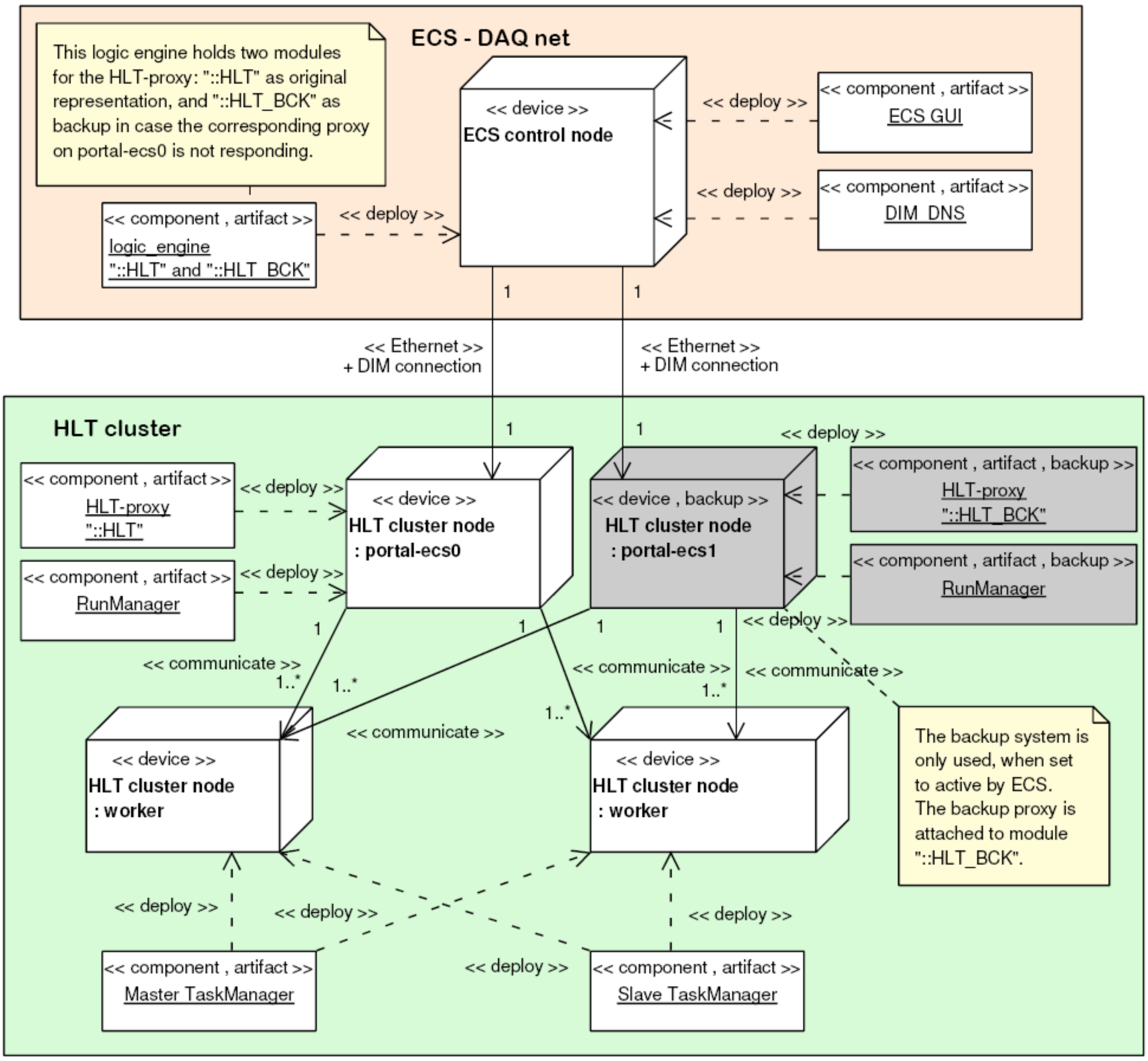


Figure 4.11: UML Deployment Diagram of the HLT-ECS interface. In ALICE the ECS and the DAQ are located on the same network. For simplification DIM_DNS, logic engine and ECS GUI are located on one node. As long as they are together with the HLT-proxy in the same DIM subnet, they can be distributed over several nodes. The logic engine hosts two modules for HLT-proxies: "***::HLT***" and "***::HLT_BCK***". For redundancy a backup system ("***::HLT_BCK***") is installed on a different node. It can be set to active by ECS, in case the primary interface node portal-ecs0 is not responding. Then portal-ecs1 takes over. The RunManager can run on a different node in the HLT cluster than the ECS-portal. Due to communication via the InterfaceLibrary of the HLT framework this is transparent to the HLT-proxy.

right column displays all intermediate states for the path from **OFF** to **RUNNING** with the implicit transition to the next stable state after all tasks for the corresponding transition state have been finished; the left side displays the opposite way.

As described earlier in this section, the HLT-proxy adapts the ECS control to the HLT internal control system, i.e. the TaskManager framework. Therefore the states and commands defined in this interface have to be mapped to the names used by the HLT framework. Figure 4.13 shows the mapping of states and transition commands. The same structure is applied like for figure 4.12.

During the complete configuration process several parameters are required to prepare the HLT for the upcoming run. These parameters can be divided into CONFIGURE parameters and ENGAGE parameters.

The CONFIGURE parameters are a collection of general settings that can be valid for a series of runs in a row. For all CONFIGURE parameters a default value can be set, in case the ECS signals "DEFAULT" for them. These configuration defaults as well as the connection details[6] for the TaskManager InterfaceLibrary are fetched from a dedicated property file per partition. While connection details are only read on start up of the HLT-proxy, the default values can be changed during run time and are evaluated each time a CONFIGURE command is received.

CONFIGURE parameters:

- **DETECTOR_LIST**: List of detectors participating in this partition.
- **BEAM_TYPE**: The experiment type (p + p (proton-proton), p + A (proton-heavy ion) or A + A (heavy ion)).
- **DATA_FORMAT_VERSION**: The expected data format version that the HLT writes out to the DAQ.
- **HLT_TRIGGER_CODE**: The HLT Trigger classes, which represent the desired HLT configuration for the upcoming run.
- **HLT_IN_DDL_LIST**: List of active DDL cables entering the HLT counting room and gathered in patch panels. From the patch panel the links are connected to H-RORCs in the FEPs. From the cables mentioned in the list the HLT can expect data from DAQ in the upcoming run. The items of the list encode the cable names and their corresponding detector parts as key-value-pairs[7] [68] [69].
- **HLT_OUT_DDL_LIST**: List of DDLs, through which the HLT can send its output data back to the DAQ[8] [68] [69].

[6] The connection details describe the host and port, where the RunManager/ Master-TaskManager can be contacted.

[7] For separation of each pair commas (",") are used. The cable name (key) and corresponding detector part (value) are separated by colons (":"). This leads to the following structure: <*CableName*>:<*DetectorPart*>,<*CableName*>:<*DetectorPart*>,...

[8] The items of the list are separated by commas (",").

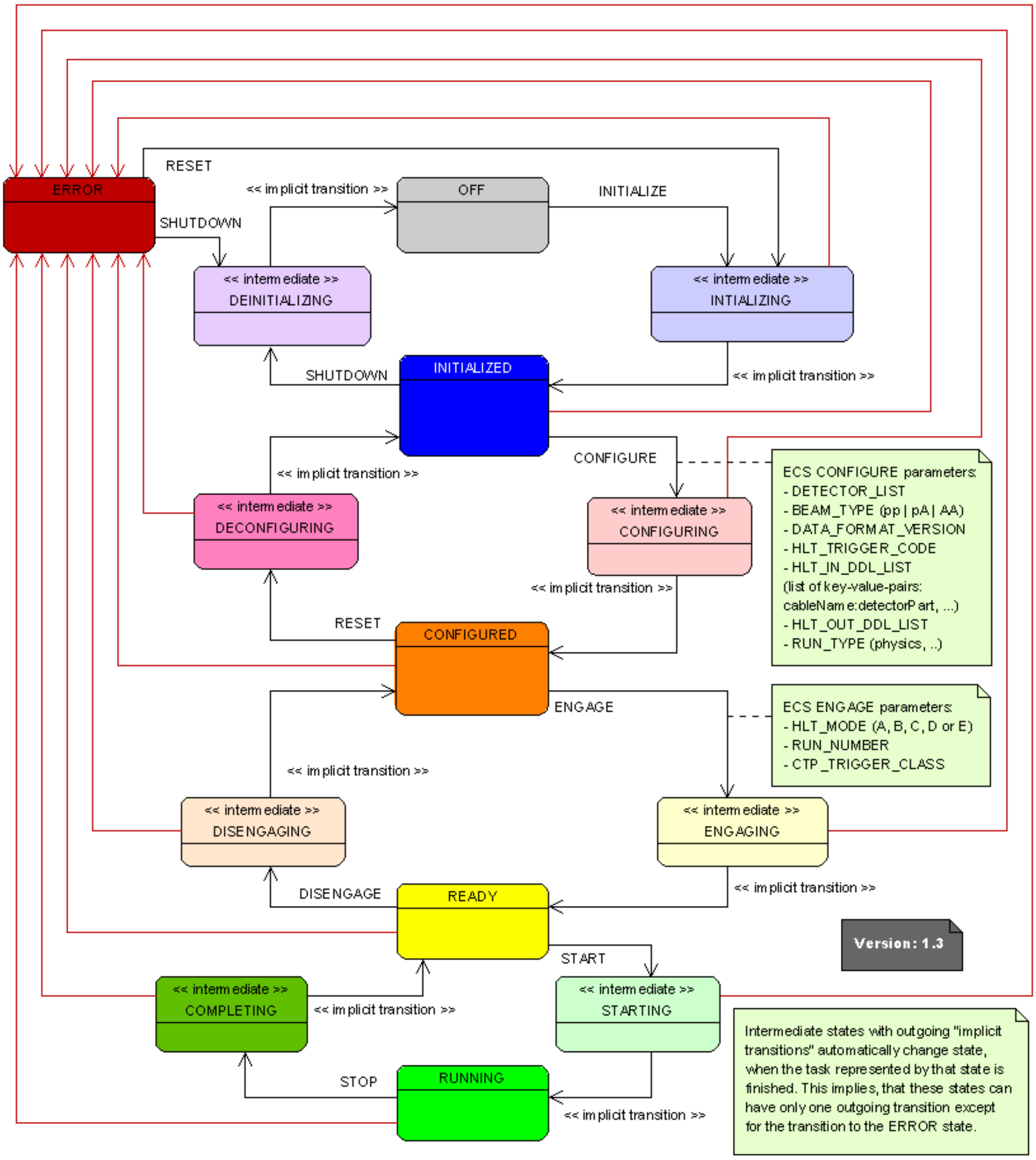
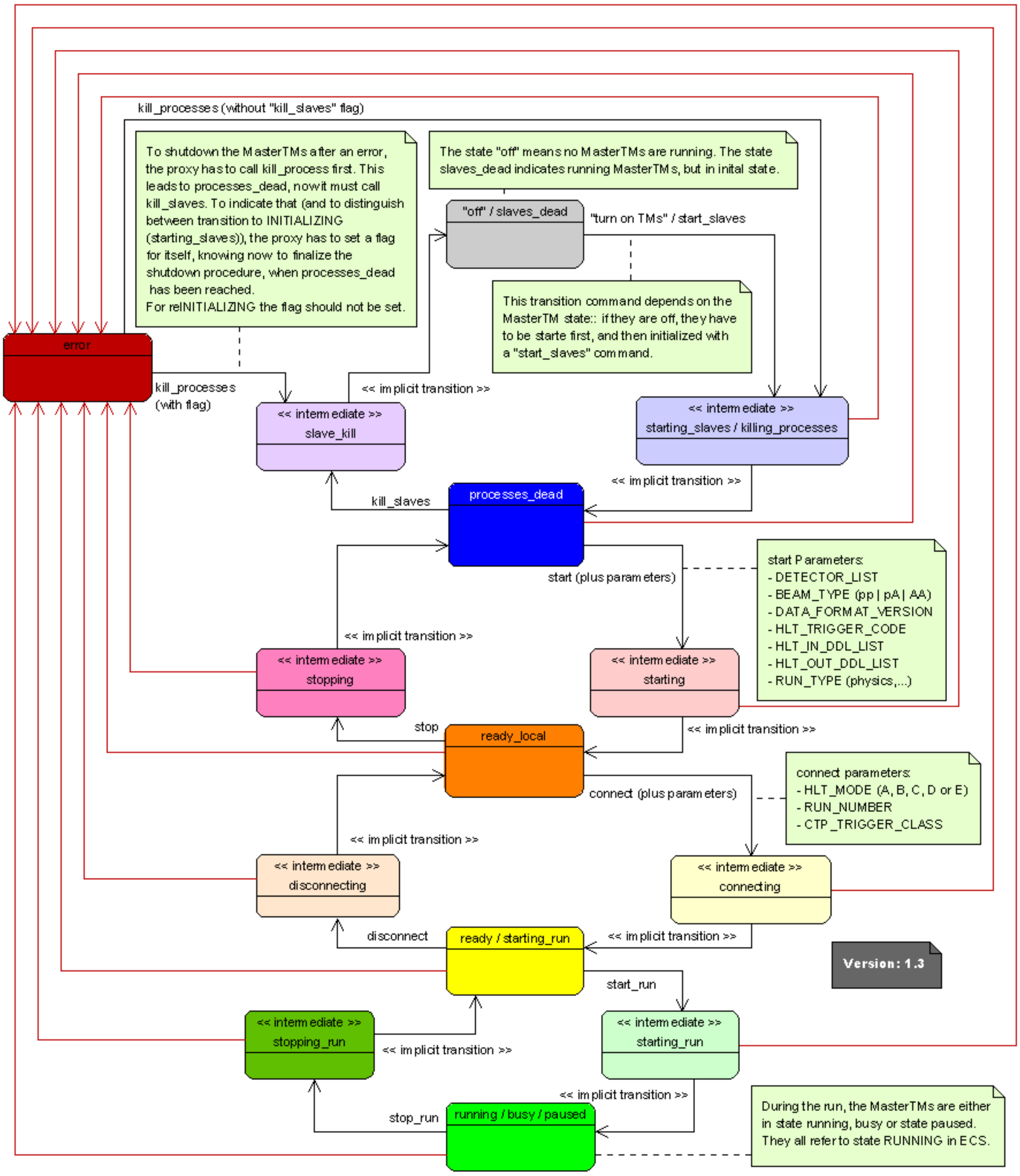

ERROR
RESET
SHUTDOWN
<< implicit transition >>
OFF
INITIALIZE
<< intermediate >>
DEINITIALIZING
<< intermediate >>
INTIALIZING
SHUTDOWN
INITIALIZED
<< implicit transition >>
CONFIGURE
<< implicit transition >>
<< intermediate >>
DECONFIGURING
<< intermediate >>
CONFIGURING
ECS CONFIGURE parameters:
- DETECTOR_LIST
- BEAM_TYPE (pp | pA | AA)
- DATA_FORMAT_VERSION
- HLT_TRIGGER_CODE
- HLT_IN_DDL_LIST
(list of key-value-pairs:
cableName:detectorPart, ...)
- HLT_OUT_DDL_LIST
- RUN_TYPE (physics, ..)
<< implicit transition >>
RESET
CONFIGURED
ENGAGE
ECS ENGAGE parameters:
- HLT_MODE (A, B, C, D or E)
- RUN_NUMBER
- CTP_TRIGGER_CLASS
<< implicit transition >>
<< intermediate >>
DISENGAGING
<< intermediate >>
ENGAGING
DISENGAGE
READY
<< implicit transition >>
Version: 1.3
START
<< intermediate >>
COMPLETING
<< implicit transition >>
<< intermediate >>
STARTING
STOP
RUNNING
<< implicit transition >>
Intermediate states with outgoing "implicit transitions" automatically change state, when the task represented by that state is finished. This implies, that these states can have only one outgoing transition except for the transition to the ERROR state.

Figure 4.12: UML State Machine Diagram of the HLT-proxy. The stable states are located in the middle column, the right column shows the intermediate states from OFF to RUNNING, the left column the reverse direction.



To shutdown the MasterTMs after an error, the proxy has to call kill_process first. This leads to processes_dead, now it must call kill_slaves. To indicate that (and to distinguish between transition to INITIALIZING (starting_slaves)), the proxy has to set a flag for itself, knowing now to finalize the shutdown procedure, when processes_dead has been reached.
For reINITIALIZING the flag should not be set.

The state "off" means no MasterTMs are running. The state slaves_dead indicates running MasterTMs, but in inital state.

This transition command depends on the MasterTM state:: if they are off, they have to be starte first, and then initialized with a "start_slaves" command.

start Parameters:
- DETECTOR_LIST
- BEAM_TYPE (pp | pA | AA)
- DATA_FORMAT_VERSION
- HLT_TRIGGER_CODE
- HLT_IN_DDL_LIST
- HLT_OUT_DDL_LIST
- RUN_TYPE (physics,...)

connect parameters:
- HLT_MODE (A, B, C, D or E)
- RUN_NUMBER
- CTP_TRIGGER_CLASS

During the run, the MasterTMs are either in state running, busy or state paused. They all refer to state RUNNING in ECS.

Figure 4.13: UML State Machine Diagram showing the mapped HLT-proxy states to the state names used by the HLT TaskManager framework.

- **RUN_TYPE**: The run type of the upcoming run (physics, pedestal, etc.). For each detector several run types are defined, which can be mainly divided into production runs and calibration runs[9].

In order to speed up the configuration process, the more general settings (CONFIGURE parameters) are decoupled from the parameters which change from run to run. The latter ones are collected as ENGAGE parameters and are sent together with the ENGAGE command. This allows ECS to leave the HLT in the state CONFIGURED after a run (see figure 4.12), in case that the same configuration settings apply also for the next run. Measurements for the configuration time after a CONFIGURE command have been over one minute for the TPC alone. To minimise this time consuming step and to prevent delays in the start of ALICE runs, it is only performed once for a series of runs with the same general settings.

The ENGAGE parameters **HLT_MODE** and **RUN_NUMBER** take no default values, the HLT mode and the run number must always be defined by ECS. The **CTP_TRIGGER_CLASS** can have a default value, which is stored in the Property file of the HLT-proxy [7] [8] [40] [45].

ENGAGE parameters:

- **HLT_MODE**: The mode in which the HLT runs:
  - **A**: In this mode ALICE runs without the HLT. This leaves the control on the HLT side for calibration, cluster maintenance, testing, etc. .
  - **B**: The HLT participates in the upcoming run and performs its tasks of online monitoring and final trigger decisions but decisions are discarded by the DAQ. Nevertheless, the HLT output is stored for later analysis and comparison with offline results.
  - **C**: This is the main mode for the HLT. The HLT runs fully operational and its trigger decisions are taken into account by the DAQ.
  - **D**: The HLT receives data from the DAQ, but does not write anything back to the DAQ via the HLT-out. There is no flow control at the DDL level. This mode is also known as B\test1.
  - **E**: The HLT receives data from the DAQ, but does not write anything back to the DAQ via the HLT-out. The flow control is switched on at the DDL level. This mode is also known as B\test2.
- **RUN_NUMBER**: The run number for the upcoming run[10].
- **CTP_TRIGGER_CLASS**: The trigger classes used in the Central Trigger Processor. This can be a list of IDs.

[9] The list of different run types defined in ALICE can be found here: http://alice-ecs.web.cern.ch/alice-ecs/runtypes_3.16.html

[10] A run is defined as the continuously taking of data for some time with the same experiment conditions. Each run is identified by a run number, which is increased by one from run to run.

4.3.2 HLT RunManager

Each set of detectors, which is compiled as an HLT partition, owns a RunManager that is connected to the corresponding HLT-proxy. The RunManager is located on top of the HLT internal control structure. It steers a set of Master-TaskManagers, which are in charge of all Servant- and Slave-TaskManagers running on the (processing) nodes.

The Slave-TaskManagers are at the bottom of this hierarchical control structure and are responsible for the tasks on the corresponding nodes. They start the actual analysis and data exchange tasks on the cluster nodes and keep track of their execution. The deployment of RunManager and TaskManagers can be seen in figure 4.11.

The setup of these tasks (initialisation and configuration) and their starting and stopping are relayed as state transition commands from the HLT-proxy, via RunManager and Master-TaskManagers to the Slave-TaskManagers. All problems and errors are handed back in the chain; if a problem cannot be solved intrinsically, the general HLT state is switched to **ERROR** and the ECS gets a notification.

The different states of the HLT-proxy correspond to states of the RunManager and TaskManagers. In the following the meaning of these states are described in more detail. The **bold** state names refer to the name in the HLT-proxy, their corresponding names on the RunManager / TaskManagers side are indicated in *italic*. In **OFF** (*"off" / slaves_ dead*) either all components are off or only the RunManager is running. The command INITIALIZE switches the HLT to the state **INITIALIZING** (*starting_ slaves / killing_ processes*). This leads to the start-up of the RunManager, if it is not already running. The same applies for the RESET command, in case the HLT is in **ERROR** state (*error*), remaining processes are killed and the cluster is cleaned up for the next start. **INITIALIZED** (*processes_ dead*) is the next stable state the HLT reaches. Here the RunManager is ready to prepare the configurations for the Master-TaskManagers. The nodes are clean for the upcoming run.

With the parameters from the CONFIGURE command the HLT changes to the **CONFIGURING** state (*starting*) and the RunManager receives the basic settings for the next upcoming run(s)[11]. Now the Master-TaskManagers are started and the compiled configurations are passed to them. Furthermore, all Servant- and Slave-TaskManagers on the included processing and interface nodes are started as well. This is the most time consuming step in bringing up the HLT. In the following stable state, **CONFIGURED** (*ready_ local*), the complete TaskManager hierarchy is up and connected but no analysis components are started yet.

When the parameters of the ENGAGE command are received during the **ENGAGING** state (*connecting*), the Slave-TaskManagers start the (analysis) components with the relevant parameters, like run type and run number. Afterwards, the components try to connect to the other components according to the configuration of the analysis chain. The connection is performed first locally and then on the network. Once this is finished, HLT reaches the **READY** state (*ready / starting_ run*). Now every component is ready, configured and connected; only the data sources are not active. After the START command is received, the HLT switches to **STARTING**

[11] It can be either a single run or a series of runs with the same basic configurations.

(*starting_run*) and the transition command is percolated through the TaskManager hierarchy. Once all components signal that they are running, global HLT enters the **RUNNING** state (*running*). The TaskManager states *busy* and *paused* are mapped as well to the **RUNNING** state in the HLT-proxy; they mainly refer to different internal states the TaskManagers can take during the data taking and analysis. At the event data stream the start is signalled by inserting a special event, the Start-of-Data (SoD) event. While in **RUNNING** state the HLT analyses events, chooses Regions-of-Interest (RoI) within an event, writes HLT-ESDs, compresses event data and provides trigger decisions. These results are transferred to the DAQ via dedicated DDLs.

When the run is stopped, ECS sends the STOP signal and HLT enters the **COMPLETING** state (*stopping_run*). At the data stream an End-of-Data (EoD) event is received. During **COMPLETING** the last events are percolated through the analysis chain to the data sinks and possible calibration results or other data for offline analysis are collected in the Shuttle-Portal (see 4.4.1 for detailed description of this interface). Afterwards, the HLT changes back to the **READY** state. This transition signals ECS that the HLT has finished all tasks of the current run. The HLT can now be started for the next run. In order to do so, the HLT has to be brought back to the **CONFIGURED** state by a DISENGAGE command. Thereby the HLT undergoes the intermediate state **DISENGAGING** (*disconnecting*). This includes killing of all analysis components in the chain, the TaskManagers remain active. The transition back to **CONFIGURED** is necessary because the run specific settings, like the run number, are only transmitted together with the ENGAGE command. To reset the more general settings valid for a series of runs, the HLT has to be brought back to **INITIALIZED**, undergoing the **DECONFIGURING** state (*stopping*). This state change is achieved by a RESET command. This command stops and exits all the TaskManagers. The configuration settings are cleaned up and archived together with the log files. To bring the HLT back to the **OFF** state a SHUTDOWN command can be issued. The HLT intermediately goes to the **DEINITIALIZING** state (*slave_kill*) before switching to **OFF** [42] [43].

In case that a fatal failure has occurred while being in any of these states, except the **OFF** state, the HLT transits to **ERROR** state (*error*). From there it can be brought back either to the **OFF** or to the **INITIALIZED** state [70].

For local testing and running of an analysis chain the HLT can be steered as well by a local HLT logic engine and a corresponding ECS GUI. Alternatively it can be controlled directly from the TaskManager GUI of the HLT framework. The TaskManager GUI connects to the RunManager directly.

4.3.3 Redundant ECS portals

For the ECS portal a backup node is prepared in the HLT cluster, having a HLT-proxy and corresponding RunManager in standby. The original setup is installed on portal-ecs0, the backup system on portal-ecs1. While the real HLT-proxy is attached to the SMI object "***HLT***", the backup system uses "***HLT_BCK***". Corresponding objects are created for both in the logic engine for the HLT. The deployment of this setup is visualised in diagram 4.11.

The backup system is flagged as being passive by default, the backup RunManager is observing the states in the HLT cluster but is not interfering with its control. The HLT-proxy accepts two additional commands to set the corresponding RunManager in active or passive state. These commands are referred to the RunManager and the current state of the HLT gets confirmed. For setting an ECS portal setup active the command **SET_ACTIVE** has been introduced, **SET_PASSIVE** is used for switching to passive mode[12]. Both commands are accepted in every state - stable states and transition states -, but they do not lead to a state change.

In case the setup on portal-ecs0 crashes, ECS sends a **SET_ACTIVE** to the HLT-proxy attached to "***HLT_BCK***" on portal-ecs1. The corresponding RunManager on that node takes over the control of the HLT cluster seamlessly and executes all further state transition commands. This can be done all the time without interfering with the current run.

[12] Again the HLT-proxy has to translate these commands to the syntax understood by the RunManager, where **set_active** and **set_passive** is used.

4.4 Offline

The HLT has two different interfaces to Offline[13], which are used for data exchange. Each one takes care of the different directions of transferring data.

In the interactions between the HLT and Offline the Shuttle-Portal is designed to offer data from HLT to Offline using the Offline Shuttle mechanism. The main purpose of this mechanism is to include new objects into the OCDB. But it is also possible to store data to the Offline Reference Database or the Reference File catalogue [33] [71]. While information required for analysis calculations like calibration and condition settings should be put to the OCDB, data sent to Offline for referencing and save keeping, is stored in the reference DB and the reference file catalogue. The reference DB and file catalogue act as additional storage place.

In the opposite direction a special application called Taxi requests entries of the OCDB and stores them locally in the HLT for calibration purposes of the HLT analysis components [33]. Both interfaces are described in more details in the following sections.

4.4.1 The Shuttle-Portal

The Shuttle-Portal consists of a MySQL Database, a File EXchange Server (FXS) and a dedicated subscriber component of the HLT Data transport framework for filling database and FXS. The database, called *hlt_logbook*, is keeping track of all entries made in the FXS starting with the first ALICE run. The FXS stores the actual objects destined for the Offline storage (OCDB or reference storage).

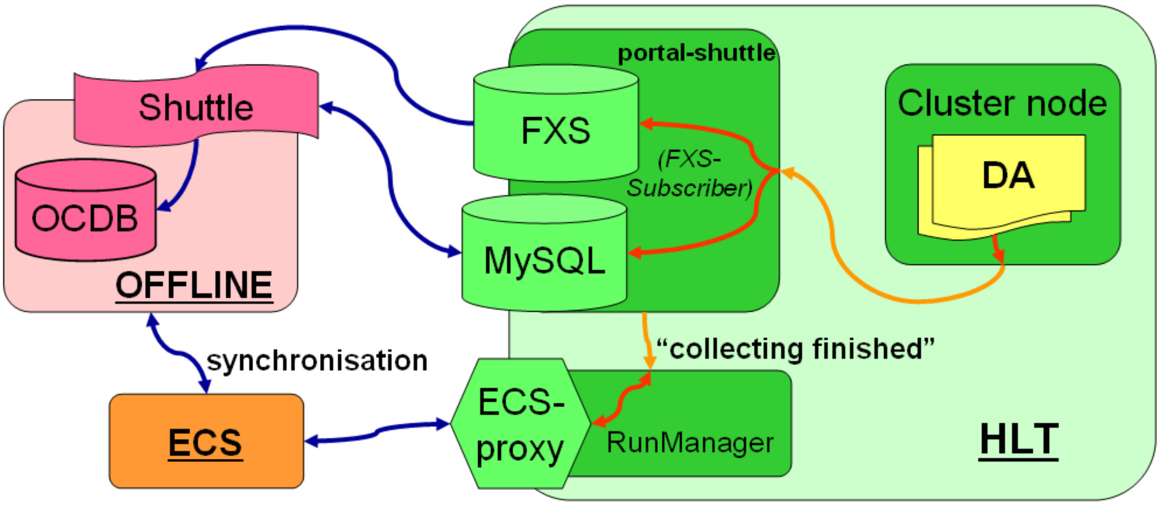


Figure 4.14: Sketch of the interface to the Offline Shuttle. New calibration data is collected by the FXS-Subscriber and offered to the Shuttle using an FXS and a MySQL DB. Synchronisation is achieved via ECS.

[13] The Offline side of this interface and the corresponding Offline components have been developed by Dr. Chiara Zampolli, Dr. Alberto Colla, Jan Fiete Grosse-Oetringhaus and Dr. Latchezar Betev from the ALICE Offline group (CERN).

The Shuttle-Portal is placed on a dedicated portal node (portal-shuttle0), which has an additional network interface to the CERN General Purpose Network (GPN). This allows the Offline Shuttle to contact DB and FXS directly without any tunnelling through the HLT cluster gateways. A direct access for the Offline Shuttle to the worker nodes inside the HLT cluster[14] is not possible. For redundancy reasons a backup setup with the below described components is installed on portal-shuttle1.

FXS-Subscriber

The FXS-Subscriber is a special Subscriber component of the HLT PubSub system. It collects all data objects that shall be fetched by the Offline Shuttle after a run. The component fills them in the above mentioned database and FXS. Although its main workload is at the end of a run, when all analysis components ship their produced calibration data to the Shuttle-Portal, it is also able to receive data during a run. Objects shipped to the Shuttle-Portal during the run are mostly intermediate results, which are sent for temporary storage and backup. Should the chain break or certain analysis components crash before the run ends, these intermediate results are at least saved and can be fetched by the Offline Shuttle.

An analysis component that wants to ship data to the OCDB (or the reference storages in Offline) has to provide additional meta data[15] for naming and classification of the object. The meta data and the actual object for the Offline Shuttle are sent as payload of the data block by a publisher component through the PubSub framework. The payload has its own protocol structure with a header and the appended actual object for the Offline Shuttle. The header contains five elements and has a complete size of 204 Bytes:

- **HeaderVersion Nr**: A 32 bit unsigned integer representing the header version (currently only version number 1 is in usage).
- **Run Number**: A 32 bit unsigned integer containing the run number.
- **Detector**: A character array of 4 Bytes containing the abbreviation of the corresponding detector.
- **FileID**: A character array of 128 Bytes, where the component can store the name of the object.
- **DDL Number**: A character array of 64 Bytes; this has to be seen as a bit field, where the participating DDLs are encoded in.

After the header the object for the Offline Shuttle is appended as BLOB. Its size is calculated from the overall size of the payload of the PubSub data block minus the

[14] The HLT cluster nodes are located in a private network. Only dedicated portal nodes have network interfaces to specified other nets, like the DCS net, the DAQ net or the GPN.

[15] The meta data provides additional information about the actual object, which is transferred.

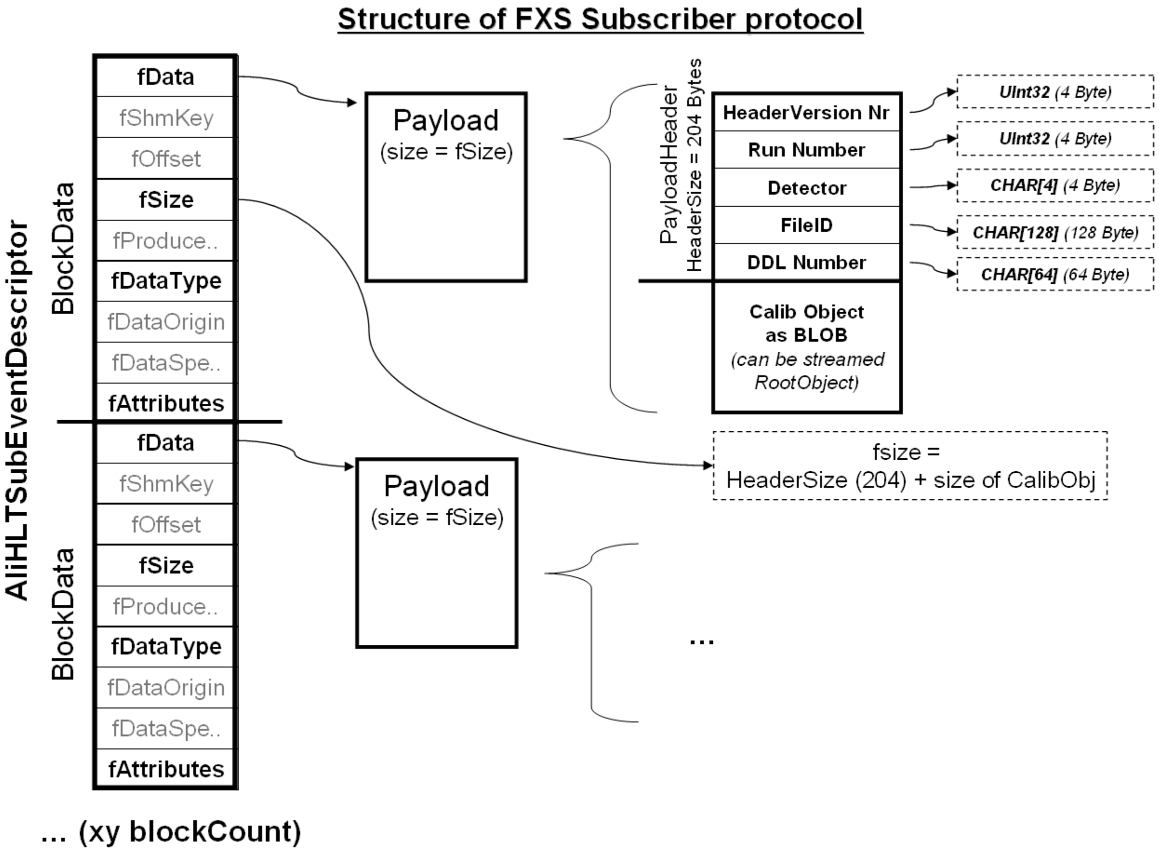


Figure 4.15: Protocol structure of the Shuttle-Portal data. The payload received by the FXS-Subscriber is divided into a header and a body part, where the header contains the meta data of the transferred object while the object itself is located in the body. The block(s) of `BlockData` can be repeated several times (`blockCount`) in the transferred protocol.

Header size of 204 Bytes. An overview displaying this protocol structure is given in figure 4.15.

The information contained in the protocol header, except for the "HeaderVersion Nr", is used for the describing meta data which is included in the *hlt_ logbook*. The object taken from the BLOB is inserted into the FXS. Each block of data is processed separately and then referred to the corresponding modules, the MySQL DB client and the FXS client.

The FXS-Subscriber consists basically of four parts. The subscriber part is used for receiving data sent through the PubSub chain. It inherits its main functionality from the HLT framework (see UML Class Diagram in figure 4.16). A database connector covers the tasks related to the MySQL DB connection. The FXS connector part is responsible for contacting the FXS and storing the objects. A logger module handles log messages issued from all parts.

On start-up the subscriber part initialises the connectors for the database and the FXS. This includes setting of the FXS base path and the contact details for the

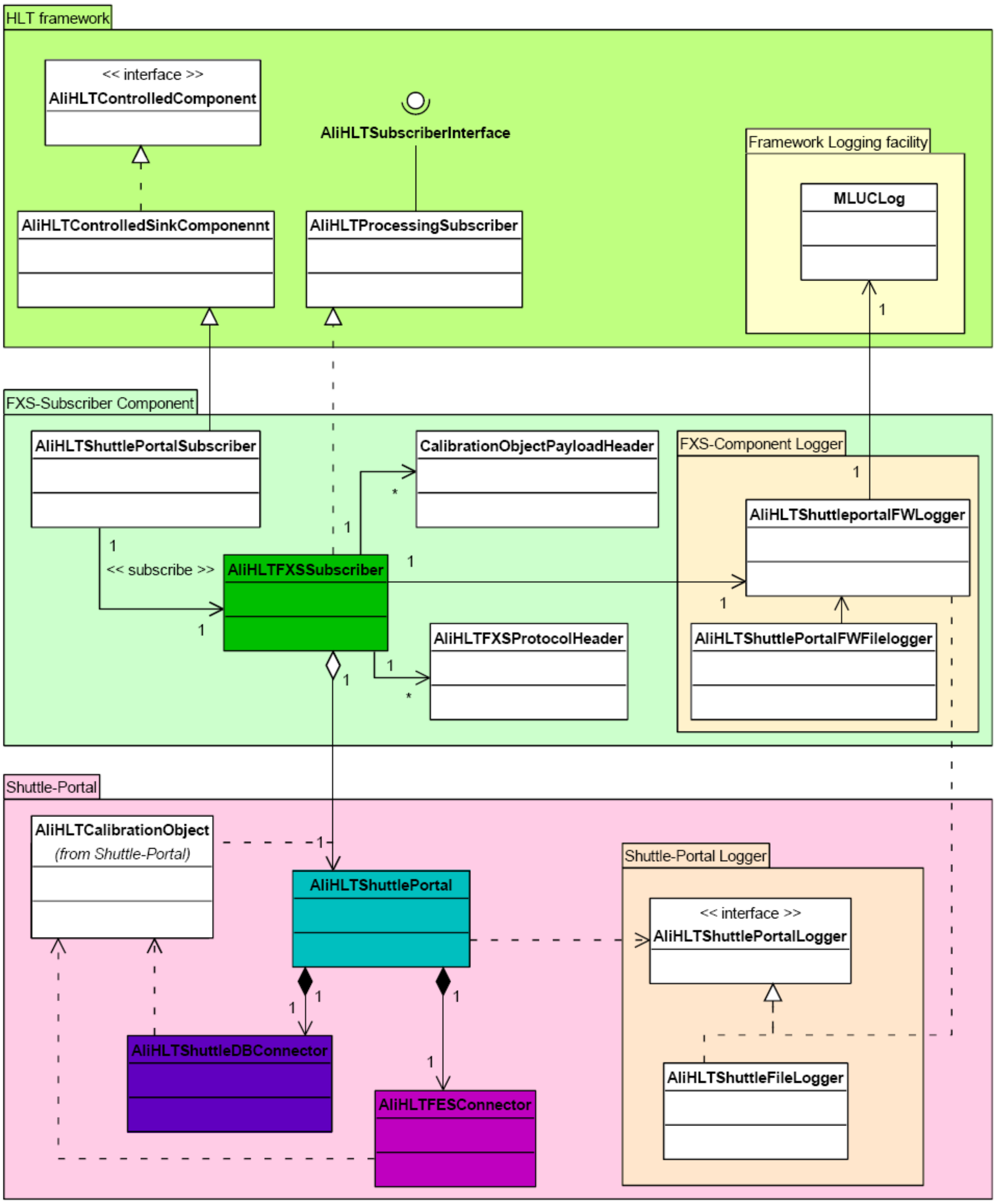


Figure 4.16: UML Class Diagram of the FXS-Subscriber. The Shuttle-Portal component inherits its functionality from the HLT data transport framework package (see page 25); the FXS-Subscriber receives data as child class from `AliHLTProcessingSubscriber`. The actual storing of received calibration data is performed by dedicated connector classes (DB and FXS - due to historical reasons "FES" instead for "FXS" is used in the name for the corresponding connector). For testing purposes the Shuttle-Portal module has its own logging class.

database[16]. Then it is ready to receive automatically data from the analysis chain through a callback function of the framework. The structure of the received data is as described above (see also figure 4.15).

All objects are filtered for the correct data type, before the connectors take over the data. The field "*fDataType*" in the transferred protocol structure contains a 64 bit unsigned integer encoding the data type. Data originated for the Shuttle-Portal have to match the bit field pattern given by the characters of `'FXS_CAL '` [17] (whitespace at the end). After filtering the payload, data is enveloped in an AliHLTCalibrationObject and handed to the database- and FXS-connector parts.

The two connectors for the Shuttle-Portal are collected in a dedicated library: *libShuttlePortal.so*. The database connector uses a third party library, mysql++, to contact the MySQL DB. Mysql++ clients wrap the MySQL native client calls, which are written in C-style, into C++ like structures [72].

In the procedure of storing an object to the Shuttle-Portal, the database connector checks first whether the *hlt_ logbook* already contains an entry for this object. This is done by testing for the unique key of the table, which is represented by the field ***filePath***. If it does not exist in the table, a new entry is made; else its version number is increased (more about the table is described in the section "HLT logbook").

Then the FXS connector stores the actual object in the file catalogue of the FXS, extracts the file size and calculates its checksum. Both values are afterwards inserted into the corresponding fields of the *hlt_ logbook* by the database connector. The sequence of storing calibration objects on the Shuttle-Portal is visualised in figure 4.17.

In addition, a logger module takes care of inserting log messages from the different stages of the storing procedure. This also includes messages from the ShuttlePortal library. All messages are inserted into the logging system of the HLT framework, which is called MLUCLog[18]. Therefore the log messages from the FXS-subscriber, like for all components of the HLT, can be observed centrally by the operator.

HLT logbook

The HLT logbook consists of a database, served by a MySQL Community server, currently running on version 5.0.21. A MySQL Community server has been chosen because it is a highly reliable, well maintained open source database, which has a high reputation in scientific research [73]. The database itself is called *hlt_ logbook*. It contains a single table: *calib_ data*. The table has columns for the run number, the detector name, the file name to which the entry refers, the encoded participating DDL numbers, the relative file path in the FXS, the timestamp for the creation of this entry, the timestamp when the corresponding object has been processed by the

[16] These contact details for the database comprise the host name of the MySQL server, database name and user name, as well as the corresponding password of the user. For the FXS the set contact detail is the path to the base directory of the file catalogue.

[17] The bit field is used to identify the transferred data type. The representation as `'FXS_CAL '` has been chosen to use it in a human readable manner.

[18] *MLUC* stands for *M*ore or *L*ess *U*seful *C*lasses, which is a proprietary utility package of the HLT framework. The MLUCLog class feeds messages in the infoLogger system used for the HLT operator.

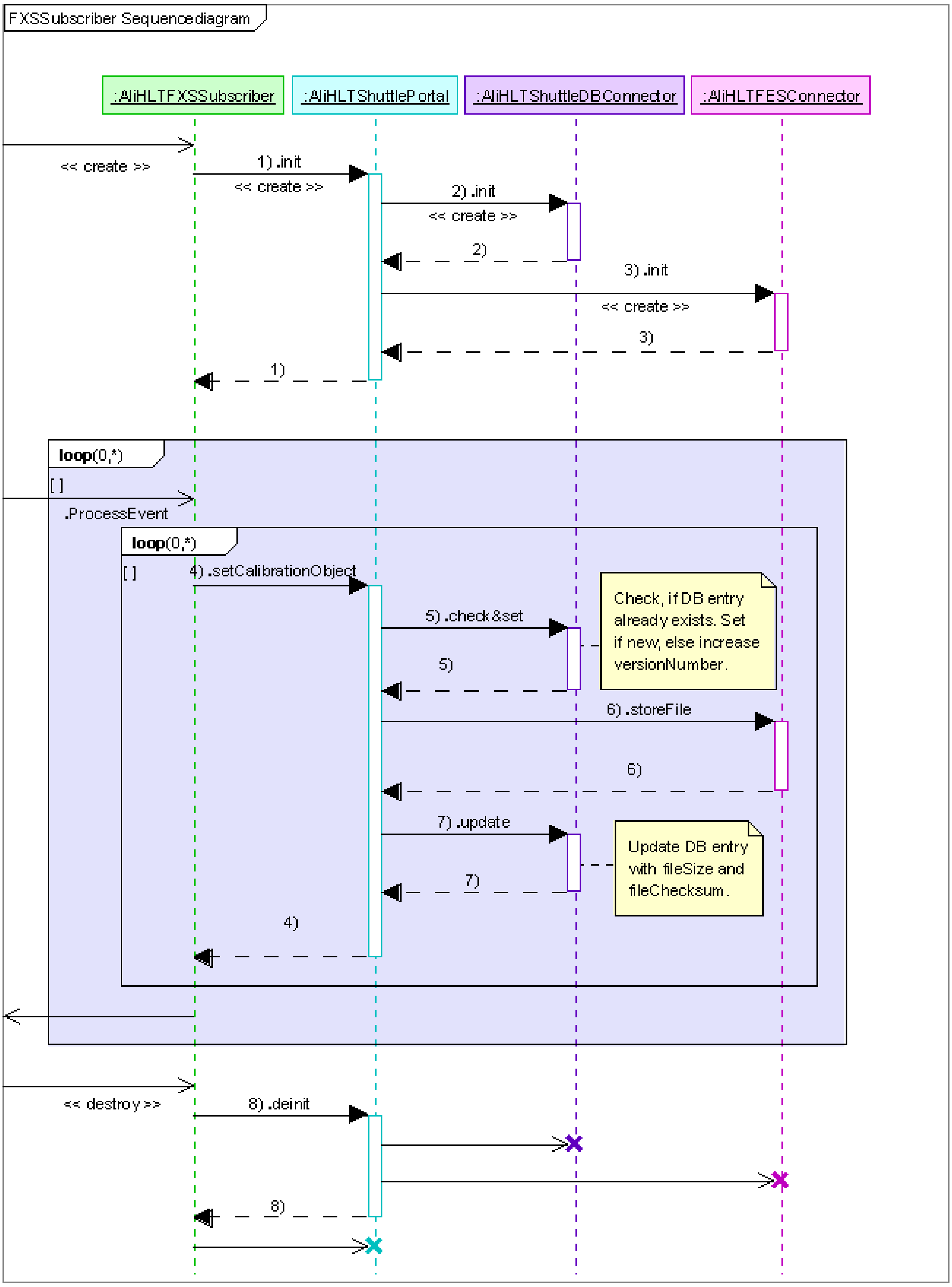


Figure 4.17: UML Sequence Diagram of the FXS-Subscriber. The blue highlighted loop shows the main task of the FXS-Subscriber: received objects and their meta data are checked and stored to *hlt_logbook* and FXS. Due to historical reasons the FXS connector module is called AliHLTFESConnector. First, Offline had used "FES" as abbreviation.

Column name	Description	Provider	Type
run	Run number of the file	HLT (FW)	INT
detector	Detector name	HLT (DA)	CHAR(3)
fileId	File name	HLT (DA)	VARCHAR(128)
DDLnumbers	Participating DDL numbers	HLT (FW)	CHAR(64)
filePath	Full path to the file	HLT (FW)	VARCHAR(256)
time_created	Timestamp of file creation	HLT (FW)	DOUBLE
time_processed	Timestamp of file processing	OfflineShuttle	DOUBLE
time_deleted	Timestamp of file deletion	HLT	DOUBLE
size	Size of file in Bytes	HLT (FW)	INT
fileChecksum	MD5 checksum of the file	HLT (FW)	CHAR(64)
versionNumber	Latest file version number	HLT (FW)	INT

Table 4.1: This table shows the entities of the meta data for ShuttlePortal entries.

Offline Shuttle, the timestamp when the corresponding object has been deleted from the FXS, the file size of the object in the FXS, the MD5 checksum[19] (Message-Digest algorithm 5) of the file and its version number. Table 4.1 displays the structure of the table and the data types of the columns.

The HLT logbook stores the meta data for every object included in the FXS from all runs. The Offline Shuttle uses the meta data to identify and classify the entries in the FXS. They are requested after each run. The same mechanism is also used by the Shuttle when requesting the DAQ and the DCS.

The ***run*** number is given by the ECS at the start of a run. The HLT framework takes this run number and refers it to the FXS-Subscriber for each entry in the HLT logbook, where it is stored as an integer number.

The ***detector*** name is given by the Detector Algorithm (DA), which has produced the object. It is abbreviated by the official three-letter acronym of ALICE detectors used by the online and offline systems [75].

The file name has to be given as well by the DAs. It is stored under ***fileId***, which consists of a variable character array with a maximum size of 128 Bytes. In principle the name should describe its content. It must be unique for a given "run/detector/DDLnumber".

The participating DDL numbers are retrieved from the framework. The multiple numbers are encoded in a 64 Byte large bit field. The TPC as largest detector in ALICE is taken here for an example of the encoding. A number of 216 DDLs connects the TPC with the HLT, therefore a bit field with 216 positions would be required. To encode this in hex numbers, each number can encode 4 bit → 216 / 4 = 54. A bit field of 54 Bytes is needed, the usage of a character array of 64 allows for some buffer. Hex numbers have been chosen in the encoding to make the bit field representing the ***DDLnumbers*** human readable. They are saved as character representation in the corresponding fixed size char array [69].

[19] The MD5 checksum allows for an integrity check by using a 128-bit hash function [74].

The full path of the file, relative to the FXS base folder, is stored in a variable character array of 256 Bytes. It is stored under ***filePath***. The scheme of the file path looks like the following: "run/detector/DDLnumbers/fileId". This represents the unique key of the *calib_ data* table in the *hlt_ logbook*.

There are three different timestamps stored in the HLT logbook. The first one, ***time_ created***, gives the time, when the file has been inserted in the FXS. It is set by the FXS-Subscriber component. The second one, ***time_ processed***, is written by the Offline Shuttle, when it fetches and processes the file after a run. The last timestamp, ***time_ deleted***, gives the time, when the file is deleted from the FXS by HLT. All three timestamps are stored in a DOUBLE. A DOUBLE has been chosen, because it is guaranteed to be 8 Bytes long. This allows for using the double precise timestamp given by a timespec struct. The timespec struct has two members: a `time_t`[20] is used to store the seconds passed since Jan 1, 1970 00:00:00 UTC and a long integer for storing nanoseconds. In the current implementation a single precision timestamp is inserted. This means only the part for the seconds in the timespec struct is taken. If a file has not been processed or deleted yet, the corresponding entries in the table are set to NULL.

The file ***size*** of the FXS entry is stored as an integer number in ***size*** by the connector.

A MD5 checksum of the file is inserted by the framework, too. A character array of 64 Bytes is used for it [74]. The field for the checksum in the *hlt_ logbook* is called ***fileChecksum***. Both, file size and MD5 checksum, can be used by the Offline Shuttle for verification of correct file transportation when fetching FXS entries.

The last column in the table stores the latest version number of the file in the FXS as an integer. The version number is used to keep track of different versions of the same type of files in the FXS. It is for HLT internal usage only. If a component sends an object with an already existing unique ID (see ***filePath***), then the database connector in the FXS-Subscriber increases the corresponding version number entry in the *hlt_ logbook* and the subscriber uses this incremented number for uniquely storing the file in the FXS. Therefore the Offline Shuttle always gets only the last version of an entry, although all versions are stored for bookkeeping. This is described in more detail in the following section. The entries of one type and one run in the FXS should be cumulative, so only the last version is needed by Offline. This mechanism has been introduced to be able to ship at least the latest intermediate version of an FXS entry to Offline, in case a component should crash during the run.

File Exchange Server (FXS)

The FXS is the last part of the Shuttle-Portal interface on the HLT side. It mainly consists of a file catalogue that is owned by a dedicated HLT user. It is used as a temporary storage for the calibration objects produced inside the HLT cluster.

Each time the FXS-Subscriber receives an object and its corresponding meta data, it checks first if there is already an entry existing with the same unique key

[20] The default implementation is a 32 bit integer.

("run/detector/DDLnumber/fileId"). If not, the version number is set to one and the object is stored in the FXS. The *run number*[21], *detector*[22] and *DDLnumber*[23] are used as names for the sub directories, where the object is stored. An underscore and the version number are appended to the *fileId* and then the latter one is used as file name for saving the object in the FXS. The original *fileId* name is used for a symbolic link pointing to the latest entry for this type. In the case that it is the first entry of this type for a given run number, detector and DDLnumber, the file is named "*fileId*_1".

If the *hlt_logbook* already contains an entry with that unique key, the version number of the corresponding entry is incremented and the object is stored with the new version number appended, e.g. "*fileId*_2". The symbolic link pointing to previous latest entry is deleted and a new link is created pointing to the actual latest version of this object ("*fileId*_2"). This allows for bookkeeping of the different versions, although the Offline Shuttle has only a well defined access to the latest version without knowing about version numbers. The latter ones are not foreseen in the Offline Shuttle mechanism.

After the Shuttle has fetched and processed the entries produced during a run, they can be deleted from the file catalogue. This is indicated by the field ***time_processed*** in the *hlt_logbook*, which is then not filled any longer with "NULL". When the HLT is performing a clean-up of old entries, the field ***time_deleted*** should be set accordingly.

Offline Shuttle

The Offline Shuttle has been introduced by the ALICE Offline project. Its purpose is to fetch calibration and reference data from the HLT, DAQ and DCS after each run and fill them in the OCDB or the Offline Reference Storage[24]. Therefore the Offline Shuttle receives a notification from the ECS, when all of the online systems have finished their processing after a run.

First the Shuttle contacts the databases of the corresponding systems: on HLT side this is the *hlt_logbook*; on DAQ side it is the *daq_logbook*; and on DCS side two different databases are requested, the database with the meta data for the DCS FXS and the DCS Archive DB, where the monitored DCS values are archived. DCS values can be sensor measurements like temperatures. After requesting the meta data of the FXS entries the new files are transferred to Offline non-interactively using an ssh-key[25]. The ssh-key is stored on the HLT, DAQ and DCS side respectively. On the Shuttle side preprocessors for each detector[26] process the data from the databases and

[21] The number of the run when this object has been created.

[22] The detector name of the DA, which has created that object.

[23] The encoded number of the participating DDLs, which have been used for creating that object. This is mainly used to distinguish between similar objects from one run.

[24] Entries in the Offline Reference Storage are put on the GRID as well. The generic path for these entries is: ***<baseGridReferenceFolder>/<DET>/<runNumber>_ <gridFileName>*** ; the base path for the reference storage is: ***/alice/data/<year>/<LHCPeriod>/Reference*** [71].

[25] Ssh-keys have been chosen to automatise the requests and not need manual authentication for the copy processes.

[26] There are 20 different preprocessors; one for each of the 18 detectors of ALICE, plus a preprocessor

the FXSs before they are included in the OCDB or the Offline Reference Storage[27]. The purpose of the preprocessors is to prepare and combine the fetched data with information from other systems or merge them with former calibration objects. Since the Shuttle can fetch any format of data the preprocessors have to "rootify" the data, if this has not already been done before. The Shuttle supervises the execution of the different preprocessors. The Shuttle itself can be monitored using MonALISA [71].

"Rootification" of or "to rootify" an object describes the process of enveloping data in ROOT-objects, which then can be stored as ROOT-files [76]. ROOT-objects and ROOT-files are widely used in the analysis procedure of ALICE. The OCDB is designed to store only objects enveloped in ROOT-files [33].

As an example the HLT preprocessor is presented here. At the current state the HLT preprocessor has two different tasks: preparation of created Huffman tables and temperature histograms.

Huffman tables for the encoding of data for various detectors are calculated during calibration runs inside the HLT. After each calibration run these tables are stored in the OCDB for later usage, e.g. encoding TPC data during physics runs. The Offline Shuttle requests these tables for the HLT Preprocessor and hands them over. The preprocessor puts the tables of one detector into the dedicated ROOT-collection and stores them to the OCDB [77].

In addition, temperature histograms produced in the HLT are taken by the HLT Preprocessor and archived in the Offline Reference File Catalogue. For the Reference File Catalogue no rootification of the files is required [71].

4.4.2 The Taxi portal

In Offline all important calibration settings are stored in the OCDB. The OCDB is a database that is located in the ALICE GRID[28]. It can be accessed using AliEn, the GRID middleware for ALICE. For offline analysis the required settings in the OCDB have to be fetched via AliEn requests. Depending on the GRID availability there can be some delays in the request.

For online analysis, like it is performed inside the HLT, these requests would significantly reduce the performance of the system. Even worse would be a temporarily inaccessibility of the OCDB during a run. The HLT would be blocked from processing events. Therefore the HLT holds a local cache of necessary OCDB entries inside its cluster.

This local cache exists in two separate versions:

- **T-HCDB** (Taxi-HCDB, where HCDB stands for HLT Conditions DataBase): After fetching new objects from the OCDB, they are stored here first. The T-

for the HLT and one for GRP (General Run Parameters).

[27] The base class for the preprocesssors with the functionality for accessing and storing data, as well as some helper functions, are provided by the Shuttle framework in AliRoot.

[28] The logical path where the OCDB is stored in the GRID uses this generic scheme: ***/alice/data/<year>/<LHCPeriod>/OCDB/*** . Replicas of the OCDB exists on the various GRID sites [71].

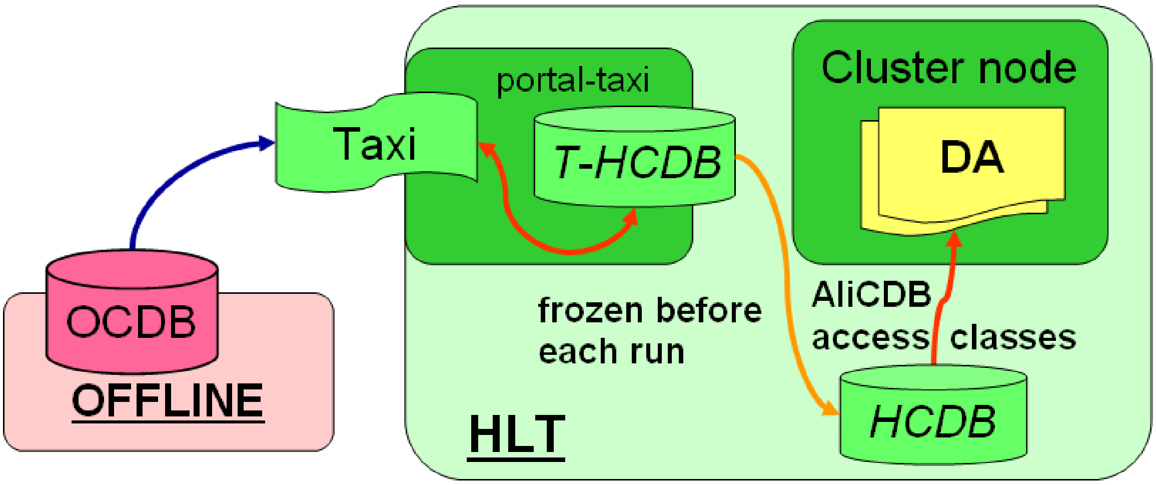


Figure 4.18: The HLT Taxi fetches OCDB entries and stores them locally in the T-HCDB. A frozen version per run of these entries is offered to the DAs via the HCDB.

HCDB can be updated all the time without interfering with a current run (see also "T-HCDB" on page 72).

- **HCDB** (HLT Conditions DataBase): This version is requested by the online analysis components inside the HLT cluster during a run. It is a fixed version of the T-HCDB, and it is not updated with new OCDB entries during the run[29]. In addition the HCDB includes DCS values retrieved during a run and configuration entries made with the HCDBManager script (see "HCDB" on page 87).

T-HCDB

The T-HCDB has been introduced to store fetched OCDB calibration and alignment objects. Its structure is similar to the one of the OCDB. It contains a defined subset of the OCDB entries. In the largest case this subset can cover all OCDB entries, depending on the lists prepared by detector experts for the Taxi. No other objects from other systems than Offline are included in the T-HCDB[30]. This guarantees that the T-HCDB entries have only the official names and version numbers. It is required for later comparison of online and offline analysis. Like the OCDB, it mainly consists of a file catalogue organised in directories with a three layers of hierarchy [31]. Figure 4.19 illustrates the structure of the file catalogue representing the T-HCDB / HCDB / OCDB.

Objects stored in the OCDB are described by a unique name, which consists of three parts representing these three layers. Each of these parts is represented by a

[29] The version is fixed per run to have a coherent environment during the run.

[30] An exception here are certain GRP entries, which are run-specifically given by the ECS (see section 4.5.1).

[31] On the HLT cluster the environmental variable **ALIHLT_T_HCDBDIR** defines the base directory of the T-HCDB. Normally it should be set to ***/opt/T-HCDB*** .

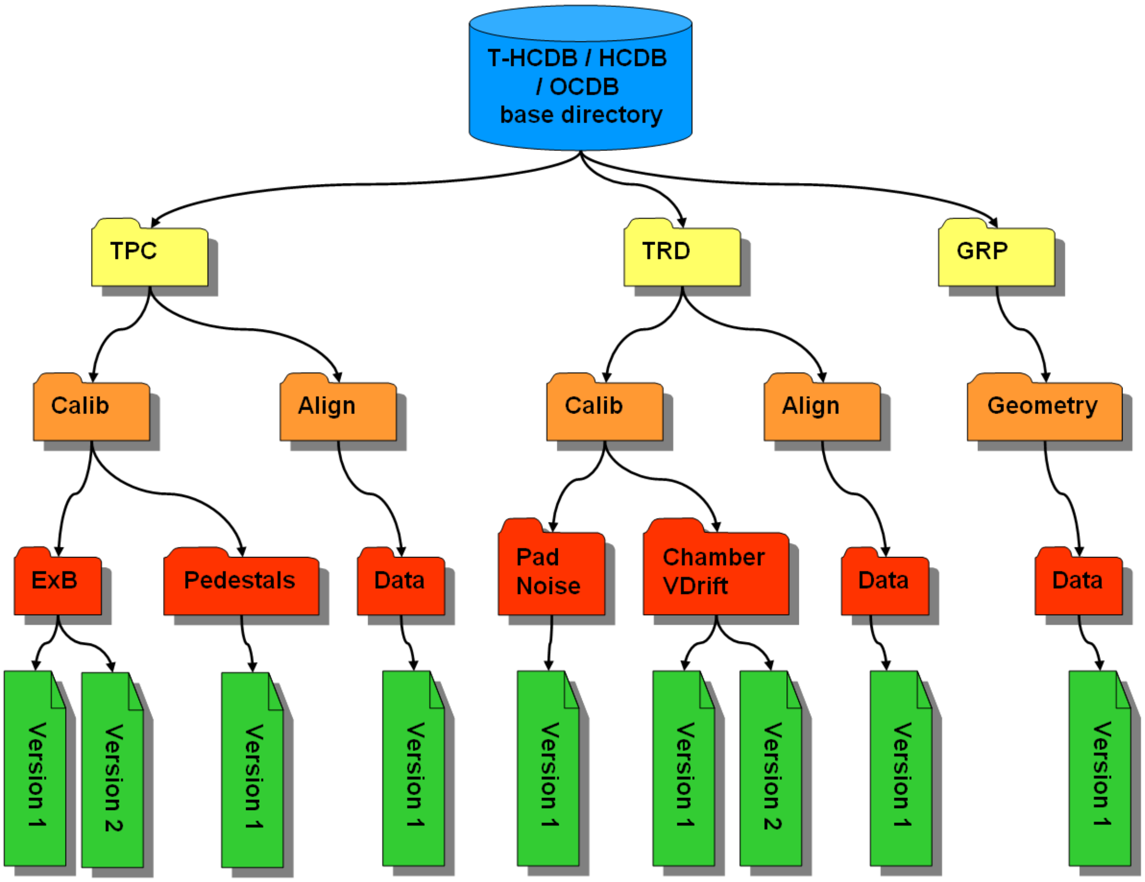


Figure 4.19: This figure illustrates the file catalogue structure of the T-HCDB / HCDB / OCDB. On top is the base directory of the file catalogue located (blue). The first layer is given by the directories representing the various detectors (yellow). Each of them contains directories accumulating the general purposes of the entries - second layer (orange). The third layer is given by directories with the actual name of the entries (red). The actual files (green), representing the entries in the different available versions, are stored in these directories. A general entry path is given by: "***DetectorAcronym/ObjectPurpose/ObjectName***" - each part is represented by a directory. The mentioned directories in this figure display only a subset of the real T-HCDB / HCDB / OCDB to visualise their structure.

directory.

The first part defines the corresponding detector. The official three-letter acronyms of ALICE detectors are used for representing the detector name [75]. Global objects are stored under "GRP". The HLT acts as a detector in this perspective and has its own detector directory. The second part of the name describes roughly the intention of the object such as *calibration*, *alignment* or *geometry*. The last part gives the actual name of the object, like *Pedestals*, *ExB* or *ChamberVdrift*. This leads to the following general structure for OCDB objects: "***DetectorAcronym/ObjectPurpose/ObjectName***".

The actual objects are stored in ROOT-files inside the directories on the lowest level of this hierarchy (directories consisting of the object name). Different versions of an object are stored in separated files in the same directory (see Run validity and versioning of calibration objects on page 74) [33].

On the GRID side a database keeps track of all the entries, using the identification of the objects (pathname, version validity) and their corresponding meta data. Both schemes are shown by table 4.2 and table 4.3. The actual objects are entries in an AliEn file catalogue. All Entries in this file catalogue are read-only, new entries with already existing names result in an increment of the corresponding version number. The version numbers are encoded in the actual file name, thereby avoiding clashes in the storage.

Specifier	Data type	Description
first_run	INT	First valid run for this CDB entry
last_run	INT	Last valid run for this CDB entry
version	INT	Version number of this CDB entry
path_level_0	VARCHAR(255)	Corresponds to the detector acronym
path_level_1	VARCHAR(255)	Corresponds to object purpose
path_level_2	VARCHAR(255)	Corresponds to object name

Table 4.2: CDB table scheme, that is used for keeping track of all files in the AliEn file catalogue corresponding to the OCDB entries [33].

Specifier	Data type	Description
object_classname	VARCHAR(255)	Class name of the corresponding object
responsible	VARCHAR(255)	Name of the responsible person/creator
beam_period	INT	Beam period of creation of the object
aliroot_version	VARCHAR(255)	Used AliRoot version for the creation
comment	VARCHAR(255)	Comment given by the creator

Table 4.3: CDB table scheme that is used for storing the meta data of OCDB entries [33].

Run validity and versioning of calibration objects

All entries in the OCDB (and so in the T-HCDB / HCDB as well) are run dependent. Therefore each object has a run validity and a set of version numbers. Both are encoded in the actual file name using a special versioning scheme.

The general structure on a file name in this storage looks like this:

$$\text{Run}\boldsymbol{X}\_\boldsymbol{Y}\_\text{v}\boldsymbol{A}\_\text{s}\boldsymbol{B}\text{.root}$$

The run validity is encoded in $\boldsymbol{X}$ and $\boldsymbol{Y}$, with $\boldsymbol{X}$ defining the first run this object is valid for and $\boldsymbol{Y}$ defining the last one. An infinite validity is indicated by $\boldsymbol{Y}$ set to 999999999[32].

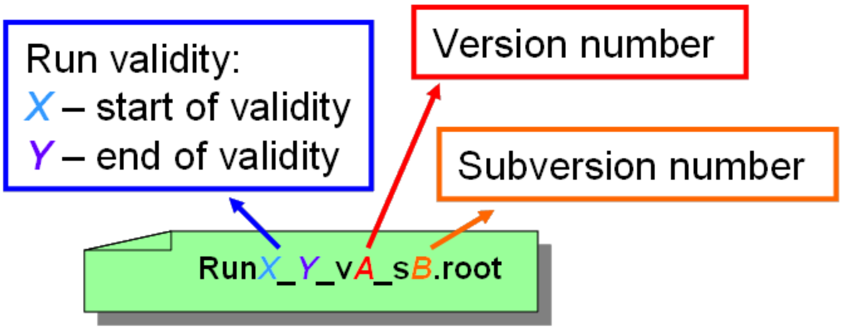


Figure 4.20: The file name scheme of entries in the CDB (OCDB, T-HCDB, HCDB, local dump) encodes run validity, version number and subversion number. The subversion number is only used when the file is stored outside the OCDB (GRID).

The versioning scheme for CDB entries uses a set of two different numbers: a version number and a subversion number. The version number is given by 'v$\boldsymbol{A}$', the subversion number by 's$\boldsymbol{B}$'. While objects in the OCDB (GRID) are stored only with their version number, the subversion number is only appended when objects are stored locally or, as in the HLT case, in the T-HCDB (respectively HCDB).

Normally an entry is first created locally getting tagged with version number and subversion number 0 together with its run validity. Contingently it gets updated locally before it it is transmitted to the OCDB. These updates result in an increment of the subversion number. When the object is transferred to the OCDB the first time, it gets registered in the corresponding database and tagged with version number 1, while removing all former subversion numbers. Further updates of this object in the OCDB result in an increase of the version number, ignoring and removing any subversion numbers. This means the version number is increased only each time an updated version is transmitted to the OCDB. If an object is fetched from the OCDB and stored locally, it is tagged again with a subversion number, reset to 0. Every local update on this object afterwards leads to an incrementation of the subversion number only. Figure 4.21 visualises this versioning scheme. The attempt to store an object with a version number that has already been cached and updated locally (same version number but subversion number higher than 0) will return an error.

[32] In the AliCDB access classes the value defining *infinity* can be retrieved by the static member function `AliCDBRunRange::Infinity()`.

Inclusion of an object with already existing name and an overlap in their run validity automatically results in an increase of their version number, respectively subversion number in case of a local update.

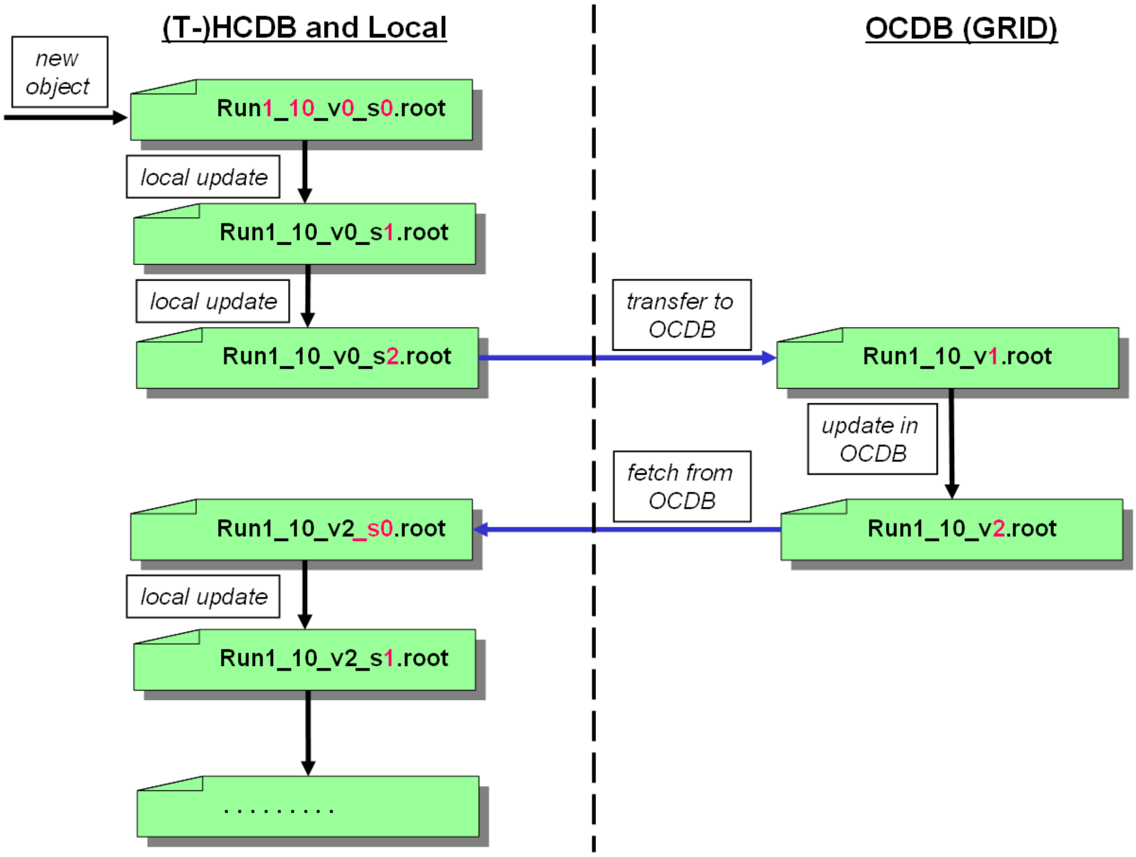


Figure 4.21: Versioning scheme of CDB entries: an update locally of an existing entry increases the subversion number, updates in the OCDB (GRID) increments the version number [33].

The Taxi

In the HLT a dedicated interface application called ***Taxi*** is responsible for regular updates of the local cache with new OCDB entries. The Taxi uses the T-HCDB for this cache. It runs independently and asynchronously to any run. It has regular time intervals for opening a connection to the OCDB and checking for new or updated entries. For performance reasons the requests are decoupled from the run start-up configuration of the HLT. This means the HLT does not have to wait for a Taxi request to finish in order to go to **RUNNING** mode.

Each detector in the HLT has its own list(s), where it can specify required OCDB objects. These lists are stored in the T-HCDB and read by the Taxi before each request interval. This enables HLT to keep track of the fetched object names. The detector experts can administrate their lists and add or remove items depending on their needs. For the Taxi these entries are transparent and only the names are referred to the OCDB during the request.

The Taxi has two different ways to request objects from the OCDB. The more efficient procedure in terms of time spent for queries, checks for local availability before the actual retrieval and storage. It uses single requests for the entries in the OCDB. In this method the Taxi requests the OCDB first for the latest available version number for a calibration object in combination with a given run number. Then it checks, if the corresponding object has not already been cached in the T-HCDB. If not, the object is fetched from the OCDB and stored to the T-HCDB. This procedure avoids unnecessary network traffic, since it only fetches objects with latest version numbers, which are not already cached locally.

The more flexible method allows for the usage of the wildcard "*" in the entry name of the request. The asterisk can be included in the name of the list entries to acquire several objects at once. Therefore either the part of the ***ObjectName*** has to be replaced by the asterisk or the ***ObjectPurpose*** with the ***ObjectName*** part left blank. If the asterisk is used for the ***ObjectName***, then all objects inside the subdirectory of the given ***DetectorAcronym/ObjectPurpose*** are fetched. If it replaces the ***ObjectPurpose***, all objects of the corresponding detector are requested. In both cases the Taxi receives a collection containing all desired objects. Before inserting them to the T-HCDB, it checks the version number of each object and stores it only, if it has not already been cached. Due to the fact that the check can only be performed after retrieving all objects connected to this list entry with the wildcard, the second method is more time consuming. In most requests to the OCDB the Taxi will step on to the next request after checking the version numbers, due to already local availability of the latest object.

Both procedures, single requests and the usage of wildcards in the list entries, can be mixed in one Taxi run. This means that the lists with the object names to be fetched can contain the exact name for an object combined with instructions to get all objects from a given OCDB subdirectory. The activities performed by the Taxi are presented in the UML diagram in figure 4.22.

As run number the Taxi uses the last given run number from the ECS and increments this number by one, in order to request objects for the next upcoming run.

The Taxi itself consists of a set of bash scripts, C++ libraries and an AliRoot macro. The scripts initialise the environment for accessing the OCDB (see section "The Taxi GRID access – AliEn") and prepare the lists of calibration objects to be fetched. After these preparations a macro in AliRoot is started to execute the actual tasks of the OCDB requests. For the access to the OCDB and the storage of the calibration objects in the T-HDCB the Taxi uses the AliCDB access classes of the AliRoot package. Each time the Taxi contacts the OCDB, the AliRoot environment is started freshly. Therefore it makes no sense for the Taxi to use the caching ability that is built-in in the AliCDB access framework.

Since the T-HCDB is hosted by the HLT file server and distributed to the dedicated nodes on the HLT cluster via AFS, the Taxi can run on any node that has an ethernet interface to the CERN GPN as well. For redundancy reasons two nodes are equipped

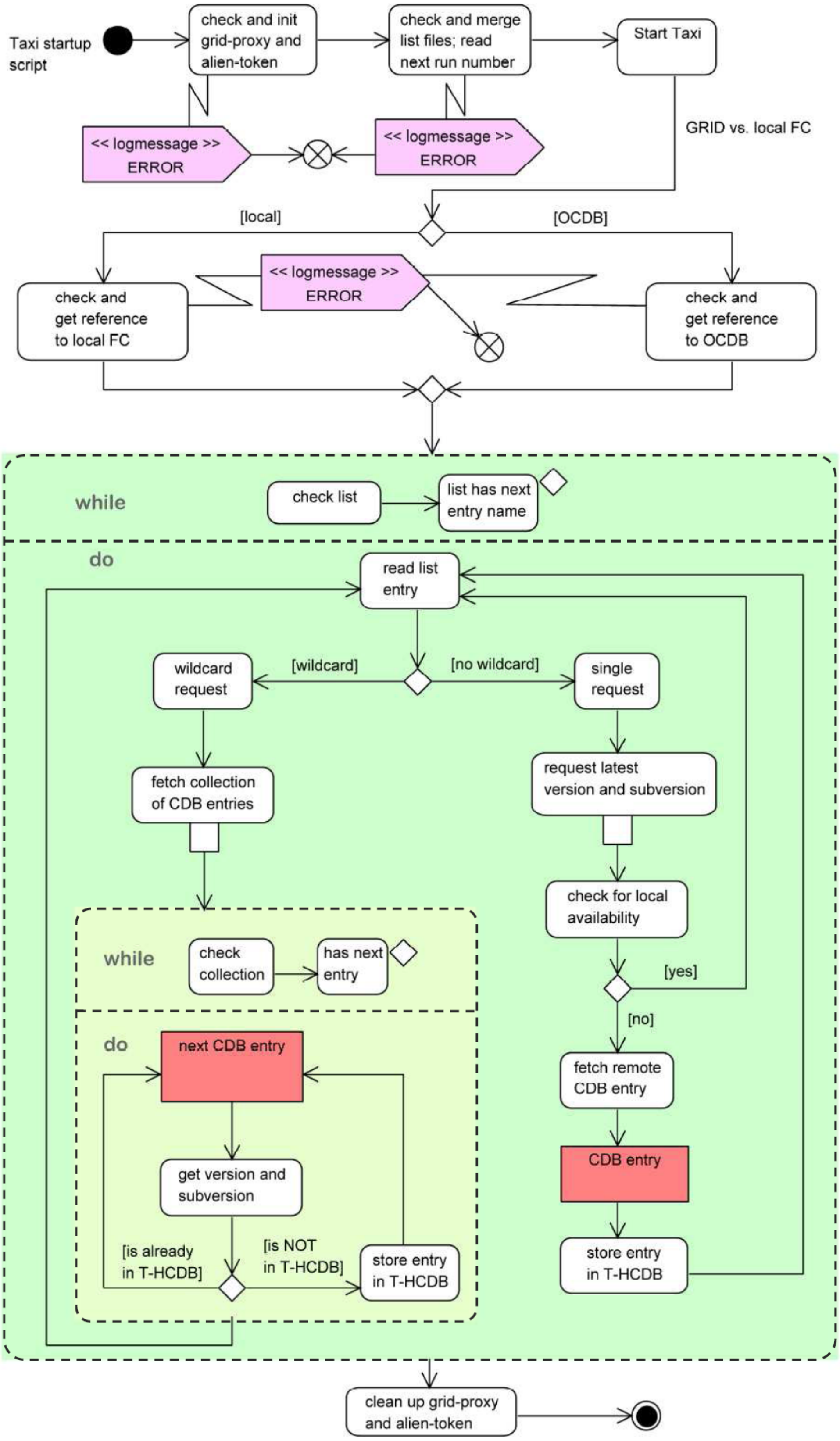


Figure 4.22: UML Activity Diagram of the Taxi: The diagram visualises the activities performed by the Taxi in order to request a CDB FC (OCDB or local).

with this hardware setup: portal-taxi0 and portal-taxi1. Should one of these nodes fail, the Taxi can be started on the other one. The log output of all components used by the Taxi are streamed together into a dedicated log file on the corresponding portal node.

Logging on the Taxi portal

The log output of the Taxi is stored locally to file on the Taxi-portal node[33]. SysMES, the cluster monitoring tool used in the HLT, monitors the files. Each time an error is signalled, the corresponding message is presented in the user front-end of the cluster management system, so that the operator can perform appropriate actions.

The Taxi GRID access – AliEn

To access data from the GRID a user has to authenticate himself / herself with a GRID certificate. To gain access to the OCDB this certificate has to be registered in AliEn and the ALICE VO.

Before accessing the OCDB the Taxi uses the certificate to generate a grid proxy and to acquire an AliEn token. Both are used for the communication with the ALICE GRID. During this process the certificate is verified by a cross check to AliEn. Normally proxy and token are valid for 24 hours, if not destroyed earlier. Their purpose is to ease requests to the GRID, so that the user does not have to authenticate himself / herself each time when accessing the GRID during this period. But in order to perform a proper clean-up after each round of OCDB requests and for safety reasons the Taxi destroys proxy and token after each Taxi run. They are recreated before starting the next request interval.

[33] The Taxi log file can be found under "/tmp/taxi.log" on portal-taxi0 (portal-taxi1) in the HLT cluster.

4.5 DCS

As described in section 2.4.6 the DCS takes care of the control of the various detectors and installed equipments in ALICE. Thereby DCS reads out a lot of different device measurements like temperatures, voltages, magnetic field values, etc.. They are collected by the PVSS and stored in the DCS Archive DB[34]. Both parts are interfaced by the HLT:

- To retrieve DCS values, the HLT requests data from the DCS Archive DB using an application called Pendolino. The retrieved DCS values are parameters required for the analysis performed inside the HLT.
- For feeding data into the DCS, the HLT has implemented the FedServer part of the FED-API (cf. description of the FED-API in section 3.4). DCS PVSS panels can subscribe to the values published by this server.

The two interfaces are described in the following sections.

4.5.1 The Pendolino portal

DCS related observables of the detectors and ALICE devices are monitored by corresponding DCS PVSS panels. These values are stored in the DCS Archive DB, where they can be fetched by other systems. All measured values are timestamped with the time they have been recorded by the PVSS panels. To reduce the amount of data stored in the DCS Archive DB, entries are only inserted when the according value has changed more than a given dead band. All entries in the DCS Archive DB are identified by their DP (DataPoint) name. In addition, they own an Alias name which can be used in the retrieval. The shipping of data from the PVSS panels to the DCS Archive DB is governed by the PVSS RDB (Relational DataBase) Manager. It can take some time until the data are included in the DCS Archive DB[35]. A subset of these stored values are requested by the HLT Pendolino to make them available to the online analysis components [59] [78].

Since the detectors and devices are not only monitored during data taking, but also in-between, it can happen that certain values have no entries for the time period of a given run. To avoid missing parameters in the analysis when requesting data for this run, DCS has implemented a script which updates all entries in the DCS Archive DB with the current value and timestamp at the Start-of-Run (SoR) signal[36]. This guarantees especially the HLT to be able to retrieve initial values when requesting the DCS Archive DB online[37]. The Pendolino has to cover the time period of the SoR in

[34] The design and implementation of the DCS Archive DB has been done by Dr. Peter Chochula (CERN), Svetozar Kapusta (CERN) and Vladimir Fekete (CERN) from ALICE DCS.

[35] Currently, transfer times of up to one minute are measured.

[36] The ECS sends this signal to the DCS after the HLT has been brought to **CONFIGURED** state; see state machine diagram on page 56.

[37] "Online" in this perspective means that the analysis can be performed in real time during a run with all required parameters. The requested values are supposed to vary very little under normal

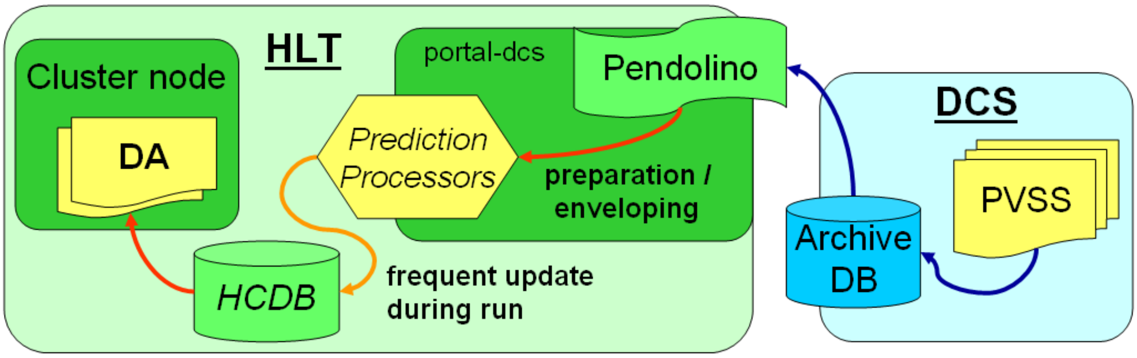


Figure 4.23: Sketch of the HLT Pendolino interface. The Pendolino requests data from the DCS Archive DB in regular time intervals. The fetched values are prepared by detector-specific PredictionProcessors and then provided to the DAs inside the HLT via the HCDB.

its first requests. To provide sufficiently up-to-date values to the analysis components inside the HLT the Pendolino repeats its requests in regular time intervals during a given run. The timing is visualised in the UML Sequence Diagram of figure 4.24. The different steps of the Pendolino are explained in more detail on the following pages.

Layout

The Pendolino is started by the RunManager before each run, when an **ENGAGE** command is received from the ECS. Before the actual requests to the DCS Archive DB are issued, this application takes care of the preparation of the HLT-internal conditions database, the HCDB. First, the initial values for certain GRPs are set, which are given by the ECS together with the **CONFIGURE** and **ENGAGE** commands[38]. These values are stored as T-HCDB entries in the corresponding folder of this FC. In addition the Pendolino informs the Taxi about the current run number[39].

Then the current version of the T-HCDB is frozen, using the release features of AFS with read-only and read-write partitions. This is done in order to have a fixed set of T-HCDB entries, so the general analysis condition settings are not changed during the run. Before the HCDB gets filled now with links to these entries of the frozen T-HCDB version, it is cleaned-up from old entries of the previous run[40]. After the HCDB is prepared, the according FC is released to all computing nodes in the cluster.

As soon as the HCDB is prepared and released, the analysis components and DAs

run conditions, therefore the time shift given by the shipping time from PVSS panels over the DCS Archive DB to the HLT is insignificant.

[38] This includes the parameters of **BEAM_TYPE**, **RUN_TYPE** and **RUN_NUMBER**.

[39] The Taxi uses this run number in the contact to the OCDB for upcoming runs, therefore increasing the given run number by one in the requests.

[40] To allow the HLT components having configuration objects stored permanently in the HCDB, a dedicated folder (*"HLTPermanent"*) has been introduced, which is not cleaned up at the start-up. These objects do not come from the OCDB and are normally included via the HCDBManager script (see section 5.2).

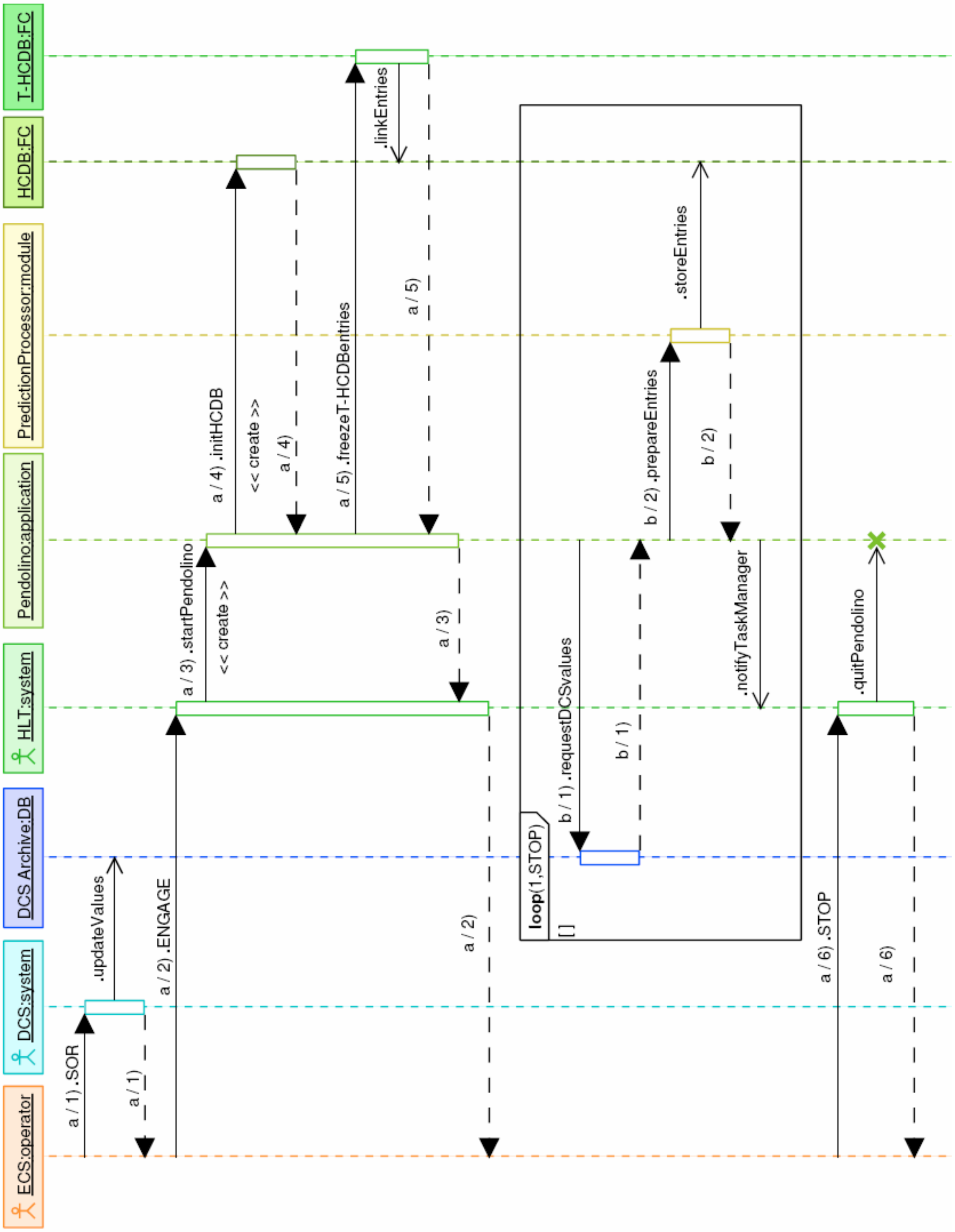


Figure 4.24: UML Sequence Diagram of the Pendolino: At the SoR signal run-initial values are set to the DCS Archive DB. With the **ENGAGE** command sent to the HLT, the HCDB is prepared for the upcoming run and the Pendolino(s) are started. Links to a frozen version of the T-HCDB FC are set to the HCDB. During the run, DCS values are fetched in regular time intervals, prepared by detector-specific PredictionProcessors and stored to the HCDB. Analysis components are notified about updated data.

on the HLT computing nodes can be started and initialised. The Pendolino starter script informs the RunManager about that event via a dedicated signal[41]; afterwards the TMs begin to instantiate their assigned analysis components.

Next, the interface application prepares the lists with Alias names of DCS values to be fetched. The Pendolino owns a dedicated list folder, where detector experts can create, edit and delete list files with DCS Alias names referring to the DCS Archive DB entries. All files inside this folder ending with *".list"* are used for the request, allowing the detector experts having different files for different detectors and purposes. All Alias names provided in these lists are used in the Pendolino requests. They have to be prefixed by their corresponding detector name in the list files in order to assign the retrieved values to their corresponding, detector-specific PredictionProcessors.

The above mentioned preparation tasks are performed by the Pendolino starter script. When they are finished, the script starts the actual Pendolino(s). The preparation time has to be taken into account as well, when calculating the time interval for the first request of the Pendolino(s) (see description above about the SoR update of DCS Archive DB entries).

The Pendolino itself consists mainly of a steering bash script[42], an AliRoot macro and AliRoot libraries. These libraries are: the so called DCS-Client for the DCS Archive DB contact, the PredictionProcessor framework (including the corresponding detector-specific PredictionProcessors for preparation of the fetched values) and general utilities like logging and benchmarking. The bash script serves for starting the AliRoot macro repeatedly with the given frequency of the Pendolino and to calculate the time interval for its requests[43]. The frequency of the Pendolino determines the time interval after which the Pendolino is restarted.

For load balancing and due to the fact that not each DCS value changes in the same manner and is not required with the same frequency, three different Pendolinos have been introduced. Each one is requesting a different subset of DCS values and each is running at a different frequency, meaning the time interval after which the request is repeated.

In the current setup, the fast Pendolino runs every minute. It requests only a limited amount of entries in the DCS Archive DB. The normal Pendolino has a frequency of five minutes and owns the highest load of entries to request. For the slow Pendolino a request interval of every ten minutes has been chosen. It fetches DCS values which changes very rarely. All three Pendolinos are started at the **ENGAGE** command from the ECS. After preparation tasks they begin immediately with the requests to the DCS Archive DB, in order to get the initial values stored at the SoR signal for their subset of DCS Alias names.

The starter script and the different Pendolinos write their log output to separated,

[41] For sending this signal the InterfaceLibrary of the TaskManager framework is used.

[42] The Pendolino starter script comes in addition to this bash script.

[43] Lest not to miss a value update, the start point of a coming request interval is set to the end point of the last request minus the maximum shipping time it can take to transfer the values from the PVSS panels to the DCS Archive DB. In the current setup: `startTime = lastEndTime - 60` (sec). The very first request time interval in a given run uses a different calculation (SoR update).

dedicated log files on the corresponding interface node. Different log files for these modules have been chosen, because their tasks run in different cycles. Their log output should not be mixed in order to achieve an easier association of these messages to the corresponding Pendolino. All four log files are monitored by SysMES, which notifies the operator in case of an error.

At the end of a run, the RunManager terminates the Pendolino by sending a dedicated Unix signal. This signal is caught by the Pendolino starter script and relayed to the different Pendolino(s) allowing them to perform a proper clean-up before termination. The tasks for HCDB preparation and filling performed by the Pendolino are presented in UML Activity Diagram in figure 4.25.

DCS-Client

The Pendolino sorts the Alias names for the request by there corresponding detector names and sends the request for the set of one detector by a time. The actual request is performed via the DCS-Client module of the AliRoot package.

The DCS-Client packs the Alias names in a special protocol, ADMP2 (AliDCS Message Protocol 2), and sends the requests to AMANDA (ALICE MANager for Dcs Archives). AMANDA is the server which handles the requests on the DCS Archive DB side. It fetches the desired values from the DB[44] and puts them in the appropriate answer protocol, which is sent back to the DCS-Client. Important for the retrieval from the DB is the requested time period for these values. AMANDA can handle the real DP names and the Alias names for the DB entries. It is a proprietary development of the ALICE DCS group [79] [80].

The DCS-Client wraps the retrieved values in AliDCSValue objects[45] and relays them over to the Pendolino in a collection of all objects of one request. This collection is also referred to as DCS-Map [71]. The DCS-Map is handed to the PredictionProcessor of the corresponding detector for data preparation and enveloping in ROOT files[46].

PredictionProcessor

The PredictionProcessors are detector-specific components of the Pendolino, because they have to prepare and envelope the DCS values according to the needs of their respective detector analysis components, which require these values. They extract the values and corresponding timestamps from the DCS-Map and process them. Old entries of the HCDB and other calibration settings might be required in the preparation

[44] The DCS Archive DB is implemented in an ORACLE Real Application Cluster (RAC). The DCS Archive DB and AMANDA run on Microsoft Windows based machines.

[45] The class AliDCSValue is defined in the AliRoot framework. It contains the actual DCS value, its data type and the timestamp of the value recording.

[46] In case no values of the requested Alias names have been updated during the request time interval, an empty DCS-Map is returned by the DCS-Client. It is also possible that only a subset of the requested DCS values is returned in the DCS-Map, because only these values have been updated. These are normal situations in the Pendolino environment, so Pendolino and the PredictionProcessors have to able to handle them.

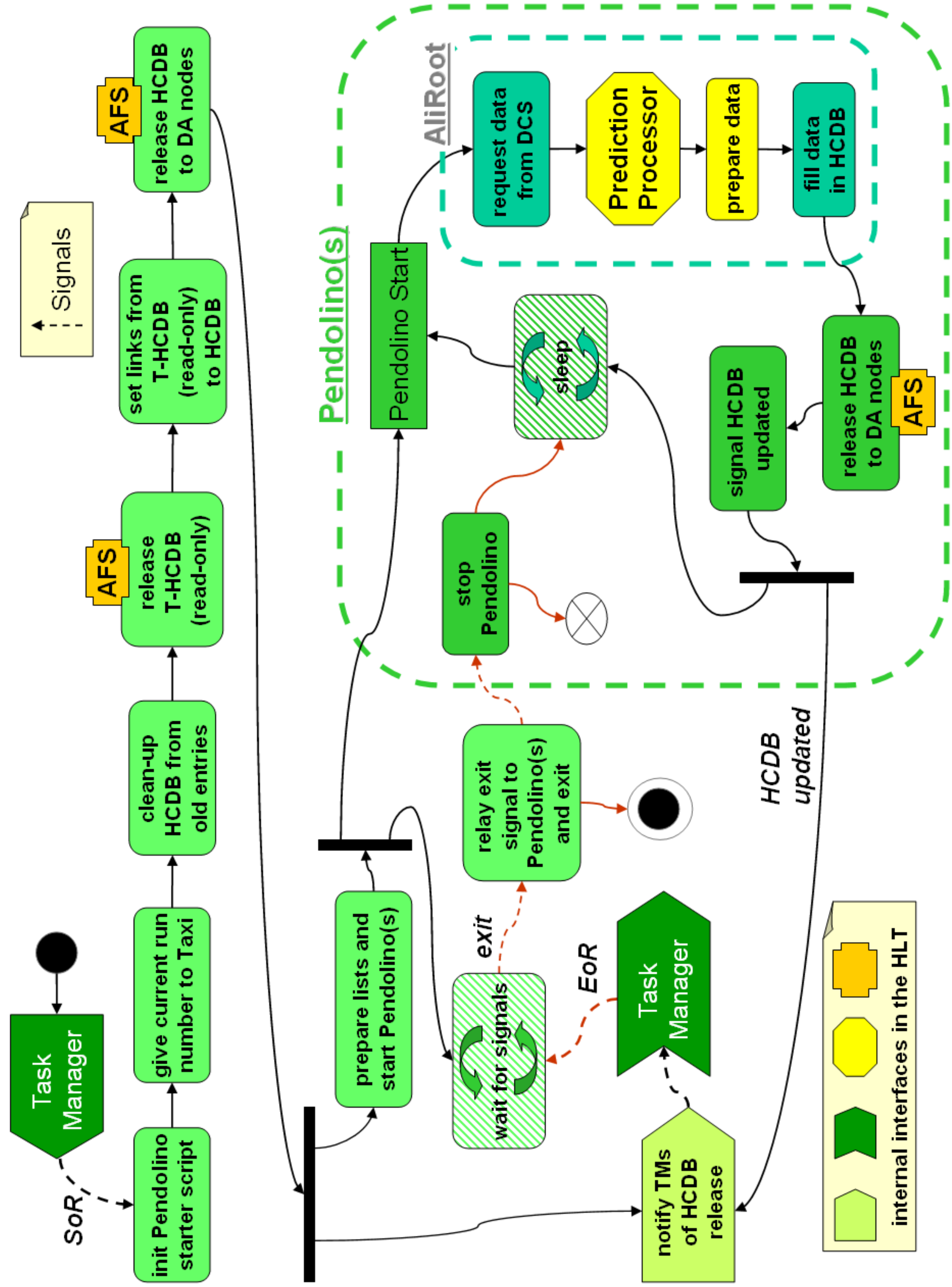

Task Manager
SoR
init Pendolino starter script
give current run number to Taxi
clean-up HCDB from old entries
AFS
release T-HCDB (read-only)
set links from T-HCDB (read-only) to HCDB
AFS
release HCDB to DA nodes
Signals
prepare lists and start Pendolino(s)
wait for signals
exit
EoR
Task Manager
relay exit signal to Pendolino(s) and exit
notify TMs of HCDB release
HCDB updated
Pendolino(s)
Pendolino Start
AliRoot
request data from DCS
Prediction Processor
prepare data
fill data in HCDB
release HCDB to DA nodes
AFS
signal HCDB updated
sleep
stop Pendolino
internal interfaces in the HLT

Figure 4.25: UML Activity Diagram of the Pendolino: The diagram illustrates the activities performed by the Pendolino and its starter script. The Pendolino prepares the HCDB for a run, requests the DCS Archive DB, processes the retrieved values and stores the resulting entries to the HCDB. HCDB preparation and updates are notified to the TMs.

and can be used in the compilation of the resulting objects[47]. Depending on the detector needs these PredictionProcessors must / can incorporate some prediction processing in the DCS data before the latter on are stored in the HCDB. This is due to the fact that the timestamps of the retrieved DCS values are older than the event data streamed through the cluster for analysis. This feature has given them their name: PredictionProcessors.

Important in the preparation by the PredictionProcessors is the rootification of the DCS values. The data have to be enveloped in ROOT files in order to be able to store them in the HCDB and to apply the run validity and versioning scheme common to CDB entries. Moreover the PredictionProcessors have to give the entries the appropriate names by which the analysis components can fetch them from the HCDB, where the data are stored when preparation and rootification are finished.

After new entries have been made in the HCDB, the Pendolino informs the TM framework about the update[48]. The TMs then percolate this information through the analysis chain, allowing the analysis components and DAs to reload their required HCDB entries with the updated values.

The Pendolino can host one PredictionProcessor per detector. In addition, it owns an HLT- and a GRP-PredictionProcessor. Each PredictionProcessor can store objects in the HCDB of its corresponding detector folder; HLT and GRP count as detectors in this perspective.

The interface defined between Pendolino and PredictionProcessors allows the Pendolino to initialise and hand over the DCS-Map to the PredictionProcessors. Additionally, by using this interface the PredictionProcessors can retrieve additional information like the run number and other relevant settings like already included HCDB entries, which are required for further processing of the DCS data. This interface also takes care of the storing of new entries to the HCDB, hiding the CDB access details to the PredictionProcessors. The Pendolino acts in that sense as a framework for the detector-specific PredictionProcessors.

The mechanism of detector-specific modules preparing the data for the CDB has been adapted from the Offline Shuttle, where Preprocessors prepare the data retrieved from the other ALICE systems before the resulting objects are included in the OCDB. This concept allows a clear separation between detector-specific and detector-independent operations. The interface between Pendolino and PredictionProcessors is an extension of the Shuttle - Preprocesssor interface in order to allow detector experts the re-usage of already existing components [71].

[47] For example: the TPC uses a temperature map of its detector in their analysis scheme. In the preparation of this map certain geometry settings are utilised, which are normally stored in the OCDB. These settings are fetched from the OCDB by the Taxi and copied to T-HCDB, respectively HCDB. From the latter one they are provided to the TPC PredictionProcessor.

[48] This notification is sent through the InterfaceLibrary of the TM framework, using a dedicated application called TM_Notifier.

HCDB

As described before the HCDB contains a per-run-frozen version of the T-HCDB. The entries are included as links to the corresponding files in the T-HCDB before each run[49].

In addition, entries from the PredictionProcessors with DCS data and configuration entries made by the HCDBManager script are included. The HCDB is distributed to the computing and interface nodes inside the HLT cluster via AFS. Analysis components and DAs inside the HLT have only access to the HCDB, but not to the T-HCDB.

Like the T-HCDB and OCDB, the HCDB consists of a file catalogue[50]. Its structure, which is shown in figure 4.19, is the same as for the T-HCDB and OCDB. In addition, the HCDB contains a folder called *"HLTPermanent"*, where detector experts can store configurations for their online analysis components. This folder is only filled by the HCDBManager script (see description on page 100). Its substructure is identical to the one of the normal detector folders.

For the new entries produced by the PredictionProcessors the same rules apply for run validity and versioning scheme as described for the T-HCDB on page 72. But, in contrary to the links to the T-HCDB entries, these entries are included as real files. Since the HCDB is only a local copy of the OCDB, updates of already existing entries receive only an increased subversion number. The version number stays the same. Depending on the frequency of the corresponding Pendolino and the changes of the DCS values, a large amount of updated objects with increased subversion numbers for one entry can be stored. Contrary to updates from the OCDB, these entries are updated during a given run.

All entries in the HCDB, except the ones inside the *"HLTPermanent"* folder, are cleaned up before the next run[51]. The clean-up is necessary because the frequent requests of the Pendolino(s) can result in a large amount of updated entries[52] and the current conditions retrieved from DCS should not be ported from previous runs.

4.5.2 The FED-Portal

Normally detector data are inserted into the DCS via the FED-API (cf. section 3.4). The HLT uses the same mechanism to feed data to the DCS. Therefore the Service channel part of the FED-Server has been implemented. A dedicated subscriber component, the FED-Subscriber, collects the values provided by DAs inside the HLT cluster. The collected values are translated to the corresponding FED-Services and published

[49] The usage of links has been chosen to reduce the time spent in the preparation phase before each run.

[50] In HLT, the base directory of the HCDB is defined by the environmental variable **ALIHLT_HCDBDIR** and is normally located under ***/opt/HCDB*** .

[51] The clean-up of the HCDB is not performed at the end of a run, because analysis components might still have events in the queue to analyse and issue requests to the HCDB, while the Pendolino has already received an exit signal.

[52] Each update is stored in a new file with the old one remaining in the FC.

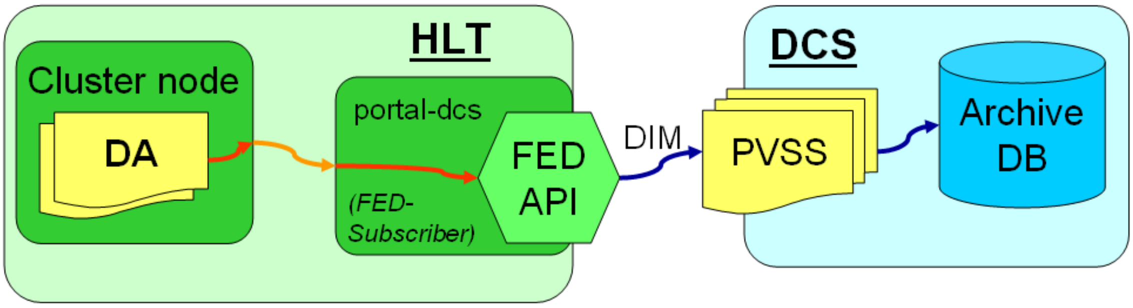


Figure 4.26: Sketch of the HLT FED-Portal interface: The HLT can send data, produced by DAs inside the HLT, to DCS via the FEP-API. These data are received by PVSS panels and included into the DCS Archive DB.

to the DCS. On the DCS side PVSS panels can subscribe to the published values and store them to the DCS Archive DB. This also offers the possibility to let the HLT system be monitored by the DCS. In general, all values transferred to the DCS PVSS panels can be monitored online by operators. On the HLT side, the required components for this interface are aggregated in the FED-Portal.

The UML diagram in figure 4.27 presents a use case, displaying the FED-Portal in providing the calculated value of the TPC drift velocity. Event data which is streamed from the FEEs to the HLT, are used by a TPC DA for calculating the TPC drift velocity. The results are handed to the FED-Subscriber at the FED-Portal, where they are published as a FED-Service. A dedicated PVSS panel on DCS side retrieves the values and presents them for online monitoring to the operators. For later physics analysis the values are stored in the DCS Archive DB.

The FED-Portal employs two different transportation frameworks in the data exchange. The FED-Subscriber receives HLT data via the HLT PubSub framework; values sent out to the DCS are published via DIM Services. The implementation of the FED-Portal uses the Adapter Design Pattern to combine both functionalities. The Adapter Pattern is used to map one interface to the domain specific interface of another system or package. There are two ways for the Adapter Pattern to be implemented:

1. The adapter class inherits from both interfaces (multiple inheritance - *class adapter*).

2. The adapter class composes an instance of one interface in the implementation (inheritance) of the second interface (*object adapter*).

These two possibilities for an Adapter Pattern are depicted in figure 4.28 [67]. The FED-Portal component is a hybrid of both methods in its implementation. Its subscriber class is derived from the `AliHLTProcessingSubscriber` (see Class Diagram in figure 4.29) to receive data through the PubSub chain. Additionally, it inherits from `DimServer` to get the DIM Server functionalities, like starting and stopping the server.

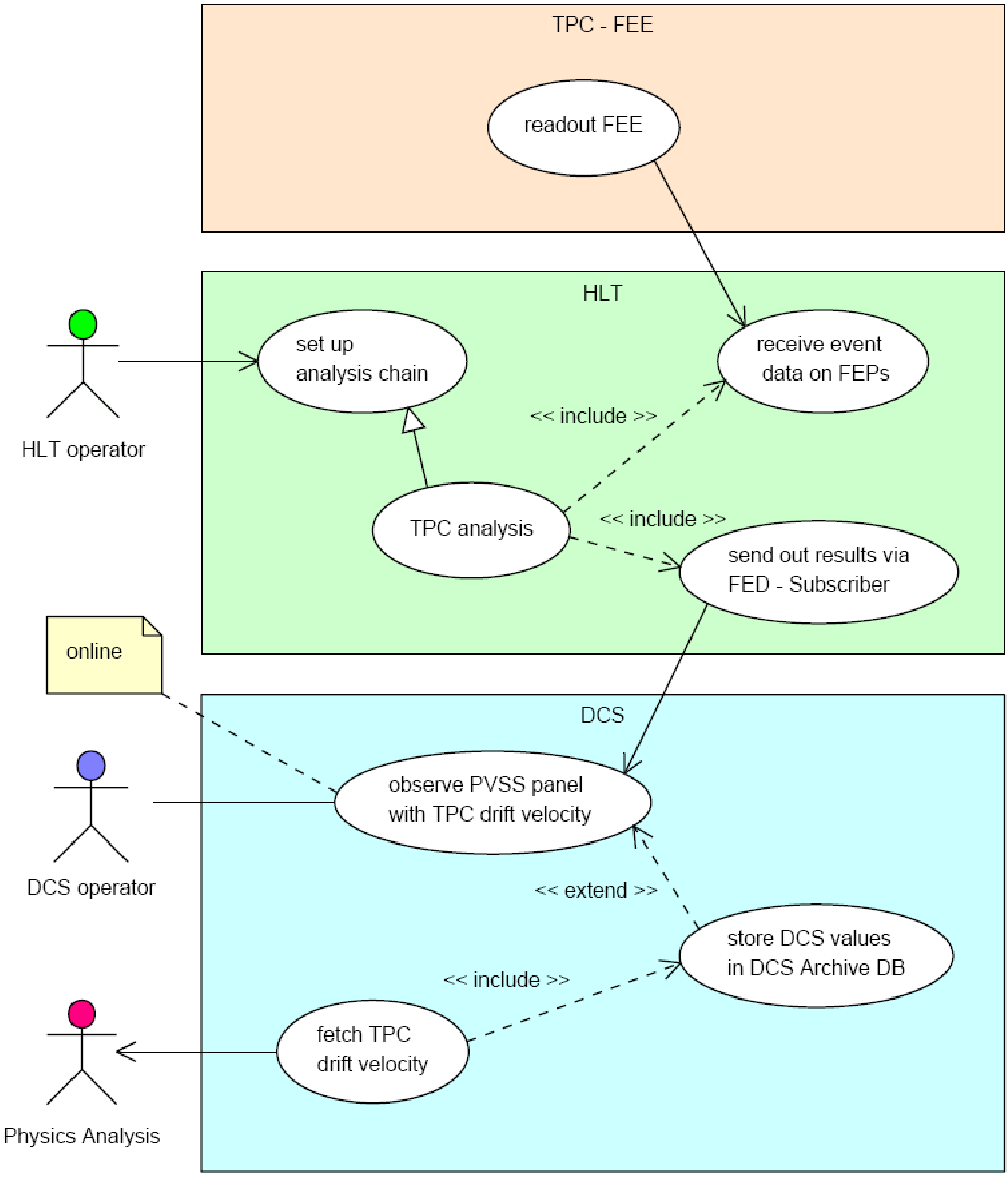


Figure 4.27: UML Use Case Diagram of the FED-Portal for the TPC drift velocity example. Event data from the TPC FEEs are streamed to the HLT, where they are analysed. The TPC drift velocity is calculated by a dedicated TPC DA. The results are sent to DCS PVSS panels via the FED-Portal. On the DCS side they can be monitored online by operators. Physics analysis can fetch them from the DCS Archive DB later-on.

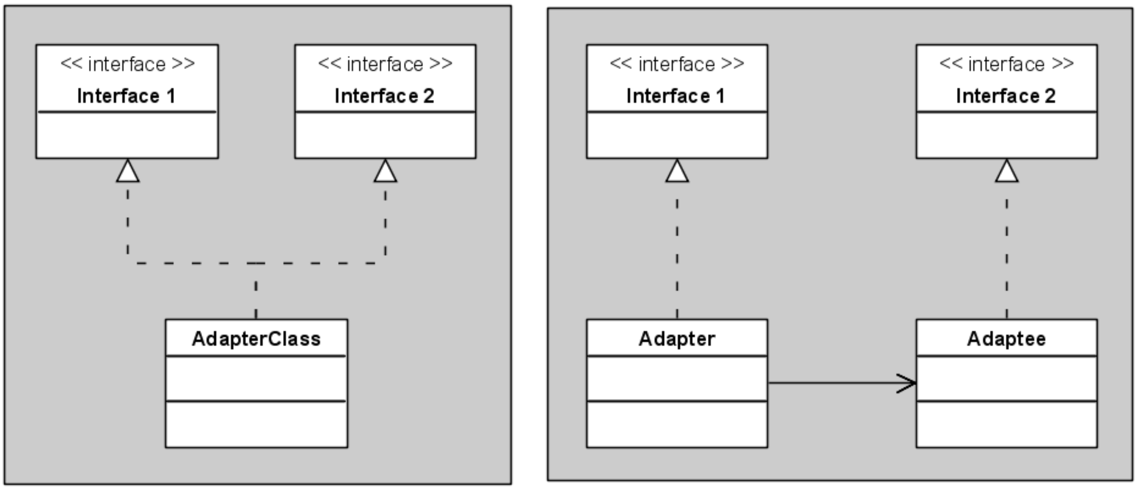


Figure 4.28: This UML Class Diagram shows the two possibilities for an implementation of the Adapter Design Pattern. On the left side the ***AdapterClass*** is inherited from ***Interface 1*** and ***Interface 2***, thereby combining both functionalities (*class adapter*). The right side shows the ***Adapter*** being inherited from ***Interface 1*** and getting the functionality of ***Interface 2*** via a composition of its implementation in the ***Adaptee*** class (*object adapter*).

Thereby it implements the *class adapter*. The actual FED-Service channels[53] are given by objects which are compositions in the FED-Subscriber class. Therefore it represents an *object adapter* as well. The class structure of the FED-Portal is shown in the UML Class diagram of figure 4.29.

The FED-Portal has to be located on a node having at least two ethernet interfaces: one to the HLT cluster and one to the DCS net in the DCS CR (Counting Room)[54].

The FED-Subscriber

A dedicated component of the PubSub framework, the FED-Subscriber[55], is responsible for collecting specific DA results inside the HLT cluster. These results are destined for sending to the DCS PVSS panels. The collected values are sorted into the assigned FED-Service channels. The actual channel depends on the designed setup for the running chain and on what is provided by the DAs. In case the received data are not log messages, they can be assigned to either Grouped- or Single Service channels.

In the protocol transferred to the FED-Subscriber, the field "*fDataType*", received in the PubSub header, has to match the bit field pattern given by the characters of `'FED-SRV '` (cf. general protocol structure of data shipped by the PubSub system in figure 4.30). The transferred payload of the PubSub protocol has to match the

[53] FED-Service channels are represented by objects of the `DimService` class or derived subclasses.

[54] On the HLT cluster the nodes *portal-dcs0* and *portal-dcs1* fulfil these requirements.

[55] The FED-Subscriber is represented by the class `AliHLTFEDSubscriber`, see UML Class Diagram 4.29.

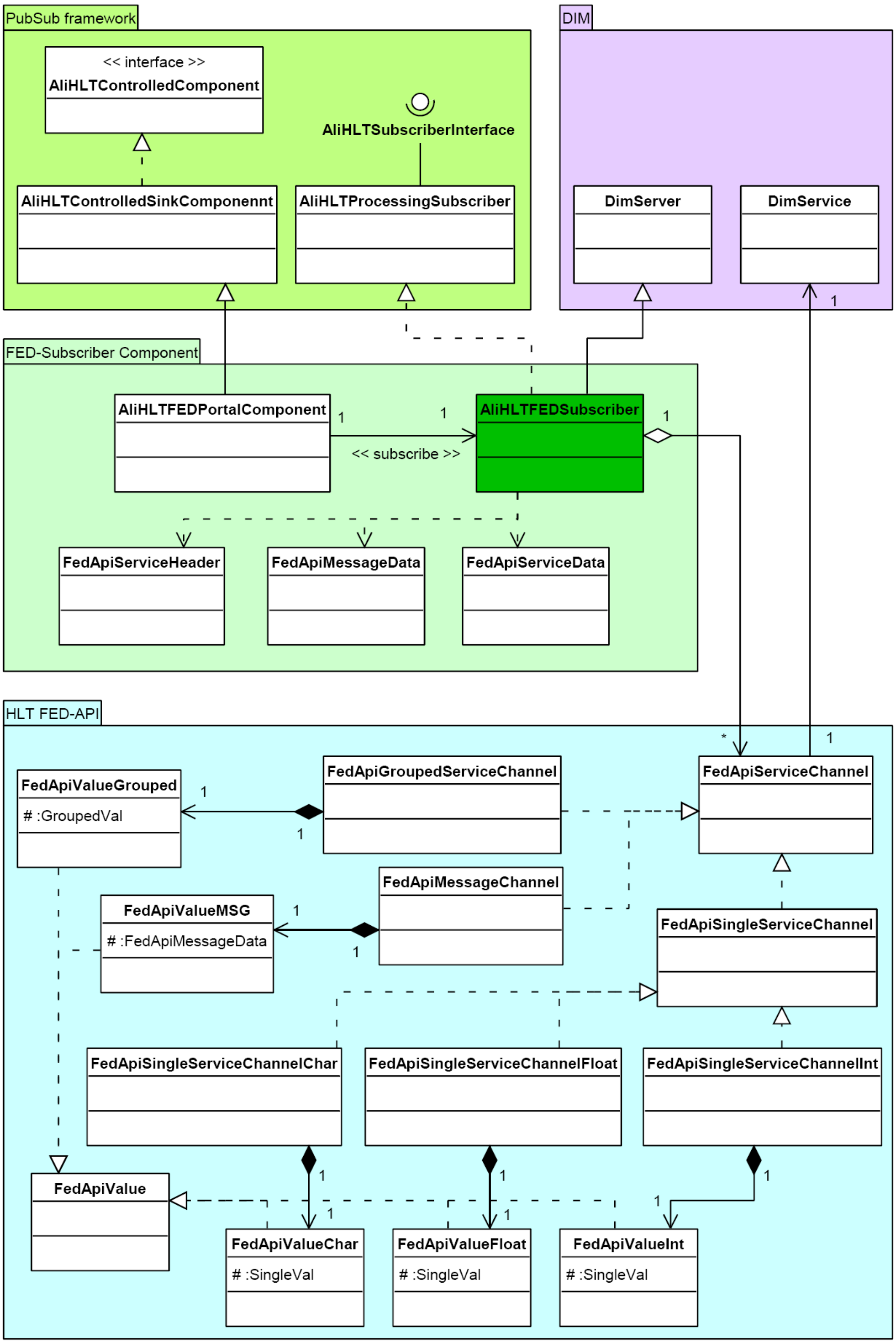


Figure 4.29: UML Class Diagram of the FED-Subscriber. The FED-Portal component is derived from the PubSub framework; the FED-Subscriber receives data as child class from `AliHLTProcessingSubscriber`. Additionally, it inherits the DIM Server functionalities from `DimServer` of the DIM package. The FED-Services are interfaced by a composition of specially wrapped `DimService` objects.

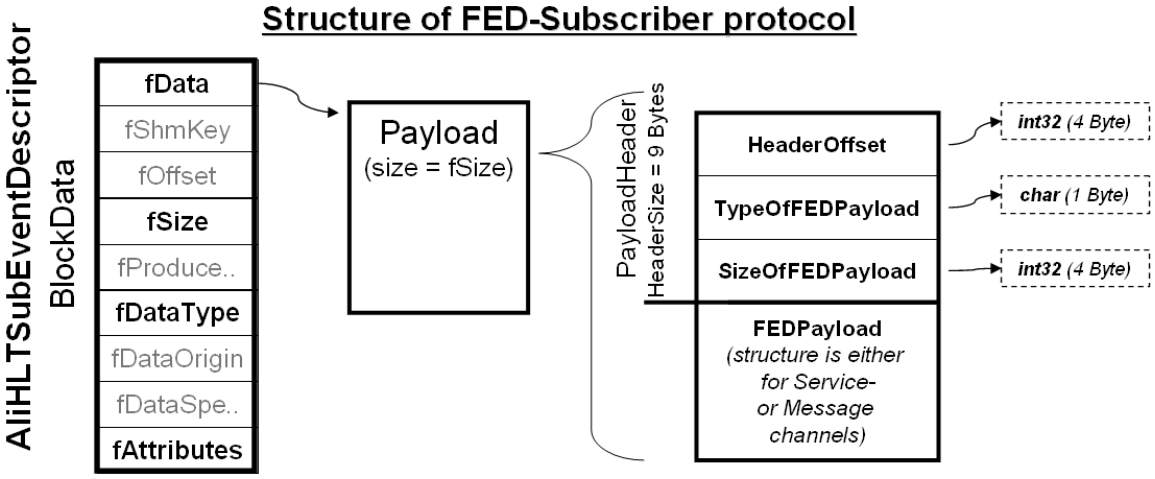


Figure 4.30: Protocol structure for the FED-Subscriber. The payload received at the FED-Subscriber is divided again into a header and a body part. The header contains information about offset, body type and body size. The body is called FEDPayload and contains the data for either Service- or Message channels.

structure in table 4.4 in order to be understood by the FED-Subscriber.

Protocol structure for FED-Subscriber			
HeaderOffset	*TypeOfFEDPayload*	*SizeOfFEDPayload*	*FEDPayload*
`int32`	`char`	`int32`	`byte array[]`

Table 4.4: The table presents the protocol structure of the data transferred to the FED-Subscriber through the PubSub system. *HeaderOffset*: Size of this header in Bytes, used as offset to the *FEDPayload* data. The offset is stored in a 32 bit integer. *TypeOfFED-Payload*: Type of the *FEDPayload*, stored in a single char (*'S'* = Single Service, *'G'* = Grouped Service, *'M'* = Message Service). *SizeOfFEDPayload*: Size of the *FEDPayload* in Bytes, stored in a 32 bit integer. *FEDPayload*: The actual payload used for the corresponding FED-Service channel. It can vary in its size, depending on the given service data. The possible structures of the *FEDPayload* are shown in the tables 4.5 and 4.6.

The *FEDPayload*, given in the above shown protocol in table 4.4, can have different structures, depending on the destined FED-Service channel (defined by the *TypeOfFEDPayload* field). The tables 4.5 and 4.6 visualise the possible protocols. Single- and Grouped Services are using the same *FEDPayload* structure.

After mapping of the FEDPayload to the required structure by the chosen FED-Service, the data are handed to the corresponding FED-Service channel.

The FED-API in the FED-Portal

The FED-API, as used in the DCS, defines data exchange in both directions: DCS can issue commands to the connected (sub-)systems, results and monitored values are sent

FEDPayload **for Single- and Grouped Service channels**					
NameLength	*charValLength*	*intValue*	*floatValue*	*ServiceName*	*charValue*
`int32`	`int32`	`int32`	`float`	`char[]`	`char[]`

Table 4.5: The table shows the *FEDPayload* structure for a Single- or Grouped Service channel. The *NameLength*, the *charValLength* and the *intValue* are stored in 32 bit integers. The *floatValue* is transferred by a float variable. The *ServiceName* is delivered in a char array of arbitrary size. It has to be NULL terminated and has a size given by *NameLength* (including the NULL termination). For the *charValue* a char array of arbitrary size is used. Its size is defined in *charValLength*. The unused value types should contain defined *"Don't-Care"* - values. The aggregated size of this structure can vary.

FEDPayload **for Message channel**			
logType	*source*	*message*	*timestamp*
`int32`	`char[256]`	`char[256]`	`char[20]`

Table 4.6: The table shows the *FEDPayload* structure for a Message channel. The *logType* is given by a 32 bit integer (see appendix A.3 for possible log types). The *source* (name of the issuing component) is transmitted in a 256 Byte char array. The detector name has to be encoded in *source* as well. A 256 Byte char array is used for the actual *message*. The *timestamp* in the format "YYYY-MM-DD hh:mm:ss\0" is taken from a 20 Byte char array. All char arrays have to be NULL terminated. This structure has a fixed size of 540 Bytes.

back to the DCS. Since the HLT is steered directly by the ECS (and not by the DCS), only the FED-Service channels of the FED-API are used in the FED-Portal interface. The FED-Command channels are omitted[56]. Via the FED-Services the HLT can publish calculated results as integer, float or char values to the DCS. The structure of the corresponding channels matches the description of the ***Grouped Service channel***, the ***Single Service channel*** and the ***Message channel*** in section 3.4.

The FED-Services

The channels defined in the FED-API are construed to be implemented in DIM Services and Commands. Thereby DIM serves as transportation layer.

The created service channels are represented by `DimService` objects from the DIM framework[57]. The data received from the DAs are mapped to the according fields of the corresponding service channel. The service channel name, which is responsible for offering the data, depends on the provided service name. After filling the channel an update call for that channel is issued to DIM and the data of the channel fields are

[56] The ***Acknowledge channel*** is also obsolete, because no command channels are offered by the HLT FED-Server.

[57] `DimService` is the (parent) class for services published by DIM Servers, when using the C++ version of DIM. For commands the class `DimCommand` is used.

transferred to the subscribed DIM Client(s).

A FED-Client using the DIM-PVSS module enables the DCS to subscribe to the published FED-Services. In order to have the FED-Server of the FED-Portal connected to the PVSS panels on the DCS side, it has to contact the DIM_DNS used by the corresponding DCS PVSS panels. There it announces its services. The role of the DIM_DNS in the connection between DIM Servers and DIM Clients is described in section 3.3.

FED-Portal logging

Normally, logging on the FED-Portal is done using the logging facility provided by the PubSub framework. These log messages can be centrally monitored by the HLT operator. Additionally, log output can be offered as FED-Service to the DCS via the **Message channel** of the FED-API. In the latter case the defined channel members have to be provided, i.e. the *log type*[58], the *detector name*[59], the *source* (component name), the actual *message* and the *timestamp*.

[58] Appendix A.3 shows the different log types defined in the FED-API.

[59] For log messages transferred via the FED-Portal, the detector name is always set to ***"HLT"***. Should the message come from a detector-specific component inside the HLT, then the corresponding detector name has to be included in the component name, which is set in the *source* field.

4.6 DAQ

The interface to DAQ consists mainly of a hardware interface. It is constituted of the DDL connections between the D-RORCs and the H-RORCs[60]. The HLT receives raw event data from the D-RORCs on the DAQ LDC via 365 DDLs: 36 ITS (SPD and SSD) links, 216 TPC links, 18 TRD links, 20 PHOS links, 24 EMCal links, 20 HMPID links, 23 DiMUON links, 3 FMD links and 1 link for each of V0, T0, ACORDE, ZDC and Trigger. It is foreseen that the missing detectors (SDD[61], TOF and PMD) are joining as well, having in the end 454 DDL links. The data is currently received by 185 H-RORCs, most of them being equipped with two DIUs. These H-RORCs are hosted by dedicated 87 FEP nodes.

For handing back processed results and trigger decisions (see section 2.4.5 on page 21) the same transport mechanism is used. The HLT has 12 DDLs to sent them to the DAQ LDCs. These DDLs are mounted on 10 H-RORCs, having two of them equipped with two SIUs for sending. Different sets of physic triggers are to be defined and communicated to the DAQ [8] [40].

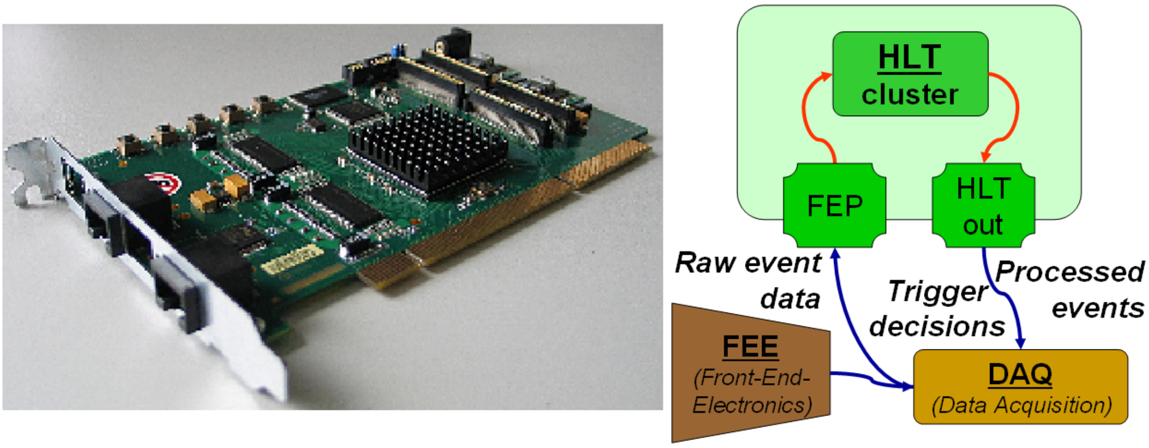


Figure 4.31: The left side shows a picture of an H-RORC, the two connectors to the DDLs are visible on the left end. On the right side the use cases of the two interfaces to the DAQ are presented.

[60] The H-RORCs, and therefore also the interface to the DAQ, has been developed by Thorsten Alt from the Kirchhoff Institute of Physics, Ruprecht-Karls-University Heidelberg (Germany).

[61] The SDD connection has been added lately in Autumn 2008.

4.7 AliEve

The Offline event monitoring tool AliEve can be used in online analysis as well. To get online access to this event display, HLT provides HOMER (HLT Online Monitoring Environment including ROOT)[62]. This interface allows to send histograms and reconstruction results to AliEve. It enables the HLT to perform direct detector performance monitoring.

HOMER can be used to send results produced from any step of the HLT analysis chain to an AliEve application running in the ALICE Control Room (ACR). Thereby it enables online event monitoring to the operators. It connects to the PubSub chain either by TCP/IP or shared memory connections. A screenshot of the AliEve usage in online monitoring is given by figure 4.32 [8] [40].

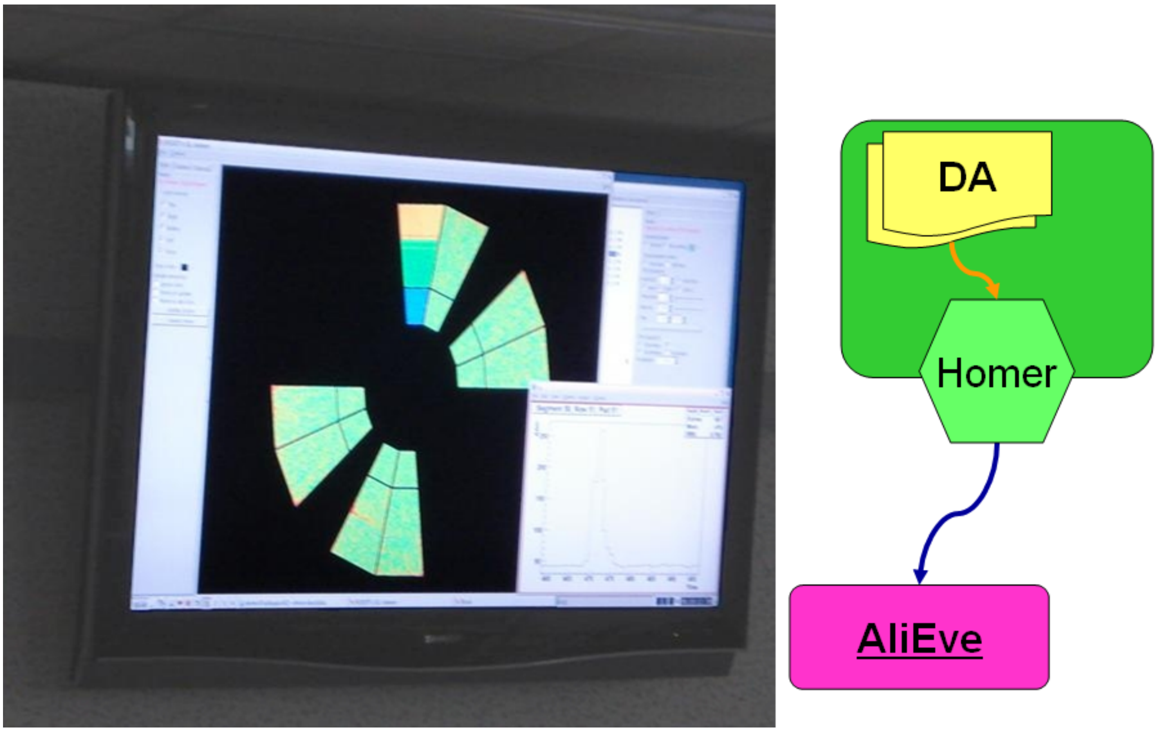


Figure 4.32: The left picture shows a real "screenshot" from the AliEve online usage in the ACR. On the right side the use case of HOMER is displayed.

[62] The HOMER interface has been developed by Dr. Timm M. Steinbeck and Jochen Thäder from the Kirchhoff Institute of Physics, Ruprecht-Karls-University Heidelberg (Germany).

Chapter 5

HLT calibration framework

5.1 Putting the bits and pieces together

The afore described interfaces constitute the major components of the ALICE HLT calibration framework. All these interface applications enable and support DAs inside the HLT to perform their assigned tasks. Each of them plays a dedicated part in the provision of calibration and condition settings from and in the distribution of results to the connected ALICE systems. The applications handle the data exchange generically, the actual data are transparently transferred by the framework[1]. Detector-related subcomponents, integrated in these interfaces, take care of the specific preparation of the data, according to the needs of the corresponding detector.

During their execution, these interfaces have also to interact and exchange data among each other. Especially the HLT-proxy plays an important role here. Apart from steering ALICE and synchronising the actions of its systems and subsystems, the ECS distributes chosen general run parameters (see CONFIGURE- and ENGAGE parameters on page 55*f*). Some of them are provided to the components through the PubSub chain, inserted by the TMs. Others are included into the HCDB, where they can be fetched by the DAs. Additionally, they are distributed to dedicated interface applications by the RunManager. The run number, for example, is needed by most of the interfaces. The Taxi requires it to issue requests to the OCDB for the next run[2]. At the Shuttle-Portal, it is utilised as name for the first folder level in the FXS and to create unique keys for the meta data stored in the involved MySQL DB. The Pendolino receives the run number as well and hands it further to the PredictionProcessors. They need it in the encoding of the real file names, under which the produced HCDB entries are stored[3]. Additionally, the Pendolino provides other ECS parameters to the PredictionProcessors, like the run type.

The assignments of the calibration framework can be mainly divided into two

[1] Except of certain settings, which are required for the preparation of the contact to the other systems, like object- or corresponding detector names.

[2] "Next run" means the run after the one that is about to be started.

[3] The actual creation of the file name is done by the CDB access classes, but the PredictionProcessors have to inform them about the current run number.

different operations:

- Bringing the calibration settings to the HLT cluster and distributing them to the analysis components inside (Calibration input).
- Collecting newly calculated calibration data and sending them to the destined system(s) (Calibration output).

5.1.1 Calibration Input

In the HLT calibration input procedures, the HCDB plays the central role. This is the location, where most of the calibration- and run condition- settings are accessed by the analysis components. The tasks involved in its filling and provision to the DAs are visualised in figure 5.1. They are summarized in the following.

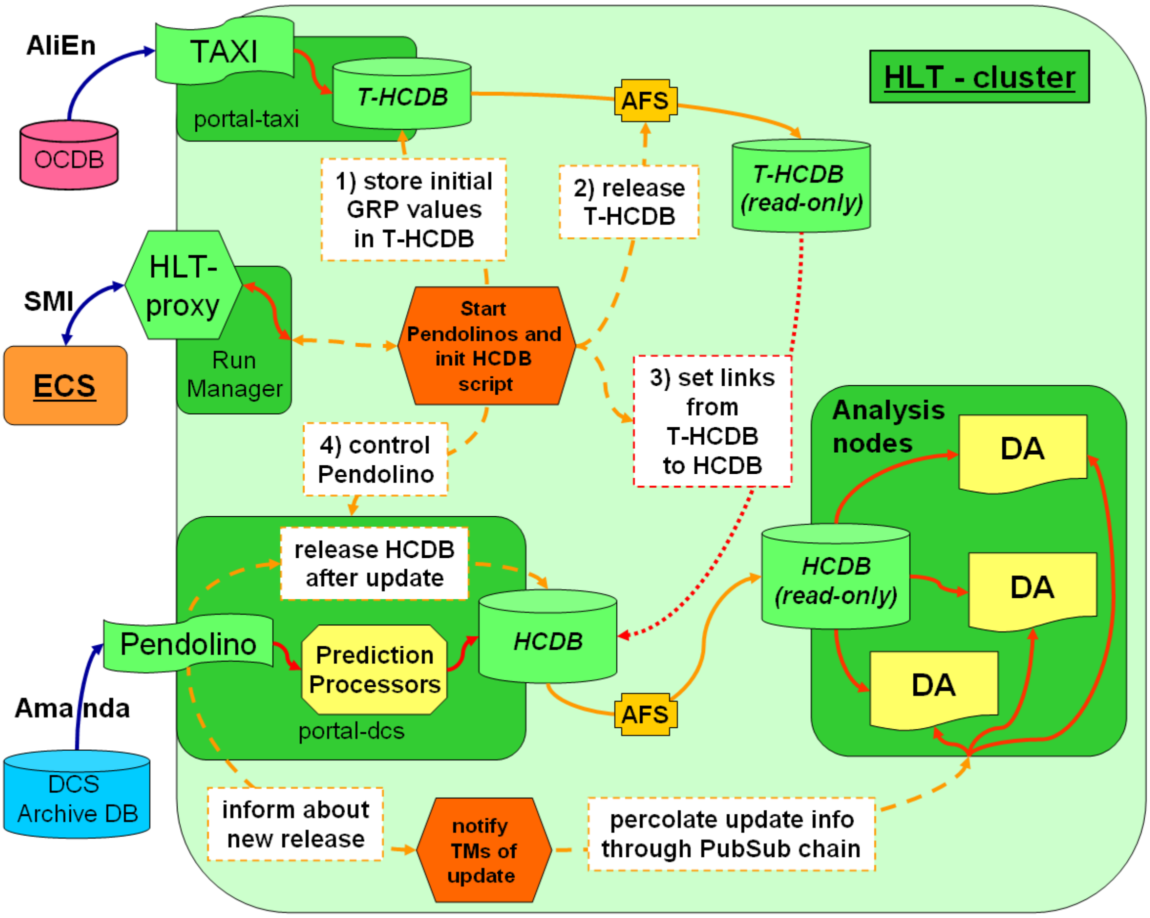


Figure 5.1: This figure visualises the complex tasks involved in filling the HCDB with calibration- and condition data for the DAs.

Using AliEn the Taxi fetches calibration settings from the OCDB and fills them to the T-HCDB. When a run is started, the HLT-proxy receives from the ECS the according commands and parameters. They are relayed to the RunManager, which starts the operations chosen for the upcoming run. In this process the initial GRP values are stored to the T-HCDB as well. Then the T-HCDB is released using the AFS read-only partition of the corresponding FC and its entries are linked to the HCDB. Afterwards the Pendolino(s) are started and the HCDB is released to the computing nodes. Here again, in the release process the AFS features of the read-only partitions for the defined folders are used. The DAs have only read access to the HCDB on the cluster.

The Pendolinos with their PredictionProcessors continuously fill retrieved DCS values into the HCDB. After each update interval, the HCDB is released again and the DAs receive according notifications. These notifications are issued by the Pendolinos and sent first to the TMs. They percolate them then as special events to the components through the PubSub chain.

Manual (re-)configuration of the components can be achieved via the HCDBManager script (see section 5.2 below), which uses the HCDB for storing of configuration data as well.

A major focus in the development of the calibration input arrangement has been on providing an environment, which makes it transparent for the DAs to run under online or offline conditions.

5.1.2 Calibration Output

While processing event data, the HLT components produce new calibration settings. The results are regarded as the HLT calibration output. They are shipped from the DAs to the corresponding interface applications via the PubSub chain.

DCS related data are received at the FED-Portal and sent further to the corresponding DCS PVSS panels. There they can be monitored online by operators. In addition, they are inserted into the DCS Archive DB.

New calibration objects, which are destined for the OCDB, are collected by the Shuttle-Portal. When the collecting procedure has finished, the Offline Shuttle can start fetching them from the HLT FXS and the corresponding MySQL DB. After preparation by Preprocessors, they are included into the OCDB. Synchronisation between HLT and the Offline Shuttle is achieved via the ECS.

A cycle of calibration objects production and usage later-on starts with DAs calculating them online on the HLT computing nodes. The new objects are included into the OCDB as described above[4]. During this procedure the official version numbers are applied to the new objects. The HLT Taxi requests the latest OCDB entries and caches them locally in the T-HCDB. Before each run, the HLT Pendolino prepares the HCDB, inter alia by setting links to the content of the T-HCDB, and distributes it to the computing nodes. There the DAs access them again to perform their assigned tasks.

The cycle of sending first newly calculated calibration objects to the OCDB, before they are fetched again for provision to the DAs, is required in order to use the official

[4] This thesis refers here only to online produced calibration objects. Offline produces calibration objects as well and stores them to the OCDB.

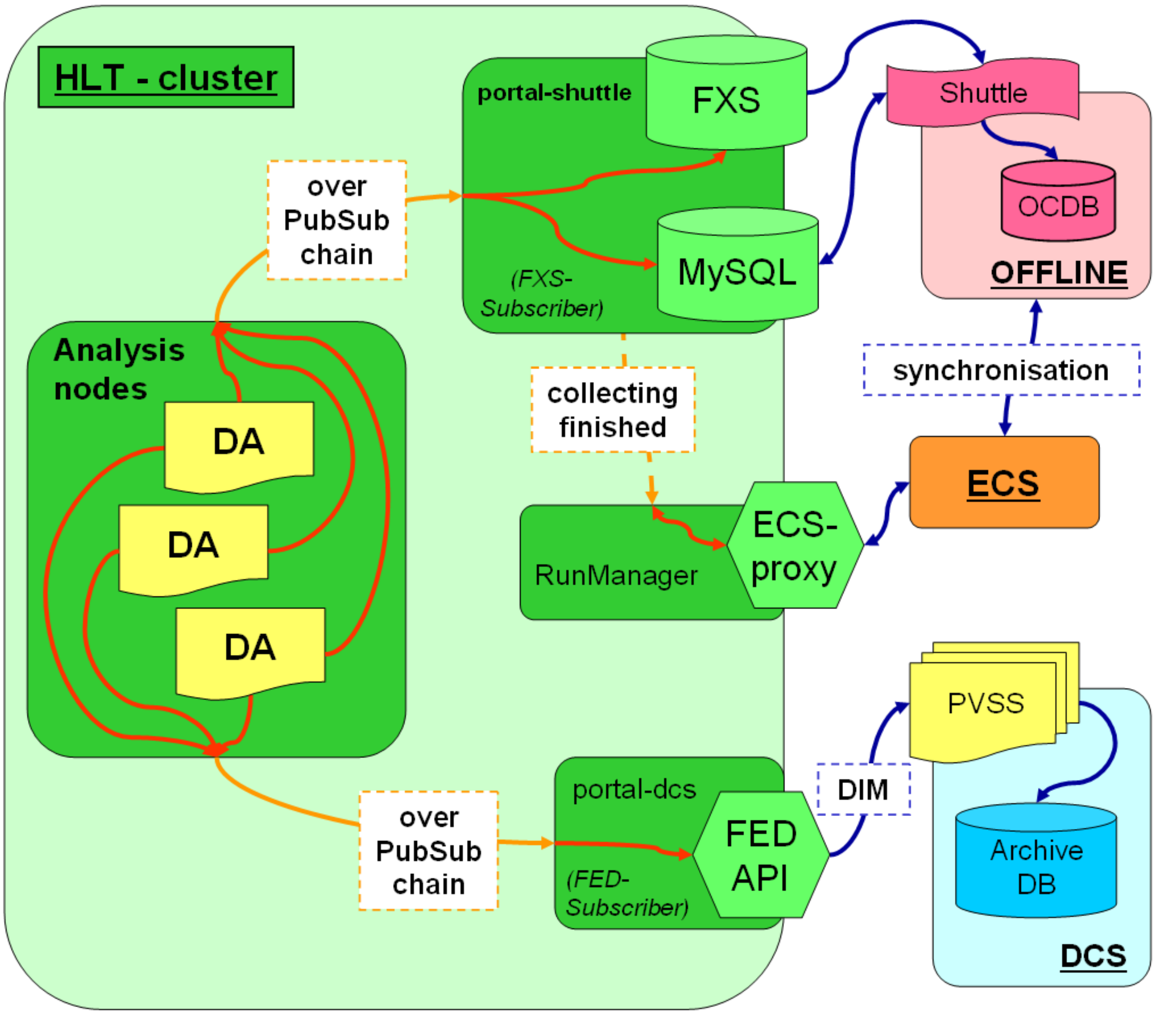


Figure 5.2: The figure shows the tasks for sending out newly produced calibration data to the destined systems.

version numbering set by ALICE Offline. This is necessary to be able to compare results with Offline analysis later-on.

5.2 The HCDBManager

The HCDBManager script is used to set configuration entries for the HLT analysis components into the HCDB. This can also be done online to perform a re-configuration of certain components during a run. After the entry is set, the script notifies the TMs, which percolate an according reconfigure event through the analysis chain[5]. This informs the dedicated analysis components to call their reconfigure function and reload the corresponding configuration entry from the HCDB. For the notification, the script uses the *TM_Notifier* tool, which is based on the InterfaceLibrary of the

[5] Configuration entries without issuing reconfigure events can be set as well.

TM framework. The issued signal carries special information about the dedicated component that shall perform the reconfiguration.

The HCDBManager script can load root files and transform them into the format required for CDB entries[6]. Alternatively, it can accept strings, which are wrapped in `TObjString` objects[7], before they are included into the HCDB. The entries are either stored under the normal detector folders of the HCDB or the script can put them in the *"HLTPermanent"* folder. The latter one is not cleaned up at the start of a run, therefore this folder has to be used, when the configuration entry is intended to be re-used in several runs (cf. HCDB description on page 87).

The usage of the HCDBManager script is described in appendix B.6.

5.3 Synchronisation sequence

Since the HLT calibration framework applications interact with the other ALICE systems, synchronisation is a vital issue for their interplay. It is achieved via communication with the ECS, which connects to all ALICE systems. In the timeline of the interface employment, their usage can be divided into five stages.

1. Asynchronous to the runs: Before the start of a series of physics runs, and then repeated in regular time intervals independent from any run, the Taxi requests the OCDB for latest calibration settings. The fetched objects are stored to the T-HCDB.

2. Preparation period at the Start-of-Run (SoR): The ECS informs the HLT about an upcoming run and provides certain general run settings, i.e. run number, beam type, trigger classes, etc.. In the following configuration steps, the HCDB gets prepared and distributed to the computing nodes. Updates from the OCDB after the start are not referred to the HCDB, in order to keep a coherent version of calibration settings during a run. Once the HCDB is prepared, the analysis components can be initialised. The Pendolino begins to request the DCS Archive DB, the detector-specific PredictionProcessors process the retrieved values and store them to the HCDB.

3. During a run: The start of data taking is signalled with the SoD event, which should go along with the START command from the ECS. Raw event data are received from the FEEs on the HLT FEP nodes. Over several analysis steps the analysis components process the data and reconstruct events. The results, together with trigger information, are streamed out to DAQ for permanent storage via the HLT-out nodes. In addition, the DAs produce new calibration settings.

[6] This "transformation" is mainly the extraction of the ROOT- or AliRoot object, containing the calibration parameter. They must be derived from `TObject`. The extracted object is handed to the CDB access classes for wrapping and storing it in the HCDB.

[7] `TObjString` is a class provided by the ROOT framework. It is derived from `TObject` and therefore suitable for the use in CDB entries.

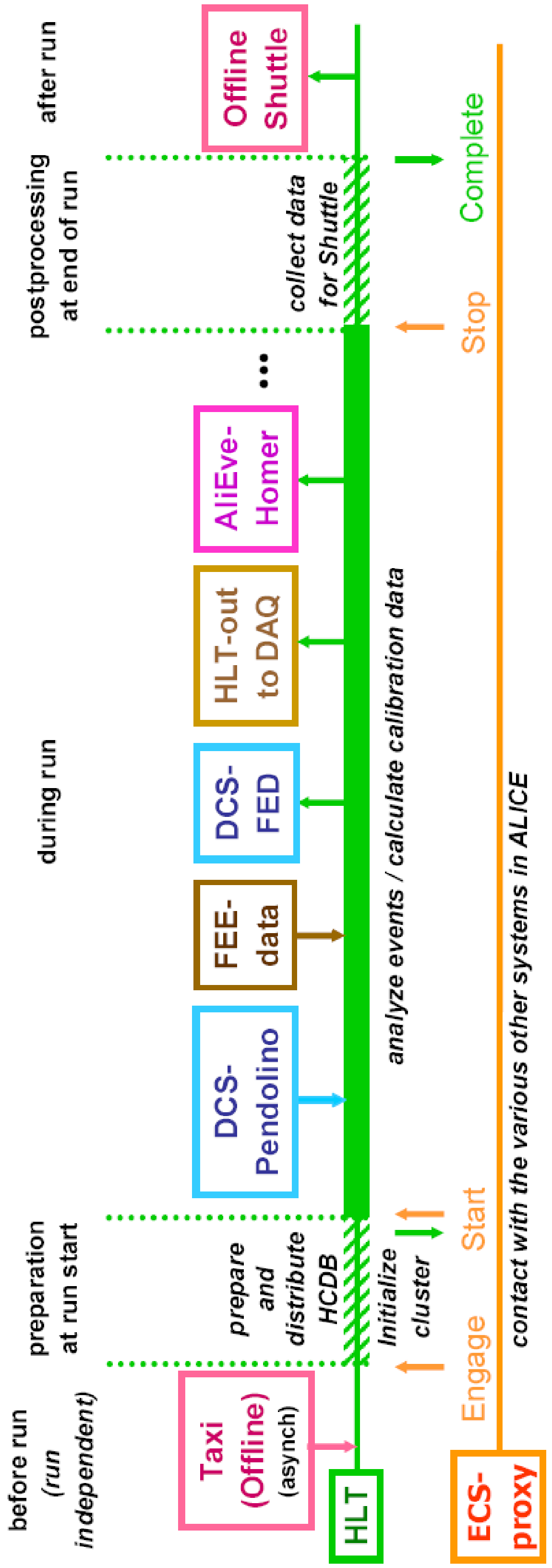


Figure 5.3: The diagram displays the timeline of the usage of the different HLT interfaces. Synchronisation with the other ALICE systems is achieved via the ECS.

Current environment conditions (i.e. temperatures, pressure values, voltages, B-field measurements, etc.) are continuously requested from the DCS via the Pendolino and stored to the HCDB. After an update of the HCDB, the latter is freshly released to the computing nodes and the DAs receive a notification. Certain parameters, which are calculated by the HLT DAs and are relevant to the DCS, like the TPC drift velocity, are published via the FED-Portal. DCS PVSS panels can subscribe to them for online monitoring and store them to the DCS Archive DB. Detector performance monitoring can be achieved online via the HOMER - AliEve connection. Using this interface, selected events and calibration results can be visualised during the run as well.

4. Postprocessing period at the End-of-Run (EoR): At the end of data taking the ECS issues a STOP command and the EoD event is percolated through the HLT chain. The HLT transits to the **COMPLETING** state. It can take some time until all event queues on the computing nodes are emptied by the corresponding analysis components. During this period the Shuttle-Portal collects newly calculated calibration settings and stores them to the FXS. Corresponding meta data are included into the MySQL DB (*hlt_ logbook*). After these tasks are finished, the HLT switches implicitly to the **READY** state. This state change is signalled to the ECS.

5. After a run: Once the HLT is back to the **READY** state, the ECS can notify Offline to start its Shuttle. The Offline Shuttle requests the HLT FXS and the *hlt_ logbook* and after some detector-specific preprocessing it puts the new calibration objects to the OCDB. While the Shuttle is running, the HLT can already be started for the next run.

The synchronisation timeline of the interfaces together with their corresponding tasks are visualised in figure 5.3 [40]. The current HLT states play an important role in the synchronisation interplay, the major course of activities related to certain states or state transitions are shown in figure 5.4.

The above mentioned HCDBManager script can be executed by the operator at any time. Beyond a run, it is employed for setting configuration entries for components to the *"HLTPermanent"* folder. During a run, the operator can use it to issue a reconfiguration of certain components.

5.4 Applications

5.4.1 General procedures

Alignment and component calibration

The prime example for the usage of this framework is the calibration of analysis components with detector properties and configurations. In order to process received event data properly, the analysis components need to know the pedestal settings, which are

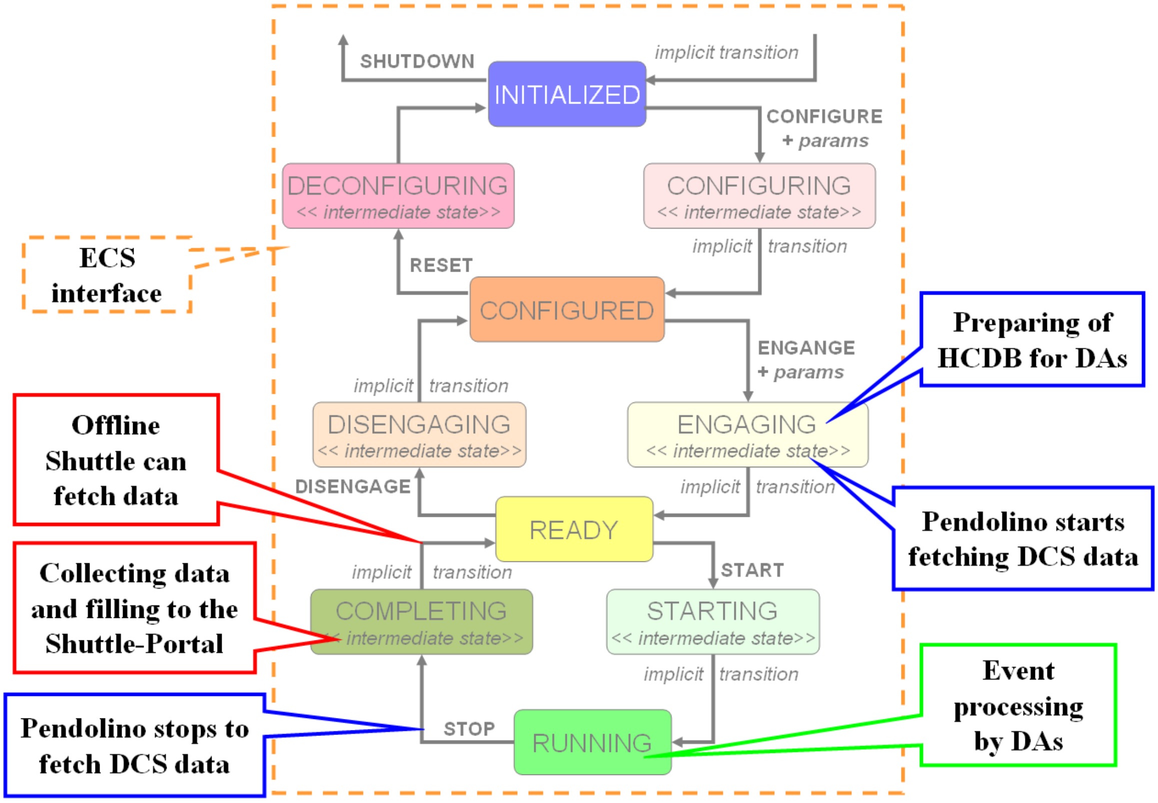


Figure 5.4: The figure sketches how the different states in the interface to the ECS synchronise the tasks of the other interfaces with the connected systems. With the ENGAGE command, the HCDB gets prepared and the Pendolino begins to request the DCS Archive DB. During the **RUNNING** state, the DAs process event data and produce new calibration objects. In the **COMPLETING** state the Pendolino stops and the new calibration data are shipped to the Shuttle-Portal. When these tasks are finished, the HLT returns to the **READY** state and the ECS can inform Offline to start its Shuttle.

applied to the corresponding FEEs. In addition, they use information about the detector alignment parameters. They are required to achieve an accurate reconstruction of the events. These calibration settings are essential to send correctly reconstructed results to the DAQ for storage and to AliEve for displaying them online.

Both parameters are fetched from the OCDB by the Taxi and cached to the T-HCDB. During a run, the HLT calibration framework provides them to the components via the HCDB. Their access is transparent to the DAs, independent of running in an online or offline environment.

Huffman data compression

Another example for the usage of the HLT calibration interfaces is the Huffman encoding[8] of event data by dedicated DAs inside the HLT cluster. In so called "training

[8] The Huffman coding is an entropy encoding algorithm for lossless data compression. In this compression method the representations of the most common source symbols / blocks of data are

runs" the Huffman component(s)[9] learn about the structure of event data and create according tables for encoding. These tables are used as pattern for compression and decompression of event data. After the training run they are collected at the Shuttle-Portal and requested by the Offline Shuttle. A dedicated HLT Preprocessor prepares the tables and envelopes them in ROOT objects. The resulting file is stored to the OCDB. The next time the Taxi requests the OCDB for updated entries, the Huffman calibration object is fetched and cached to the T-HCDB. At the start of the next run, the HCDB gets prepared with the T-HCDB entries, thereby distributing the Huffman calibration object to the computing nodes. Huffman components access the object from the HCDB and extract the included Huffman table for its detector type. During the run these tables are used for online encoding of received event data. The described chain is presented in figure 5.5 [77].

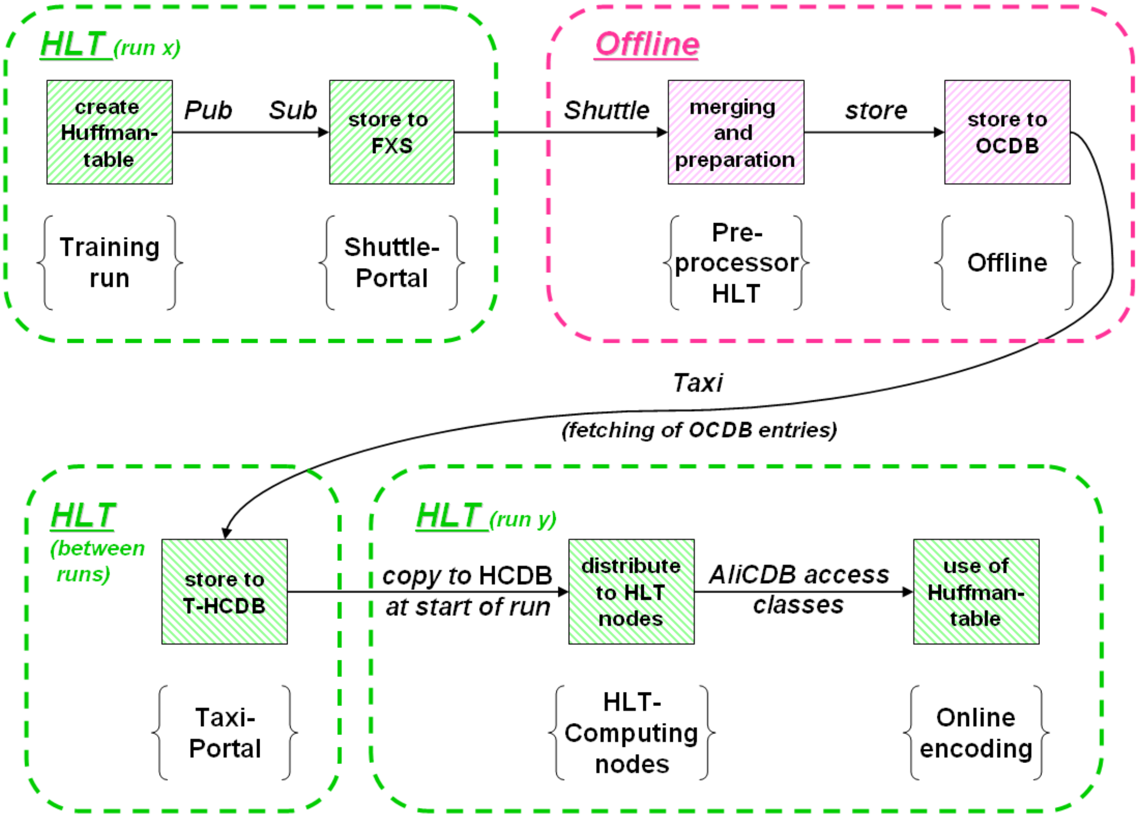


Figure 5.5: The sketch shows the Huffman table use case of the two HLT-Offline interfaces. Huffman tables produced by dedicated DAs inside the HLT cluster are shipped to the OCDB after so called *"training runs"*. Before the next *physics runs*, the tables are retrieved from the OCDB and put to the HCDB, where they can be accessed by corresponding data compression components. They use them for Huffman encoding of event data.

assigned to shorter strings of bits than the ones of less common source symbols / blocks of data. The coding table is represented in a so called Huffman tree [81].

[9] The Huffman components have been developed by Jennifer Wagner (KIP – University of Heidelberg, Germany).

B-field calibration

In the reconstruction of events, tracking components combine clusters into particle tracks. Due to the B-field of the L3- and dipole magnet, the tracks of charged particles are bent. In order to calculate the track curvature correctly, these components need information about the B-fields. The corresponding values can be requested from the DCS via the Pendolino. They are either given by direct B-field measurements from the according sensors or can be calculated in the PredictionProcessors by the retrieved values for the magnet currents and polarities.

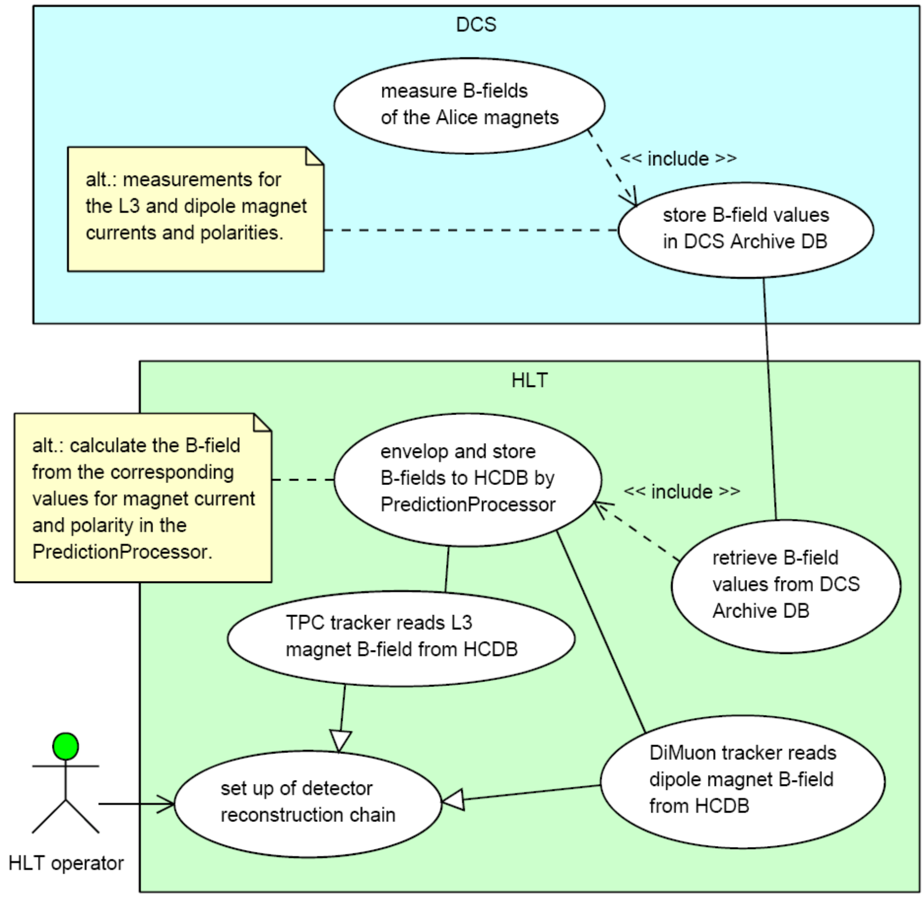

DCS
measure B-fields of the Alice magnets
<< include >>
alt.: measurements for the L3 and dipole magnet currents and polarities.
store B-field values in DCS Archive DB
HLT
alt.: calculate the B-field from the corresponding values for magnet current and polarity in the PredictionProcessor.
envelop and store B-fields to HCDB by PredictionProcessor
<< include >>
retrieve B-field values from DCS Archive DB
TPC tracker reads L3 magnet B-field from HCDB
DiMuon tracker reads dipole magnet B-field from HCDB
set up of detector reconstruction chain
HLT operator

Figure 5.6: The figure presents the UML Use Case diagram for the B-field example. The B-field measurements by the DCS are fetched via the Pendolino and inserted to the HCDB. Tracking components retrieve these values from the HCDB during the run. Alternatively to direct measurements, the B-field can be calculated by retrieved values for the magnet currents and polarities.

Since the B-field is relevant to all tracking detectors, the preparation and enveloping of the data is handled by the HLT or GRP PredictionProcessor[10]. After storage to the

[10] At the moment the preparation is performed by the HLT PredictionProcessor, but there are strong arguments to move this part to the GRP PredictionProcessor. Only the GRP PredictionProcessor can store the results into the GRP folder of the HCDB.

HCDB, the B-field values can be accessed by the corresponding tracking components. The use case of the B-field retrieval is visualised in figure 5.6.

5.4.2 TPC procedures

Temperature histogram

The interfaces of the HLT calibration framework can also be used for creating and displaying temperature histograms of the ALICE detectors, especially for temperature distributions of the TPC. Therefore DCS temperature measurements are fetched from the DCS Archive DB via the Pendolino and stored to the HCDB. A dedicated DA retrieves these values from the corresponding HCDB entry each time an "HCDB-updated" notification has been triggered. New temperature values are filled to the according histograms. These histograms can be sent to AliEve via the HOMER interface for online monitoring by ALICE operators during the run. Using these histograms, a temperature map showing the temperature distribution over the ALICE barrel can be created and displayed to operators in the ACR as well.

At the end of a run, the histograms are shipped to the Shuttle-Portal, where the Offline Shuttle collects them afterwards. After preparation by a dedicated Preprocessor, these histograms are stored to the GRID. The UML Use Case Diagram displaying this example is given in figure 5.7.

Drift velocity

Knowledge about the drift velocity is important to calculate correctly the time electrons need to drift towards the end plates of the TPC. The drift time gives the sensitive window of the TPC. Several procedures allow for calculating the drift velocity, most of them can be performed online. Therefore the according algorithms require input from various other systems in ALICE. For the calculation using a temperature map, the according calibration object has to be taken from the HCDB (fetched from the OCDB by the Taxi and stored inside the HLT). It can also be calculated from the current temperature and pressure measurements. Both values can be requested online from the DCS using the Pendolino. In addition, the drift velocity can be measured and monitored by laser tracks. The drift velocity results can be provided to Offline via the Shuttle-Portal and to the DCS via the FED-Portal.

$\mathbf{E}\times\mathbf{B}$

$\mathbf{E}\times\mathbf{B}$ effects have influence on the drift path[11] [11]. These effects are parametrised and wrapped into a calibration entry of the OCDB. Normally, $\mathbf{E}\times\mathbf{B}$ corrections are created offline by analysing lasers tracks. But their calculations can also be done by online DAs. The results are send to the OCDB via the Shuttle-Portal afterwards.

[11] $\mathbf{E}\times\mathbf{B}$ effects result from non-uniformities in the electric- and magnetic fields at the end caps of the TPC, where the $\mathbf{E}$ and $\mathbf{B}$ vectors are not parallel.

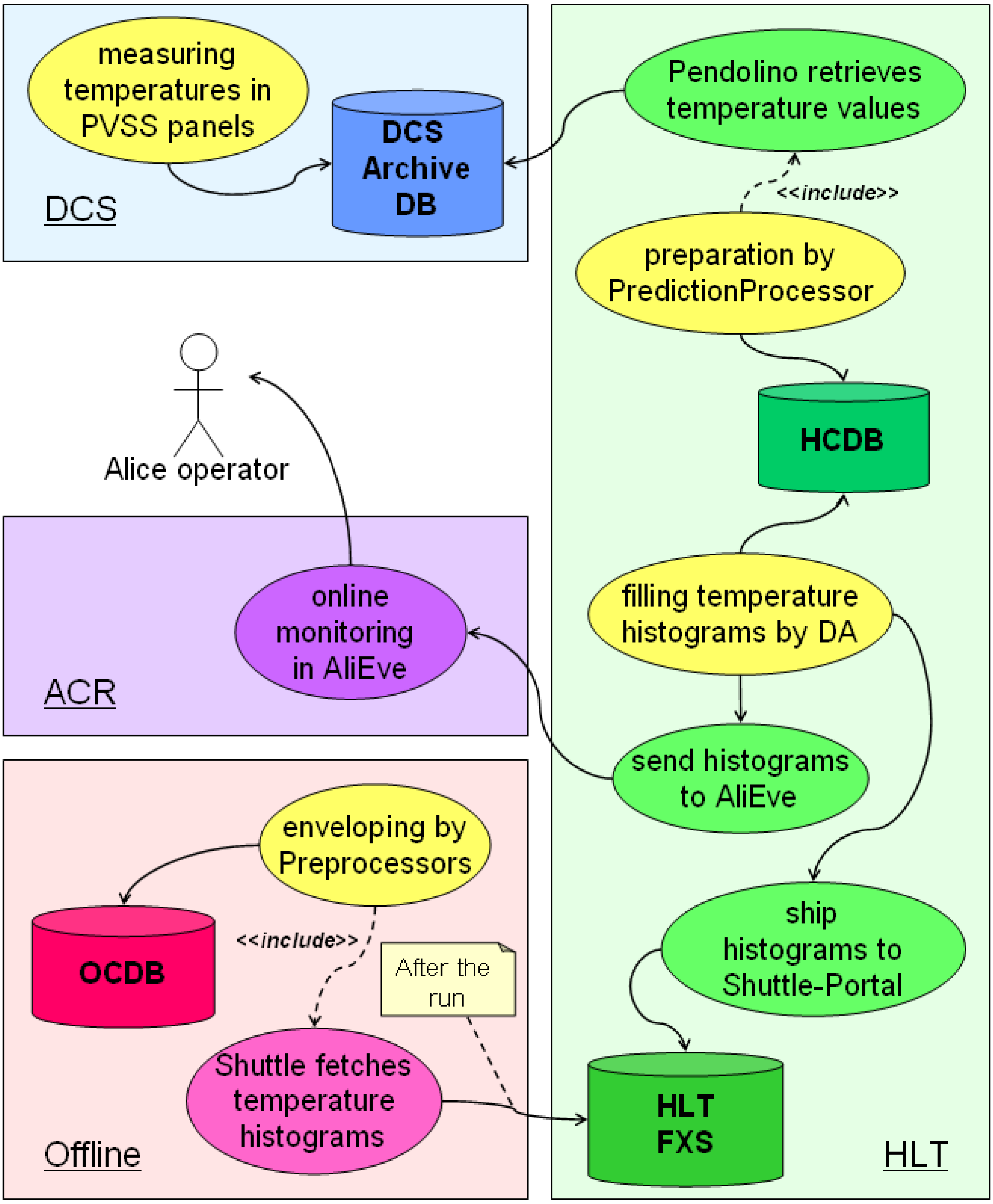

measuring temperatures in PVSS panels
DCS Archive DB
DCS
Pendolino retrieves temperature values
<<include>>
preparation by PredictionProcessor
HCDB
Alice operator
online monitoring in AliEve
ACR
filling temperature histograms by DA
send histograms to AliEve
ship histograms to Shuttle-Portal
enveloping by Preprocessors
OCDB
<<include>>
After the run
Shuttle fetches temperature histograms
Offline
HLT FXS
HLT

Figure 5.7: The UML Use Case Diagram depicts the usage of the HLT interfaces in temperature histogramming inside the HLT. Temperature values are acquired from the DCS via the Pendolino and stored to the HCDB. A dedicated DA retrieves these values from the HCDB and fills them into histograms. The histograms can be monitored online via AliEve in the ACR. Additionally, they can be shipped to the Shuttle-Portal, where the Offline Shuttle fetches them at the end of a run and includes them to the OCDB.

HLT analysis components can fetch these parameters from the HCDB to apply the corrections to the coordinates of the space points (clusters).

5.4.3 PHOS procedures

Gain equalisation

Establishing the response of the calorimeter to the deposition of a certain photon energy is essential to the physics performance. In PHOS the signal produced by the electronics is proportional to the amount of energy deposited. The gain factor for the signal also depends on the voltage applied to the APDs measuring the light produced in the crystals. This requires that the high voltage for each channel is adjusted to equalise the gains between the channels. However, the precision of that voltage equalisation does not suffice for the physics requirements. This means that during reconstruction each channel needs an individual gain factor. In addition the conditions may change between runs, requiring a re-calculation of these parameters for each single run. Only after this procedure is carried out it is possible to correctly establish the energy response of the calorimeter. A table of gain values is retrieved from the OCDB before a run and an updated table is stored after the HLT has processed the raw data. In this procedure the HLT Taxi and the Shuttle-Portal are involved.

Dead channel map

Another important task in the PHOS calibration process for each run is to investigate and establish which channels are useful for physics, and to create a so called bad channel map. Since a highly energetic particle entering PHOS will deposit energy in several crystals producing a cluster, measuring the total energy is highly dependent on the state of the channels involved in this cluster. The bad channel map is also retrieved from OCDB before a run and made available to PHOS components via the HCDB. After the run an updated version is stored back to the OCDB. This involves both Offline interfaces.

5.5 Benchmarks

During ALICE commissioning, certain benchmark tests have been performed for the Taxi and the Pendolino. The according interface applications were running under normal working conditions (other applications running on the corresponding portal nodes, additional network traffic on the involved ethernet connections). In these measurements, outliers with significantly longer request times have been observed from time to time. It is assumed that they are related to heavy network traffic or CPU load at the request moment on the involved devices.

For the other interfaces, no real sensible performance measurements can be made. The HLT-proxy exchanges only very limited amount of data with the ECS. The data are transferred via DIM, for which performance measurements are available on the web[12]. The same applies to the FED-Portal. Only a small amount of data have to be extracted from the received PubSub protocol, which are then shipped to the DCS via DIM. On the Shuttle-Portal, objects are fetched by the Offline Shuttle, which takes part in the interface, but it is not an HLT application. On the HLT side, only collecting of the according data, and therefore the PubSub performance, is relevant. The PubSub benchmarks are extensively discussed in: *"A Modular and Fault-Tolerant Data Transport Framework"* [42].

Taxi benchmark

The Taxi requests are not time critical, since they are performed asynchronously to the normal HLT operation cycle[13]. Nevertheless the tests have shown that the Taxi requests OCDB entries in reasonable time limits; even the average request time for 100 OCDB objects is below one minute (see benchmark graph of the Taxi in figure 5.8).

The request time raises almost linearly and the Taxi performance scales very well. Except for the single request, the average time spent in each entry retrieval is around half a second (cf. table 5.1).

# of OCDB entries	av. total time $[s]$	av. time per entry $[s]$
1	2.768	2.768
5	3.037	0.608
10	4.416	0.442
25	12.984	0.519
50	25.864	0.517
75	48.276	0.644
100	55.218	0.552

Table 5.1: The table displays the measured performance of the Taxi. The average total request time and the average request time per entry, spent for a given number of OCDB entries in the Taxi request, are presented.

During these benchmarks the T-HCDB had been cleaned completely before each request interval. Therefore, none of the requested objects had been locally available and all were transferred freshly from the OCDB. The measurements were taken from the moment of establishing first contact to the OCDB[14], until the last object had been fetched and stored to the T-HCDB. The real Taxi-Portal node[15] were used for the measurements.

[12] `http://dim.web.cern.ch/dim/DIM_Performance.pdf`

[13] The normal HLT operation cycle is: Initialise and configure HLT cluster and involved tasks, start run and process data, stop run and complete unfinished jobs, clean up and get ready for the next run.

[14] The time spent for acquisition of the AliEn token and the GRID proxy are not included in the measurement.

[15] The specs of this node (portal-taxi0) are: Linux 2.6.15-28-amd64-server #1 SMP Thu May 10 09:58:22 UTC 2007 x86_64 GNU/Linux (Ubuntu 4.0.3-1ubuntu5); dual board node with dual core

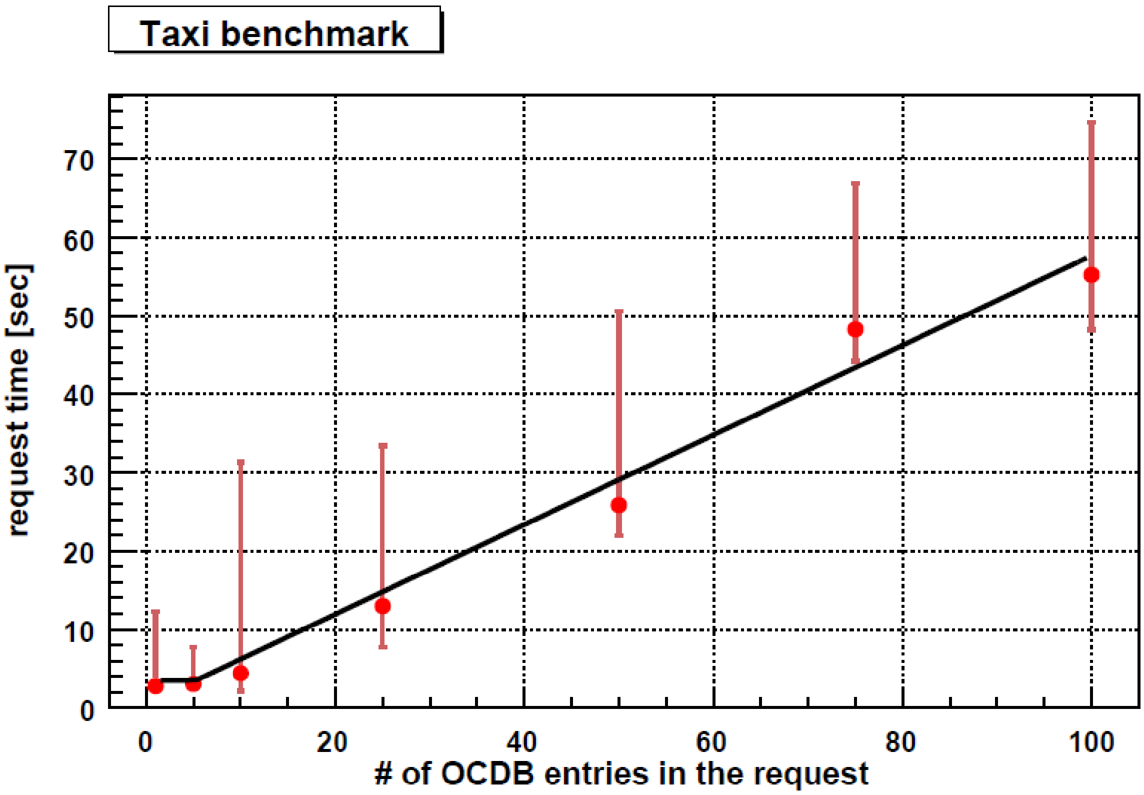


Figure 5.8: Benchmark graph of the Taxi. The circles display the average measured request time for a given number of OCDB entries. The vertical lines at each point depict the outliers, observed from time to time in the measurements. Especially outliers to much higher request times can be denoted, presumably due to heavy workload on the network or the GRID hosting the OCDB. They illustrate that a fast a access to the OCDB cannot always be guaranteed. The fit shows that after a short constant plateau (contact overhead overweights the retrieval time), the request time rises roughly proportional with increased numbers of fetched entries.

Annotation: The performance might look a little bit different, when real ALICE runs have started. A lot more calibration entries will be produced and put to the OCDB. Additionally, the storage procedure of event data to CASTOR files in AliEn will take a huge fraction of the GRID bandwidth.

Pendolino benchmark

The measured request time for the Pendolino stays relatively constant at about one second for the requests up to an amount of 75 demanded Aliases. In this section the request protocol overhead seems to be the dominant part. With larger amounts of Alias names, the performance scales linearly. Here, the retrieval of data from the DCS

CPUs (4 processing units on the node); CPU model name: Dual Core AMD Opteron(tm) Processor 265, CPU MHz: 1800.012, cache size: 1024 KB; total memory size: 4 GB, ethernet: 1 GBit/s .

Archive DB by AMANDA and the data transportation seem to take a heavier weight in the benchmarked time. But still, with an average query time of around 5 seconds for 500 Alias values, the speed of the Pendolino remains in an acceptable range. The average benchmark numbers are presented in table 5.2. These measurements have been taken on the real Pendolino-Portal node[16].

# of Aliases	av. total time $[s]$	av. time per Alias $[s]$
1	0.725	0.725
10	0.930	0.093
20	0.797	0.040
50	0.930	0.019
75	0.876	0.012
100	1.098	0.011
200	1.885	0.009
300	2.800	0.009
400	3.780	0.009
500	4.932	0.010

Table 5.2: The table displays the measured performance of the Pendolino. The average total request time and the average request time per Alias, spent for a given number of Alias names in the Pendolino request, are presented.

For the performance of the Pendolino the current workload of involved nodes and networks seems to have a even bigger impact. The size of the outliers have had a significant larger deviation from the measured average. Figure 5.9 shows the measured values for the Pendolino benchmarking. They were taken from the moment the DCS Client gets prepared for the query until the answer from AMANDA is received at the Pendolino[17].

To optimise the execution of the Pendolino, a prioritisation of the requested Aliases and their assignment to the three different Pendolinos introduced for the HLT can be done. Only a limited amount of relatively often changing and in high accuracy needed Alias values should be retrieved by the Pendolino with the highest frequency. The rest has to be distributed among the two others. Nevertheless, it is not expected that the Pendolino reaches critical performance regions. Only in the first request, when the SoR signal and the consecutive update of all values in the DCS Archive DB is covered, heavy workload is expected.

[16] The specs of this node (portal-dcs0) are: Linux 2.6.15-28-amd64-server #1 SMP Thu May 10 09:58:22 UTC 2007 x86_64 GNU/Linux (Ubuntu 4.0.3-1ubuntu5); dual board node with dual core CPUs (4 processing units on the node); CPU model name: Dual Core AMD Opteron(tm) Processor 265, CPU MHz: 1800.011, cache size: 1024 KB, total memory size: 4 GB, ethernet: 1 GBit/s .

[17] The preparations of these values by the PredictionProcessors are not included in the measured time period, because they might differ broadly depending on the tasks performed by the latter one.

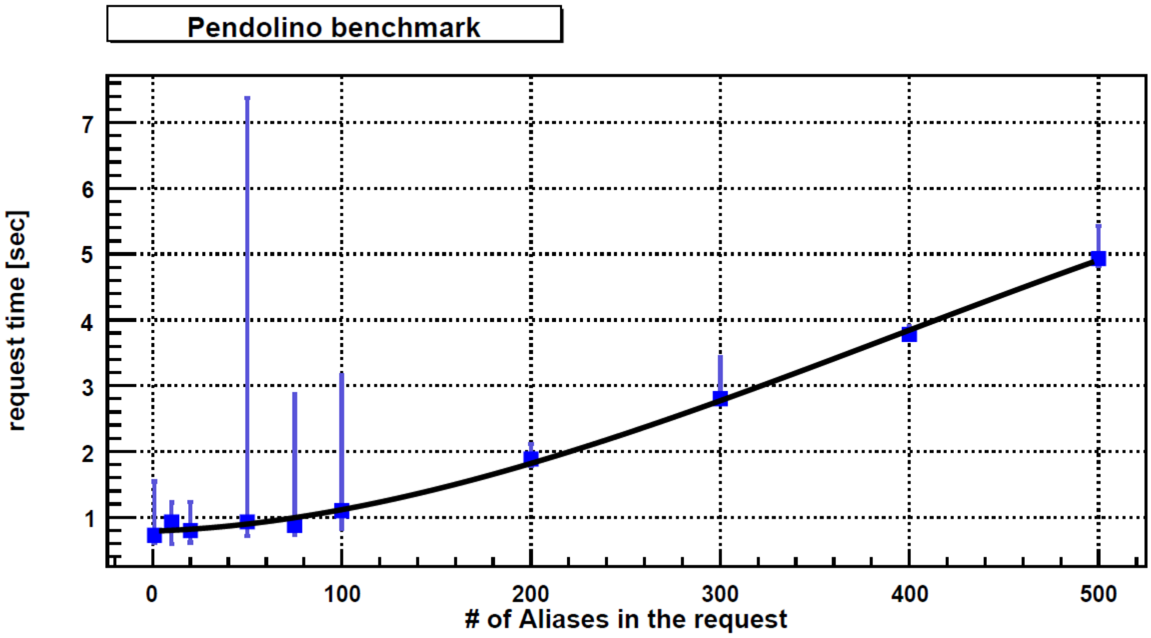


Figure 5.9: Benchmark graph of the Pendolino. The squares mark the average request time for a given number of Alias names, the blue vertical lines visualise the outliers in the measurements, especially to higher retrieval times. As seen for the numbers of 50 to 100 requested Aliases, the outliers can take three to seven times longer than the average. The fit shows a relatively stable request time for low amounts of Aliases in the request (75 and below), afterwards it seems to rise linearly.

Chapter 6

Summary and Outlook

The afore described interfaces constitute the HLT Calibration Framework. It is used to provide the analysis components inside the HLT cluster with current condition and calibration settings. In addition, produced calibration results are relayed to destined targets.

The ECS steers and controls the execution of the HLT. Moreover, general run parameters for each run are provided. The supervision is performed by states and state transition commands with additional parameters. Most recent calibration settings which are stored in the OCDB are fetched by the Taxi and cached locally to the T-HCDB. The official versioning and validity scheme is kept in order to be able to compare HLT results with offline analysis. The retrieval from the OCDB is employed continuously and run-independently.

When the dedicated SoR signal from the ECS is received, the Pendolino is started at the beginning of a run. First, it prepares the HCDB with links to a frozen version of the T-HCDB and distributes the corresponding FC to the computing nodes. The frozen version of the T-HCDB is required to have coherent calibration settings during the run. Then the Pendolino starts requesting the DCS Archive DB for current experiment conditions. The retrieved values are prepared and enveloped by detector-specific PredictionProcessors, before they are stored to the HCDB. The structure of the stored objects has to be similar to the one in the OCDB, in order to have analysis components running regardless of being in an online or offline world. The analysis components receive notifications about HCDB-updates. The HCDB poses as the central part for the input given by the calibration framework. All DAs can fetch their configuration- and calibration settings there. The HCDB is run specific and therefore cleaned up and freshly prepared before each run. Its structure matches the one introduced by Offline for the OCDB.

Specific, DCS relevant conditions, which are calculated online by DAs, are provided to the DCS via the FED-API on the FED-Portal. The FED-API is common among all detectors interfacing DCS and it is (re-)used for the HLT-DCS connection. The FED-Portal is one of two interfaces which allow for online monitoring of produced results. The other one is the HOMER-AliEve interface, which is used for displaying reconstructed events and monitoring detector performance during a run.

The actual event data are received as direct copies from the DAQ-LDCs. DAs analyse the data and reconstructed events. After making trigger decisions, selecting RoIs (data reduction) and compressing event data, the results are streamed out to the DAQ via the HLT-out nodes.

After a run, newly calculated calibration objects are collected at the Shuttle-Portal and offered to Offline. They are fetched by the Offline Shuttle and included as new entries into the OCDB. The Shuttle-Portal and the FED-Portal are the two interfaces providing calibration output to other ALICE systems. Overall synchronisation is achieved via the ECS. In addition, manual reconfiguration of HLT components during a run can be done by the HCDBManager script.

All these interfaces connect to the heterogeneous system environment of ALICE and integrate the HLT into them. The different protocols and techniques, which are involved in the external data exchange, are mapped to the mechanisms used HLT-internally. Constraints in the structure and versioning of calibration objects are to allow comparison of results with Offline analysis later-on. In addition, they enable analysis components running independently of being in an online or offline environment. The access to the calibration settings are kept transparent to the DAs.

Several examples for the usage of the HLT calibration framework have been explained. These examples reveal the diversity of the systems and the interaction inside the framework. Additionally, performance measurements for dedicated interface applications haven been presented.

An outlook for possible future upgrades of the calibration framework includes additional interfaces to the DAQ and the DCS. System tests during ALICE commissioning have shown that it can take a longer time period until data are fetched from the FXS of the different systems by the Offline Shuttle. Therefore it also takes longer time after a run until they are available in the OCDB. In order to have them included faster into the HLT, an interface application could be implemented requesting the FXS of the DAQ and the DCS directly. This would require a technique analogue to the Offline Shuttle mechanism, involving data preparation similar to the work performed by the Preprocessors. The drawback of this connection is that the new entries are no longer following the official versioning scheme of the OCDB entries. It would not be possible any more to correlate results to the used calibration settings in the comparison of online and offline results. It is also unclear, whether this would lead to a faster availability of these calibration settings, when the new interface requests a similar amount of data like the Offline Shuttle. Taking these arguments into account, it remains to be discussed if this is really an option in the extension of the calibration framework.

Another possible enhancement of the calibration framework is the introduction of an LDAP (Lightweight Directory Access Protocol) server to combine the interface application requests to general run parameters. The LDAP server could provide the run parameters given by the ECS to the requesting interfaces. Additionally, this would allow for an HLT-internal book-keeping of past run settings. This technique is similar to a mechanism used on the DAQ side with the *"daq_run_logbook"*.

More discussions on enhancements will show, whether they could be integrated to the HLT before ALICE really starts up - now, hopefully in spring 2009. But already with the current setup, the calibration framework provides the HLT with an environment, where all required calibration settings can be accessed by the analysis components and, vice versa, the HLT can provide new calibrations to the other systems in ALICE.

Appendix A

FED-API FeeCom commands

The following two sections show the commands for ***ConfigureFeeCom*** and ***ControlFeeCom*** channels of the FED-API defined in the FeeCom chain. The generic structure used in the description looks like this:

Command name | "Command ID" | <used opt. value>

The third section shows the encoding of the different log levels in the FeeCom chain.

A.1 ConfigureFeeCom commands

Defined commands for the ***ConfigureFeeCom*** channel:

- **Set FeeServer names:** Tells the InterComLayer to which FeeServers it shall subscribe. The additional integer value represents the tag by which the FeeServer names can be retrieved from the Configuration DB.
  ○ Only possible target is the InterComLayer.

  SET_FEESERVER_NAMES | "0x0001" | <integer tag>

- **Set FeeService names:** Tells the InterComLayer to which FeeServices it shall subscribe. The additional integer value represents the tag by which the FeeService names can be retrieved from the Configuration DB.
  ○ Only possible target is the InterComLayer.

  SET_FEESERVICE_NAMES | "0x0002" | <integer tag>

- **Set log level:** Sets a new log level at the specified target. The InterComLayer features two log levels: a remote and a local one. The local log level filters messages issued by the InterComLayer itself, the remote log level filters messages received from the FeeServers. To distinguish between both log level filters in the InterComLayer, "_local" or "_remote" is added to the "icl" as target

name. Only messages, which pass this filter are delivered further to the PVSS. All received log messages at the ICL stage are written to a dedicated log file independent of their delivery to PVSS. The possible log levels are described later in section A.3.
◦ Possible targets: FeeServers and InterComLayer.

SET_LOG_LEVEL | `"0x0003"` | <log level (integer)>

- **Get log level:** Requests the current log level of the specified target. To distinguish between local and remote log level filters of the ICL see description above.
◦ Possible targets: FeeServers and InterComLayer.

GET_LOG_LEVEL | `"0x0004"` | <no opt. value used>

- **Set update rate:** Sets the update rate for checking FeeService values against their dead band in a FeeServer. The update rate is set in milliseconds [ms]. The optional integer value contains the new update rate.
◦ Only FeeServers can be targets.

SET_UPDATE_RATE | `"0x0005"` | <update rate (integer)>

- **Get update rate:** Requests the update rate of a specified FeeService. The update rate is returned in milliseconds [ms].
◦ Only FeeServers can be targets.

GET_UPDATE_RATE | `"0x0006"` | <no opt. value used>

- **Set time out:** Sets the watchdog time out for issued commands in milliseconds [ms]. After this time out a FeeServer or CE is regarded as not responding.
◦ Possible targets: InterComLayer (timeout in the contact to the FeeServers), FeeServers (timeout in the contact to their CEs).

SET_TIME_OUT | `"0x0007"` | <time out (integer)>

- **Get time out:** Requests the current watchdog timeout for issued commands at the specified target. The result is given in milliseconds [ms].
◦ Possible targets: InterComLayer (timeout in the contact to FeeServers), FeeServers (timeout in the contact to their CEs).

GET_TIME_OUT | `"0x0008"` | <no opt. value used>

- **Set dead band:** Sets a new dead band for the specified FeeService. The new dead band is given as a float number. A service update to PVSS is only triggered at the FeeServer side, if the corresponding FeeService value exceeds the given dead band around its last transmitted value. The mechanism of its application is shown in figure A.1.
  ◦ Only possible targets are FeeServices; the InterComLayer has to determine the hosting FeeServer and issues the command to the corresponding FeeServer.

  SET_DEAD_BAND | `"0x0009"` | <dead band (float)>

- **Get dead band:** Requests the current dead band of the specified FeeService.
  ◦ Only possible targets are FeeServices, the InterComLayer has to determine the hosting FeeServer and issues the command to the corresponding FeeServer.

  GET_DEAD_BAND | `"0x000A"` | <no opt. value used>

A.2 ControlFeeCom commands

The ControlFeeCom commands do not return acknowledgements from the FeeServers. The only command using the optional tag is the command "Update FeeServer". In this case the tag determines the DCS Configuration DB entry for the binary of the new FeeServer version. Defined commands for the ***ConfigureFeeCom*** channel:

- **Update FeeServer:** Copies a new binary of a FeeServer to the DCS board(s) and restarts the new FeeServer(s) (uses the optional integer tag for retrieval of the binary from the DCS Configuration DB).

  FEESERVER_UPDATE_FLAG | `"0x0004"` | <integer tag>

- **Restart FeeServer:** Restarts the specified FeeServer(s).

  FEESERVER_RESTART_FLAG | `"0x0008"` | <no opt. value used>

- **Reboot DCS board:** Reboots the DCS board(s) of specified FeeServer(s).

  FEESERVER_REBOOT_FLAG | `"0x0010"` | <no opt. value used>

- **Shutdown DCS board:** Shuts down the corresponding DCS board(s).

  FEESERVER_SHUTDOWN_FLAG | `"0x0020"` | <no opt. value used>

- **Exit FeeServer:** Exits specified FeeServer(s).

FEESERVER_EXIT_FLAG | "0x0040" | <no opt. value used>

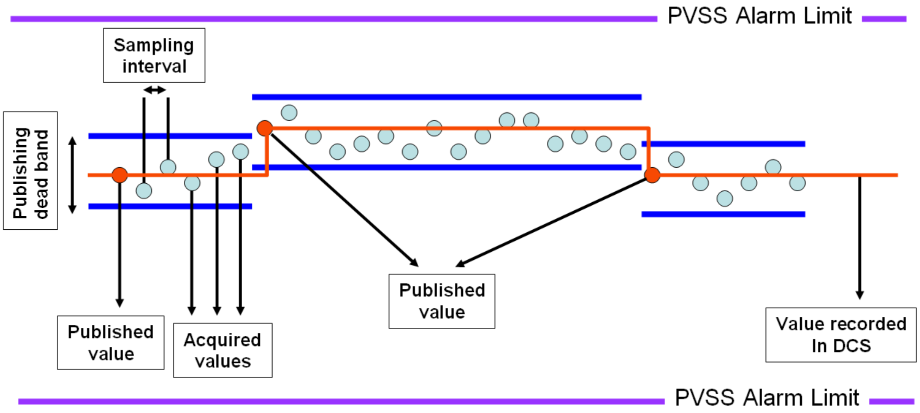


Figure A.1: Sketch of the dead band mechanism used in DCS-FEE. Only measurements, exceeding the given dead band around its last transmitted value, are relayed to the DCS.

A.3 Message channel log levels

The log messages in the FeeCom chain, as well as in the general ALICE DCS system, can have the following log types (severity of the message):

log type	*integer representation*
MSG_INFO	"0x01"
MSG_WARNING	"0x02"
MSG_ERROR	"0x04"
MSG_FAILURE_AUDIT	"0x08"
MSG_SUCCESS_AUDIT	"0x10"
MSG_DEBUG	"0x20"
MSG_ALARM	"0x40"

Appendix B

Usage of the HLT interfaces

B.1 Usage of the HLT-ECS interface

B.1.1 Start of the Logic Engine

The Logic Engine is normally started on the ECS side, but for local tests it is necessary to have it running inside the HLT.

```
start-logic-engine.sh [Domain_Name]

    Domain_Name: The domain name defines the partition of the HLT,
        that shall be steered by this logic engine. The parameter is
        optional, per default "ALICE_HLT" is used.
```

The starter script checks in regular time intervals, if the Logic Engine is still running and restarts it automatically, if it has stopped. A `'Ctrl-C'`(`SIGINT` signal) ends the logic engine and the start-up script.

B.1.2 Start of the HLT-proxy

The HLT-proxy is running on the HLT side. Its start-up script mainly passes its parameters further to the HLT-proxy binary. Certain default settings for the parameters are provided by the script and referred to the binary.
Usage of the start script:

```
start-hlt-proxy.sh [--domain <domain_name> --dimDnsNode <node_name>
                   --propertyFile <file_name> --logFile <file_name>
                   --debug --help] [-h]

    GENERAL REMARK: All parameters are optional and passed to
    'hlt_SM_proxy', except for '-h', which prints out this usage.
    The script has a default for the domain ('ALICE_HLT'), which
    is set, if absolutely no parameter is provided. As soon as one
    parameter is set, the '--domain <domain_name>' has to be set
    as well.

    --domain: gives the domain name for the HLT-proxy. If no
            parameter is given, 'ALICE_HLT' is assumed and added to
            the call of 'hlt_SM_proxy' (proxy binary).

    --dimDnsNode: defines the node, where the DIM_DNS is running.
            (Can be set also as environment variable via
            'DIM_DNS_NODE').

    --propertyFile: defines the name of the property file.
            (Can be set also as environment variable via
            'HLT_PROXY_PROPERTY_FILE').

    --logFile: defines the name of the to be used log file.
            (Can be set also as environment variable via
            'HLT_PROXY_LOGFILE_NAME').

    --debug: if provided, additional debug output is switched on.
            (This parameter must not have any value.)

    --help: prints out the help of 'hlt_SM_proxy'
            (other parameters are ignored).

    -h: prints out this usage, all other parameters are ignored.
```

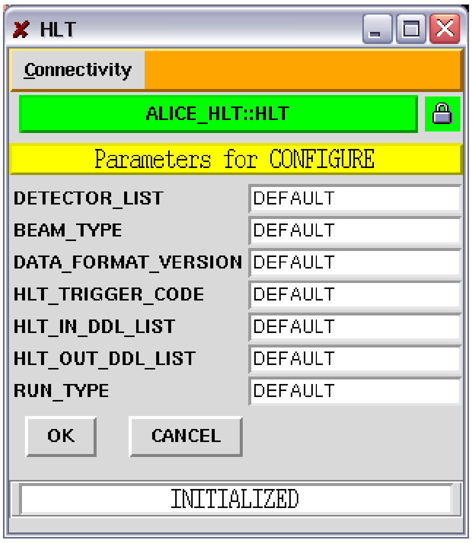
The HLT-proxy binary accepts the following parameters:

```
hlt_SM_proxy --domain <domain name> [--dimDnsNode <node name>
             --propertyFile <file name> --logFile <file name>
             --debug --help]

        --domain: necessary parameter, gives the domain name for the
                HLT-proxy (mandatory).

        --dimDnsNode: defines the node, where the DIM_DNS is running
                (Can be set also as environment variable via
                'DIM_DNS_NODE').

        --propertyFile: defines the name of the property file
                (Can be set also as environment variable via
                'HLT_PROXY_PROPERTY_FILE').

        --logFile: defines the name of the to be used log file
                (Can be set also as environment variable via
                'HLT_PROXY_LOGFILE_NAME').

        --debug: optional parameter, if provided, then additional
                debug output is switched on in the HLT-proxy.

        --help: prints out this help.
```

B.1.3 Start of ECS test-GUI

The ECS test-GUI allows to monitor the current state of the HLT and to send transition commands plus additional parameters, if required. It is started by a dedicated script for setting required environment variables. An example picture of the ECS test-GUI is shown in figure B.1. Its usage is as the following:

```
start-test-gui.sh [Domain_Name]

    Domain_Name: The domain name defines the partition of the HLT,
        that shall be monitored by the ECS test-GUI. The parameter
        is optional, per default "ALICE_HLT" is used.
```



DETECTOR_LIST	DEFAULT
BEAM_TYPE	DEFAULT
DATA_FORMAT_VERSION	DEFAULT
HLT_TRIGGER_CODE	DEFAULT
HLT_IN_DDL_LIST	DEFAULT
HLT_OUT_DDL_LIST	DEFAULT
RUN_TYPE	DEFAULT

INITIALIZED

Figure B.1: Screenshot of the ECS test-GUI, showing the HLT in INITIALIZED state and preparing a CONFIGURE command plus parameters to be sent.

Remember, that Logic Engine, HLT-proxy and ECS test-GUI require a DIM_DNS running in the same net. Its location has to be given by the environment variable

`DIM_DNS_NODE`. In order to have a correct setup running all three components have to use the same domain name. The default for the domain name in all three components is "ALICE_HLT", which refers to an ALICE global run.

B.2 Usage of the Shuttle-Portal

B.2.1 FXS-Subscriber parameters

The FXS-Subscriber, that handles the Shuttle-Portal, needs certain settings given as parameters on start up (This component has to be started on one of the two interface nodes to the CERN GPN [portal-shuttle0 or portal-shuttle1 on the HLT cluster]):

```
AliHLTFXSSubscriber -FXSBase <FXS base path> -DBName <data base name>
                    -DBHost <DB host name> -DBUser <DB user name>
                    -DBPasswd <DB password>

    -FXSBase <base path>:
        Base path of the File Exchange Server (most likely "/opt/FXS")

    -DBName <data base name>:
        Database name of Shuttle-Portal (most likely "hlt_logbook")

    -DBHost <DB host name>:
        Host name of the Shuttle-Portal's MySQL DB (use "localhost";
        the MySQL DB server has problems with the usage of
        "*.internal", which has been chosen as internal HLT cluster
        subnet name).

    -DBUser <DB user name>:
        User name to access the MySQL DB on the Shuttle-Portal node
        (use "hlt").

    -DBPasswd <DB password>:
        Password for the given user of MySQL DB on the Shuttle-Portal
        node (ask the Administrator).

  e.g.: AliHLTFXSSubscriber -FXSBase /opt/FXS -DBName hlt_logbook
                            -DBHost localhost -DBUser hlt -DBPasswd XX
```

B.2.2 XML example file for FXS-Subscriber configuration

In order to include the FXS-Subscriber in an analysis chain, it has to be included in the chain configuration. The chain configuration is given by an XML file, which is interpreted by the TMs. A sample XML file for the usage of the FXS-Subscriber is shown below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SimpleChainConfig2 ID="HLT" verbosity="0x78">
    <infoblock>
        <author> SB </author>
        <date> 09/17/08 </date>
        <description>
            XML example for the usage of the FXS-Subscriber in a chain
        </description>
    </infoblock>

    <ALICE>
        <Sources type="DDL">
            <!-- Here comes in the definition of the SOURCE -->
        </Sources>

        <FXS>  <!-- FXS-Portal component -->
            <Base> /opt/FXS </Base>
            <!-- base directory for FXS, mandatory -->

            <DB> hlt_logbook </DB>
            <!-- The name of the DB to use, mandatory -->

            <User> hlt </User>
            <!-- the username for accessing the DB, mandatory -->

            <Password> ***** </Password>
            <!-- the password for accessing the DB, mandatory -->

            <!--<Node> portal-shuttle0 </Node>-->
            <!-- An additional node on which an FXS should be run,
                 optional, can occur multiple times -->

            <!--<NodeExclude> portal-shuttle1 </NodeExclude>-->
            <!-- Exclude the given node, run no FXS on this node,
                 optional, can occur multiple times -->

            <Parent> TPC/DA4FXS </Parent>
            <!-- at least one is mandatory -->
        </FXS>

        <!-- Here comes in the Publisher options -->

        <TPC>
            <Component ID="DA4FXS">
                <ComponentID> TPCCalibComp </ComponentID>

                <Options>
                    <!-- Options of the calibration component -->
                </Options>

                <Shm blocksize="2M" blockcount="100"/>
                <Multiplicity> 1 </Multiplicity>

                <Library>
                    <!-- The Library containing the component -->
                </Library>
            </Component>
        </TPC>

    </ALICE>
</SimpleChainConfig2>
```

B.3 Usage of the Taxi

The Taxi application for fetching calibration objects from the OCDB is started by a script called "*TaxiDriver.sh*". It has the following usage:

```
Usage (Version: 2.1.1):
TaxiDriver.sh <AliEn_user> <OCDB_path> <THCDB_path> <Calibration_list>
              <PW_file> <Request_type> [Time_interval]

    AliEn_user: the name of the corresponding AliEn user
                (registered for the used GRID certificate).

    OCDB_path: path to the used OCDB file catalogue in AliEn (GRID)
                or to a local file catalogue storing entries to be
                fetched. Which type (AliEn-GRID or local) is defined
                by parameter '<Request_type>'.
                To take the path matching for the current LHC period
                use the environment variable 'LHC_PERIOD_OCDB_PATH'.

    THCDB_path: path to the local T-HCDB file catalogue. This should
                be set to the environment variable 'ALIHLT_T_HCDBDIR'.

    Calibration_list: file name or folder name containing the list(s)
                of to be fetched calibration objects. If a folder name
                is given, the lists from this folder are merged.
                Since version 2.0, the Taxi can handle the wildcard
                character '*' in the entries of the lists. The '*'
                can be written instead of a complete folder name. It
                can be used to fetch all entries for a detector
                (e.g. "TPC/*").
                NOTE: This applies for the list entries, not for the
                parameter 'Calibration_list'.

    PW_file: file containing the certificate passphrase (incl. path).

    Request_type: defines, if a AliEn-GRID request or local request
                shall be made. If AliEn-GRID, the parameter has to be
                set to 'alien', if local, the parameter has to be set
                to 'local'. For both types the path for the request
                is given by the parameter 'OCDB_path'.

    Time_interval: the time interval, after which the Taxi should
                repeat its task. The interval is given in minutes.
                (optional parameter: If it is left empty or is set
                to 0, then Taxi exits after a single turn -
                no automatic repetition.)

  e.g. TaxiDriver.sh bablok /alice/data/2008/LHC08d/OCDB /opt/T-HCDB
                     /opt/T-HCDB/lists/lists-taxi /path/xY alien
  or
       TaxiDriver.sh roehrich $LHC_PERIOD_OCDB_PATH /opt/T-HCDB
                    /opt/T-HCDB/lists/lists-taxi /path/xY alien 30

    NOTE: In order to have the Taxi monitored by SysMES, the log
          output of the Taxi has to be piped to the following file:
          ’/tmp/taxi.log’.
          Monitoring is only done, if the Taxi runs on ’portal-taxi0’
          or ’portal-taxi1’. Use the following command for proper log
          monitoring of the Taxi:

       TaxiDriver.sh roehrich /alice/data/2008/LHC08d/OCDB /opt/T-HCDB
                     /opt/T-HCDB/lists/lists-taxi /path/xY alien 30
                     >> /tmp/taxi.log
```

B.4 Usage of the Pendolino

The Pendolino application prepares the HCDB and fetches DCS values from the DCS Archive DB during a run. It is started by a script called "*StartPendolino.sh*", with the following usage:

```
Usage (v1.2.0):
StartPendolino.sh <Current_Run_Number> [Master_TaskManager_Address]
           [Num_of_Pendolinos] [Beam_Type] [Run_Type] [Detector_List]

    Current_Run_Number: The run number of the current run,
                (should be provided by the starting RunManager).

    Master_TaskManager_Address (optional): The address, where the
                RunManager can be contacted for notification about
                HCDB updates. When this parameter is not provided,
                the default is taken.
                (Default: tcpmsg://portal-ecs0.internal:20100)

    Num_of_Pendolinos (optional): The number of Pendolinos, that
                shall be started. Only a number of 1 to 3 Pendolinos
                are accepted; default is number 1.
                Note: If the number of Pendolinos is provided, the
                'Master_TaskManager_Address' has to be given as well.

    Beam_Type, Run_Type, Detector_List (optional): Initial run
                parameters given by ECS at start up. NOTE: When these
                parameters are given, 'Master_TaskManager_Address'
                and 'Num_of_Pendolinos' have to be provided as well.

  e.g. StartPendolino.sh 66 tcpmsg://portal-ecs0.internal:20100 2
           pp PHYSICS ALICE_ALL

  NOTE: In order to have the Pendolino starter log monitored by
        SysMES, the command line output has to be piped to:
        '/tmp/pendolino-starter.log'.
        (Monitoring only on 'portal-dcs0' and 'portal-dcs1'.)
        The actual Pendolinos store their log output in:
        '/tmp/pendolino-<type>.log', where <type> is either 'fast',
        'normal' or 'slow'. These files are also monitored by SysMES.

  e.g. StartPendolino.sh 8 tcpmsg://portal-ecs0.internal:20100 3
           pp PHYSICS ALICE_ALL > /tmp/pendolino-starter.log 2>&1

  Additional remark: When starting the Pendolino from any operator,
        but the hlt-operator, dedicated operator log files for the
        actual Pendolinos are used:
        '/tmp/pendolino-<type>-test.log-<operatorName>'.
        In addition the real HCDB ('/opt/HCDB') is only used for the
        hlt-operator; other operators get a HCDB copy in their home
        folder: '/afsuser/<operator>/HCDB'. The environment variable
        'ALIHLT_HCDBDIR' normally defines the used path to the HCDB.
```

In order to have the Pendolino application automatically started by the HLT RunManager, a wrapper script around the actual starter script has to be called by the RunManager[1]:

```
RunManager-PendolinoWrapper.sh <Current_Run_Number>
                <Master_TaskManager_Address> <Num_of_Pendolinos>
                <Log_File_Name> [Beam_Type] [Run_Type] [Detector_List]

    Current_Run_Number: The run number of the current run,
                (has to be provided by the RunManager).

    Master_TaskManager_Address: The address, where the RunManager
                can be contacted for notification of HCDB updates.

    Num_of_Pendolinos: The number of Pendolinos, to be started.
                Only a number of 1 to 3 Pendolinos are accepted.

    Log_File_Name: The log file name, where the Pendolino starter
                script puts its log output. Normally it should be
                set to: '/tmp/pendolino-starter.log'.

    Beam_Type: The beam type, used for the initial GRP T-HCDB entries
                (this is given by the ECS).

    Run_Type: The current run type, used for the initial GRP T-HCDB
                entries (this is given by the ECS).

    Detector_List: List of participating detectors, used for initial
                GRP T-HCDB entries (this is given by the ECS).


  e.g. RunManager-PendolinoWrapper.sh 66
                tcpmsg://portal-ecs0.internal:20100 2
                /tmp/pendolino-starter.log  pp PHYSICS ALICE_ALL
```

[1] This wrapper script has been introduced to have proper logging of the Pendolino application, when started remotely by the RunManager.

B.5 Usage of the FED-Portal

The FED-Subscriber for publishing data over the FED-API needs certain settings given as parameters on start up. The component has to be started on a node connected to the DCS CR as well (*portal-dcs0* or *portal-dcs1*):

```
AliHLTFEDSubscriber -dimServiceName <name>
                    -channelType <SingleServiceChannelInt |
                                  SingleServiceChannelChar |
                                  SingleServiceChannelFloat |
                                  GroupedServiceChannel>
                    [-dimDnsNode <node name>]



    -dimServiceName <name>:
        Name of the Dim Service to create.

    -channelType <SingleServiceChannelInt | SingleServiceChannelChar |
                  SingleServiceChannelFloat | GroupedServiceChannel>:
        Determines the FED-API channel type. The channel structure
        is defined implicitly ([I:1;F:1;C] - Single Services;
        [I:1;F:1;C:256;C:256] for Grouped Services).

    -dimDnsNode <node name>:
        Specifies the node, where the DIM_DNS is running. If not
        provided as command line parameter, it has to be set as an
        environment variable. (semi-optional)


  e.g.: AliHLTFEDSubscriber -dimServiceName TPC-drift-velocity
                    -channelType SingleServiceChannelFloat
                    -dimDnsNode alidcs-hlt01.cern.ch
```

B.6 Usage of the HCDBManager script

The HCDBManager script is used to insert objects into the HCDB manually. This script can either be called before a run to enter an initial configuration or during a run to change current configurations. Most of the settings are given as command line parameters, but a few settings have to be given interactively during the execution:

```
Usage (v1.3.1):
InsertInHCDB.sh <[--string <Configuration_String>]
                 [--file <filename>]> <--name ObjectName>
                 <--range startRun endRun> <--detector DetectorName>
                 [--permanent] [--help]

    This command line tool allows to enter configuration entries in
    the HCDB. After inserting the entry into the HCDB, the script
    prompts to let the user specify, which component shall be notified
    about the new entry. The notification is performed by the Master-
    TaskManagers using a special software event. Targets can be a
    single component or a list of components separated by a blank ' '.
    If no component shall be specified, just hit 'RETURN' when asked
    for the component names. Wildcards are possible.
    In addition the partition / detector corresponding the current
    Master-TaskManager has to be specified; for a global run use
    'ALICE'.


    --string: Following parameter is a configuration string, that
            shall be converted to a configuration object. A string
            containing blanks has to be surrounded by ' ' .
            (e.g. --string 'HV=5 LV=3')

    --file: Following parameter is ROOT-filename (inclusive path),
            that shall be inserted to the HCDB.
            (e.g. --file /tmp/TrackerConf.root)
            NOTE: The name of the TObject inside the ROOT-file has to
            be the same like the 'ObjectName' provided by parameter
            '--name' -> the configuration object name will be the same
            like the name of the inserted TObject.

    --name: Following parameter is the name of the configuration
            object, that shall be used in the HCDB for this object.
            NOTE: The name will be prefixed by the following subfolder
            structure in the HCDB:
            HLT/Config<DetectorName>,
            in case of storage in temporary mode; else HLT is replaced
```

```
           by HLTPermanent, (see below).
           Accessable by:
           HLT/Config<DetectorName>/<ObjectName>.
           NOTE: The first part of the path will be either HLT for
           temporary storage in HCDB (deleted before (!) next run)
           or HLTPermanent for permanent storage inside the HCDB
           (see also '--permanent').

   --range: Following two parameters define the run range ('startRun'
           until 'endRun'), for which the configuration object shall
           be valid.
           (e.g. --range 13 678; use '-1' as 'endRun' for infinite
           validity)

   --detector: Following parameter defines the detector for which
           this configuration object is designed. The detector name
           is added to the subPath2 in the HCDB folder structure;
           see description of '--name' above. Use the detector
           abbreviations (TPC, TRD, ...).

   --permanent (optional): Defines, if configuration object shall be
           stored to the permanent section of the HCDB. When
           provided, the permanent storage will be used.
           Temporary storage path is:
           'HLT/Config<DetectorName>/<ObjectName>'
           Permanent storage path is:
           'HLTPermanent/Config<DetectorName>/<ObjectName>'

   --help: Prints out this usage.

 Either '--string <Configuration_String>' OR '--file <filename>' has
 to be provided together with '--name ObjectName'.

 NOTE (I): The order of the given parameters is important:
       First '--string' (or '--file'), then '--name' followed by
       '--range' and '--detector', as last (optionally) '--permanent'

 NOTE (II): The environment variable 'ALIHLT_HCDBDIR' has to be set
       in order to run the script. Current value is: '/opt/HCDB' .


e.g. InsertInHCDB.sh --string 'HV=5 LV=3' --name TrackerVoltageSet
                     --range 13 678 --detector MUON
```

```
e.g. InsertInHCDB.sh --file AliESDs.root --name esdTree --range 12 -1
                    --detector PHOS --permanent
```

After the script has inserted the calibration object into the HCDB, the user can type in according information about which component shall be notified and on which partition the corresponding Master-TaskManager is running. Both parameters are taken by the *TM_notifier* tool and used by the TM framework to create and percolate the according notification event through the analysis chain. This is only relevant in case a reconfiguration of (a) component(s) shall be issued.

```
Please specify the component(s) [separated by ' ' (blank space)],
that shall be notified about the new entry. If left blank, all
current components are notified:
```

$ *<insert component name>*

```
And now please enter the partition (detector) of the corresponding
Master-TaskManager; for global run partition enter 'ALICE',
(if left blank the default ('ALICE') is used):
```

$ *<insert partition name of Master TaskManager>*

Appendix C

ALICE HLT Collaboration

Kenneth Aamodt[1], Torsten Alt[2], Harald Appelshäuser[3], Sebastian R. Bablok[4], Bruce Becker[5], Stefan Böttger[2], Sukalyan Chattopadhyay[6], Corrado Cicalo[5], Jean Cleymans[7], Indranil Das[6], Gareth de Vaux[7], Øystein Djuvsland[4], Roger Fearick[7], Øystein Haaland[8], Håvard Helstrup[8], Kirstin F. Hetland[8], Per Thomas Hille[1], Kalliopi Kanaki[4], Sebastian Kalcher[2], Udo Kebschull[2], Camilo Lara[2], Dag Larsen[4], Volker Lindenstruth[2], Gunnar Lovhoiden[1], Davide Marras[5], Joakim Nystrand[4], Ralf Panse[2], Mateusz Ploskon[9], Matthias Richter[4], Dieter Röhrich[4], Sabyasachi Siddhanta[5], Bernhard Skaali[1], Kyrre Skjerdal[4], Timm M. Steinbeck[2], Artur Szostak[5], Jochen Thäder[2], Trine Tveter[1], Kjetil Ullaland[4], Zeblon Vilakazi[7], Boris Wagner[4], Pierre Zelnicek[2], Gaute Øvrebekk[4],

[1]Department of Physics, University of Oslo, Norway
[2]Kirchhoff Institute of Physics, Ruprecht-Karls-University Heidelberg, Germany
[3]Institute for Nuclear Physics, University of Frankfurt, Germany
[4]Department of Physics and Technology, University of Bergen, Norway
[5]I.N.F.N. Sezione di Cagliari, Cittadella Universitaria Cagliari, Italy
[6]Saha Institute of Nuclear Physics, Kolkata, India
[7]UCT-CERN, Department of Physics, University of Cape Town, South Africa
[8]Faculty of Engineering, Bergen University College, Norway
[9]Lawrence Berkeley National Laboratory, University of Berkley, USA

(42 physicists from 9 institutions)

Appendix D

Publications

As main contributor

- S. Bablok, E.S. Conner, G. Hartung, R. Keidel, C. Kofler, T. Krawutschke, V. Lindenstruth and D. Röhrich: *Front-End-Electronics Communication software for multiple detectors in the ALICE experiment*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment Volume 557, Issue 2, 15 February 2006

- S. Bablok, et al.: *ALICE HLT interfaces and data organisation*, Proc. of Computing in High Energy Physics Conf. 2006 (CHEP 2006) - Mumbai (India), 2006, ed S. Banerjee vol 1 (India: Macmillian India Ltd) pp 96-99, ISBN-10: 0-230-63016-2; ISBN-13: 978-0230-63016-1

- The ALICE Collaboration, K. Aamodt, et al.[1]: *The* ALICE *Experiment at the CERN LHC*, Journal of Instrumentation, 2008_JINST_3_S08002, 2008

- S. Bablok, et al.: *High Level Trigger Online Calibration framework in ALICE*, Proc. of Computing in High Energy Physics Conf. 2007 (CHEP 2007) - Victoria BC. (Canada), Journal of Physics: Conference Series (JPCS) vol 119, 022007, by IOP Publishing, 2008

- S. Bablok: *Communication Software for the ALICE Detector Control System at CERN*, VDM Verlag Saarbrücken (Germany), Mai 2008, ISBN: 3-639-01913-X, ISBN-13: 978-3639-01913-1

As collaborator

- M. Richter, J. Alme, et al.: *DCS Communication Software for the ALICE TPC Front-End-Electronics*, CERN Reports, 2005-011. (358-362)

[1] My contribution to this paper is located in the HLT chapter, mainly on section 6.3.5 *"HLT interfaces to other online systems and offline"*.

- J. Alme, M. Richter, et al.: *A distributed, heterogeneous control system for the ALICE TPC electronics*, International Conference Workshops on Parallel Processing 2005, ICPP 2005 Workshops, Oslo (Norway), 14-17 June 2005

- M. Richter, et al: *Communication software for the ALICE TPC front-end electronics*, Prepared for 11th Workshop on Electronics for LHC and Future Experiments (LECC 2005), Heidelberg (Germany), 12-16 September 2005 - Published in "Heidelberg 2005, Electronics for LHC and future experiments"

- M. Richter, J. Alme, et al.: *The control system for the front-end electronics of the ALICE time projection chamber*, Proc. of the Real Time Conference 2005, 14th IEEE-NPSS, Stockholm (Sweden), 4-10 June 2005 - published in IEEE Transactions on Nuclear Science, Volume 533 No 3 Part 1, June 2006, ISSN: 0018-9499

- The ALICE Collaboration, et al.: *ALICE Technical Design Report of the Computing*, CERN-LHCC-2005-018, ALICE TDR 012, Printed at CERN, June 2005, ISBN 92-9083-247-9

- T. Alt, H. Appelshäuser, et al.: *Benchmarks and implementation of the ALICE High Level Trigger*, IEEE Transactions on Nuclear Science, Volume 53, Issue 3, Part 1, June 2006

- The ALICE Collaboration, et al.: *ALICE: Physics Performance Report, Volume II*, Journal of Physics G: Nuclear and Particle Physics 32 (1295-2040), September 2006

- M. Richter, et al.: *High Level Trigger Applications for the ALICE Experiment*, IEEE Transactions on Nuclear Science, Volume: 55, Issue: 1, Part 1, February 2008

- The ALICE Collaboration, et al.: *ALICE Electromagnetic Calorimeter - Technical Design Report*, CERN-LHCC-2008-014, ALICE-TDR-014, Printed at CERN, September 2008, ISBN 978-92-9083-320-8

Additionally, a total number of 93 publications credited as member of the ALICE Collaboration and the ALICE TPC Collaboration are listed since 2005 (based on results at SPIRES-HEP search).

Bibliography

[1] The ALICE Collaboration, et al.: *ALICE*: Physics Performance Report, Volume II J. Phys. G: Nucl. Part. Phys. 32 (2006) 1295-2040, 2006

[2] S. Eidelman, et al.: Particle Physics Booklet 2004, extracted from the Review of Particle Physics, Physics Letters B592, 1 (2004), July 2004

[3] CERN faq LHC the guide, Communication Group, CERN-Brochure-2008-001-Eng, January 2008
`http://cdsweb.cern.ch/record/1092437/files/CERN-Brochure-2008-001-Eng.pdf`

[4] B. Povh, K. Rith, C. Scholz, F. Zetsche: Teilchen und Kerne (6. Auflage), Springer Verlag Berlin Heidelberg New York, ISBN: 3-540-21065-2, Januer 2004

[5] Particle Data Group: Particle Physics Booklet (2006), Institute of Physics Publishing, July 2006

[6] *ALICE* Technical Proposal for A Large Ion Collider Experiment at the CERN LHC, Printed at CERN, CERN-LHCC-95-71, ISBN: 92-9083-077-8, December 1995

[7] *ALICE* Technical Design Report of the Trigger, Data Acquisition, High-Level Trigger and Control System, ALICE TDR 010, Printed at CERN, CERN-LHCC-2003-062, ISBN: 92-9083-217-7, December 2003

[8] The ALICE Collaboration, K. Aamodt, et al., The *ALICE* Experiment at the CERN LHC, 2008_JINST_3_S08002, 2008

[9] L. Betev, P. Chochula: Definition of the ALICE Coordinate System and Basic Rules for Subdetector Components Numbering, ALICE internal note ALICE-INT-2003-038, EDMS: 406391; `https://edms.cern.ch/document/406391/2`

[10] *ALICE* ITS TDR - Technical Design Report of the Inner Tracking System (ITS), ALICE TDR 04, Printed at CERN, CERN-LHCC-99-12, ISBN: 92-9083-144-8, June 1999

[11] *ALICE* TPC TDR - Technical Design Report of the Time Projection Chamber (TPC), ALICE TDR 07, Printed at CERN, CERN-LHCC-2000-001, ISBN: 92-9083-155-3, January 2000

[12] *ALICE* TRD TDR - Technical Design Report of the Transition Radiation Detector (TRD), ALICE TDR 09, Printed at CERN, CERN-LHCC-2001-021, ISBN: 92-9083-184-7, October 2001

[13] *ALICE* TOF TDR - Technical Design Report of the Time-Of-Flight (TOF) detector, ALICE TDR 08, Printed at CERN, CERN-LHCC-2000-12, ISBN: 92-9083-159-6, February 2000

[14] *ALICE* PHOS TDR - Technical Design Report of the Photon Spectrometer, ALICE TDR 02, Printed at CERN, CERN-LHCC-99-4, ISBN: 92-9083-138-3, December 1999

[15] *ALICE* HMPID TDR - Technical Design Report of the High Momentum Particle Identification Detector (HMPID), ALICE TDR 01, Printed at CERN, CERN-LHCC-98-19, ISBN: 92-9083-134-0, August 1998

[16] *ALICE* DiMUON TDR - Technical Design Report of the Dimuon Forward Spectrometer, ALICE TDR 5, Printed at CERN, CERN-LHCC-99-22, ISBN: 92-9083-148-0, August 1999

[17] *ALICE* Addendum to the Technical Design Report of the Dimuon Forward Spectrometer, Addendum 1 to ALICE TDR 5, Printed at CERN, CERN-LHCC-2000-046, ISBN: 92-9083-173-1, December 2000

[18] *ALICE* Technical Design Report on Forward Detectors: FMD, T0 and V0, ALICE TDR 011, Printed at CERN, CERN-LHCC-2004-025, ISBN: 2-9083-229-0, September 2004

[19] *ALICE* PMD TDR - Technical Design Report of the Photon Multiplicity Detector, ALICE TDR 6, Printed at CERN, CERN-LHCC-99-32, ISBN: 92-9083-153-7, September 1999

[20] *ALICE* ZDC TDR - Technical Design Report of the Zero Degree Calorimeter, ALICE TDR 3, Printed at CERN, CERN-LHCC-99-5, ISBN: 92-9083-139-1, March 1999

[21] *ALICE* EMCal TDR - Technical Design Report of the Electromagnetic Calorimeter (EMCal), Printed at CERN, CERN-LHCC-2006-014, ISBN: 92-9083-270-3, April 2006

[22] ROOT homepage: `http://root.cern.ch`

[23] AliRoot homepage:
`http://aliceinfo.cern.ch/Offline/AliRoot/Manual.html`

[24] Microsoft Encarta 2005 about Monte Carlo simulations

[25] FLUKA homepage: `http://www.fluka.org/`

[26] GEANT homepage:
`http://wwwasd.web.cern.ch/wwwasd/geant/`

[27] Geant 4 homepage:
`http://geant4.web.cern.ch/geant4/`

[28] LCG homepage: `http://lcg.web.cern.ch/LCG/`

[29] MONARC homepage: `http://monarc.web.cern.ch/MONARC/`

[30] P. Buncic, A. J. Peters, P. Saiz: The AliEn system, status and perspectives, Proceedings of the CHEP 2003 - Conference for Computing in High-Energy and Nuclear Physics, La Jolla (California), pp MOAT004, March 2003

[31] P. Saiz, et al: AliEn - ALICE environment on the GRID, Nuclear Instruments and Methods A502: 437-440, 2003

[32] A. J. Peters, P. Buncic, P. Saiz: AliEnFS: A Linux file system for the AliEn grid services, Proceedings of the CHEP 2003 - Conference for Computing in High-Energy and Nuclear Physics, La Jolla (California), pp THAT005, March 2003

[33] The Offline Conditions DB framework:
`http://aliceinfo.cern.ch/Offline/Activities/ConditionDB.html`

[34] *ALICE* Technical Design Report of the Computing, ALICE TDR 012, Printed at CERN, CERN-LHCC-2005-018, ISBN: 92-9083-247-9, June 2005

[35] The ALICE Offline Manual:
`http://aliceinfo.cern.ch/export/download/OfflineDownload/OfflineBible.pdf`

[36] J. Alme, et al: Radiation-Tolerant, SRAM-FPGA Based Trigger and Readout Electronics for the ALICE Experiment, IEEE Transactions on Nuclear Science, Volume 55, Issue 1, Part 1, February 2008, Digital Object Identifier 10.1109/TNS.2007.910677

[37] M. Munkejord, A. Stangeland, et al: Busy Generation in a large Trigger Based Data Acquisition System, in Proceedings of the 4th FPGAworld Conference, Lund and Stockholm (Sweden), 11. - 13. September 2007, ISSN 1404-3041 ISRN MDH-MRTC-215/2007-1-SE

[38] M. Munkejord: Development of the ALICE Busy Box, Master Thesis at Department of Physics and Technology (University of Bergen), October 2007

[39] S. Chapeland, et al: Commissioning of the ALICE Data Acquisition system, Proceedings of the CHEP 2007 - International Conference on Computing for High Energy Physics, Victoria BC. (Canada), published in Journal of Physics: Conference Series (JPCS) vol 119, 022006, by IOP Publishing, 2008

[40] S. Bablok, et al: High Level Trigger Online Calibration framework in ALICE, Proceedings of the CHEP 2007 - International Conference on Computing for High Energy Physics, Victoria BC. (Canada), published in Journal of Physics: Conference Series (JPCS) vol 119, 022007, by IOP Publishing, 2008

[41] Lemon homepage:
http://lemon.web.cern.ch/lemon/index.shtml

[42] T. M. Steinbeck: A Modular and Fault-Tolerant Data Transport Framework (Dissertation), University of Heidelberg (Germany), April 2004,
http://www.ub.uni-heidelberg.de/archiv/4575

[43] T. M. Steinbeck, et al: A Control Software for the ALICE High Level Trigger, Proceedings of the Computing in High Energy Physics 2004 (CHEP04), Interlaken (Switzerland), 2004

[44] T. M. Steinbeck, et al: New experiences with the ALICE High Level Trigger Data Transport Framework, Proceedings of the Computing in High Energy Physics 2004 (CHEP04), Interlaken (Switzerland), 2004

[45] S. Bablok, et al: ALICE HLT interfaces and data organisation, Proceedings of the CHEP 2006 - International Conference on Computing for High Energy Physics, Mumbai (India), Macmillian India Ltd, ed S. Banerjee vol 1, pp 96-99, ISBN-10: 0230-63016-2; ISBN-13: 978-0230-63016-1, 2007

[46] J. Wagner, et al: Lossless Data Compression for ALICE HLT, ALICE internal note ALICE-INT-2008-020 v1, EDMS: 948159, August

2008; https://edms.cern.ch/file/948159/1/ALICE-INT-2008-020.pdf

[47] C. Kofler: Design and realisation of a communication software to configure and control distributed embedded devices in a large scale research project at CERN, Diploma Thesis, University of Applied Sciences, Worms, February 2005

[48] B. Schockert: Development of Command and Database interfaces for a distributed Control System in the context of a large scale research project at CERN, Diploma Thesis, University of Applied Sciences, Worms, June 2006

[49] S. Bablok: Development and implementation of a safe and efficient communication software in a heterogeneous system environment of a major research project, Diploma Thesis, University of Applied Science Worms, June 2004

[50] S. Bablok: Communication Software for the ALICE Detector Control System at CERN, VDM Verlag, Saarbrücken (Germany), Mai 2008, ISBN: 3-639-01913-X

[51] S. Bablok, E.S. Conner, et al: Front-End-Electronics Communication software for multiple detectors in the ALICE experiment, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment Volume 557, Issue 2, 15 February 2006

[52] J. Alme: Firmware Development and Integration for ALICE TPC and PHOS Front-end Electronics (Dissertation), University of Bergen (Norway), ISBN: 978-82-308-0658-6, October 2008

[53] J. Alme, M. Richter, et al: A distributed, heterogeneous control system for the ALICE TPC electronics, International Conference Workshops on Parallel Processing (ICPP) 2005 Workshops, 14-17 June 2005

[54] M. Richter, J. Alme, et al: The control system for the front-end electronics of the ALICE time projection chamber, Proceedings of Real Time Conference, 2005. 14th IEEE-NPSS, 4-10 June 2005; published in IEEE Transactions on Nuclear Science / Volume 533, No 3, Part 1 / June 2006 / ISSN 0018-9499

[55] M. Richter, J. Alme, et al: DCS Communication Software for the ALICE TPC Front-End-Electronics CERN Reports, 2005-011. (358-362)

[56] M. Richter et al: Communication software for the ALICE TPC front-end electronics, Prepared for 11th Workshop on Electronics for LHC and Future Experiments (LECC 2005), Heidelberg (Germany), 12-16 September 2005; Published in "Heidelberg 2005, Electronics for LHC and future experiments"

[57] C. Gaspar, J. Schwarz: A Highly Distributed Control System for a Large Scale Experiment, Presented at: 13th IFAC workshop on Distributed Computer Control Systems - DCCS 95, Toulouse (France), September 1995

[58] C. Gaspar, M. Dönszelmann, Ph. Charpentier: DIM, a Portable, Light Weight Package for Information Publishing, Data Transfer and Inter-process Communication, Presented at: International Conference on Computing in High Energy and Nuclear Physics 2000, Padova (Italy), February 2000

[59] Private communication with Dr. Peter Chochula (CERN, Geneva, Switzerland)

[60] RFC of the Internet Engineering Task Force about Multicast: `http://tools.ietf.org/html/rfc3170`

[61] S. Bablok, P. Chochula, et al: FedServer API for ALICE DCS, `http://alicedcs.web.cern.ch/AliceDCS/Documents/FedServerAPI.pdf`

[62] Private communication with Benjamin Schockert (Zentrum für Technologietransfer und Telekommunikation - ZTT, University of Applied Science Worms, Germany)

[63] M. Jeckle, et al: UML 2 glasklar, Carl Hanser Verlag, München (Germany), ISBN: 3-446-22575-7, 2004

[64] B. Franek, C. Gaspar: SMI++ object oriented framework for designing and implementing distributed control systems, Nuclear Science Symposium Conference Record, page 1831-1835 Vol. 3, 2004 IEEE, 16-22 Oct. 2004, E-ISBN: 0-7803-8701-5, ISSN: 1082-3654, ISBN: 0-7803-8700-7

[65] B. Franek, C. Gaspar: SMI++ - Object Oriented Framework for Designing Control Systems for HEP Experiments, Proceedings of the CHEP 97 - International Conference on Computing for High Energy Physics, Berlin (Germany), Apr 7-11 1997

[66] SMI++ homepage: `http://smi.web.cern.ch/smi/`

[67] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Addison-Wesley Professional Computing Series, Boston (USA), ISBN 0-201-63361-2, April 2005

[68] R. Divià, T. M. Steinbeck: Data format and specifications for the HLT-to-DAQ interface, ALICE internal note ALICE-INT-2007-015 v2, EDMS: 871995; `https://edms.cern.ch/file/871995/2/ALICE-INT-2007-015.pdf`

[69] C. Cheshkov, R. Divià, T. M. Steinbeck: Identification of DDL links in ALICE data, ALICE internal note ALICE-INT-2007-016 v3, EDMS: 871996; `https://edms.cern.ch/file/871996/3/ALICE-INT-2007-016.pdf`

[70] Private communication with Dr. Timm Morton Steinbeck (Kirchhoff Institute for Physics, University of Heidelberg, Germany)

[71] A. Colla, J. F. Grosse-Oetringhaus: The Shuttle Framework - A system for automatic readout and processing of conditions data, ALICE internal note ALICE-INT-2008-011 v1, EDMS: 924807, May 2008; `https://edms.cern.ch/file/924807/1/ALICE-INT-2008-011.pdf`

[72] MySQL++ homepage: `http://tangentsoft.net/mysql++/`

[73] MySQL Community Server pages: `http://dev.mysql.com/downloads/mysql/5.0.html`

[74] RFC of the Internet Engineering Task Force about MD5 checksum: `http://tools.ietf.org/html/rfc1321`

[75] L. Betev, P. Chochula: Naming and Numbering Convention for ALICE detector part identification - Generic Scheme, ALICE internal note ALICE-INT-2003-039, EDMS: 406393; `https://edms.cern.ch/document/406393/1`

[76] ROOT documentation on ROOT files: `ftp://root.cern.ch/root/doc/11InputOutput.pdf`

[77] J. Wagner: Data compression for the ALICE detector (Diploma thesis), University of Heidelberg (Germany), 2008,

[78] Private communication with Svetozar Kapusta (CERN, Geneva, Switzerland)

[79] S. Kapusta, et al: Data Flow in ALICE Detector Control System, Poster at the 11th International Vienna Conference on Instrumentation, Vienna (Austria), February 19Ű24, 2007

[80] ALICE DCS AMANDA pages: `http://alice-project-dcs-amandaserver.web.cern.ch/alice-project-dcs-amandaserver`

[81] D. A. Huffman, A Method for the Construction of Minimum-Redundancy Codes, Proceedings of the I.R.E., pp 1098-1102, September 1952

Glossary

ACK	ACKnowledge
ACORDE	ALICE COsmic Ray DEtector
ACR	Alice Control Room
ADMP2	AliDCS Message Protocol 2
AFS	Andrew File System
ALICE	A Large Ion Collider Experiment
AliEn	Alice Environment
AliEve	Alice Event Visualisation Environment
AliRoot	Alice Root
alt.	alternative(ly)
AMANDA	Alice MANager for DCS Archives
AMD	Advanced Micro Devices
APD	Avalanche Photo Diode
API	Application Programming Interface
ARM	Advanced RISC Machines
ATLAS	A Toroidal LHC ApparatuS
av.	average
BLOB	Binary Large OBject
BNL	Brookhaven National Laboratory
$C_2H_2F_4$	Tetrafluoroethane
C_6F_{14}	Perfluorohexane
CA	Control Agent
CASTOR	CERN Advanced STORage manager
CDB	Conditions DataBase
CDH	Common Data Header
CE	Control Engine
CERN	Conseil Europeen pour la Recherche Nucleaire (European Organisation for Nuclear Research)
cf.	confer (compare to)
CH_4	Methane
CHARM	Computer Health And Remote Management
CIA	Cluster Interface Agent
CINT	C INTerpreter
cm	centimeter
Cmd	Command

CMS	Compact Muon Solenoid
CO_2	Carbon DiOxid
CoCo	CommandCoder
Cool	Cooling
CPU	Central Processing Unit
CPV	Charged Particle Veto
CR	Counting Room
CsI	Caesium Iodide
CTP	Central Trigger Processor
CU	Control Unit
D-RORC	DAQ - ReadOut Receiver Card
DA	Detector Algorithm
DAQ	Data AcQuisition
DB	DataBase
DCS	Detector Control System
DDL	Detector Data Link
DDR-SDRAM	Double Data Rate Synchronous Dynamic Random Access Memory
DELPHI	DEtector with Lepton, Photon and Hadron Identification
DID	Distributed Information Display
DIM	Distributed Information Managment
DIU	Destination Interface Unit
DMA	Direct Memory Access
DNS	Domain Name Service
DP	DataPoint
DS	DAQ Services
DU	Device Unit
ECS	Experiment Control System
EDM	Event Distribution Manager
EMCal	ElectroMagnetic Calorimeter
EoD	End-of-Data
EoR	End-of-Run
EPICS	Experimental Physics and Industrial Control System
ESD	Event Summary Data
et al	et alii (and others)
etc.	et cetera (and so forth)
eV	electron Volt
EVGEN	EVent GENerator
FAIR	Facility for Antiprotons and Ions Research
FC	File Catalogue
FEC	Front End Card
FED...	Front End Device ... (alt. Fed)
FEE...	Front End Electronics ... (alt. Fee)
FeeCom	Fee Communication
FEP	Front End Processor

FERO	Front-End Read-Out
FES	File Exchange Server - old abbreviation used by Offline
FLUKA	FLUktuierende KAskade
FMD	Forward Multiplicity Detector
FORTRAN	FORmula TRANslating System
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FW	FrameWork
FXS	File EXchange Server
G3	GEANT3
G4	Geant4
GDC	Global Data Concentrator
GEANT	GEometry ANd Tracking - since version 4: Geant
GPN	General Purpose Network
GRP	General Run Parameter
GSI	Gesellschaft für SchwerIonenforschung
GUI	Graphical User Interface
H-RORC	HLT - ReadOut Receiver Card
HCDB	HLT Conditions DataBase
HEP	High Energy Physics
HLT	High Level Trigger
HMPID	High Momentum Particle Identification Detector
HOMER	HLT Online Monitoring Environment including ROOT
HV	High Voltage
Hz	Herz
i-C_4H_{10}	Isobutane
i.e.	id est (that is)
ICL	InterCommunication Layer
ID	IDentifier
InterComLayer	InterCommunication Layer
IP	Internet Protocol
IROC	Inner ReadOut Chamber
ITS	Inner Tracking System
kHz	kilo Herz
LCG	LHC Computing GRID
LDAP	Lightweight Directory Access Protocol
LDC	Local Data Concentrator
Lemon	LHC era monitoring
LEP	Large Electron Positron Collider
LFN	Logical File Names
LHC	Large Hadron Collider
LHCb	Large Hadron Collider beauty
LHCf	Large Hadron Collider forward
LTS	Long Term Support

LTU	Local Trigger Unit
LV	Low Voltage
m	meter
MA	Monitor Agent
MD5	Message-Digest algorithm 5
MonALISA	Monitoring Agents using a Large Integrated Service Architecture
MONARC	Models Of Networked Analysis at Regional Centres for LHC Experiments
MRPC	Multi-gap Resistive-Plate Chamber
MSG	MeSsaGe
MWPC	Multi Wire Proportional Chamber
N	Nitrogen
Ne	Neon
NFS	Network File System
OCCI	Oracle C++ Call Interface
OCDB	Offline Conditions DataBase
OMG	Object Management Group
OpenGL	Open Graphics Library
opt.	optional
OROC	Outer ReadOut Chamber
OS	Operating System
p	proton
Pb	Plumbum (lead)
PC	Personal Computer
PCI	Peripheral Component Interconnect
PDS	Permanent Data Storage
PFN	Physical File Names
PhD	Philosophiae Doctor
PHOS	PHOton Spectrometer
PLC	Programmable Logic Controller
PMD	Photon Multiplicity Detector
PubSub	Publish Subscriber
PVSS	ProzessVisualisierungs - und Steuerungs - System
QCD	Quantum-Chromo-Dynamics
QGP	Quark-Gluon-Plasma
RAID	Redundant Array of Inexpensive Disks
RAM	Random Access Memory
RCU	Readout Control Unit
RDB	Relational DataBase
RHIC	Relativistic Heavy Ion Collider
RICH	Ring Imaging CHerenkov
ROB	Read Out Board
RoI	Region-of-Interest
RTTI	Run Time Type Information

SCADA	Supervisory Control And Data Acquisition
SDD	Silicon Drift Detector
sec	second, alt. s
SF_6	Sulfur hexafluoride
SIS	SchwerIonenSynchroton
SIU	Source Interface Unit
SLAC	Stanford Linear Accelerator Center
SLC	Scientific Linux CERN
SM	Standard Model
SMI	State Management Interface
SML	State Manager Language
SoD	Start-of-Data
SoR	Start-of-Run
SPD	Silicon Pixel Detector
SPS	Super Proton Synchrotron
SQL	Structured Query Language
SRAM	Static Random Access Memory
SSD	Silicon Strip Detector
SuSy	Super-Symmetry
SysMES	System Management for Networked Embedded Systems and Clusters
T	Tesla
t	ton
T-HCDB	Taxi - HLT Conditions DataBase
T0	Time 0 Detector
TCP	Transmission Control Protocol
TDR	Technical Design Report
TDS	Transient Data Storage
TDSM	Transient Data Storage Mover
TM	TaskManager
TOF	Time - Of - Flight
TOTEM	TOTal Elastic and diffractive cross section Measurement
TPC	Time Projection Chamber
TRD	Transition Radiation Detector
TRG	TRiGger system
TTC	Timing Trigger and Control
UML	Unified Modelling Language
V	Volt
V0	Vertex 0 Detector
VME	Virtual Machine Environment
VMS	Virtual Memory System
VO	Virtual Organisation
Xe	Xeon
ZDC	Zero Degree Calorimeter