Czech Technical University in Prague

Faculty of Nuclear Sciences and Physical Engineering

Doctoral thesis

Analysis and proposal of the new architecture of the selected parts of the software support of the COMPASS experiment

Vladimír Jarý

Supervisor: Doc. Ing. Miroslav Virius, CSc.

Advisor: Ing. Tomáš Liška, PhD.

A thesis submitted to the Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (PhD)

in the branch of study Mathematical Engineering
of the study program Applications of Natural Sciences

Prague, August 2012

Statutory declaration

I hereby declare that I have elaborated this disseration with topic *Analysis and proposal of the new architecture of the selected parts of the software support of the COMPASS experiment* independently and used no other aids that those cited. In each individual case, I have clearly identified the source of the passages that are taken word by word or paraphrased from other works.

In Prague, 9th August 2012 ....

Vladimír Jarý

Název práce:
Analýza a návrh nové architektury vybraných částí softwarového zabezpečení fyzikálního experimentu COMPASS

Abstrakt: Tato práce se zabývá systémem pro sběr dat použitým v rámci fyzikálního experimentu COMPASS v laboratoři CERN. Nejprve je studován stávající databázový podsystém, který se během roku 2009 potýkal s problémy způsobenými nárůstem zátěže. Jako první jsou analyzovány příčiny problémů, poté je představena a implementována nová architektura, která používá replikaci, zálohování a dohled pro dosažení vysoké dostupnosti a spolehlivosti. Popsány a otestovány jsou některé pokročilejší databázové technologie včetně dělených tabulek nebo datových úložišť. Poté je vysvětlen proces implementace vzdáleného řízení a dohledu experimentu. Stávající systém pro sběr dat je částečně založen na dnes již zastaralých technologiích, proto započal vývoj nové architektury. V práci je představena analýza požadavků a návrh nového dohledového a řídicího systému pro hardwarovou platformu založenou na FPGA technologii. Podle návrhu by měl být systém nasazen v heterogenním síťovém prostředí. Návrh definuje základní role v systému, chování je popsáno pomocí stavových automatů. Dále jsou shrnuty výsledky prvních testů výkonu a stability. V závěru jsou zmíněny další vývojové kroky zahrnující portování na cílový hardware.

Klíčová slova: COMPASS, databáze, vysoká dostupnost, vzdálené řízení, sběr dat, úložiště

Title:
Analysis and proposal of the new architecture of the selected parts of the software support of the COMPASS experiment

Abstract: This work focuses on the data acquisition system of the COMPASS experiment at CERN. At first the database current subsystem that suffered from increased load during year 2009 is analysed. The reasons of problems are identified and new architecture that includes replication, backups, and monitoring for achieving the high availability and reliability is proposed and implemented. Several advanced database features including partitioned tables or storage engines are described and tested. Then, the process of implementation of the remote control and monitoring of the experiment is explained. As the existing data acquisition system is partly based on a deprecated technologies, development of a new architecture has started. We focus on requirements analysis and proposal of a control and monitoring software for the new hardware platform based on the FPGA technology. The software is to be deployed in a heterogenous network environment. According to the proposal, the system is built on the DIM communication library. Roles participating in the system are defined in the proposal, the behavior of actors is described by state machines. First results of performance and stability tests are summarized. Finally, the planned steps including porting on the target hardware are summarized.

Key words: COMPASS, database, high availability, remote control, data acquisition, storage

Contents

Introduction

Look at those inventors and at their resources
yet the stars have not deviated from their courses
look at all those people living quietly
no this isn't work nor even energy
this is adventure as on the high seas
locking oneself in one's laboratories
look at all those people living quietly
no this isn't work it's poetry
Vítězslav Nezval: Edison[1]

Today, computer systems play an integral role in the high energy physics experiments; they are used for simulations, for control and monitoring, and for data analysis. Modern experiments produce vast amount of data that cannot be processed online, therefore the data acquisition systems that gather and store data are needed.

Goal of the thesis

The dissertation thesis focuses on the data acquisition system of the COMPASS experiment at CERN. Recently, the system has experienced several scalability and performance issues caused by increases in the data rate. The purpose of the research part of the thesis was to analyze the existing data acquisition architecture of the COMPASS experiment and to identify the cause of performance issues.

The COMPASS is a high energy physics experiment with fixed target situated at the Super Proton Synchrotron at CERN built for the study of the gluon and quark structure and the spectroscopy of hadrons using high intensity muon and hadron beams, [1]. The data acquisition system of the experiment is based on custom frontend electronics and event building network based on the Gigabit Ethernet. From the software point of view, the data acquisition is powered by the DATE package [3].

We have detected that the most serious problems had been caused by database subsystem, therefore, in the research part, we have focused on the rational database management systems. Based on the investigation of the data acquisition system, the following three goals have been designed:

1. At first, the database subsystem of the experiment needed to be evaluated. The new system that would handle the increased data rates needed to be proposed, implemented, and managed. The migration to the new database architecture needed to be transparent to clients. Additionally, the new database architecture should fulfill the high availability and high reliability requirements.

[1]Translated by Ewald Oser

2. The data acquisition of the experiment is controlled and monitored from the control room situated directly in the hall with COMPASS spectrometer. As a consequence of increased beam intensity, the radiation approached safety limits in the control room, therefore the second goal of the thesis was to analyze possible methods of remote access and to implement the remote control and monitoring for the experiment.
3. Development of a new data acquisition hardware has started, therefore the third goal of the thesis was to perform requirements analysis and prepare a proposal of the control and monitoring software for this new hardware architecture.

Additionally, as the thesis describes newly implemented systems such as a database architecture or remote control room, we have been asked by members of the COMPASS collaboration to provide technical details about installation and configuration of these systems that should be used by data acquisition experts. Therefore, several sections of the thesis as indicated bellow contain these technical information.

Structure of the thesis

The thesis is divided into the following six chapters.

In the first chapter, the trigger and the data acquisition systems are briefly introduced. Two basic trigger systems (i.e. system with a periodic trigger and system with a physics trigger) are described. Then several key terms such as the dead time of the system and the trigger efficiency are explained.

The second chapter of this work focuses on the data acquisition of the COMPASS experiment. The experiment is introduced, then in the following section, the existing data acquisition system is analyzed in detail. At first the hardware of the system is described, then the trigger control system of the experiment is overviewed. Finally, the various modules of the data acquisition software DATE are presented. The problems of the current system are also analyzed in this chapter.

The third chapter explains several tools that have been used during upgrade of the data acquisition system. At first, the rational database management systems are introduced and the MySQL database software that implements an online database service of the experiment is presented. Then, the network communication based on the DIM library that we have used during development of the control and monitoring software is described. Finally, the Qt framework that has also been used during the development is introduced.

The fourth chapter covers the online database service of the experiment. At first, the original architecture and its problems is introduced. In the second part of the chapter, the proposal of the updated architecture is presented. In this section, we also compare different storage engines used by the MySQL server. Then, the migration process to the new database architecture is described in more details covering the operating system and database software installation, configuration, and the data migration and verification. These details should be used by data acquisition experts of the experiment in case a new database server needs to be added into the architecture. Then the building blocks of the highly available and reliable database systems including the database replication, the configuration process of the proxy server, the backup and monitoring system, are analyzed. Then, we present new database applications that we have developed and we also enumerate problems with the database that have been solved. Finally, we propose several possible improvements to the updated database architecture that should be used to increase reliability and performance of the database architecture.

The fifth chapter analyzes implementation of the remote control and monitoring of the experiment. At first, we compare several possible implementations of remote access. Then, the installation of the remote control room is described. During the installation, we have used the kickstart technology to achieve unattended installation of multiple workstations. The author of the thesis has lead a team of undergraduate students that participated in the installation process. Also, we present the applications that are used by members of the shift crew to control and to monitor the experiment and the data taking process.

Finally, in the sixth chapter, we introduce the control and monitoring software for the new data acquisition system. At first, the hardware based on the Field Programmable Gate Array technology is presented. Then, we analyze requirements on the control and monitoring software for this hardware platform. Based on these requirements, we propose software architecture. The proposal has been successfully implemented by undergraduate students supervised by an author of this thesis. Finally, we summarize the first results of performance and stability tests of the new software and present the following development steps.

Acknowledgement

The work on this thesis has been supported by the following grants of the Ministry of Education, Youth, and Sports of the Czech Republic: LA08015, SGS 10/094, and SGS 11/167.

I would like to thank to my supervisor, Mr. Miroslav Virius, and to my supervisor specialist, Mr. Tomáš Liška for their useful remarks and advice concerning this thesis. My sincere thanks belongs to Mr. Miroslav Finger who offered me an opportunity to participate in the Compass experiment at CERN. I also wish to express my gratitude towards Mr. Vladimir Frolov and Mr. Damien Neyret who kindly introduced me into a data acquisition system of the Compass experiment.

The work on this thesis would not be possible without continuous support of my family. My thanks belong especially to my uncle and my grandmother who provided me a home away from home during my studies. Last but not least, I would like to thank to my parents for their love. Thank you!

I would like to dedicate this thesis to my grandfather Vladimir who suddenly passed away in December 2011.

Chapter 1

Trigger and data acquisition systems

Today, modern scientific experiments often produce data in previously unseen quantities. Usually, it is not possible to analyze the data in the real time as they are produced, thus the data acquisition system is employed to prepare data for an offline analysis. This work focuses on the role of the data acquisition systems in the high energy physics, however, it can be generalized to other fields to some extent.

The data acquisition can be divided into several steps as seen on Figure 1.1. In the first step known as a *readout*, the analog signal coming from the detector channels is preprocessed and digitized in *frontend electronics*. High energy physics experiment typically consists of a large number of detectors and each of these detectors produces data in multiple channels. The data acquisition system needs to gather fragments of data from all the detector channels and assemble full events from these fragments; this step is known as an *event building*. In the data logging phase, the full events need to be deposited into a permanent storage where they wait for the offline analysis.

Besides these data related tasks, the data acquisition system also provides the control and the configuration facilities. In order to enable the control, the system also needs to support monitoring. Optionally, the data acquisition system may include tools for partial online analysis, event filtering, or data preprocession.

The experiments at the Large Hadron Collider particle accelerator at CERN are characterized by a high collision rates of up to 40 MHz. However, only very small fraction of these collisions can be processed, maximum acceptable rate is in order of $O(100)$ Hz. The limit is imposed by the computing power and also by a storage capacity. The majority of the collisions corresponds to physically non interesting events. Thus, the data acquisition system often cooperates with the *trigger system* that is used to select physically interesting (or reject non interesting) events in a high rate environment. The trigger system can be organized into several levels. On the lowest layer, the system has only few μs to perform the decision, thus only simple algorithm can be used and this layer is usually implemented in a hardware. On the higher layer, the selection can be performed by the software and more complex algorithms can be utilized.

For instance, the detectors of the ATLAS experiment at the Large Hadron Collider produce data in 10^7 channels at the collision rate of 40 MHz. The first level trigger is based on a hardware; it reduces the rate to 10^5 Hz and the second level trigger to 10^3 Hz. The third level trigger implemented by the offline computing farm reduces the rate down to 10^2 Hz. On the other hand, some experiments with a triggerless readout exist or are planned. At these experiments, the uninteresting events are excluded purely by a high level filtering software.

In the high energy physics, the signal is often hidden in the background, thus some effective algorithms that can identify the signal are required. Depending on the physics program and the setup of the experiment, the trigger decision can be based on the total energy deposited in calorimeters, muon or electron tracks, energy losses, and other quantities. The detectors are divided into multiple trigger regions. Some of these regions can be used for accepting the signal, the others for rejecting background. The resulting trigger signal is assembled by logical combination of signals from these subdetectors.

Often, the trigger system can generate the artificial (usually random) trigger signal. The artificial trigger signal is used for the performance studies of the data acquisition system and also for the noise measurements and calibration of the detectors. When the trigger system selects some interesting event, it notifies the data acquisition system that performs the readout of the detector channels. The trigger system can also be used to distribute event identification or reference time.

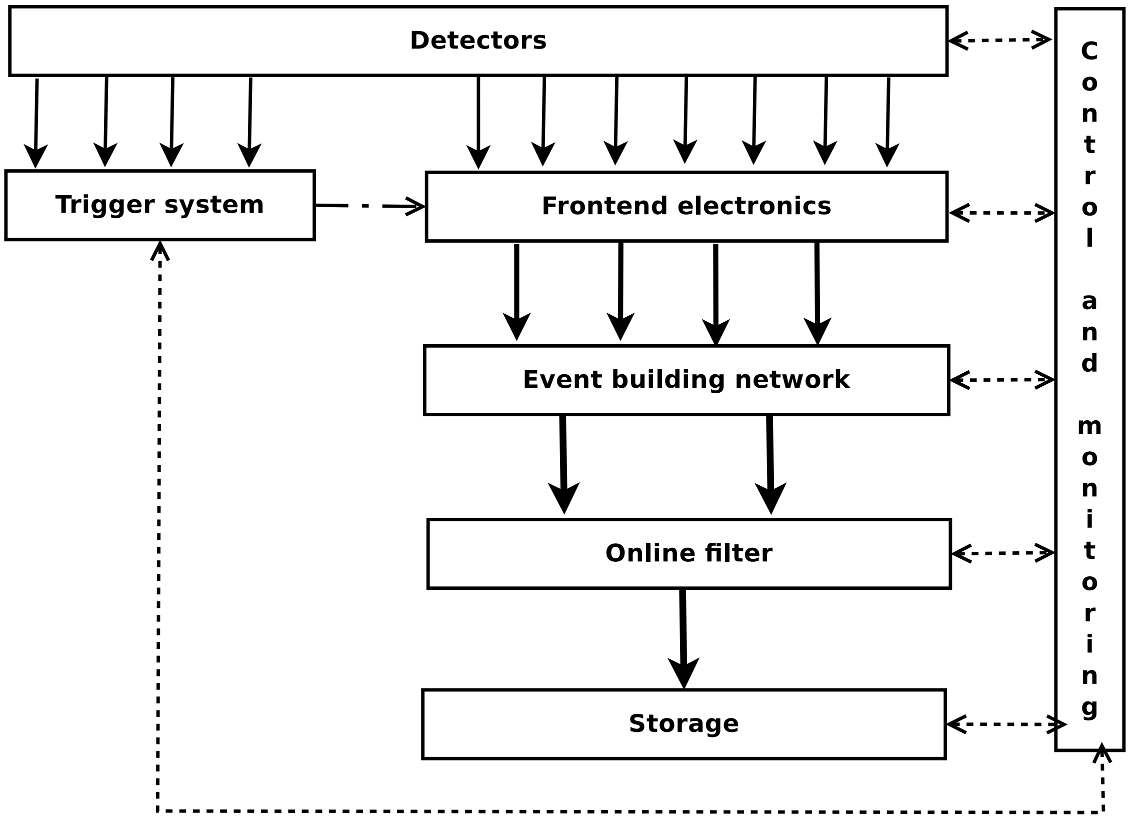


Figure 1.1: Data acquisition system

The system can be characterized by two important parameters: the dead time of the data acquisition system and the trigger efficiency. The dead time expresses the ratio between the time when the system is busy and cannot accept new triggers and total time. The dead time is caused by each processing step that takes a finite time interval to complete. The trigger efficiency represents the ratio between number of recorded good events and total number of produced good events. With increasing dead time, the number of rejected triggers increases and consequently, the efficiency of the trigger system decreases. Later, several common techniques such as a pipeline processing that are used to reduce the dead time (and improve the trigger efficiency) will be addressed.

At first, a very simple data acquisition system with the periodic trigger will be briefly described. It will be demonstrated that the trigger rate of the system is limited by a time required to process an event. Then, we will focus on an explanation of a more realistic system with a physics trigger.

1 The system with a periodic trigger

Imagine a small laboratory experiment that is used to measure and record the evolution of the room temperature in time. According to Figure 1.2, the system consists of a temperature sensor, an analog to digital (A/D) converter, a processing unit, a timer, a storage device, and communication links.

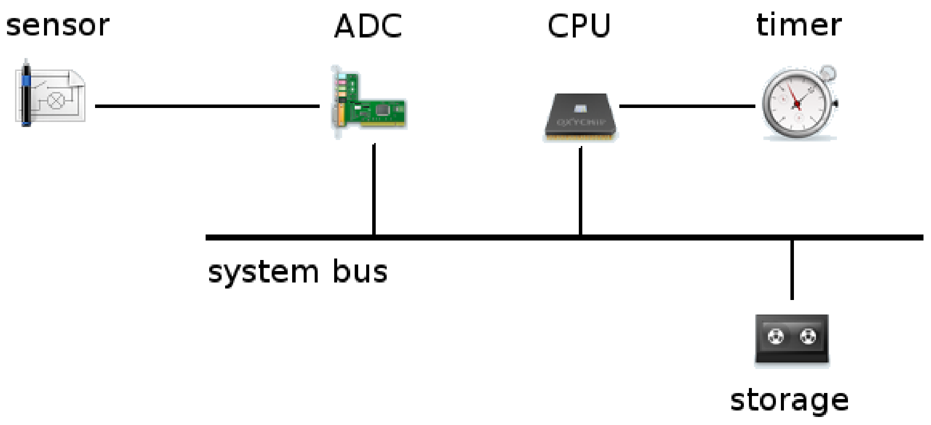


Figure 1.2: Data acquisition with periodic trigger

The timer periodically generates interrupts. When the central processing unit receives this interrupt, it executes the interrupt service routine (interrupt handler) in which the readout and digitization of the analog data from the sensor is performed. The digitized date are preprocessed and sent to the storage device. In this setup, the A/D converter corresponds to the frontend electronics layer.

The A/D converter spends a certain amount of time by converting one sample, the next time period is spent by the preprocessing and storage of data, thus it takes a finite time T to process one trigger. Consequently, this implies the upper limit on the trigger rate ν of the system. If the processing of the trigger takes 1 ms, the corresponding trigger rate $\nu < 1$ kHz.

2 The system with a physics trigger

In a physics experiment, the events come asynchronously and unpredictably. The detectors used in the high energy physics usually produce data in a large number of channels, for instance each wire in a wire chamber represents one channel, thus the probability that one particular channel is excited is relatively low and the time interval between two consecutive events is described by the Poisson probabilistic distribution. At first, a simple system with just one channel will be analyzed.

Signal from the detector frontend electronics is preamplified and converted to a voltage if necessary. This signal then serves as an input to the trigger system. In the simplest case, the trigger system compares the input signal to a threshold value and if the threshold is exceeded, it sends the start signal to the data acquisition system. Signals in the high energy physics have large variation in the amplitude, thus the threshold value should not be set too high. On the other hand, too low threshold would also trigger a noise. Pulse width contributes to the dead time and must be adapted to the desired trigger rate. The moment when the signal crosses the threshold value depends on the amplitude; larger pulses give shorter response time. This effect known as a time walk can be eliminated by using the *constant fraction discriminator* that allows triggering on a constant fraction of the peak amplitude.

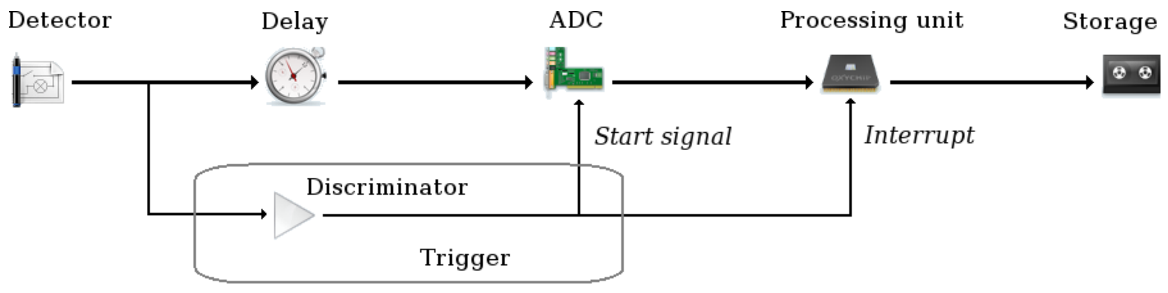


Figure 1.3: Data acquisition with a basic physics trigger

The input signal is split into two discrimination branches. In the first branch, the signal is compared against (adjustable) threshold in a normal threshold discriminator. The other branch implements the constant fraction discrimination. The signal is again split: the first copy is delayed by a delay cable, the second one is attenuated by a factor N. These two copies are subtracted and the result of subtraction is compared with a (almost) zero threshold. The two branches are finally merged by the AND gate which starts the data acquisition when the bipolar signal changes the polarity. The trigger system starts the operation of the A/D converter and sends the interrupt to the processing unit of the readout module. The readout module receives the data from the ADC, preprocesses them and sends them to the next processing stage (event building, data filtering) or to the storage device. The latency of the trigger is compensated by the delay cable between the frontend electronics and the readout module.

Each processing step that takes a finite time interval to complete contributes to the dead time of the system. Usually, there are three main sources of the dead time in the data acquisition systems: a readout dead time, a trigger dead time, and an operational dead time. The readout dead time represents a time period in which one event is fully read, the trigger dead time represents sum of trigger logic components processing times, and the operational dead time is caused by periods when the data acquisition is stopped between the consecutive runs. The readout module must be equipped with the busy logic that block producing the other triggers while the system performs the readout.

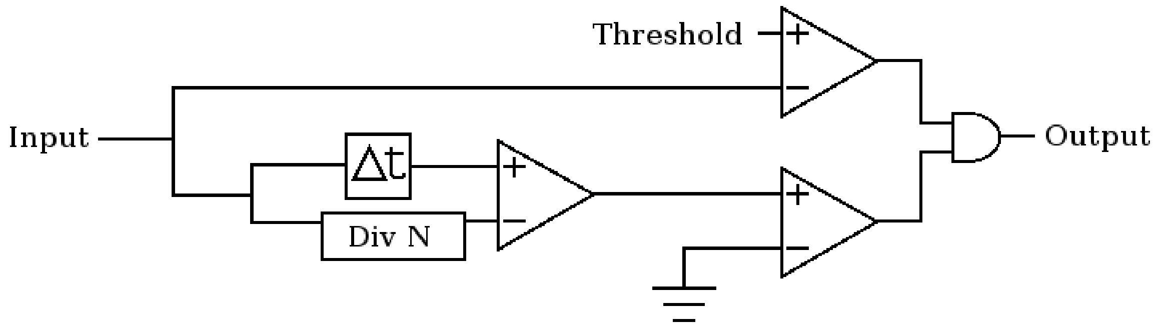


Figure 1.4: Constant fraction discriminator

Let τ denote readout time per event, ν denote the number of events read per second (DAQ rate), and ν_t denote the raw trigger rate. Thus the product $\nu \cdot \tau$ represents the fractional time when the DAQ system is busy and the $1 - \nu \cdot \tau$ is the live time of the system. From the expression of the

DAQ rate in a form $\nu = (1 - \nu \cdot \tau) \cdot \nu_t$ the limitation on the rate can be deduced: $\nu = \frac{\nu_t}{1+\tau \cdot \nu_t} < \nu_t$, This means that the DAQ rate is lower than the raw trigger rate and the efficiency $\epsilon = \frac{1}{1+\tau \cdot \nu_t}$ is always lower than 100 %.

The events are always lost if the $\nu_t > \frac{1}{\tau}$. In order to achieve high efficiency $\epsilon \sim 100\,\%$, the DAQ rate needs to approach the raw trigger rate $\nu \sim \nu_t$. This requires that the $\nu_t \cdot \tau \ll 1$ and the $\tau \ll \lambda$ where λ is the mean time between events. Suppose that the raw trigger rate is 1 kHz, i.e. $\lambda = 1\,\text{ms}$. To achieve the efficiency of 99 %, one needs to guarantee that the processing time $\tau < 0.01\,\text{ms}$ which means to over–design the system by a factor of 100.

There are two main techniques used to reduce the dead time in a high rate environment with large data flow, namely pipeline processing and parallelism. Pipeline processing is based on introducing fast memory buffers organized as a queue (FIFO, first in, first out) between processing stages. The buffers absorb fluctuations off the input rate and provide relatively stable output rate; this effect is known as a derandomization. Buffers decouple fast frontends from slow data processing and storage; this minimizes number of fast (and expensive) components. With buffering, it is possible to approach the 100 % efficiency if the A/D converter can operate at rate higher $\gg \nu_t$ and data processing and storage can work at rate comparable to the raw trigger rate ν_t.

Since the detectors consist of large number of channels it is natural to digitize, readout, and preprocess each channel in parallel, independently on other channels. At later stage, the data coming from different detector channels are assembled together to form the complete event.

Additional improvement can be achieved by implementing a pre–trigger which is a very fast first stage of the trigger system that signals the minimal activity in the detector. It sends the start signal to the digitizer units which is later confirmed or rejected by the main trigger. In this way, the main trigger can operate at later stage (after the digitization) and can be more complex.

A complex data acquisition system are usually distributed over a large number of nodes: servers, VME crates, and other equipment. Some of these nodes such as the A/D converters should be located close to the detectors, on the other hand, other nodes should be placed further away from the detectors in order to prevent the radiation damage. There are two main types of interconnection between the nodes: buses and networks.

Buses are relatively simple systems, they consist from fixed number of wires with given mechanical and electrical properties. The number of wires defines the bus width. Devices share the bus, thus some kind of arbitration is required to avoid collisions. The number of devices connected to the bus and the bus length is limited. Moreover, the bus bandwidth is shared among all the devices, thus the buses have scalability issues. Despite these issues, buses are often used, mainly for transferring data over shorter distances. The most commonly used types include VME, PCI, or SCSI buses.

All devices connected to the network are considered to be equal, they communicate directly with each other by sending messages. The devices follows a set of rules called protocol. In switched networks, special devices called switches transfer messages. They find the correct path between sources and destinations and they use buffers to handle the congestion of messages. In contrast to the buses, the networks scale well. Today, the Ethernet is probably the most used network technology.

Chapter 2

The Data Acquisition System of the COMPASS Experiment

This chapter focuses on the existing data acquisition system used by the COMPASS experiment at CERN. The scientific program of the COMPASS experiment is briefly introduced. Then, the data acquisition system of the experiment is described in more details. At first, the hardware architecture that is based on the custom electronics for readout and by the network based event building is presented. Next, the trigger system based on the Time and Trigger Control system is overviewed. Great attention is dedicated to the description of the DATE data acquisition software. Finally, the performance and stability problems of the system are discussed.

1 The COMPASS Experiment

The COMPASS[1] is a high energy physics experiment with a fixed target that operates on the Super Proton Synchrotron (SPS) particle accelerator at laboratory CERN in Geneva, Switzerland, [1]. The scientific program of the experiment was conditionally approved by the CERN in 1997. The detectors were installed in years 1999–2000, the system was commissioned during the technical run in 2001 and the physical data taking started in 2002. Today, about 250 scientists, engineers, and students from 28 institutes and 11 countries collaborate on the COMPASS experiment.

The goal of the experiment is to research the structure and the spectroscopy of hadrons. The scientific program consists of experiments with high energy muon and hadron beams provided by the SPS accelerator. The experiments with the muon beam include studies of transverse spin effects, vector meson production, or $\Delta G/G$ measurements. The program with the hadron (pion and proton) beams explores the spectroscopy of light mesons, pion and kaon polarisability, search of glueballs, or production of double charm baryons. Currently, the experiment enters its second phase known as the COMPASS-II. The program of this phase covers research of generalized parton distributions, Primakoff scattering, or Drell-Yan effect, [2].

The COMPASS spectrometer operates on the SPS particle accelerator that provides a primary beam. The accelerator works in 16.8 s long cycles; 12 s of this period is dedicated to the acceleration, the remaining 4.8 s to the extraction of particles. The extraction period is also known as a *spill*. A typical SPS spill contains approximately 1.2×10^{13} of protons with a momentum of 400 GeV/c. From the primary beam, the secondary muon and hadron beams are extracted. It is also possible to extract the tertiary low energy electron beam that is used for the calibration

[1]COMPASS is an acronym that stands for the Common Muon and Proton Apparatus for Structure and Spectroscopy

of electromagnetic calorimeters.

The primary beam hits the Beryllium target and the pions are produced. The set of acceptance magnets selects pions of momentum around 225 GeV/c which are then transferred along 600 m long beam line. During the transfer, part of the pions decays into muons and neutrinos. By absorbing the remaining hadrons and steering the muons by focusing and defocusing magnets, the secondary muon beam is produced. The radioprotection limits the maximum flux to 2×10^8 muons per SPS cycle, the momentum of muons can be adjusted between 60 and 190 GeV/c. The secondary hadron beam is produced by removing the hadron absorbers from the beam line. For momenta up to 225 GeV/c, the same settings for the acceptance magnets is used; different settings must be loaded for hadron beams with a higher momentum. The radioprotection gives the limitation on the flux of 10^8 hadrons per SPS cycle. It is possible to switch between different beam line configuration remotely, using the computer program (see Chapter 5 of this work).

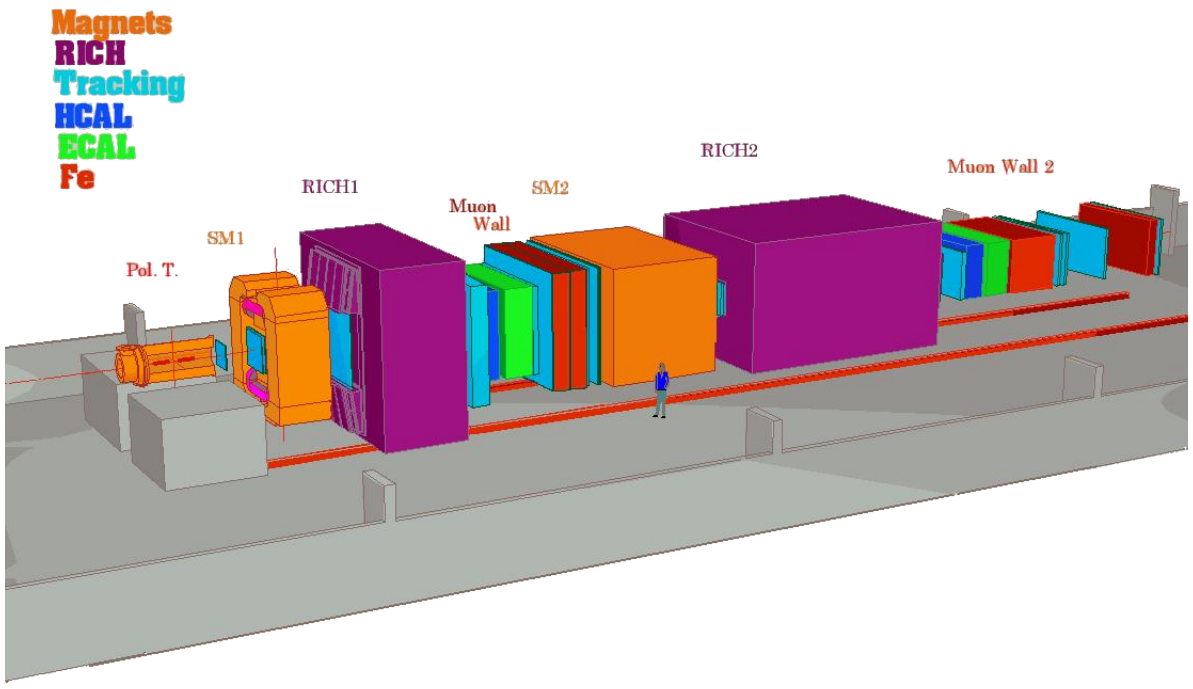


Figure 2.1: Artistic view of the COMPASS spectrometer, image taken from [30]

Depending on the physical program and the type of the secondary beam, different production targets are employed. For the experiments with muon beam, the polarized ^{6}LiD target is used, for experiments with hadron beam, solid state targets of various thickness and materials are used. When a beam particles impinge on the target, the secondary particles are produced. These secondary particles are then registered in a series of detectors that form the COMPASS spectrometer.

The COMPASS spectrometer consists of detectors that are used to identify particles, track particles, and measure energies of particles. Experiments with muon and hadron beams require different layout of detectors and target platform, thus the components are mounted on rails which enables easy manipulation. The detectors are grouped into three main parts: beam spectrometer that is located upstream (i.e. before) of the polarized target and small and large angular spectrometers that are situated downstream (i.e. after) of the target.

The main task of the beam spectrometer is to measure the momentum and position of the beam particles. Additionally, part of the beam spectrometer contributes to the veto signal and separates the beam from the halo. The large angular spectrometer is designed to detect particles at large angles (±180 mrad). The large angular spectrometer consists of tracking detectors (scintillating fibres, drift chambers,...), ring-imaging Cherenkov detector (RICH), hadronic calorimeter, and muon filter. The small angular spectrometer that follows large angular spectrometer detects particles at small angles (±30 mrad). The small angular spectrometer includes tracking detectors situated downstream and upstream of the dipole magnet, electromagnetic and hadronic calorimeters and muon filter.

2 Trigger and data acquisition systems

The data acquisition system of the Compass experiment is strongly influenced by the cycle of the SPS particle accelerator. The spill of 4.8 s that is repeated every 12 s gives a duty cycle of 30 %. The DAQ system must use the acceleration part of the cycle to reduce the peak data rate to one third of the onspill rate. The system can be divided into the following functional layers:

1. frontend electronics
2. concentrator modules
3. readout buffers
4. event builders

2.1 Data acquisition hardware

First layer of the system known as the *Frontend electronics* (also primary electronics) serves as an input of data into the data acquisition system. This layer consists of 1400 detector frontend cards that are used to preamplify and digitize analog data from detectors. In order to reduce loss of quality of the signal and to reduce cost of cables, the process of digitization is performed close to the detectors. Each frontend processes data from multiple channels, in total, there is approximately 250 000 channels. At Compass, four different types of detector frontends are used. Most tracking detectors are treated by the F1-TDC chip. This chip developed at the University of Freiburg is designed to be pipelined and thus, it is dead time free. The GASSIPLEX application specific integrated circuit (ASIC) that processes data from the RICH detector requires a fixed dead time of 3–5 ms. The readout of the GEM and silicon detectors is handled by the APV25 chip that features the analogue pipeline and buffer for 10 events. The calorimeters are processed by fast integrated analog to digital converters (FIADC) that can digitize up to 3 consecutive events in 30 ms.

The 1 400 frontend cards are connected to 150 *concentrator modules* that form the following layer of the system. These modules are used to initialize frontend cards, perform readout and assembling of data from multiple channels. From the hardware point of view, the modules are based on the VME buses. Two types of modules are used: CATCH and GeSiCA. The CATCH[2] module is designed to work with calorimeters, RICH detector, and tracking detectors. The CATCH modules are developed at the University of Freiburg. The GeSiCA[3] modules are designed for the frontends based on the APV25 chip that are characterized by high data rates and channel density. The concentrator modules receive signals from the *Trigger Control System* (TCS). When the trigger signal arrives, the concentrator modules perform a readout of date. The TCS also distributes metainformation about the event. By appending these information

[2]COMPASS Accumulate, Transfer and Control Hardware

[3]GEM and Silicon Control and Acquisition module

that include event identification and timestamp to the raw data assembled by a readout module, the *subevents* are created.

The files corresponding to the subevents are transferred to following layer using the *S-Link* communication interface. S-Link is a high speed, optical link developed for the ATLAS experiment that is capable of transferring data at speeds up to 160 MB/s, [26]. To reduce number of S-Links, subevents from multiple concentrator modules can be multiplexed on a single link using the S-Link multiplexer SMUX. Subevents coming from the S–Links are stored in a PCI cards called *spillbuffers*. Each spillbuffer is equipped with a 512 MB of on board memory organized as a FIFO. This amount corresponds to data from 2–3 spills. Spillbuffers are installed in servers called *readout buffers*. Each readout buffer can contain four spillbuffers. The spillbuffers make use of the SPS cycle: they are being filled during spills and continuously unloaded into the main memory of the readout buffers. In this way, the the load is distributed across the entire SPS cycle which reduces the data rates to one third of the onspill rate.

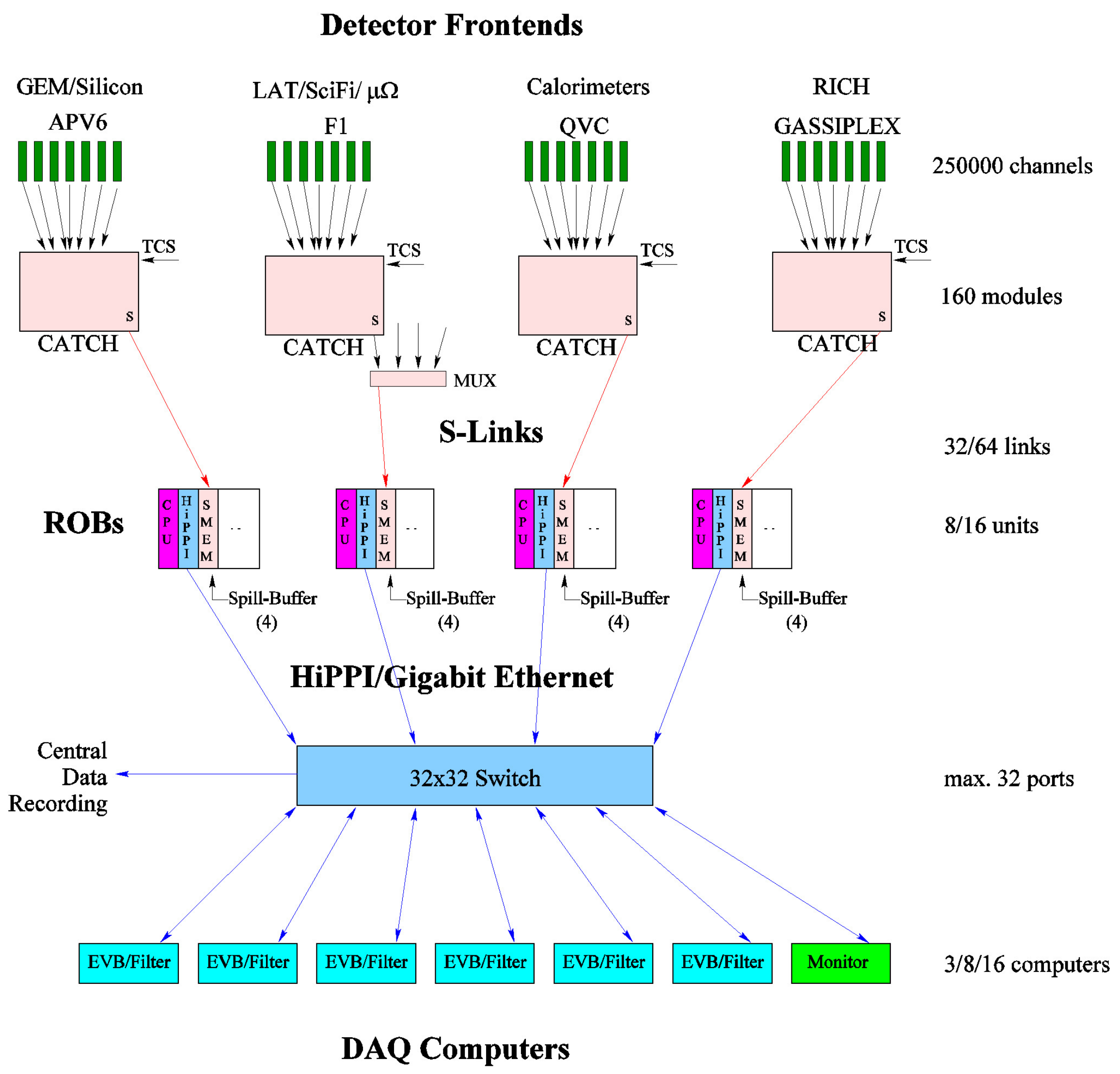


Figure 2.2: Layers of the data acquisition system according to [1]

The readout buffers send the subevents over the Gigabit Ethernet to the *event building servers* that form the last layer of the data acquisition. The main task of an event builder is to receive subevents from all readout buffers and to reorganize data from these subevents into full events (i.e. event building). During this process, a file with metainformation about the event is created. After some delay, this metainformation is stored into the Oracle database and the file with event is send via the *Central Data Recording* facility into the permanent storage *CASTOR*[4] in the CERN IT center situated 5 km away from the experimental hall. The remaining CPU power the event building machines is dedicated to the online filter and partial data analysis.

During the 2004 Run, 19 readout buffers and 13 event builders were used for the data acquisition. These machines are installed in a server room in the COMPASS experiment hall and are connected into the internal COMPASS network which is accessible from the outside through the gateway computers. In this way, the data acquisition is protected from the unauthorized access.

2.2 Trigger Control System

The trigger system selects physically interesting events or vetoes physically non interesting events in a high rate environment. The resulting trigger signal at the COMPASS experiment is based on signals from hodoscopes, energy deposited in calorimeters, and a veto system. Depending on the scientific program of the experiment, different components are combined to create the trigger signal. The trigger together with the reference time and trigger identification is distributed to the concentrator modules by the trigger control system TCS. The TCS also provides the stable clock to the experiment.

The TCS is based on the Time and Trigger Control (TTC) system that has been developed for the LHC experiments, however the TCS controller, the TCS server, and the TCS receivers have been added for the COMPASS experiment. The TCS controller is the heart of the system; its purpose is to synchronize data acquisition with the cycle of the SPS, to encode the trigger signal, to count triggers, to distribute the configuration, and to generate the dead time. The encoded signal is transmitted over the optical fibre network into the TCS receivers which are plugged into the backside of slots where concentrator modules are inserted in the VME crates. The TCS receivers receive and decode the trigger signal and provide it to the concentrator modules. The TCS server provides a command interface between the control and monitoring software and the TCS controller hardware.

Two types of the dead time are generated by the TCS controller: minimal time allowed between two consecutive triggers is generated for detector channels that can not be pipelined and maximal number of triggers accepted during certain time period is generated for pipelined channels. During the year 2004, the following configuration was used, [1]:

- 5 μs minimal time between consecutive triggers
- 3 triggers within period of 75 μs
- 6 triggers within period of 225 μs

At the nominal trigger rate 10 kHz, this configuration caused dead time of 5 %.

2.3 Data acquisition software DATE

From the software point of view, the data acquisition at the COMPASS experiment is handled by the DATE[5] package, [3]. The package has been developed for the ALICE experiment; several modification and extensions have been added for the purposes of the COMPASS experiment. The

[4]CERN Advanced Storage facility

[5]Data Acquisition and Test Environment

Date provides the data acquisition tasks in a distributed network environment. The DATE was designed to be very flexible system because the ALICE experiment operates in two modes: heavy ion collisions and proton–proton collisions. The proton–proton mode is characterized by a high interaction rates (up to 200 kHz) but relatively small size of events. The selectivity of the first level trigger must be high in order to cope with the high interaction rate. On the contrary, the heavy ion mode is characterized by lower interaction rates (upto 10 kHz) and bigger size of events (several megabytes per event). Thus the selectivity of the first level trigger may be lower but its complexity must be higher. At ALICE, the data acquisition is distributed over several hundreds nodes, on the other hand, it can be used at a small laboratory experiments with a single node for all the tasks. It was proved that the system is able to readout data at speed 40 GB/s, to perform event building at speed 2.5 GB/s, and to store events into the permanent storage at speed 1.25 GB/s, [3].

Each of these nodes must be powered by the GNU/Linux operating system and support the TCP/IP stack. Additionally, the DATE software requires to be installed on the Intel $x86$ compatible hardware architecture.

From the functionality point of the view, the Date package offers the data flow control, the event building, the load balancing, the information reporting, the run control, the event sampling, and the interactive configuration. The Date package distinguishes two basic types of nodes: *Local Data Concentrators* (LDC) and *Global Data Collectors* (GDC). In the terminology used by the Compass experiment, the LDCs correspond to the readout buffers; their purpose is to gather data from subdetectors. The GDCs correspond to the event builders in the Compass terminology; they receive the subevents from the LDCs and assemble them into the full events. For the purpose of the Compass experiment, the functionality of the online filter or the electronic logbook have been added to the Date package. These facilities will be described in the following paragraphs.

Moreover, the Date package also defines the file format that is used for storing subevents and events. The tools for the physical analysis of data need to decode this format, thus a data decoding library has been developed at the Compass experiment.

Local data concentrators

On each of the local data concentrators, two processes run: the *readout* and the *recorder*. The *readout* process receives signals from the trigger system, reads the subevents from the frontend electronics, and stores them into into the memory buffer organized as a FIFO. The *recorder* process offloads the subevents from this buffer and moves them to the appropriate recording device. Depending on the configuration, the recording device can be either a disk file (in the case of a data acquisition with a single node), or more typically the network socket connected to the global data collector; in this case each nodes must be connected to the same network that supports the TCP/IP protocols.

Since the Date package is designed to be portable between different experiments, the reading of the data from the subdetectors is done by a software module called *readList* that is linked to the readout process. Each experiment that wishes to use the Date package for the data acquisition must develop its own version of the *readList* module. The module must implement the following routines: *ArmHw*, *AsynchRead*, *EventArrived*, *ReadEvent*, and *DisArmHw*.

- The *ArmHw* routine initializes the equipment before the start of the run.
- The *AsynchRead* routine performs the readout of the detectors that produce the data asynchronously; this routine is called with the main loop of the readout program.

- The *EventArrived* routine is called in the main loop of the readout program to test if the signal from the trigger system has arrived.
- The *ReadEvent* routine performs the readout of the subevents produced by the detectors. This routine is called in the main loop of the readout program when the signal from the trigger system arrives.
- *DisArmHw* routine is called when the run stops; its task is to shutdown the equipment.

When the *readout* process receives the Start Of Run (SOR) signal, it prepares memory buffers for the subevents, executes the initialization scripts, and calls the *ArmHw* routine of the *readList* module. If the initialization phase succeeds, the readout process enters its main loop (see Figure 2.3). In the main loop, the process calls the *AsynchRead*, the *EventArrived*, and *ReadEvent* routines to read the subevents from the frontends. When the subevent is received, the readout verifies that its header contains all required information and it appends additional fields to this header (e.g. timestamp). The *readout* process also increments the counter of processed subevents that can be used to terminate the current run. The verified subevents are then passed to the *recorder* process. The main loop is finished when certain conditions are met. Most typically, the run ends when the required number of subevents or spills is collected. The run can be also terminated if some error occurs, e.g. the data do not arrive within a defined time period after the start of run. During the termination phase, the *readout* process calls several scripts and the routine *DisArmHw* from the *readList* module.

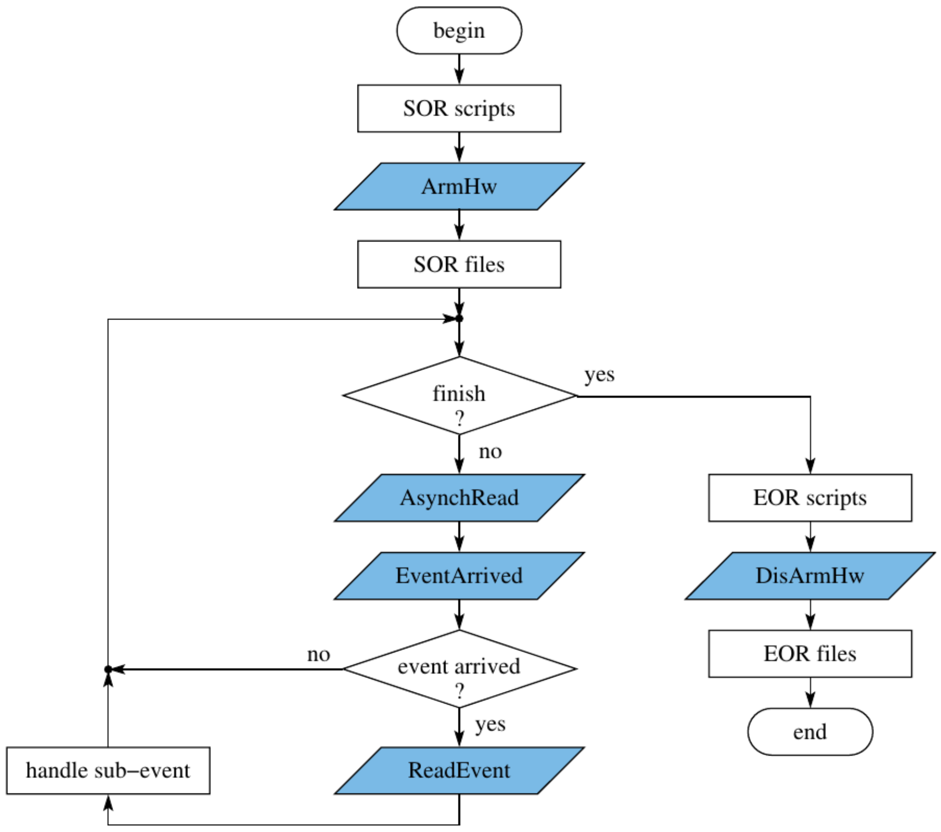


Figure 2.3: Event loop of the *readout* process according to [3]

At the COMPASS experiment, the data from the multiple frontend channels are assembled in the concentrator modules CATCH and GeSiCA and then sent to the spillbuffer PCI cards that are installed in the readout buffer servers. The readout process prepares the buffer in the main memory of the readout buffer and offloads the subevents from the spillbuffer memory via the Direct Memory Access (DMA) transfer to this buffer. The spillbuffer card can store data from 2–3 SPS spills.

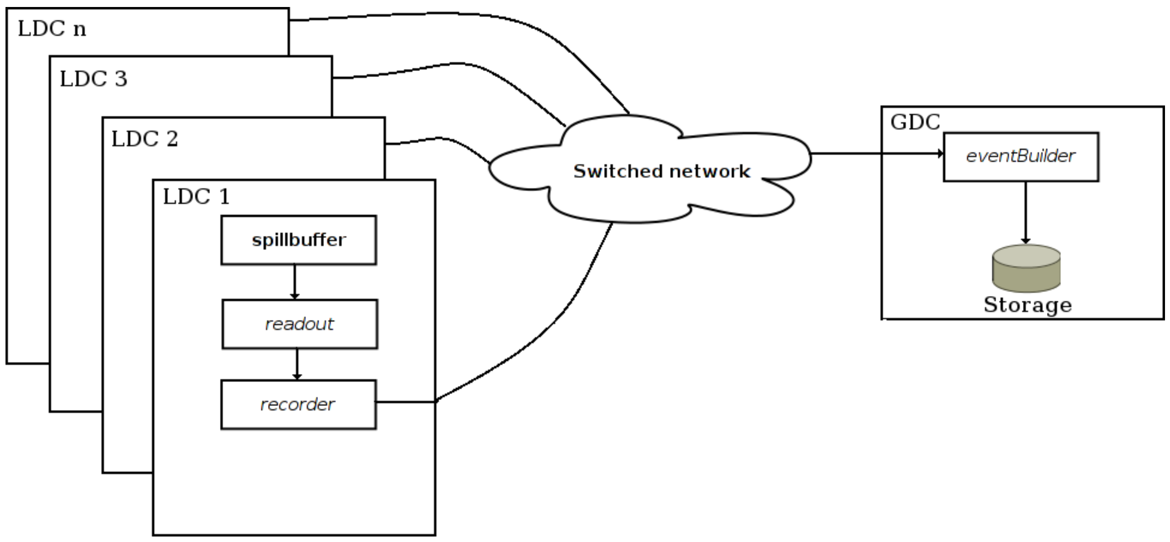


Figure 2.4: Data flow between DATE processes in the event building network

The *recorder* process that runs on each LDC records the data produced by the readout process to a recording device. The process is based on routines from the *recordingLib* package. Depending on the value of the *recordingDevice* run parameter, the device can be a set of local disk files, a set of named Unix pipes, or a set of global data collectors. At the start up, the *recorder* process connects to the memory buffer allocated by the *readout* process. It writes its own process identification number (pid) to the shared region of the buffer, thus it can be paused and resumed by the *readout* process. Depending on the configuration, the *recorder* either opens the local files, or connects to the global data collectors. After the initialization, the *recorder* process enter its main loop. In the main loop, the subevents are taken from the buffer and either saved into the opened local file, or sent to the global data collector over the event building network. When a subevent is successfully recorded, the *recorder* process removes it from the buffer to free the memory. The main loop continues until the End Or Run (EOR) command arrives. However, it can also be terminated if a requested amount of data is already recorded, there is too many errors during writing on a disk or sending to the network, or an operator requests the end of run.

Global data collectors

On each of the global data collectors, the *eventBuilder* process runs. At the start of the run, the process allocates memory buffer required to hold the incoming subevents. This memory is divided into the public and the LDC parts, the LDC part is further divided into segments corresponding to each LDC. The TCP/IP connection is established between the *recorder* processes on LDCs and the *eventBuilder* processes on GDCs. The eventBuilder process polls this opened connection for incoming data that are sent by the *recorder* process on the LDC. The data are stored in the previously allocated buffer; when the all subevents are received, the *eventBuilder* reads the subevent headers to form the complete events. The *eventBuilder* process can work either in a *direct recording mode*, or in an *online recording mode*. In the direct recording mode, the complete events are temporarily stored on the local disk (and later moved to the permanent storage). In the online recording mode, the complete events ar moved using the memory mapped scheme to the following processing stage such as an online filter or a program for an online analysis of data.

The GDCs can be added or removed to or from the system during the taking of the data. Furthermore, the data taking can continue if a GDC crashes, in the worst case, several events are lost. On the other hand, with a LDC missing, it would be impossible to reconstruct the full events, thus if any LDC crashes during the operation, then the run is stopped and marked as crashed.

Load balancing

During the event building process, all the subevents that correspond to one particular trigger are transferred from all the readout buffers/LDCs to one selected event builder/GDC. The DATE package supports the balancing of the load between the GDCs using the *Event Distribution Manager* (EDM).

The EDM functionality is implemented by the *edm* process that is deployed on the dedicated node *edmHost* and *edmClient* with *edmAgent* processes that are deployed on each of the LDCs. The task of the *edm* process is to maintain the list of available GDCs which is also denoted as the *GDC availability mask.* When the GDC connects to the data acquisition, it is added to the mask. When the GDC disconnects from the system, it is removed from the mask by the *edm* process. Additionally, the *eventBuilder* process that runs on the GDCs can send the *nearlyEmpty* and the *nearlyFull* messages to the *edm* process. If the *edm* receives the *nearlyEmpty* message, it adds the corresponding GDC to the list; if it receives the *nearlyFull* message, it removes the corresponding GDC from the list. The availability mask is valid only for certain range of events identified by the *firstEventId* and *lastEventId* variables.

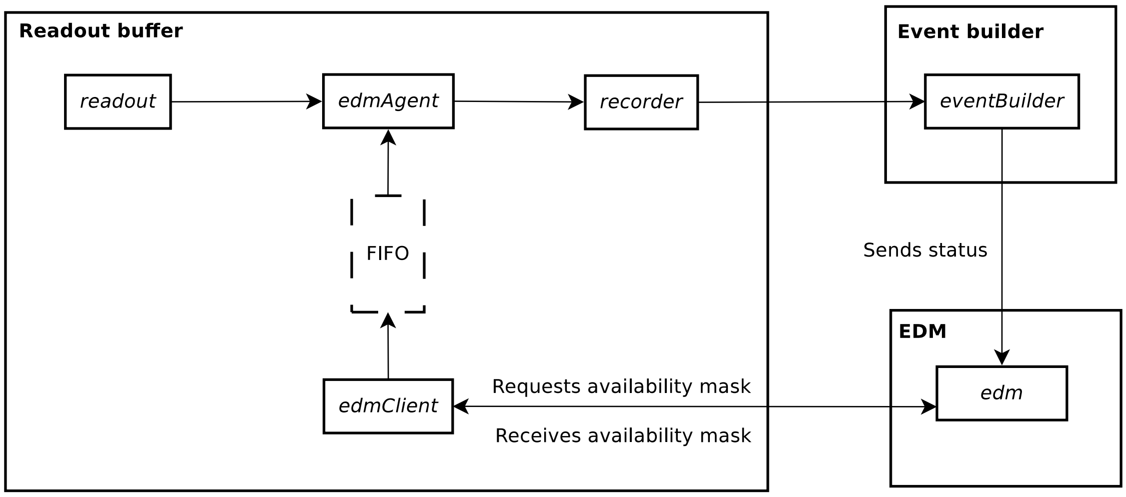


Figure 2.5: Event distribution management in the Date

On each of the LDCs, the *edmClient* and *edmAgent* processes run. The *edmClient* process communicates with the *edm* process over the TCP/IP connection. To avoid the dead time, the *edmClient* asks for the updated availability masks in advance before the validity of the current mask expires. When the updated mask is received, the *edmClients* inserts it into the FIFO in the shared memory. The *edmAgent* reads the availability mask from the shared memory, selects the appropriate destination GDC and inserts identification of this GDC to the subevent header. The subevent is then passed to the recorder process which sends it to the selected GDC. The selection of the GDC is done independently by all the LDCs using the data driven algorithm; this algorithm prevents selecting unavailable GDCs.

The EDM functionality is optional, it can be disabled by the operator. This makes sense for example for the data acquisition with just one GDCs. If multiple GDCs are included in the system and the EDM is disabled, then the GDCs are selected in a round robin fashion.

Error and information reporting

In order operate the complex distributed system such as a data acquisition, it is essential to know what happens on each of its nodes. For this purpose, the DATE package contains the logging facility that generates, assembles, stores, and publishes messages about the behavior of the system.

The logging subsystem consists of a library of functions called *infoLogger* and several processes. The *infoLoggerReader* process runs as a daemon at each node that can produce log messages. This process is started automatically when the first process that uses the *infoLogger* library is launched. The *infoLoggerReader* receives the messages generated by the local DATE processes over the named Unix socket and sends them to the *infoLoggerServer* process.

The *infoLoggerServer* process is deployed on a dedicated node, thus the communication between the *infoLoggerServer* and *infoLoggerReader* processes is based on the TCP/IP connections. The *infoLoggerServer* receives the messages sent by the *infoLoggerReader* processes and archives them in the storage. Depending on the configuration of the DATE system, the storage can be implemented by a text file or a table in the MySQL database. Finally, the *infoBrowser* is a graphical application used to browse the archived message.

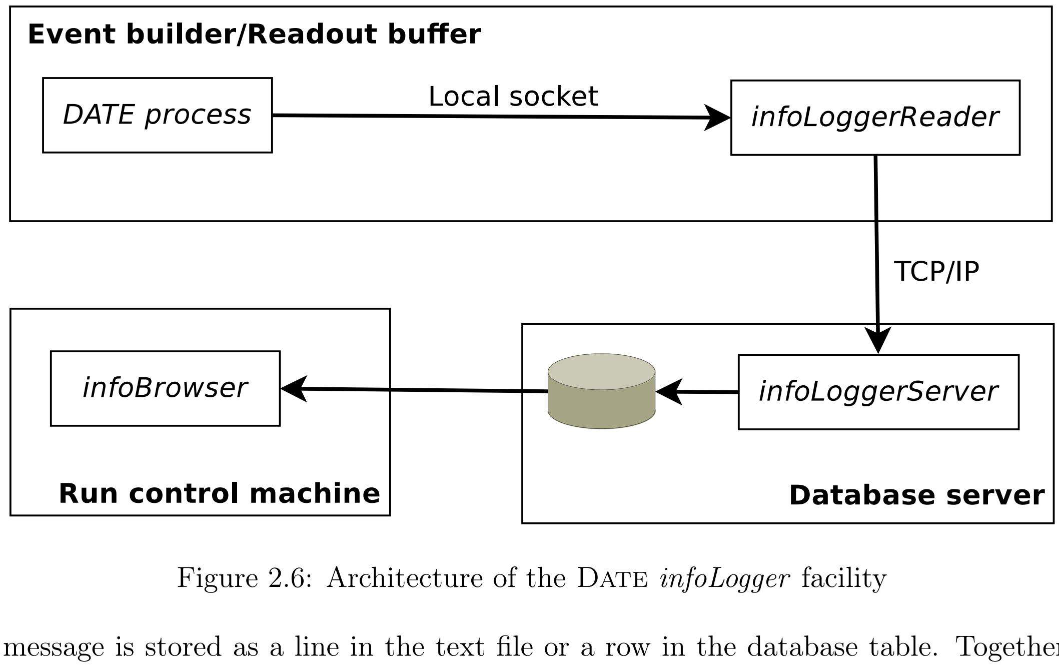


Figure 2.6: Architecture of the DATE *infoLogger* facility

Each message is stored as a line in the text file or a row in the database table. Together with the text that describes the incident, the message contains additional information that helps to identify the problem including the timestamp of the incident, the hostname of the node when the incident occurred, the identification number of the process that generated the message, the run number, or severity. According to the severity, the messages can be divided into the three groups: information messages, errors, and fatal errors. Information messages describe the normal operation of the system, such as a successful start of the data taking. The error messages are used to describe the abnormal behaviour that does not interrupt the normal operation of the system. Finally, the fatal errors are reserved for incidents that causes crash of the data taking. The complete overview of the format of the messages can be found in the chapter 14 of the Date manual, [3].

The *infoLogger* facility provides several possibilities in which a new message can be generated. Each process (and also an operator) can call the command line tool *log* that generates the message and passes it to the *infoLoggerReader* process. The *infoLogger* also contains the library of the C functions for manipulating the messages. List of available functions is defined in the header file `infoLogger.h`. The connection to the *infoLoggerReader* daemon is established when one of these functions is used for the first time in a given process and remains opened until the process exits. The library contains functions to generated information messages, errors, and fatal errors and to open and create the connection to the *infoLoggerReader* process. The subset of functions from this library can be directly called from the scripts in the Tcl language.

Control of the system

The data acquisition system tasks are performed by tens of processes that are deployed on tens of distributed nodes. These processes need to be launched and initialized before the data taking can be started. This task is handled by the run control subsystem of the Date package. Additionally, it enables the operator to configure the system, include new machines into the system, or run several data acquisition subsystems within the experiment at the same time.

The run control is implemented by several processes. The data acquisition is controlled by the *runControl* process that receives the commands issued by a human operator. The human operator uses the *runControlHI* (HI = human interface) application to enter the commands. The *runControl* guarantees that only one operator can control the system at the same time and it also rejects the commands that are not compatible with current state of the system (e.g. it rejects to start the system if it is already running). According to the configuration of system, the *runControl* starts the *LogicEngine* process that is responsible for starting and stopping of processes that participate in the data acquisition. The *LogicEngine* sends the commands from the operator to the *rcServer* processes that are deployed on the remote hosts.

During the initialization, the *runControl* process sets the configuration of the data acquisition system to an empty one and sets the run parameters to their default values. If the Date configuration is stored in the database, it is loaded and it replaces the empty configuration. In the same way, if the run parameters are stored in the database, they replace the Date default values. After the initialization, the *runControl* enters the DISCONNECTED state in which it waits for commands from the operator. The CONNECT command loads the configuration with given name from the database and spawns the appropriate *LogicEngine* process. If the command succeeds, the *runControl* enters into the CONNECTED state. In this state, the operator can issue the LOCK_PARAMETERS command that causes that the *runControl* loads the set of run parameters with given name from the database. In case of success, the *runControl* moves to the READY state and the operator can issue the START_PROCESSES command that starts all the processes required to the data acquisition. When all processes are running, the system changes its state to the STARTED state. Finally, from this state, the data taking can be started by sending the START_DATA_TAKING command. During the data taking, the system is in the RUNNING state until the run is stopped either by reaching the spill limit or it is stopped by the STOP_DATA_TAKING command which causes the *runControl* process to return back to the READY state. The STOP_PROCESSES and ABORT_PROCESSES can be issued in the STARTED state; purpose of these command is to stop the processes that participate in the data acquisition and return the *runControl* into the READY state. In contrast to the STOP_PROCESSES command, the ABORT_PROCESSES command can kill the processes that do not respond. The UNLOCK_PARAMETERS command resets the run parameters and moves the *runControl* back to the CONNECTED state. Finally, the DISCONNECT command stops the *LogicEngine* process, resets the data acquisition configuration and switches the *runControl* process back to the DISCONNECTED state.

The exchange of commands between the distributed processes is based on the DIM communication library, [8]. The behaviour of the nodes is described by the finite state machines that are implemented in the SMI++ framework, [13].

Date configuration

Data acquisition system includes many elements such as detectors, readout buffers, or event builders. In order to operate the Date based data acquisition, a configuration that describes these elements and relations between them needs to be prepared. The Date package provides the *dateDb* facility that manages creating, modifications, and retrieving the system configuration. The configuration consists of a static and a dynamic part. The static part defines setup of the triggers, detectors, or hosts (i.e. the LDCs, the GDCs, and the event distribution manager). Based on this static part that is valid across runs, an operator can select a different dynamic configuration before starting a new run. Thus the actual configuration of the system is obtained by merging the static information with the current dynamic configuration. The static part of the configuration changes only when some element is replaced or added into the system.

Each element in the system has assigned a role, e.g. a LDC or a GDC, and is identified by its name and an identification number that is unique within a given role (i.e. there cannot be two LDCs with the same id). The static part of the configuration is stored in the following essential databases:

- The *roles* database defines the elements participating in the data acquisition including hosts, detectors, or trigger setup.
- The *trigger* database defines sets of detectors that are involved for the given trigger masks.
- In the *detector* database, the connection between the frontends and LDCs is defined for each detectors and subdetectors.
- The memory buffers on each of the nodes that are defined in the roles database are described in the *banks* database.
- The information from the *event building control* database is used by the eventBuilder process to prepare the event building strategy.

The static part is completed by the equipment configuration that describes the readout system. This information is not part of the *dateDb* package, however.

The *dateDb* facility supports storing the configuration either in the online MySQL database, or in the plain text files; the format of these text files is decribed in the Date manual, [3]. DateDb package contains a tool *createtables* that creates the empty tables that will contain the static configuration. The MySQL backend is recommended because the relational model reduces the risk of entering incorrect data. Moreover, for the MySQL backend, the dateDb provides the *editDb* application that is used to create and edit the static configuration of the system. The dynamic configuration can be modified directly from the human interface of the run control application.

At the Compass experiment, it has been decided to select the MySQL database. Additionally, this database is also used to store the messages generated by the infoLogger facility, the monitoring data of some detectors, or the information about the state of the beam line. An electronic logbook has been developed for the Compass experiment; it uses the same online database as a backend for the storage of its entries. More information about COMPASS database service can be found in Chapter 4 of this work.

Performance of the system and the online filter

During the first year of the physics data taking (i.e. 2002), the COMPASS experiment has recorded 260 TB of data which corresponds to $5.5 \cdot 10^9$ events. The system was working at the 5 kHz trigger rate, thus 25 000 events were recorded during a 4.8 s long spill. The average size of event was approximately 45 kB, therefore the onspill rate approached a value of 200 MB/s. Thanks to the spillbuffers that distribute the load over the entire SPS cycle of 16.8 s, the sustained data rate was reduced to the 60 MB/s which roughly amounts to the one third of the onspill rate.

Several improvements have been implemented into the data acquisition system before the start of the data taking in the year 2003. These improvements include the reduction of the event header (and consequently reduction of the average event size to 35 kB) or more effective accessing to the disks. These modifications enabled to increase the trigger rate upto 10 kHz.

During the pilot hadron run, the experiment was able to record over 8 TB per day, i.e. over 90 MB/s. However, during the tests with recording disabled, the system was able to handle data rate of 192 MB/s. This means that the system is limited by connection to the CERN Advanced Storage facility in the computing center that can only transfer data by speeds upto 128 MB/s.

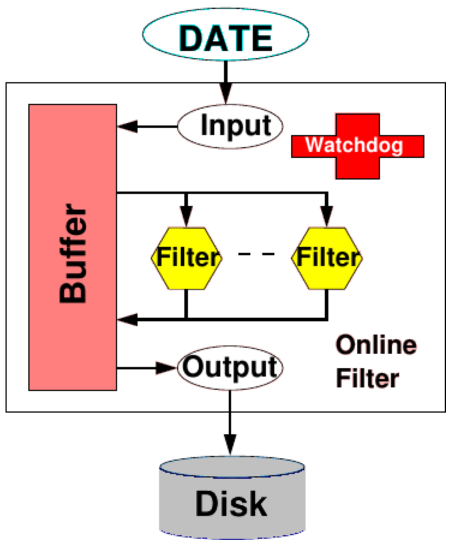


Figure 2.7: Architecture of the online filter *Cinderella* as proposed in [23]

In order to use the bandwidth of the link to permanent storage, the online filter program called *Cinderella* has been develop and integrated into the DATE package, [22]. As an additional benefit, the time required for the analysis of data and also the costs for the data storage are decreased by using the filter. The online filter can be regarded as a high level, software trigger; it rejects physically non–interesting events.

The online filter is running on the event building machines; it makes use of the fact that the event building requires only a fraction of the computing power of the machines. The *eventBuilder* process works in the online recording mode (see above); it passes the assembled events to the filter process. The Cinderella analyses and filters the events; the events that passed the filter are saved to the local disk array and are sent to the permanent storage after some delay. The Cinderella process can generate error and information messages that are processed by the infoBrowser facility. Moreover, the filter also writes additional messages into the local log file. The Cinderella can operate in several modes, e.g. filter active, mark only, or none. In the filter active mode, the Cinderella normally analyses events and rejects the bad ones. On the contrary, in the mark only mode, the bad events are only marked. This mode should be used for the testing. In the none mode, the online filter is disabled and the eventBuilder is working in the direct recording mode. The human interface of the run control facility has been updated to enable the operators to switch between the different modes of the filter. Also several information about performance of the filtering are included into the human interface.

Under the Compass conditions (i.e. the trigger rate of 10 kHz, duty cycle of the SPS of 30 %), the software has only 4 ms per event to make a decision. Thus the filter can use only part of the event because the full decoding is not possible within the time limit. However, there is a plan to deploy the online filter to the dedicated computing farm. This will allow to improve the decision algorithms and consequently increase a number of rejected events.

Data quality monitoring tools

Quality of the data needs to be continuously monitored during the process of data taking. Bad quality of data may be a symptom of a problem in a detector or a readout channel. Two main tools are used at the Compass experiment: *MurphyTV* and *COOOL*. MurphyTV is an application that checks the consistency of event headers. The application displays a list of data sources; for each source, a number of error is monitored. If the number of errors exceeds certain limit, operators are notified to fix the problem by reloading the faulty equipment.

The COOOL (Compass Object Oriented Online) is a process that runs on one of the event builders; it receives part of the events assembled by the eventBuilder process and performs analysis of these events. The COOOL presents the results of analysis in a form of the histograms produced by the ROOT framework, [7]. Users can interactively configure which detector planes should be included or removed from the analysis. By comparing these histograms to the reference values, it is possible to identify problems such as inactive channels, or noise on some detectors. The COOOL can export the histograms into the PDF file that can be added into the online logbook.

Scalability of the current data acquisition system

The number of input channels and the trigger rate increase in time: 260 TB of data have been recorded during the pilot run in 2002, this amount increased to approximately 500 TB collected in the year 2004 and more than 2 PB in 2010, [1, 23]. The demands on the data acquisition system increases, however, the hardware remains mostly unchanged for several years. Before the start of the data taking in 2006, several new servers were bought. These new servers are powered by a dual core Intel Xeon processors running on 3.6 GHz and are equipped with a 4 GB of the system memory. Unfortunately, part of the readout buffers and computers that host the concentrator modules are running on the original hardware (i.e. Pentium III processors running at 866 MHz with 1 GB of the system memory).

Nowadays, the upgrade of these machines is complicated because the spillbuffer cards are based on the PCI technology that is deprecated today, thus replacing those readout buffers would require development of a PCI Express version of the spillbuffer cards. Furthermore, as the hardware gets older, the failure rate also increases. Replacing the failed components usually means pausing the data acquisition which contributes to the dead time. Moreover, the number of a spare parts such as a spillbuffer cards is limited.

New database architecture for the Compass experiment

During the last three years, we have been participating in an upgrade of the existing data acquisition system. At first, we have analysed problems of the database subsystem that had caused major problems during the 2009 as a result of an introduction of a new monitoring application and an increased trigger rate. We have proposed and implemented a new database architecture that is able to handle the increased demands. Our works on the database subsystems will be thoroughly described in the Chapter 4 of this work.

Remote control room for the Compass experiment

At the end of the data taking in the year 2010, it has been measured that the radiation level in the COMPASS control room approaches the safety limits. Since the planned studies of the Drell-Yan process requires a higher intensity beam which would cause excess of the safety limits, it has been decided to either invest into an additional shielding of the spectrometer, or implement a remote control and monitoring of the experiment. We have studied the possibility of the remote control and based on these studies, we have implemented and successfully tested the remote control room. The remote control room will be discussed in more details in Chapter 5 of this thesis.

Control and monitoring software for a new data acquisition architecture

In parallel to the above mentioned activities, a development of a brand new data acquisition system proceeds. The new system is based on a custom hardware that is developed in a Technical University in Munich. This hardware uses the FPGA technology to control the flow of data and the event building, [18]. We are working on a software that will control and monitor the hardware. The system is to be deployed on several distributed nodes; we have defined the roles participating in the system, used the finite state machines to define a behaviour of the system, and proposed a custom communication protocol. We have implemented this proposal and tested its performance and stability. The development of the new system is covered in Chapter 6 of this work.

Chapter 3

Software used during update of the Compass data acquisition system

This chapter describes several software tools that have been used during the process of update of various parts of the data acquisition system of the Compass experiment. At first, we have analyzed problems with the online database system of the experiment and proposed and implemented updated version of this system. The system is based on the MySQL server, therefore first part of this chapter focuses on the database management systems and MySQL software. The second part of this chapter presents network programming with the DIM communication library. We used the library during implementation of the remote control and monitoring software for the new hardware platform for the data acquisition. During development of this software, we have also used the Qt framework, thus the last section of this chapter is dedicated to this framework, especially on the extensions of the object model of the C++ language introduced by the framework.

1 Database management systems

The Compass experiment uses databases to store information about conditions during data taking, runs, and events. *Database* is a collection of information whose systematic structure enables looking up these information using a computer. *Database management system* (DBMS) is a software that is designed to maintain and utilize large collection of data in database [10]. *Database systems* (DBS) consist of the database (DB) and DBMS.

The DBMS can be viewed as a virtual machine that encapsulates data stored in the database. Therefore data do not depend on application programs and application programs may access these data only through special interface such as a query language. Besides the independence of data on application programs, the DBMS offers other advantages. DBMS is optimized for efficient storing and retrieving large amount of data. Furthermore, use of DBMS enables enforcing of integrity constraints on data in database. Additionally, the DBMS supports transactions, concurrent access to data, or crash recovery. Thanks to these features, time required to develop application programs that require access to large set of data can be significantly reduced when using a DBMS.

The interface for accessing data provided by the DBMS can be divided into the following four main parts:

1. *Data definition language* (DDL) should be used to define logical and physical schema of the database.

2. *Data manipulation language* (DML) should be used to add, modify, or retrieve database data.
3. *Data control language* (DCL) should be used to define access rights to the data stored in database.
4. *Transaction control language* (TCL) serves for controlling of the database transactions.

The DBMS also includes a set of high level data description constructs known as a data model. The model insulates database users from the low level physical storage implementation, it allows users to the define the data to be managed by the database. Several types of data model exist, however this work focuses only on the relational data model that was introduced in 1970 at IBM research laboratory by Edgar Codd.

1.1 Relational data model

The *relational data model* works with one data description construct known as a *relation* that can be regarded as a set of records. Data in the relation model are described by a *schema*. The relation schema R defines a name of relation, set of *attributes* A_i (fields, columns), and *domains* (types) of the attributes D_i, therefore the schema can be written as $R(A_1 : D_1, \ldots, A_n : D_n) \equiv R(A)$. Relation is a set of n-tuples $\subset D_1 \times \ldots \times D_n$. A relation can be closely related to a table. The relation schema corresponds to the table header, names of attributes correspond to the names of columns, attributes correspond to columns, and n-tuples correspond to rows of the table. However, relation as a set cannot contain duplicitous n-tuples and does not define ordering of n-tuples. The schema of the relation can be extended by definition of *integrity constraints* that the n-tuples must fulfill. Therefore, we define the *relational database* as (R, I) where $R = (R_1, \ldots, R_k)$ is a set of relations and I is a set of integrity constraints. The *key* of a relation scheme $R(A)$ is a minimal subset of attributes from A that uniquely identifies each n-tuple of specific relation R.

The *database query* over a schema R is an expression which returns an answer in a form of schema S. The query domain includes all relations with schema R, the range includes all relations with schema S. Results of the query contain data from database and do not depend on physical implementation of the storage. *Query language* is a set of all usable expressions for construction of database queries. Several types of query languages exist. *Relational calculus* is a formal language based on the mathematical logic. *Relational algebra* is based on a set of operators for manipulations with relations. Relational algebra and calculus are equivalent in power. Query language that enables implementation of relational algebra is *relationaly complete.* Structured Query Language (SQL) is an example of the relationaly complete language.

Several operations with relations can be defined:

- *Selection* of relation R according to condition ϕ is defined as $R(\phi) = \{x | x \in R \wedge \phi(x)\}$ where ϕ is logical expression in form $t_1 \Theta t_2$ or $t_1 \Theta a$ and t_1, t_2 represent attributes, a represents some constant, and Θ represents relational operator.
- *Projection* of relation R on set of attributes $C \subset A$ is defined as $R[C] = \{x[C] | x \in R\}$.
- *Natural join* of relations $R(A)$ and $S(B)$ is a relation $T(C)$ defined as $R * S = \{x | x[A] \in R \wedge x[B] \in S\}$ where $C = A \cup B$ and n-tuples in the join are given by equality on attributes common for both relations R and S. The natural join operation can be generalized into a *general join* operation defined as $R[t_1 \Theta t_2]S = \{x | x[A] \in R \wedge x[B] \in S \wedge x \cdot t_1 \Theta x \cdot t_2\}$. Θ is a relational operator $\Theta \in \{=, <, >, \leq, \geq, \neq\}$, it is possible to construct more complex join conditions using the logical connectives $\{\wedge, \vee, \neg\}$.
- It is also possible to *rename an attribute*, this operation is denoted by the $\rightarrow$ symbol.

- Additionally, set operations are also defined for the relations. The operations include *union* $\cup$, *intersection* $\cap$, *set difference* $\setminus$, and *set (Cartesian) product* $\times$.

The relational operations selection, projection, Cartesian product, attribute renaming, union, and set difference form a minimal set of operation. Other operations can be derived from these operations.

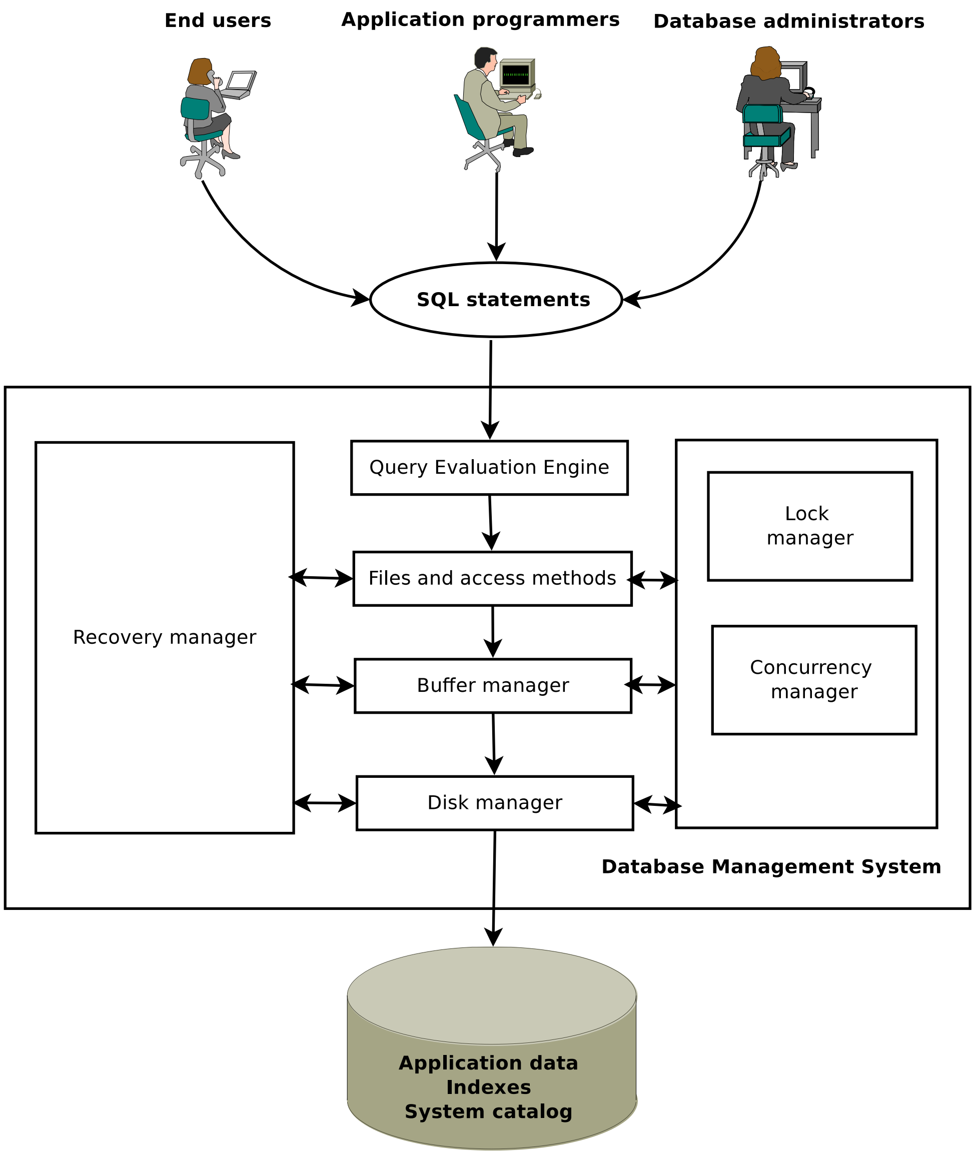


Figure 3.1: Database management system

The DBMS describes the data in three levels of abstraction: the *conceptual schema*, the *physical schema*, and the *external schema*. The conceptual (or logical) schema is used to describe data in terms of the data model. Therefore in the relational data model, the conceptual schema describes all relations managed by the database. Defining relations that model the real world system is not a trivial task, the process of designing a conceptual schema is known as a *conceptual design*. The conceptual schema shields users from the physical storage of data. This property is known as a *physical data independence*. The physical schema describes how the relations defined during the process of conceptual design are stored on a physical medium such as hard disks. Finally, the external schema allows customization of access to the database data at the level of individual users. Each database has only one physical and one conceptual schema, however it can have multiple external schemas. Each external schema consists of a set of *views* and relations from the corresponding conceptual schema. Views are similar to relations, but the n-tuples are not stored in the database but computed by the rules in the definition of the view. The views can be used to insulate users from changes made into the logical structure of data. This property is known as a *logical data independence.*

Several types of users interact with the database management systems including database implementators, end users, database administrators, and programmers of the database applications:

- Database implementators build and develop the DBMS itself. Usually, these users are employees of the database vendors such as IBM or Oracle.
- The end users are usually not computer professionals, they use the DBMS to store and access their data. Depending on the specific database systems, the end users may be customers of e-shop, students wishing to enroll to university course, or members of the shift crew accessing the electronic logbook of the experiment.
- Database application programmers use the tools that the DBMS vendors provide to develop applications that simplify access to the database data to the end users. The application programs should access the data through the external schemas.
- The database administrators manage the complex enterprise databases. They design the conceptual and physical schemas of the database. Additionally, they respond for security and authorization. Furthermore, they ensure that the database remains highly available and reliable. In case of the crash, they must restore the data back into the consistent state. They also modify the structure of the conceptual and physical schema according to changes in requirements. However, smaller databases are often maintained by the users who own and use them.

The DBMS can be viewed as a virtual machine that communicates with users through various user interfaces and client programs. The DBMS processes queries in a *query evaluation engine* that parses and optimizes the query and prepares the *query execution plan.* The plan usually takes a form of tree of relational operators that serve as building blocks for query evaluation. Bellow the query evaluation engine, the *access and files method* lie. In the DBMS, the concept of file represents a set of records or pages. The *buffer manager* retrieves pages from the disk into the buffers in the memory. On the lowest layer, the *disk space manager* operates. Its task is to read, write, allocate, and deallocate pages as requested by higher layers.

The DBMS also contains components that implement transactions, crash recovery, and concurrent access. These components include *transaction manager*, *lock manager*, and *recovery manager.* These components must cooperate with disk space manager, buffer manager, and files and access methods.

1.2 MySQL database

According to [37], *MySQL* is the world's most popular open source relational database management system (RDBMS). Development of the MySQL software was sponsored by the Swedish company MySQL AB, today it is owned by the Oracle Corporation. MySQL supports broad subset of the ANSI SQL 99 standard of the Structured Query Language. At the COMPASS experiment, the MySQL is used to store configuration and logs of the DATE and also information about data taking process.

The MySQL is available on wide range of platforms including GNU/Linux, MS Windows, FreeBSD, or Mac OS. The software is written in the C and C++ programming languages. Client applications can be developed using the provided C/C++ client library. Also, application programming interfaces for C/C++, Java, Perl, Python, PHP, Ruby, and Tcl languages also exist. Furthermore, the MyODBC interface provides standardized Open Database Connectivity connections. Additionally, the software includes embedded library that can be linked to the applications to create a standalone products. In the client–server mode, the connection between the server and its clients is most commonly based on the TCP/IP. The TCP/IP connection can be also encrypted by the Secure Socket Layer (SSL) library to improve the system security. Additionally, under the GNU/Linux system, clients can connect to the MySQL server through the Unix domain socket files. Under the Windows operating system, connection based on the named pipes or shared memory is available.

The software is designed to be fast, it is fully multithreaded system that uses the kernel threads to utilize multiple processor cores. It is very scalable software, it can be deployed on laptop or personal computer as well as on the dedicated database server. Using the *replication*, it is possible to distribute the load across multiple servers. The replication could also be used when high availability and reliability is required. Additionally, the highly available and reliable version of the MySQL suited for distributed computing environment called *MySQL Cluster* also exists. Server supports very large tables, it has been reported that some users use the MySQL to manage over $2 \cdot 10^5$ tables with $5 \cdot 10^9$ records [37]. The server's error messages has been localized into many languages including Czech, German, French, Polish, Slovak, or Russian. Server also supports different character sets for storing data and uses sorting based on the selected character set and collation. Unicode is also supported.

From the functionality point of view, the MySQL software also supports the following main features:

- views
- information schema database
- triggers and stored procedures
- caching of queries
- full–text indexing and searching (when using the MyISAM storage engine)
- transactions (when using the InnoDB storage engine)
- hot backups
- storage engines
- replication
- partitioned tables

Support for various *storage engines* is a feature unique to the MySQL software. When creating a table, database administrator can select the most suitable storage engine according to requirements, e.g. MyISAM engine should be used when high performance is the most important factor while the InnoDB engine should be used when the transactions are required. The storage engines are analyzed and compared in Section 4.2.2 of this work. The replication is a software technology that enables propagating changes made on the master server to the slave server(s). The replication implementation and application is described in details in Section 4.3.3. *Partitioned tables* are divided into several disk partitions according to value of a partitioning function. Under certain circumstances, the partitioning can be used to optimize queries as the query evaluation engine browses only partitions that can contain the requested data. This technique known as a *partition pruning* is explained in Section 4.5.1. The other mentioned features are described in details in the MySQL documentation [37].

2 DIM library

The last chapter of this work is dedicated to development of the control and monitoring software for the new data acquisition architecture for the COMPASS experiment. The new system is distributed over large number of heterogeneous nodes ranging from microcontrollers powered by some embedded linux up to workstations powered with MS Windows system. The nodes need to exchange information messages and commands between each other. The communication between the nodes should be efficient, transparent, robust, and reliable.

Because of the compatibility with the DATE package, we have decided to build the control and monitoring software on the *DIM communication library.* The DIM (Distributed Information Management System) has been designed for use with the trigger and data acquisition system of the DELPHI experiment at CERN; it provides asynchronous, one to many communication in a heterogeneous network environment, [8]. The communication is asynchronous, therefore clients do not have to poll the server regularly to verify whether some information has changed, the server notifies clients when the value changes. Additionally, the communication is one to many, this means that the server can broadcast the information about change of some monitored value to all subscribed clients. The library also provides transparency, both for coding time and run time. The *coding time transparency* means that the framework hides the differences between supported platforms (e.g. Little Endian and Big Endian byte ordering, number representation, size of data types, or data alignment) from the programmer, therefore the code is portable without changes between the platforms. The *run time transparency* means that the library allows processes to communicate without knowledge of the platform on which they are deployed. Also, processes should be allowed to freely move between the nodes. Furthermore, the library offers the reliable and robust communication: if a communication between client and server is interrupted, the affected client tries to reestablish the communication. Also, crash of a single nodes should not influence the rest of the system.

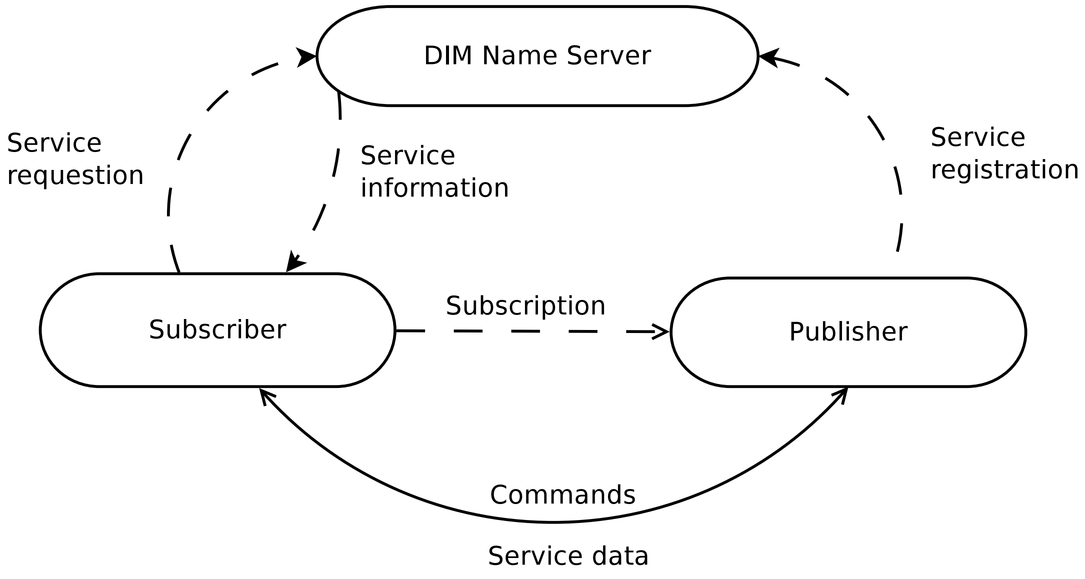


Figure 3.2: Subscription to the DIM service; full lines represent exchange of data between publisher and subscriber, dashed lines represent communication with DIM Name Server.

The DIM library is based on the TCP/IP standards, it extends the client–server paradigm with a concept of the *DIM name server* DNS. The DNS keeps a list of services available in the system.

When a server (or a *publisher* in the DIM terminology) wishes to publish some service, it must pass the description of the service to the name server. The description of the service contains the format of message and unique service name. The name server registers this information with the address of the publisher. When a client (or a *subscriber* in the DIM terminology) wishes to subscribe to a particular service, it must pass its unique name to the name server. The name server looks up the requested service and returns address of the corresponding publisher to the subscriber. Then, the subscriber uses the address to connect to the publisher. The communication with the name server is transparently handled by the library. As the services are identified by the name, there is no need to manually specify IP address or hostname of the server in the client code. On each node that uses the DIM library, the `DIM_DNS_NODE` environmental variable should be set to contain address of the name server. Otherwise, one needs to specify this address manually in code.

The DIM library distinguishes four types of messages. The TIMED services are updated at regular intervals. Using the MONITORED services, the subscribers are notified by the publisher when the monitored quantity changes. The ONLY ONCE service should be used when a subscriber wishes to receive non recurring information. Finally, subscribers can also send COMMANDS to the publishers.

2.1 Application Programming Interface

The DIM library is implemented in the C programming language because of the portability. However, it also provides interfaces to the C++, Java (through the Java Native Interface calls), FORTRAN, and Python languages. The library is available on a wide variety of platforms including GNU/Linux, MS Windows, Solaris, or Darwin and also on real time systems such as LynxOS or VxWorks.

During the implementation of the control and monitoring software, we have compared Java and C++ version of the library, the results are summarized in the last chapter of this work. Based on the test results, we have decided to use the C++ interface that can be divided into three parts: server classes, client classes, and utility classes. The server classes include:

- The `DimServer` class represents DIM publishers. It provides mainly static method that should be used to start the publisher and to set exit and error handlers.
- The `DimService` class corresponds to ONLY ONCE, MONITORED, and TIMED services. The class contains several constructors that should be used to describe the service and register it under the name server and various methods that should be used to update the service data. One can also use this class to set time stamps and quality flags of the messages.
- The `DimCommand` class implements the DIM COMMANDS. Constructors of this class enable to describe and register the DIM commands to a list of available services managed by the name server. Additionally, the class contains methods that enable accessing command data sent by subscribers.

The client part of the interface of the library contains two important classes: `DimClient` and `DimInfo`. The `DimClient` class implements several static methods related to the DIM subscribers. Most importantly, this class enables sending DIM commands to the subscribers. The class can also be used to set the error and exit handlers. The `DimInfo` class serves for subscription and reception of the service data. The class contains several methods that should be called to access the service data and also the information handler that is called back when the service data changes on the publisher. The handler is a virtual method, therefore user must subclass the `DimClient` in order to define behavior of the handler.

Finally, the utility part of the library consists of several support classes. The `DimBrowser` class provides service discovery, it can be used by processes to get a list of subscribers, publishers, and services known by the name server. The `DimErrorHandler` implements handling of DIM related errors to both subscribers and publishers. The class contains only one virtual function that is called back when an error is detected. The error code, the error message, and the severity is passed to the method. The `DimExitHandler` implements exit handling for the DIM servers, the class contains one virtual method that is called back when the server is requested to exit. This may happen under several circumstances: when the `Exit` command is received either from some subscriber or the name server, the address of the name server is not defined, the name server refuses the connection, or some service provided by the server is already provided by some other server.

The complete description of the library can be found in the online documentation, [31]. In the last chapter of this work, the source codes of example DIM publisher and DIM subscriber is analyzed in more details.

3 Framework Qt

The *Qt framework* is the most commonly known as a multiplatform library of the C++ classes that represent the graphical widgets that are used to design graphical user interfaces (GUI). These classes include primitive widgets such push buttons, text labels, or check boxes, as well as more complex widgets such as main application windows, tool bars, text editors, graphical canvases, or menu systems. However, the framework also provides classes for other areas frequently used during programming of graphical applications including networking, multithreading, databases, 2D and 3D graphics, state machines, or XML. The framework also contains container classes similar to the classes from the standard template library of the C++ language such as lists, vectors, or maps. Additionally, the Qt framework introduces introspection, signal and slot mechanism, or guarded pointers into the object model of the C++ language. In addition to the classes, the Qt installation also contains several support tools:

- The *Qt Designer* is a graphical application that serves for designing of the layout of application windows and forms. The layout is stored in a custom *ui* (ui = user interface) files that use format based on XML.
- The *uic* (user interface compiler) tool translates the ui files created by the Qt Designer into the C++ headers and source files that implement the designed layout.
- The *moc* (meta object compiler) is a preprocessor that implements the signal and slot mechanism.
- The *qmake* tool manages the Qt projects and generates `Makefile` files with rules required to build the project. Normally, the moc and uic tools need to be called on source codes and ui files to generate the C++ source codes that can be compiled by standard compilers such as gcc (GNU C Compiler).
- *Qt Creator* is an integrated development environment (IDE). It supports project manager, distributed version control systems, code completion, syntax highlighting, code profiling, or debugging.
- *Qt Assistant* is an interactive documentation browser.

The Qt is designed to be portable between the major platforms. Besides the MS Windows and GNU/Linux with X window system, the Qt applications are also supported on the Mac OS, Solaris, or various *BSD systems. According to the motto *"Write once, compile everywhere"*, the Qt applications need to be recompiled when a change of platform is required; i.e. it provides source code compatibility between supported platforms.

3.1 The object model of the Qt framework

The object model of the Qt framework includes *introspection* (self knowledge), *signal and slot* mechanisms, *properties*, or *object trees*. The model is build on three elements: the `QObject` class, the `Q_OBJECT` macro, and the moc preprocessor. Each class that wishes to use the signal and slot mechanism must be derived from the `QObject` base class and must contain the `Q_OBJECT` macro in the private section of its declaration. During building of the application, the moc preprocessor replaces the `Q_OBJECT` macro with a code that implements signals and slots. The resulting code generated by the moc tool can be compiled by any C++ compiler. Normally, user does not have to call the moc manually as the qmake project manager prepares corresponding build rules stored in the `Makefile` file.

Signal and slot mechanism

During development of the application with graphical interface, different parts of the application often need to exchange messages, e.g. a canvas needs to be notified when user clicks some push button. For this purpose, the Qt introduces the signal and slot mechanism; each subclass of the `QObject` class can contain slots and can emit signals.

Slots are very similar to common member functions, they can be called directly or as a reaction when the connected signal is emitted. The slots are introduced by the `slots` keyword in the class declaration. Like member functions, the slots can be divided into three groups according to access rights:

1. *Public slots* can be called from everywhere.
2. *Protected slots* can be called from the class in which they are declared and its descendants.
3. *Private slots* can be called only from the class in which they are declared.

Signal are introduced by the `signals` keyword in the class declaration. Signals cannot return any value, therefore they must be declared as `void`. Signals are emitted by the `emit` keyword. When signal is emitted, the execution of current method is interrupted until all slots connected to the signal are executed. One signal can be connected to multiple slots, also one slot can be triggered by multiple signals. Signal can also be connected to another signal.

Connection between signal and slot is established using the static method `connect` of the `QObject` class:

```
QObject::connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

The `sender` and the `receiver` parameters are pointers to the sender and receiver of the signal. The `signal` and `slot` parameters are function signatures without names of parameters. Signal and slot must use the same list of parameters. Signal and slot mechanism is type safe, bad connection does not cause crash of the application; only warning is printed on the standard output. Signals do not know anything about slots that are called when it is emitted. Furthermore, the slots do not know if it is called directly or as a reaction on some emitted signal. Thus, signals and slot are *loosely coupled* which enables creation of independent software components.

Introspection

Introspection or *self knowledge* enables retrieving information about classes at run time. In the Qt framework, the introspection is based on the base `QObject` class. The class contains several methods and properties that can be used to access metainformation about type. The method `className` returns the name of the class as a character array. Method `isA` tests if the current instance is an instance of given class. Similarly, the `inherits` method tests is the current instance is descendant of given class.

Furthermore, each object in Qt is associated with a meta object. Pointer to this meta object is returned by the `metaObject` method of the `QObject` class. The meta object provides additional information about signals (methods `numSignals`, `signalNames`), slots (methods `numSlots`, `slotNames`), and properties (methods `numProperties`, `propertyNames`) of given class. Complete list of methods can be found in the reference documentation [39].

When constructing an instance of some Qt class, it is possible to pass a pointer to owner of the instance to the constructor. Therefore, the instances are organized into a tree in a memory according to the ownership. When instance is destroyed, the Qt automatically and recursively destroys all owned instances. The tree of ownership can be printed by calling the `dumpObjectTree` method.

The implementation of introspection is independent on compiler thanks to the meta object compiler preprocessor.

Properties

In addition to signals and slots, the classes in Qt can also contain *properties*. Properties are data fields that are associated with read and possibly write methods. Therefore, properties extends the concept of *encapsulation of the information* stored in the data field. List of properties of given class can be accessed through the meta object system (see above). Properties are also displayed in the Object Inspector in the Qt Designer tool. Properties are registered by the `Q_PROPERTY` makro:

```
Q_PROPERTY(type Property READ getProperty WRITE setProperty)
```

Parameter `type` represents data type of the property, parameter `Property` represents the name of the property, and parameters `getProperty` and `setProperty` that follow the `READ` and `WRITE` keywords represent the read and write access methods. Both methods must be public member methods of the class. The read method must return the `type` type and must take no parameters. The write method does not return any value (i.e. type `void`) and takes one parameter of the type `type`. Often, the write method is also specified as a slot, it can also emit signal that notifies about change of value of the property. The `QObject` class contains `property` and `setProperty` methods that can be used to access value of given property.

Chapter 4

New database architecture for the COMPASS experiment

Some information about the conditions of the spectrometer and about the events are needed to be quickly retrieved during the data acquisition and data analysis. For this purpose, the COMPASS experiment uses the relational database management systems (RDBMS). In these systems, the data and also relations between the data are stored in the tables; the RDBMS provides a query language for the retrieval of data.

The COMPASS experiment uses two types of the database systems: *offline* and *online*. During the event building process, the catalog files with metainformation about events are prepared by extracting data from the event headers. The files with events are send to the permanent storage facility using the Central Data Recording (CDR) facility, the catalog files are inserted into the offline database. The offline database is distributed over nine machines powered by the Oracle system. These servers are installed in the CERN IT center which is located approximately 5 km away from the COMPASS experimental hall and are managed by the CERN IT division. The information stored in the offline databases is used during the offline physics analysis by the data reconstruction and analysis software.

This chapter focuses on the online database subsystem of the COMPASS data acquisition. This subsystem is powered by several MySQL servers that are installed in the server room of the COMPASS experiment which is situated directly in the experimental hall. The online database is used to manage the metainformation about the conditions of the data acquisition system, the beam line, and the spectrometer, taken directly (online) during the data taking. During the data taking in the year 2009, the online database service experienced serious performance problems that caused several crashes of the data taking process. Before the start of the data taking in the year 2010, we have analyzed the causes of these problems, proposed and implemented more robust and reliable database architecture.

The rational database management systems are briefly introduced in the previous chapter. In the first part of this chapter, the existing online database service (as in 2009) and its performance problems are analyzed. In the following part, we present our proposal of an updated database architecture. With some modifications, the proposal has been approved by a technical coordinator of the experiment. The process of migration to the proposed architecture is described in more details; the new architecture features database replication, connection through a proxy server, regular backups, and monitoring to achieve a high availability and reliability of the database service. We have analyzed the most frequently used tables and added additional indexes to these tables in order to optimize query execution times and consequently to reduce the load of the system. Moreover, we have studied and compared the different storage engines of the MySQL server. Furthermore, we have developed a new database application called *daqmon* that is used for monitoring of a performance of the nodes that participate in the data acquisition system. Finally, the performance of the database service during the data taking in the year 2010 is summarized and several possible improvements to the systems including load balancing and partitioned tables are discussed.

Part of this chapter contains technical information about installation and configuration of the database servers. These information were requested by the COMPASS data acquisition experts and supervisor of this thesis and should serve as a manual for future database administrators of the experiment.

1 Analysis of the original database architecture

In Chapter 2, it has been shown that the *dateDb* facility of the DATE package can use either a text file, or a MySQL database software for storing data. At the COMPASS experiment, the MySQL database has been selected as a backend for storing the data. The DATE package uses this database for storing the configuration of the data acquisition system and the debug messages. However, additional applications have been developed for the needs of the COMPASS experiment: the database also manages information about conditions of the spectrometer, the beam line, or the target; it also servers as a storage for the data of the electronic logbook. The data is being inserted into the database directly during the data taking, thus we call this subsystem an online database.

Up to the year 2010, the database service was powered by two physical servers called *pccodb01* and *pccodb02*. These servers were configured to operate in a *master–master replication* mode. In this mode, the server *pccodb01* was acting as a *replication master* of the *slave server pccodb02*. At the same time, the server *pccodb02* was acting as a replication master of the slave server *pccodb01*. This configuration guaranteed that both server were kept synchronized because each change of one server is replicated into the other server. However, the client applications connected to the database service through a virtual address *pccodb00* which normally pointed to the *pccodb01* server which was running on more powerful hardware configuration. On the *pccodb02* server, backups were regularly created. The backup process locked the tables for writing, thus replication was temporarily suspended. Since the clients were connected to the *pccodb01* server, the normal operation was not affected by the backup process. A *watchdog process* was monitoring the physical servers; if it detected a crash of the *pccodb01* server, it rewrote the virtual address to point to the *pccodb02* server; this allowed the *pccodb01* server to recover without interrupting the data taking. After the recovery, the server synchronized itself using the replication and the virtual address could be reset. Each physical server was managing approximately twenty logical databases. The most important logical databases are enumerated in the following list:

- The *DATE* database contains the static configuration of the data acquisition system defined in the dateDb facility as defined in Chapter 2 or DATE manual, [3].
- The *DATE_log* database contains the messages table used by the *infoBrowser* facility of the DATE package to store the debug messages generated by various processes that participate in the data acquisition. The content of this database can be browsed by the *infoBrowser* application that is used by members of the shift crew to monitor the behavior of the system.
- The *beamdb* database holds the tables *beam_profile* and *SPSinformation* with information about the beam line provided by the SPS control room. Additionally, monitoring information about the electromagnetic calorimeters (table *ECAL_mon*) or the online filter process

(table *FilterInfo*) are stored in this database.

- The content of the electronic logbook is handled by the *runlb* database which contains a table with messages inserted by the shift members and also tables with metainformation about the data taking which are filled automatically by the DATE processes. These tables include information about conditions in the experimental hall, runs, SPS spills, or configuration of the trigger system.
- The database *mysql* contains system information about users and user privileges.

In order to reduce the size of the above described databases (with the exceptions of the runlb and the mysql databases) and to chronologically separate data, every a few years new databases with the same name but different suffix are created, e.g. the *DATE2006* database contains the DATE configuration that was used in the year 2006, the *DATE2009_log* stores debug messages generated in the year 2009, etc. In a similar fashion, new messages table in a *DATE_log* database is automatically created every day.

Multiple clients connect to the database service; some clients are only retrieving records, while the others are also inserting or modifying records. For safety reasons, users accounts have been created for these clients. For example, a *cinderella* account is used by the online filter, a *dcs* account is used by the *Detector Control System*, or a *daq* account is used by the DATE processes. Additionally, some tables are accessible for reading by an anonymous account.

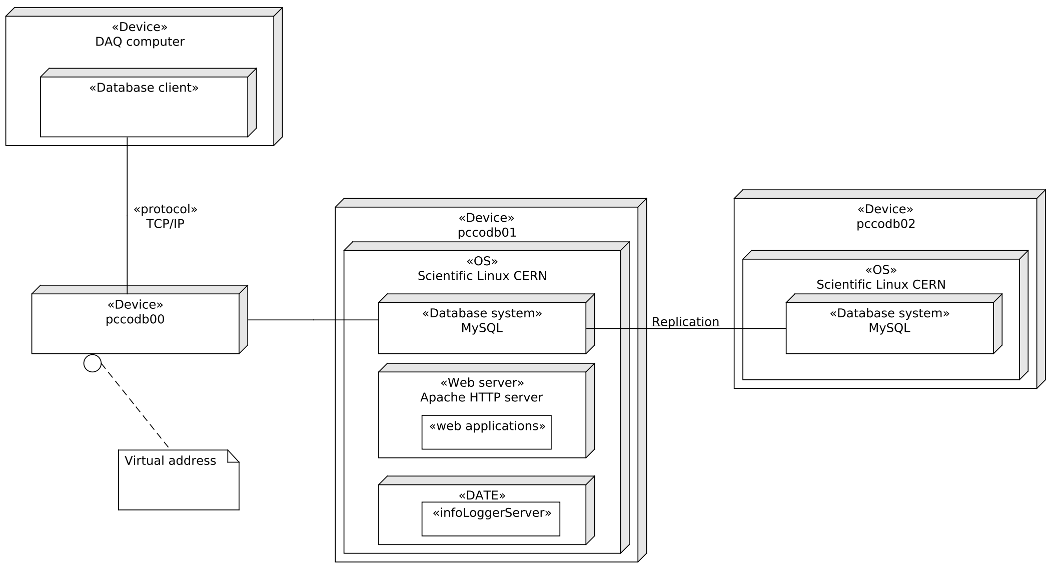


Figure 4.1: Original database architecture of the COMPASS experiment

Several additional processes were installed on the physical server *pccodb01*. These services included web server Apache that was used to host some web application used by a shift crew, the *infoLoggerServer* of the DATE infoLogger facility, or a monitoring system Ganglia.

During the data taking in the year 2009, the load of the database service has increased as a consequence of increased trigger rate and newly added monitoring applications. The increased load caused several crashes of the main server *pccodb01*; consequently the virtual address was remapped to the backup server *pccodb02*. Because the backup server was running on a weaker hardware than the main server, it crashed soon after the crash of the main server. To recover from the crash, the database service needed to be restarted. During the restart, the data taking was paused which increased the dead time and caused loss of the beam time. As it was expected that the trigger rate would increase even more in the year 2010, it was decided to modify the database architecture in order to prevent the future instabilities.

2 Proposed update of the database architecture

We were asked to investigate the database subsystem during the winter shutdown of the experiment, a few weeks before a start of data taking in the year 2010. It was not possible to identify the exact cause of the instabilities as we had not been able to inspect the system under the real conditions and additionally, the logging had been partially disabled. We have concluded that the database service had been powered by a weak servers that could not withstand the increased load. The hardware configuration of the original machines is summarized in Table 4.1; especially the amount of the installed Random Access Memory (RAM) seemed to be a critically low for the database servers. Operating system Scientific Linux CERN and the MySQL database software that were powering the servers were installed in an outdated versions, too. Furthermore, the operating system was installed in only 32-bit version that does not allow addressing of more than 4 GB of RAM.

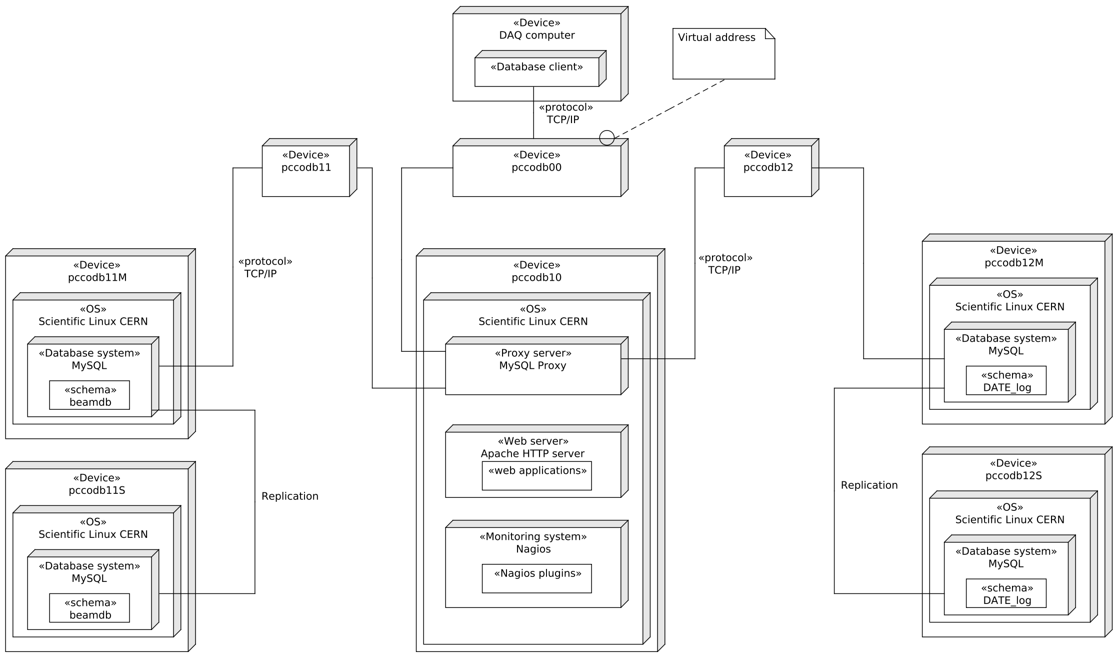


Figure 4.2: Proposed database architecture for the Compass experiment

The migration to the new architecture needed to be transparent to the client applications, thus the main requirement on the new architecture was to keep the compatibility with the original system. Effectively, this forced us to keep the MySQL software. We have proposed an updated version of the database service that is based on the following two key demands:

1. distribute the logical databases on more physical servers
2. use more powerful hardware and more recent version of the software

We have proposed to reduce the load of the physical servers by separating the two largest logical databases (i.e. *DATE_log* and *beamdb*) on two pairs of physical servers; one pair of servers would handle the *DATE_log* database, the other would handle the *beamdb* database. The remaining smaller database would be approximately evenly distributed among these two pairs of servers. In order to achieve the high availability and reliability of the service, the servers in pairs would be synchronized by a master–master replication. The fifth server would be used for additional tasks, mainly as a proxy server. The clients would connect to the proxy server through the virtual address *pccodb00*; according to the requested logical database, the proxy software would forward the queries to the corresponding pair of database servers. Since the same virtual address would be used, it would not be necessary to reconfigure client connection parameters. A monitoring system would be also installed on this fifth server. Its main task would be watching the state of the physical servers; if a problem would be detected, the monitoring system would try to reconfigure the proxy server to forward the traffic to the remaining servers and it would inform an operator via an SMS or e-mail about the incident. The HTTP server *Apache* and the *infoLoggerServer* process that were originally deployed on the *pccodb01* server would be also moved to the proxy server.

	Old server	*New server*
Processor	2 cores at 3 GHz (Xeon)	8 cores at 2.5 GHz (Xeon)
Memory	3 GB	16 GB
OS	32b SLC 4.7	64b SLC 5.4
Kernel	2.6.9	2.6.18
Database	MySQL 4.1.22	MySQL 5.1.45

Table 4.1: Configuration of database servers

2.1 Comparison of different versions of the MySQL server

According to the above described proposal, we have requested five new physical servers. The configuration of new servers is summarized in Table 4.1. While the original server was powered by one dual core Intel Xeon processor, the new servers are powered by two quad core Xeon processors. The amount of the system memory increased from 4 GB to 16 GB. For the data storage, several SATA disks in a RAID-5 configuration are used; approximately 4.5 TB of disk space is available in total. In order to fully utilize the 16 GB of the main memory, we have decided to install 64b version of operating system. Since the technical support of the version 4 of the Scientific Linux CERN (SLC) that was installed on the original lasted only until December 2010, a more recent version 5 of the SLC has been selected for installation on new servers. More recent version of operating system contains security and performance enhancements and also features a more recent linux kernel that has better support for hardware.

On the old database servers, the MySQL software was installed in version 4; however, a more recent version 5 already existed. Several new features were implemented together with security and performance fixes in this newer version of the software. We designed a test that should compare performance of several common database operations in the old and the new versions of the MySQL software and SLC operating system.

Design of the test

While waiting for a delivery of new servers, we have evaluated the performance of these two different versions of the MySQL server software. Since we could not reinstall the old servers and the new server were not delivered, we performed the tests in the *virtual machine*. As a virtualization software, the *qemu* processor emulator with the Kernel Virtual Machine KVM acceleration has been used. The host has been powered by the Intel Core2 Duo T9660 (two cores running on 2.8 GHz) with the 64b version of the Arch Linux distribution with linux kernel 2.6.32. Using the *qemu-img* tool, two harddisk images of 4 GB have been prepared; the 64b SLC 4 has been installed on the first image named `slc4.qcow`, the 64b SLC 5 has been installed on the second image named `slc5.qcow`. The qemu emulator has been launched with the following arguments:

```
qemu-kvm -cpu pc -machine x86 -smp 1 -m 1024\\
         -hda slc4.qcow -cdrom /dev/cdrom\\
         -net nic -net user
```

The guest uses one physical processor of the host machine running at 2.8 GHz. By using the `m` parameter, it is possible to specify amount of RAM that is assigned to the guest system (1 GB in this case). The image `slc4.qcow` acts as an IDE hard disk, the guest system uses the optical drive of the host system. The net parameters instruct the qemu to emulate the e1000 PCI network card with a network stack that bridges to the host's network. The user mode networking acts as a firewall and does not allow any incoming connections; only TCP and UDP protocols are supported.

During the installation of the operating system, the packages with the MySQL database and the PHP scripting language have been installed from the official repositories. The MySQL software has been available in the version of 4.1.22 for the SLC 4 and in the version 5.0.77 for SLC 5 system. Using the Create Database statement, a testing database called *test* has been created. The database contains a single table *test_ tbl*, therefore we do no present the entity-relation diagram, The table *test_ tbl* has been inserted into the *test* database with the structure shown in Listing 1.

Listing 1 Structure of the test table

```
CREATE TABLE test_tbl(
  id bigint(20) NOT NULL AUTO_INCREMENT,
  param varchar(25),
  val bigint(20),
  tag varchar(25),
  x float,
  y float,
  z float,
  t float,
  PRIMARY KEY(id)
) ENGINE=MyISAM;
```

The table is stored in the MyISAM engine which is a default storage engine in MySQL 4. The integer column *id* is used as a primary key of the table. We have assumed that there are only 30 000 different possible values for the column *param*, 10 000 different values for the column *tag*, and columns *x*, *y*, *z*, and *t* are random values from the interval $[0, 200]$. We have prepared a script in the PHP language that fills the table with a random records that fulfill these requirements:

```
#!/usr/bin/php
<?php
```

```
  require 'functions.php';
  require 'config.php';
  srand(make_seed());
  mysql_connect($host, $user, $pass) or die(mysql_error());
  mysql_select_db($db) or die(mysql_error());
  //code continues bellow...
```

The PHP scripts are normally interpreted on the web server. However, we have used the command line interface to interpret the script; the first line of the script defines a path to the interpreter. Then, the file `functions.php` that contains definitions of some custom functions for generating random strings is included. The PHP function `srand` initializes the built in generator of pseudorandom numbers. The custom function `make_seed` generates the seed from the current system time. The more recent versions of the PHP should initialize the pseudorandom generator automatically. Then, the script tries to connect to the database by calling the `mysql_connect` function and to use database by calling `mysql_select_db` function. The database name and connection parameters are defined in the `config.php` file. If these functions fail, a error message is printed and execution of the script is terminated.

After the connection to the database, two arrays of random strings are generated: first array with 30 000 items serves for filling the *param* column, second array with 10 000 items for filling the *tag* column. The random string are generated in the custom `getRandomString` function defined in the `functions.php` file, the code is based on example from [24]:

Listing 2 Function that generates random string of given length

```
function getRandomString($length){
  $result = "";
    for($i=0; $i<$length; $i++){
      $x = mt_rand(0,61);
      if($x < 10){
        $result .= chr($x+48);
      }else if($x < 36){
        $result .= chr($x+55);
      }else{
         $result .= chr($x+61);
      }
    }
  return $result;
 }
```

The function takes one parameter that defines the requested length of the random string. The generated string consists of the three types of characters: digits, capital letters of English alphabet, and small letters of the English alphabet. In the `for` cycle, the string is built a letter by letter by using the operator "." which returns the concatenation of its right and left arguments and `mt_rand` and `chr` functions. The `mt_rand` function returns a pseudorandom integer from the given interval; the function is based on the Mersenne–Twister generator, [21]. The generated integer is used as an ASCII code of a next character of the string. The ASCII code is transformed into a corresponding character by calling the `chr` function. The interval of codes $[48, 57]$ corresponds to digits (0, . . . , 9), the interval $[65, 90]$ to capital letters (A, . . . , Z), and interval $[97, 122]$ to small letters of English alphabet.

In the remaining part of the script, random records are generated and inserted into the table in the for cycle:

```
for($i=0; $i < $count; $i++)
{
  $x = (float)mt_rand(0, 20000)/100;
  ...
  $query = "INSERT INTO test_tbl
   VALUES('','$param','$val','$tag','$x','$y','$z','$t')";
  mysql_query($query) or die(mysql_error());
}
```

The `mt_rand` function returns an integer from the given interval. The value is divided by 100 and explicitly converted to a float in order to get a pseudorandom rational number from the interval $[0, 200]$ which is required for the x, y, z, and t columns. The remaining numerical columns are generated in a similar fashion. The values of the *param* and *tag* string fields are selected from the previously generated arrays. The random values are then substituted into the Insert query and the query is sent to the database using the `mysql_query` function (the value of the *id* column is not substituted as at the column is defined as an Auto_Increment. If the insertion fails, an error message is printed and execution of the script is terminated.

We have used the script to fill the *test* table with one million of random records; we have measured and compared the time required to execute the script on both SLC 4 and SLC 5 installations. On this table, two tests have been performed. At first, we have measured time needed to write all the records from the table into a disk file. We have used a *mysql_dump* client program to dump the table data. Then we have measured time required to execute the following query:

Listing 3 Test query

```
SELECT param, val, tag, sum(x) as S1, sum(y) as S2, sum(z) as S3,
                        sum(t) as S4 FROM test_tbl
                        GROUP BY param, val ORDER BY NULL;
```

Note that at this phase we have not created any indexes on this table. Examination of the query by the Explain statement reveals that temporary tables are created during evaluation of this query. By adding proper indexes it would be possible to remove the need to create the temporary table and consequently to significantly optimize the query execution time. However, the Explain statement and query optimization techniques are covered in the following sections of this chapter. Finally, we have deleted the content of the table using the Delete From statement and repeated the test case for the table with 2 and 4 million of rows.

Results of the test

The results of the tests are summarized in Table 4.2. The results demonstrate that our assumption that using a combination of more recent version of the Scientific Linux CERN distribution and MySQL server would increase the performance of the database service has been correct. For the larger tables, the time required to fill the table improved by approximately 29 %, time required to evaluate the given query improved by 20 %, and time required to dump the table data improved by 15 %.

It can be seen that the times required to fill and dump the table scale approximately linearly with the size of the table which is not case of the time required to evaluate the query. This effect is caused by the temporary tables that are created during the query evaluation. For smaller tables, the corresponding temporary table fits in a memory buffer. If the size of the temporary table exceeds the size of the memory buffer, the temporary table is created on the disk which cause the delay and observed non–linearity.

Test	**Insert**		**Select**		**Dump**	
Size	SLC 4	SLC 5	SLC 4	SLC 5	SLC 4	SLC 5
1 000 000	123	96	37	36	9	7
2 000 000	251	196	117	102	21	13
4 000 000	493	386	346	289	41	36
SLC 4: MySQL 4.1.22, PHP 4.3.9, Linux 2.6.9						
SLC 5: MySQL 5.0.77, PHP 5.1.6, Linux 2.6.18						
Hardware: Qemu single core CPU at 2.8 GHz, 1 GB RAM						

Table 4.2: Comparison of different versions of the MySQL server, times are in seconds

Besides the performance improvements, several new features have been implemented in the release 5.0 of the MySQL software. The new features include an implementation of the information schema database that contains metadata about databases on server, instance manager that can be used to start or stop the server, or a new fixed point arithmetic library. MySQL 5.0 has also added a limited support for triggers and stored routines. Additionally, new storage engines Archive and Federated have been included and performance of the InnoDB storage engine has increased in the MySQL 5.0. Furthermore, additional features such as partitioned tables, event scheduler, row based replication, or tables with server logs have been added in the subsequent 5.1 release of the MySQL server. Since the changes between version might cause issues with compatibility, it has been required to verify the integrity of data on new servers after the migration.

2.2 Comparison of storage engines

After comparing of different releases of the MySQL database, we have evaluated various *storage engine* supported by the MySQL server software. Storage engines are modular, they can be dynamically loaded into the running server by calling the Install Plugin statement and removed from the server by the Deinstall Plugin statement. List of currently supported engines can be displayed by calling the Show engines statement. The storage engine of the table can be specified directly as part of the Create Table statement. For example, the following statement

```
CREATE TABLE test(id INT NOT NULL) ENGINE=MEMORY;
```

will create the table *test* in the Memory storage engine. If no engine is specified, the table is created in the default engine. In one logical database, different tables can be stored in different storage engines. It is possible to modify an engine of the existing table by the Alter Table statement, for example:

```
ALTER TABLE test ENGINE=MyISAM;
```

statement will convert the *test* table into the MyISAM engine. If the desired storage engine is not supported by the server, a warning will be issued and the table will be converted into the default storage engine.

The storage engine architecture shields the client applications and Application Programming Interface (API) from the low level implementation details of the storage. Different engines serve different purposes, some are targeted for a highly available system, some for a transaction processing, some for an archiving. However, set of routines and interfaces common to all the storage engines is defined; this minimizes a need of changes in the client code required by the change of the underlying storage engine.

The storage engines can be divided into two groups: *transaction* and *non–transaction* engines. The transaction safe tables (i.e. tables created in a storage engine that supports transactions) are much safer than the non–transaction safe tables. In case of a server crash, it is possible to restore data from the transaction safe tables either by an automatic recovery, or from the transaction logs. If an update fails, all changes to the transaction safe tables are reverted; changes can also be reverted by a ROLLBACK statement. Additionally, more SQL statements can be combined and accepted together by the COMMIT statement. On the other hand, a transaction processing imposes a performance overhead, thus the non–transaction safe tables are usually much faster and also the storage requirements are lower. The InnoDB engine is the most commonly used transaction storage engine, while the MyISAM is the most common non–transaction storage engine.

Besides the support for the transactions, the storage engines differ in more details. For example, some engines can compress data, the others support foreign key integrity constraint. Also, different engines provide different support for the table indexing. Some engines can lock a single row, while other engines can only lock the entire table. Most of the engines stores the table data into a disk files, however some engines can store data into a memory heap, or into the cluster. Some engines support caching of indexes and queries. All these aspects needs to be considered when choosing an appropriate storage engine that will be used for a specific applications.

Types of the storage engines

In the following paragraphs, MyISAM, InnoDB, Archive, Blackhole, CSV, Exampled, Federated, and Memory storage engines are described. More detailed information about these specific storage engines can be found in the documentation, [37].

MyISAM engine

The MyISAM is a default storage engine as of the version 5.1 of the MySQL server. The engine does not support transactions and foreign keys, on the other hand, it supports replication, B–tree and full–text search indexes, query and index caching, data compression, and concurrent inserts. The data values are stored with the low byte first, while the numeric values are stored with the high byte first. The table is portable between platforms provided that the target system supports two's complement signed integers and IEEE format for floating point. MyISAM table can contain up to 2^{32} rows; the limit can be increased to $(2^{32})^2$ if the server is compiled with support for large tables (i.e the `--with-big-tables` flag needs to be passed to the *configure* script during installation). A table can contain store up to 256 TB of data. A table created in the MyISAM engine is stored in three disk files: a file with the FRM extension contains the structure of the table, a file with MYD extension contains the table data, and a file with the MYI extension contains table index.

A table in the MyISAM engine can be stored either in a *static* (fixed length), or in a *dynamic* format. The static format is used when the table does not contain any columns with a variable length (i.e. columns of the types VARCHAR, VARBINARY, BLOB, and TEXT). Since each row has the same length, it is very simple to calculate the position of the requested row. For the same reason, tables in this format are easily recoverable in case of a server crash. Furthermore, tables in static format are less prone to the fragmentation. On the other hand, disk requirements are higher. Rows in the dynamic format have different lengths; the length is stored in a header that is added to each row. The header also contains information about empty strings and zero values stored in the row. A row can become fragmented if it is extended during an update, thus a defragmentation may be necessary. Since each row uses only as much space as it is required, the dynamic format is more storage space efficient than the static format. Additionally, by using the *myisampack* tool, it is possible to compress the MyISAM tables in both static and dynamic format. After the compression, the table is in a read only mode, however it can still be dropped and emptied. A packed table can be uncompressed using the *myisamchk* tool. This tools can be also used to repair and optimize the MyISAM tables.

InnoDB engine

The InnoDB is storage engine that supports the transactions that fulfill the *atomicity*, *consistency*, *isolation*, and *durability* requirements (ACID). The atomicity guarantees that in an atomic transaction (that consists of several operations) either all operations are executed, or none operation is executed. The consistency requirement means that each transaction brings a database from a valid state into another valid state. The isolation requirement defines that a change performed by one operation is visible to another concurrent operations. The durability guarantees that when a transaction is committed, it remains committed regardless of crashes, power failures, or other errors.

The InnoDB storage engine supports query and index caching, foreign key referential integrity, data compression, replication, B–tree indexes. On the other hand, hash and full–text search indexes are not supported by the engine. In contrast to the MyISAM tables that support locking on the table level, the InnoDB engine supports locking on the row level. An InnoDB table can store up to 64 TB of data. In contrast to the MyISAM engine that creates separate files for all tables, the InnoDB uses a concept of the *tablespaces* that contain data and indexes for multiple tables. Together with a tablespace file, a file with *transaction log* is created. It is possible to distribute a tablespace and a log on different disks to improve the performance. The data and log files are binary compatible between the major platform that have the same floating point format. A table in the InnoDB engine can store rows in either *compact*, or *redundant* format. The format can be selected when creating a table; if none is specified, the compact format is used. The tables with rows in the compact format are smaller by approximately 20 %, however some operations on these tables might be slower.

Archive engine

The ARCHIVE storage engine is used for storage of large amount of data. The engine does not support transactions, indexes, and modifications of rows. The INSERT and SELECT operations are available, however the DELETE and UPDATE operation are not permitted by this engine. New rows are inserted into a compression buffer that is flushed when necessary (e.g. when the SELECT is called upon the table). The engine uses the lossless compression based on the *zlib* library. During a SELECT statement, a full table scan is performed, the rows are uncompressed on demand. Each table in the ARCHIVE engine is stored in a multiple files: a file with the FRM extensions holds the table structure, while the table with the ARZ extension contains the table data. The size of the data file is limited only by the available disk space.

Blackhole engine

The Blackhole storage engine accepts inserted rows but does not store them. However, if the replication is enabled, the statements are written to the binary log and replicated to the slave servers. The engine is aware of transactions, only committed transactions are written into the binary log. The engine supports all types of indexes, the Select statement on the table always returns an empty set. On the disk, each Blackhole table is represented by a single FRM file that contains the table structure. Due to the fact that all operation on the Blackhole table have no effect, it is possible to use the tables in the Blackhole engine for finding performance bottlenecks and evaluation of the overhead caused by the binary logging.

Csv engine

The Csv engine stores table data to the text files in the Comma Separated Values format (the CSV extension). This file can be viewed and edited by any spreadsheet application such as MS Excel or Openoffice Calc. The table structure is saved in the FRM file, while the CSM file contains state of the table and number of rows. The Check statement verifies the integrity of the table (number of columns, matching separators, ...). If the check fails, the table is marked as crashed and must be repaired by the Repair statement. However, all rows beyond the first corrupted row are lost.

Example engine

The Example storage engine does nothing. The engine is a stub that can be used as a starting point when implementing a new custom storage engine. A table in the engine is represented by a single FRM file that contains the table structure. No data can be inserted into a table in the Example engine, all selects return an empty set. Indexes are also not supported.

Federated engine

The Federated engine enables accessing data from a table on a remote MySQL server without a need of using cluster technology or replication. At first, the table needs to be created on the remote server. The remote table consists of the FRM file with the table structure and a file with table data; the remote table is stored in normal engine such as a MyISAM or an InnoDB. Then a table with the same structure in the Federated engine is created on the local server. During creation of the local server, a connection string to the remote server needs to be passed to the Create table statement. The engine supports the Insert, Update, Delete, Truncate, and Select statements; the queries that would select, insert, update, or delete rows on the local table are send for execution to the remote server where the corresponding operation is performed on the remote table and the corresponding result is returned to the local server. The Alter Table statement is not supported, in fact, the Drop Table is the only supported data definition language statement. When using the Drop Table statement, only the local table is removed, the remote table is unaffected. A table in the Federated engine does not support indexes since they are handled by the remote tables. Engine does not support transactions, the replication is supported on the other hand.

Memory engine

The tables created in the Memory engine store its data in the random access memory (RAM) of the server; table structure is stored on the disk in the FRM file. Since the content of the RAM is lost during every restart or crash of the system, this engine is mainly used for temporary tables and applications that deal with non critical data such as session management on web servers. The maximal size of the memory table is defined by the *max_heap_table_size* system variable. The memory for the rows is being allocated in small blocks. When deleting individual rows, the memory is not released. However, new rows can reuse the memory because rows use have fixed length. The memory is released in three situations: either table is dropped, or all rows are deleted, or the table is rebuilt by the ALTER TABLE statement.

The replication is possible with this engines, however the slaves are not aware that the table on master is truncated in case of restart which causes desynchronisation. Engine supports hash and B–tree indexes, it does not transactions. The MySQL server uses temporary memory tables internally during evaluation of some queries. However, an internal table can be swapped to the disk, if its size exceeds the size defined by the *max_heap_table_size* system variable. The MEMORY tables created by users are never swapped to the disk.

Design of the test

Under normal data taking conditions many processes frequently insert new records into the online database, therefore we have tested how many rows can different engines store per second. Additionally, we wanted to verify whether some storage engine is capable of storing rows at rate of 1 MHz. In the first test, we have measured time required to insert 10 million rows into a test table with one integer column. The test has been performed on the dual core processor Intel Core2 Due T9660 running on 2.8 GHz supported by 4 GB of the system memory. A 64b operating system based on the linux kernel in version 2.6.35 with multiprocessor support enabled has been installed on the SATA disk. As a database software, a binary distribution of the MySQL 5.1.50 has been used.

To utilize both cores of the CPU, we have prepared a multithreaded client application based on the C application programming interface provided by the client library of the MySQL; one thread fills the test table with all the integers from the interval [0; 5 000 000), the second thread covers the interval [5 000 000, 10 000 000).

The code of the client application will be briefly commented; at first the `fork` function is called to create a child process:

```
pid_t pid = 0;
pid = fork();
```

The variable `pid` (pid = process identification number) is used to distinguish a parent and a child threads; in the parent thread, the variable remains 0, in a child thread, it has a non zero value. In the parent thread, a current timestamp which will be used to calculate the elapsed time is taken. After that, both threads attempt to connect to the MySQL server:

```
MYSQL *conn = mysql_init(NULL);

if (conn == NULL){
 printf("Error %u: %s\n", mysql_errno(conn), mysql_error(conn));
 return 1;
}
if (mysql_real_connect(conn, "localhost", "root", "pass", "test",
                        0, NULL, 0) == NULL)
{ /* error handling */}
```

At first, a `MYSQL*` handle that will be used for communication with the database is initialized in the `mysql_conn` function. If the initialization fails, the function returns the `NULL` value; in this case an error is printed on the standard output and application is terminated. The handle together with a server address, user name, password, and working database is passed to the `mysql_real_connection` function that tries to establish a connection to the database server. If the connection fails, the function returns the `NULL` value and the application is terminated. On the other hand, if the connection is successfully established, the threads can fill the table with data:

```
int i;
char query[256];

if(pid == 0){ /* parent thread */
 for(i = 0; i < 5000000; i++){
   sprintf(query,"INSERT INTO speedtest VALUES(%d)", i);
   mysql_query(conn, query);
} else { /* code for the child thread */
```

The `pid` variable is used to decide if a parent or child code should be executed. Then, the query is prepared. The `mysql_query` function uses the `MYSQL*` handle to execute the provided query. When the table is filled, the connection to the database is closed by calling the `mysql_close` function. Finally, in the parent thread, an elapsed time is calculated and printed on the standard output.

Engine	*Data directory on disk*	*Data directory in memory*
ARCHIVE	13.9 s	14.2 s
BLACKHOLE	8.7 s	8.8 s
CSV	32.2 s	19.3 s
INNODB	33.0 s	39.7 s
MEMORY	9.3 s	9.2 s
MYISAM	13.7 s	12.3 s

Table 4.3: The speed of the *INSERT* operation on different storage engines

Results of the test

At first, a performance of the MyISAM storage engine has been evaluated. It has taken 288.8 s (which corresponds to a rate of approximately 35 kHz) to fill the table. Since this result has not corresponded to our expectations, we have analyzed the behaviour of the server. Using the `iotop` utility [25] we have discovered that the disk usage was minimal during the test. Furthermore, MySQL server consumed 4 times more CPU power than the client application. We have concluded that the bad performance had been caused by overhead associated with inserting of rows into the table. We have added support for *bulk inserts* into the client application; in the bulk mode, one query can insert multiple rows which reduces the overhead significantly. Indeed, by storing 25 rows per query, we have been able to reduce a time required to fill the table to 14 s (approximately 650 kHz); the server process has been writing to disk at average speed of 5 MB/s during the test. After the test, the table was dropped from the database using the DROP statement and the test was repeated for Archive, Blackhole, CSV, InnoDB, and Memory storage engines.

We have also tried to move the directory used by the MySQL server to store databases from the disk partition with the *ext3* file system to the virtual RAM drive with the *tmpfs* file system. At first, the RAM drive has been created by launching the mount application on the root console with the following arguments:

```
mount -t tmpfs tmpfs -o size 1024m /mnt/ramdisk
```

The command will use 1 GB of the RAM to create the ramdisk and will mount it under the `/mnt/ramdisk` directory. Then, we have assigned this address to the variable `datadir` that defines the directory that is used by the MySQL server to store database. Note, that change of the *datadir* variable requires restart of the server.

The results of these tests are summarized in Table 4.3. The best performance has been achieved with the Blackhole engine which is not surprising as this engine discards data instead of storing them. For the same reason, the engine cannot be used for the needs of the data acquisition. The second fastest engine in the test, the Memory engine is also not suitable candidate for deployment on the Compass experiment because it stores data only temporarily. The Archive engine also cannot be used because it does not support indexing and modifications of already inserted data. However, old databases with data from previous years could be converted into this engine to save disk space. We rejected the Csv engine as it does not support indexing. Thus only MyISAM as an example of transaction engine and InnoDB as an example of non-transaction engine remained our candidates after this first series of tests. The MyISAM has been approximately three times faster than the InnoDB engine, on the other hand the InnoDB engine supports fully ACID compliant transactions. We have performed additional tests to select the most suitable engine for the needs of the Compass online database. From Table 4.3, one can also see that only CSV and MyISAM engines can benefit from moving the database directory to the RAM disk.

Language	*Speed [s]*	*Ratio*	*Comments*
C	41.2	1.00	uses MySQL client library, gcc 4.5.1
Pascal	42.2	1.02	Free Pascal 2.4.0
PHP	45.9	1.11	PHP 5.3.3 (interpreted on command line)
C++	46.3	1.12	uses Qt framework 4.6.3
Java	59.4	1.44	JDBC MySQL driver, java version 1.6.0/OpenJDK
Python	87.3	2.12	Python 2.6.5
Perl	503.4	12.22	uses Perl DBI, Perl 5.12.1

Table 4.4: The comparison results of the selected programming languages

In connection to the above described test, we have been asked to verify whether the MySQL server is capable of inserting rows at the rate of 1 MHz. The results in Table 4.3 show that the speed is feasible with the Memory, Archive, and MyISAM engines provided that a buffering is implemented on the client side and clients can insert rows in a bulk mode. By using the MySQL server in an embedded mode, additional performance improvement would be possible. In the embedded mode, MySQL server runs as a part of client application. This greatly increases the performance as there is no overhead caused by a network communication between the client and the server. Also, the management of the embedded application is simpler compared to the client/server application. On the other hand, the embedded server is available only from the C and C++ language as the embedded library is written in the C/C++ languages.

Libraries that provide a connection to the MySQL server have been developed for many programming languages thanks to the popularity of the MySQL software. Some of these libraries are provided directly by the MySQL developers, other are provided by third parties. We have compared a speed of the insert operation of the C, C++, Java, Pascal, PHP, Perl, and Python client libraries. Using these libraries, we have developed single threaded applications that insert 10 million rows (in a bulk mode, 10 rows insert per each query) into a table in the MEMORY engine. The test results are summarized in Table 4.4. As expected, with the exception of the PHP language, the interpreted languages were the slowest in the test. The C application was fastest in the test, perhaps because the client library is developed directly by the MySQL developers. However, the comparable speed was also achieved by the Pascal, PHP, and C++ applications. On the other hand, the Perl application was more than 12 times slower than the C application. The performance drop may be explained by the fact that the Perl language is targeted to the processing of text in contrast to the other general purpose languages that participated in the test.

Comparison of the MyISAM and InnoDB storage engines

For the further comparison of the performance of the MyISAM and InnoDB engines, we have used the table *test_tbl* whose structure is described in Listing 1. This time, we have created the table in the MySQL 5.1.46 installed on the above described physical hardware. We have measured the time required to execute the query from Listing 3 for different numbers of rows in the table. We have already demonstrated that without indexing the table, the temporary table needs to be created and sorted during evaluation of the query. For smaller numbers of rows the time required to process the query scales linearly with number of rows (see Figure 4.3). Starting from approximately 5 million rows, the temporary table does not fit into a memory and must be converted to disk table which significantly reduces the speed of the query execution. The query on table with more than 10 million rows consumes also the swap memory and the client is killed by the out of memory mechanism (*oom-killer*) of the operating system.
We have used the information provided by the EXPLAIN statement to propose an index (or indexes) that would remove the need to create the temporary table during the query evaluation. Unfortunately, such an index would be based on data from all the columns that are included in the query. The key length would exceed 1000 B which is the maximal length supported by the MyISAM engine. Thus, we have tried the following changes to the structure of the table:

- At first, we have tried to include only prefixes of the column values into the index. Unfortunately, this index could not be used.
- We have tried to replace the FLOAT columns to the DECIMAL data types, however, the performance remained the same.
- According to the documentation [37], the overhead associated with the VARCHAR type exceeds the overhead of the CHAR type, thus we have replaced the VARCHAR(255) columns by CHAR(255) type in the table structure. However, since the table contains strings with random length, replacing the variable length strings by the fixed length strings five times increased the size of the table. Consequently, the query execution time increased from 6 to 9 minutes for 2 500 000 rows.
- We have also tried a feature implemented in the new version of the MySQL server software: partitioned tables. We have divided a table into 300 partitions according to a hash of the

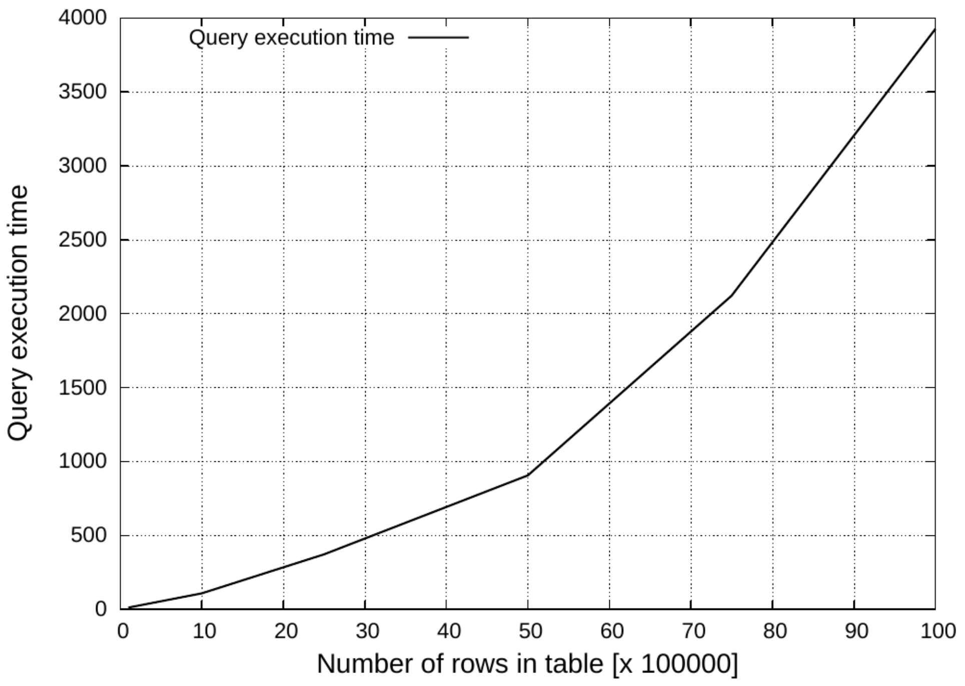


Figure 4.3: Query evaluation time

primary key *id*. Partitioning did not improve the behaviour, probably because all partitions were stored on the same disk.

- We have reduced a length of the VARCHAR columns in order to be able to create the key. On this modified table, the proper indexing reduced the query evaluation time five times. By reducing the length of the VARCHAR columns from 255 to 25 characters, the size of the table with 5 million rows decreased from 1.5 GB to 400 MB. The time required to process the query decreased from 935 s to 385 s. By adding an index to the modified table, it was possible to increase the query evaluation speed five times. On the other hand, indexing the table increased a time required to fill the table and greatly increased the disk space required to hold the table.

We have achieved the best improvement by reducing an amount of data that needs to be stored into the database table. Reducing the amount of data is also recommended by the MySQL manual [37].

Operation	*MyISAM*	*InnoDB*
Filling of the table	373 s	1945 s
Query without index	906 s	2042 s
Indexing	N/A	10740 s
Query with index	N/A	25 s

Table 4.5: Comparison of performance of basic operation on the test table with 5 million rows in MyISAM and InnoDB storage engines. Creating index on the MyISAM was not possible due to the limitation on the key length.

After the evaluation of the MyISAM storage engine, we have used the ALTER TABLE statement to convert the table into the InnoDB engine. This engine does not limit a length of the key, thus it was possible to create index based on data from all columns of the table. Although the query execution on the table without an appropriate index was twice slower on the table in InnoDB engine than on the table in the MyISAM engine, the indexing reduced the time by two orders.

The test results are summarized in Table 4.5. Although, it is possible to create index on the InnoDB engine, the operation is very slow. Additionally, the overhead caused by the transaction handling negatively influences the speed of the insert operation. On the other hand, the query on the indexed table is executed almost 40 times faster than on non indexed MyISAM table. To sum it up, the InnoDB should be used either in the applications that require transactions, or when the speed of selects is more important then speed of inserts. On the contrary, the MyISAM storage engine should be preferred in the database systems that are characterized by high frequency of insert operations and the transactions are not required. Since the database service of the Compass experiment consists of multiple clients that are frequently inserting new rows into multiple tables and occasional lost of some rows is not critical, we have decided to use the MyISAM storage engine for the new database architecture. However, the table with historical data could be converted into the Archive storage to save disk space. Furthermore, these tests have confirmed our assumption that the amount of the system memory is very important for the database servers. Thus, the new database servers are equipped with 16 GB of RAM in contrast to 3 GB RAM on the original *pccodb01* server and 2 GB on the original *pccodb02* server, cf. Table 4.1.

3 New database architecture for the Compass experiment

The proposed new database architecture has been presented and approved by the Compass collaboration at the Frontend electronics meeting, [11]. Unfortunately, only three new servers have been delivered, thus the proposal had to be updated. Consequently, the implemented database architecture merges some features of the existing (as in 2009) architecture with some features of the original proposal.
In the implemented architecture, two physical servers called *pccodb11* and *pccodb12* are used to power the database service. As in the original architecture, these servers are kept synchronized by the means of the master–master replication. One physical server is also replicated into the CERN computing center which can be regarded as a form of a geographical backup. The third physical server called *pccodb10* is used mainly as a proxy server: it receives the requests from the clients and dispatches them to the database servers. Furthermore, this server also manages additional services such as a HTTP server that were originally deployed on the *pccodb01* server. A new monitoring tool Nagios has been installed on the *pccodb10* server; it watches the state of the physical servers and in the case a problem is detected on one of these servers, it automatically tries to reprogram the behaviour of the proxy software to reroute the traffic to the remaining server and it also notifies an operator about the incident. A regular database backup has been scheduled on one of the physical servers. Continuous monitoring, replication, and backups should guarantee the required high availability and reliability of the database service. To guarantee that the migration to new architecture is transparent to the clients, we have used the same virtual address *pccodb00*. In the new architecture, it always points to the proxy server *pccodb10* which decides where to redirect the requests issued by clients.

In the spring 2010 when the migration was performed, all servers participating in the data acquisition at COMPASS were powered by the Scientific Linux CERN in version 4. However, given the fact that the official support of the SLC 4 would end in December 2010 and the test results described in previous section of this work, we have decided to install SLC in version

5 on the new servers. For the same reason, we have upgraded the MySQL database software from version 4.1.22 that had been used on the old server to version 5.1.45 which was the most recent stable version when the migration started. However, many changes were incorporated in the new versions of MySQL and some of these changes could affect the compatibility. Thus, as an important part of the migration process, we needed to carefully verify that the data were transferred unchanged to the new servers. As a proxy software, we have decided to install the *MySQL Proxy*. For the web server, we have used the *Apache* software with enabled server side scripting based on the PHP language. Finally, the *Nagios* package has been selected as a monitoring software.

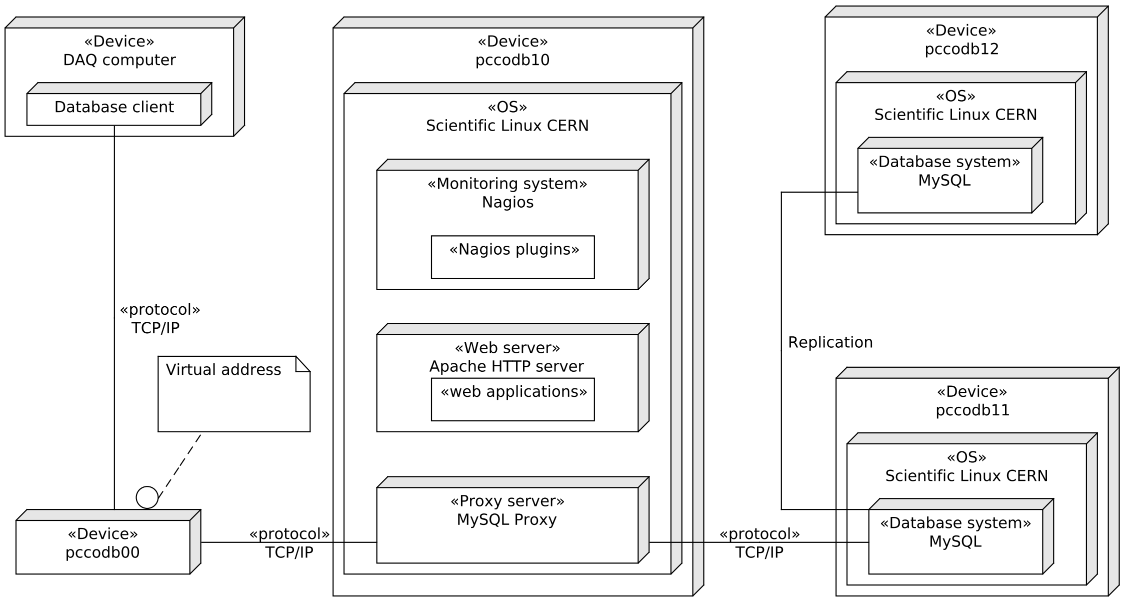


Figure 4.4: Implemented database architecture

The process of a migration to the proposed database architecture is a complicated; it can be divided into several steps that will be discussed in more details in following paragraphs of this chapter:

1. Installation of the operating system on the new servers and connecting them to the Compass internal network.
2. Installation and configuration of the MySQL software on the new *pccodb11* and *pccodb12* servers including performance tuning, logging, and establishing the master–master replication between *pccodb11* and *pccodb12* servers.
3. Backup of the data and configuration on the old database servers, export of table structure and table data from the old servers.
4. Import of the users, user privileges, table structure, and table data to the new servers.
5. Verification of the compatibility between the old and the new servers.
6. Installation and configuration of the MySQL Proxy, Nagios, and Apache software on the *pccodb10* server,
7. Disconnecting the old servers, connecting the new database architecture by reprogramming the virtual address to point on the proxy server.
8. Continuous monitoring and maintenance of the new database architecture.

The following two subsections contain technical details concerning installation and configuration of the operating system and database software and should serve as a manual for future administrators of the COMPASS database.

3.1 Operating system installation and configuration

The new servers have been delivered without an operating system installed, thus as a first step during implementation of the new database architecture, we needed to install the system on these servers. As already discussed, we have decided to install the 64–bit version of the *Scientific Linux CERN 5* (SLC 5). The system is based on the *Scientific Linux* distribution that is being developed by joint effort of the CERN and Fermi National Accelerator Laboratory (Fermilab), [34]. In turn, the Scientific Linux is basically a *Red Hat Enterprise Linux* (RHEL) distribution recompiled from the source codes. The aim of the Scientfic Linux is to have a common system compatible with RHEL and usable for various experiments. A SLC contains modifications and packages that enables integration of the system into the CERN networking infrastructure including the distributed file system AFS. Additionally, a custom repositories with a CERN software and libraries are available during the SLC installation. However, the SLC still remains compatible with RHEL.

As an RHEL derivative, the SLC uses the *RPM* packages with the *yum* package manager. The system installation can be started from several types of media: hard disks, optical disk, or network. As for the system installer, the *Anaconda* program is included. As Anaconda is written in the C language, it is available for a wide range of hardware platforms. The program offers a text–mode as well as a graphical installation; it also provides an automated installation through the *kickstart* configuration files. This method will be described in the following chapter as we have used it during an installation of the new run control machines in a remote control room of the experiment.

As the new server are not equipped with an optical drive, we have decided to start the installation from the USB flash disk and to perform the network installation. In order to start the system from the flash drive, the flash disk needs to made bootable and formatted. To make a disk bootable, one needs to copy boot instructions in a machine code into the *master boot record* (MBR) of the disk. During the start up of the computer, the *primary loader* called *Basic input/output system* (BIOS) loads these instructions into the memory and instructs the processor to execute them. Since the MBR is stored in first 512 bytes of the disk, the instructions are very limited; typically a *secondary boot loader* is copied from the medium into the memory and executed. As a secondary loader, we have used the *syslinux* which is designed to load the linux kernel from various types of media: flash disks, optical disks, network. The syslinux installation is distributed together with a pre–prepared binary image of the MBR in the `mbr.bin` file. To install this image into the first 512 bytes of the disks, one can employ the *dd* tools which is one of the core utilies of the GNU/Linux operating system. Suppose that the flash disk is recognized as a `/dev/sdX` device by the linux kernel (the corresponding device node can is printed into a kernel log when the flash disk is inserted into a USB drive), then:

```
dd if=mbr.bin of=/dev/sdX
```

command issued with the root privileges copies the image (the *if* parameter) onto the device (the *of* parameter). The *dd* tool serves for low level copying and conversion of raw data; its usage is documented in the corresponding manual page (*man dd*).

After the preparation of the MBR of the flash disk, a primary partition needed to be created by the *fdisk* program, formatted to the FAT 16 file system using the *mkdosfs* program. Finally, the syslinux boot loader could be installed on the formatted partition:

```
fdisk /dev/sdX
mkdosfs /dev/sdX1
syslinux /dev/sdx
```

At this stage. the flash drive is prepared and it is possible to transfer data from installation image to it. From the SLC website, we have downloaded a minimal bootable image called `boot.iso`. Purpose of this image is to launch a network installation. During the process of network installation, all selected software packages are download from a network repository using an HTTP or an FTP protocol. Since the packages in a repository are being regularly updated, a network installation leads to fully updated system (in contrast to an installation from the DVD image).

In order to copy content of the ISO image to the flash drive, it is necessary to use the *mkdir* tool to create empty directories called *mount points* and the *mount* tool to add ("mount") both medias into the directory tree:

```
mkdir /tmp/usb
mkdir /tmp/iso
mount -t vfat /dev/sdX1 /tmp/usb
mount -o loop boot.iso /tmp/iso
```

Then, the directory structure contained within the ISO image can be recursively copied to the flash disk using the *cp* command.

```
cp -r /tmp/iso /mnt/usb
```

As the `boot.iso` image is intended to be burnt on an empty CD-ROM, a small modification of the directory structure needed to be created in order to successfully boot from the flash disk. The CD-ROM contains the secondary boot loader binary stored under the `isolinux` folder whereas flash drive expects the boot loader binary in its root directory. Additionally, boot loader configuration for CD-ROM is saved in the `isolinux.cfg` file while flash disk requires the `syslinux.cfg` configuration file. Both issues can be easily resolved using the *mv* command that serves for moving and renaming of files and directories:

```
cd /mnt/iso
mv isolinux/* .
mv isolinux.cfg syslinux.cfg
```

Installation process should be started after the system reboot provided that the bootable flash drive is inserted in a USB port and the primary boot loader BIOS is configured to boot from removable media. The primary loader loads and executes the secondary loader syslinux which in turn prepares a computer for start of the installation program Anaconda that guides user through the installation process step by step. At first, we have tried running the installation in a graphical mode; unfortunately, the installation process was aborted due to a software bug. Thus, we had to restart the installation process, this time in a text mode. The text mode is based on the *ncurses* library that uses semigraphical characters to draw interactive dialogs on screen.

In the first step, user selects system language and the preferred keyboard layout that should be used during installation process. Then, it is required to configure network interface and enter location of the package repository. The installation program detects wide range of network cards, it can use a static configuration, as well as a dynamic configuration based on the *Dynamic Host*

Configuration Protocol (DHCP). Additionaly, the Anaconda supports both version 4 and 6 of the Internet Protocol (IPv4, IPv6). The program supports various protocols for downloading data, we have decided to use the *File Transfer Protocol* FTP. When the network configuration is completed, an image of second stage is downloaded from the specified repository and installation continues by definition of a hard drive partitioning.

Each of the new servers is equipped by several hard disks with a capacity of 1 TB that are configured as *RAID 5* arrays. The RAID (redundant array of independent/inexpensive disks) technology combines multiple disks into a single logical unit. The aim of the technology is to increase a performance and reliability of the storage by distributing data across multiple physical disks and by introducing the redundancy. The technology distinguishes several levels. RAID 5 requires at least three disks. Data is stripped between these disks and also the parity data is distributed between all these disks. The RAID 5 is able to work, although in degraded performance, if one of the member disks fails because the data can be reconstructed using the parity data from remaining disks. When a faulty disk is replaced, the system automatically rebuilds the array. The technology may be implemented on both hardware and software levels.

Partition	Mount point	File system	Capacity
sda1	/boot	ext3	100 MB
sda2	/tmp	ext3	42 GB
sda3	/	ext3	8 GB
sdb1	N/A	swap	32 GB
sdb2	/var	ext3	32 GB
sdb3	/data	ext3	4.2 TB

Table 4.6: Defined partitions

Since the new servers contain a hardware RAID controller, the installer program Anaconda sees the disk array as two disks with capacities of 50 GB (recognized as a `/dev/sda` device) and 5 TB (the `/dev/sdb` device). However, Anaconda also supports creating of software RAID arrays. In Unix and Unix–like systems, it is a good practice to install a system on several disk partitions. Separating data of the operating system from the data produced and managed by user processes simplifies system maintenance including backups and reinstallation of the system. Although installer program can automatically partition hard disks, we have defined a custom partitioning scheme that is summarized in Table 4.6. The `/boot` partition contains the linux kernel and a configuration files of the boot loader. We have decided to create a separate partition for the `/tmp` directory. By convention, this directory is used for storage of temporary data. Filling the temporary directory by some faulty process would cause a crash of the operating system. By moving the temporary directory on a dedicated partition this danger is eliminated. For the same reason, we have created a separate partition for the `/var` directory which serves as a storage for data (logs, web pages) produced by various process. Additionally, we have created a partition for the `/data` directory to store the database tables and logs. We have decided to format these partition using the *Linux Extended* (ext3) file system which supports journaling and is considered to be stable. Finally, a dedicated `swap` partion has been created; its size has been set as a double of the amount of the RAM.

When the defined partitions are created, the installation process continues with a configuration of the boot loader. As a derivative of the RHEL distribution, the SLC uses the *Grand Unified Boot loader* (grub). Anaconda automatically configures the loader to boot the SLC, user can specify where to install it. We have decided to install the boot loader into the Master Boot

Record of the hard disk. In the following steps of the installation, the network interface needs to be configured: at first, user decides if support for the IPv4 and IPv4 should be enabled, then the IP address and adresses of DHCP servers are defined, and finally a hostname of the machine is chosen. Then, user selects a timezone in which the machine is located. Before the start of the installation of packages, user must enter the root password. For a compatibility with the old database servers, we have kept the password unchanged. Finally, it is possible to define a purpose of the machine; this information is used to select groups of packages that should be installed. We have selected a predefined profile called *Server*; additional packages can be installed after the system installation using the yum tool. The selected packages are download from the network repository and unpacked into the directory tree. After this step, the machine needs to be restarted in order to complete the installation process.

Post–install tasks

After the restart of the machine, system needs to be configured. Installation program Anaconda helps user with setting up of the firewall *iptables*, *Security Enhanced linux* (SElinux), or time zone. Because database servers are part of the Compass internal network, we have decided to disable SELinux. The *ntsysv* utility has been used to disable unnecessary service to reduce system load. We have decided to disable the *Bluetooth* service, the printing service *cups*, or the daemon for sending mails which are not required on the database server. Additionally, we have disabled automatic system update which is normally executed at each boot of the system as the automatic update would contribute to the dead time of the data acquisition system. On the other hand, we have enabled several other services such as a system scheduler *cron*, system logger *syslog*, network file system *nfs*, or SSH server *sshd*. Scientific Linux CERN boots into the *runlevel 5* after the installation. Runlevel 5 represents multiuser environment with X–Window server. MySQL server does not need X–server, therefore the default runlevel has been changed to level 3 (i.e. full multiuser mode). To change the default runlevel, one has to edit a configuration file called `inittab` and change line containing *initdefault* to

```
id:3:initdefault:
```

Then, it has been necessary to install packages for building software from the source codes: *GNU C compiler* GCC, standard C library, and build tools. Scientific Linux CERN comes with package management tool called *yum*. Yum can install, update, and remove package, moreover it handles package dependencies and downloads packages directly from the repository. To install package (or packages), simply issue the install command, for example:

```
yum install gcc gcc-c++ glibc glibc-common autoconf automake
```

The yum tool can be also used to remove (the *remove* command), find (the *search* command), and query (the *info* command) packages. Yum is an advanced tool, it can manage package repositories, it checks the dependencies of the packages. Furthermore, using the update command, yum is also able to update the entire system.

Then, a configuration file `/etc/hosts.allow` needed to be edited in order to enable network connection to the MySQL server. This file contains a list of machines that are allowed to access local network services. To enable all computers from the CERN network to access the database server, one needs to put the following line into `hosts.allow` file:

```
mysqld: .cern.ch
```

Multiple values can be appended to one line, one can also specify the hosts by IP addresses. To increase security, we have only allowed connections from the both database servers *pccodb11* and *pccodb12*, from the proxy server *pccodb10*, from the computers from the COMPASS internal network, and from one computer from the CERN computing center that acts as an additional replication slave.

Finally, several steps needed to be performed in order to include the new servers into a data acquisition network. At first, we needed to create a special user account *daq* that is used for the common tasks. As with a root account, we have kept the password for this account unchanged for the compatability reason. Then, a shared network directory provided by the file server *pccofs01* needed to be connected into the `/online` mount point. In the original architecture, the *infoLoggerServer* process of the DATE package had been installed on one of the database servers. Unfortunately, this process failed to start under 64–bit operating system. After a discussion with a data acquisition expert, we installed the *infoLoggerServer* on one of the event building servers as they were powered by 32–bit system.

At this stage, the installation and configuration of the system is finished and the system is prepared for MySQL installation. Complete installation log can be found in the `install.log` file in root's home directory (i.e. `/root`).

3.2 MySQL installation and configuration

The official repository for the SLC 5 distribution offers an RPM package with a prebuilt MySQL software in version 5.0. To benefit from the new features of the version 5.1 of the MySQL, that include support for the table partitioning, the installation had to be based on building the source codes. The compilation from the source codes requires more effort and time then installing a binary package. On the other hand, it is often possible to increase the performance and stability of the software by disabling unneeded features. Additional performance improvement can be achieved by instructing a compiler process to produce binaries optimized for given hardware architecture.

In April 2010 when the installation was performed, MySQL 5.1.45 was the latest stable release. The source codes are distributed in a form of *tar* archives compressed by the *gzip* program that can be downloaded from an official MySQL website[1]. This package contains source codes of not only the MySQL server but also of the various client applications. Additionally, documentation and tools required to configure the installation are included in the package. Several steps needs to be taken before a compilation of the source codes can be started. At first, a special user account and group called *mysql* has to be created. This account will be used by both server and client parts of the database. A new group can be created using the *groupadd* tool; the *useradd* tool creates a new user account:

```
groupadd mysql
useradd -g mysql mysql
```

Note, that the *mysql* user is immediately added into the *mysql* group. These commands must be executed with root privileges. Preparation continues by decompressing of the source archive, suppose that it is downloaded into the `tmp` directory:

```
cd /tmp
tar xzvf mysql-5.1.45.tar.gz && cd mysql-5.1.45
```

[1]http://www.mysql.com/downloads/mysql

It is advisable to define and export several environment variables, namely *CHOST*, *CFLAGS*, and *CXXFLAGS* in order to help the compiler to produce executable files optimized for given processor architecture (Intel Xeon in this case). By using a correct compiler options, it is possible to achieve performance increase of up to 30 %, [37].

```
CHOST="x86_64-pc-linux-gnu"
export CHOST
CFLAGS="-fPIC -march=nocona -O2 -pipe -fno-strict-aliasing"
export CFLAGS
CXXFLAGS="${CFLAGS}
export CXXFLAGS
```

The *CHOST* environment variable defines a system architecture, in this particular case a 64–bit system based on the linux kernel. Then, the parameters that should be passed to the C (the *CFLAGS* variable) and C++ compilers (the *CXXFLAGS* variable) are defined and exported. The parameter *march* passed to the compiler defines a type of processor (Nocona is a code name of processor architecture that contains the Intel Xeon). The *fPIC* (Position Independent Code) flag is required for building shared libraries, the *pipe* flag instructs the compiler to use Unix pipes instead of temporary files for the inter process communication which significantly increases the speed of the build process. The *fno-strict-aliasing* flag is used to prevent a compiler from producing unwanted optimizations that could generate invalid code. The *gcc* compiler supports several levels of optimizations that can be enabled by the *O* flag. We have used the second level *O2* that enables almost all optimizations with exceptions of loop unrolling, function inlining, and register renaming. The second level increases time required to complete the compilation, however it also produces faster code. More information can about compiler flags can be looked up in the manual [41].

Listing 4 Configuration of the build process of the MySQL software

```
./configure --with-readline \
            --with-ssl \
            --without-debug \
            --enable-shared \
            --enable-assembler \
            --sysconfdir=/etc/mysql \
            --localstatedir=/var/lib/mysql \
            --datadir=/data/mysql \
            --with-unix-socket-path=/var/lib/mysql/mysql.sock \
            --with-mysqld-user="mysql" \
            --with-extra-charsets=complex \
            --with-embedded-server \
            --with-big-tables \
            --enable-local-infile \
            --enable-thread-safe-client \
            --with-named-thread-libs="-lpthread" \
            --with-plugins=partition,innobase,innodb_plugin
```

The source package of the MySQL includes the `configure` script that searches the system for various components that are required to build the source codes (e.g. compiler, linker) and also run the MySQL software. If the script detects any problems, it generates a corresponding error message. In case of success, it prepares the `Makefile` file which contains instructions needed to build the MySQL server and client applications. By passing additional parameters to the configure script, it is possible to enable (or disable) optional features or change various default variables such as the installation directory. The parameters that have been passed to the script are shown in Listing 4.

Because the server will be used in the production environment, support for debugging has been disabled by the *without-debug* flag in order to increase performance. On the other hand, support for multithreaded client has been enabled. We have enabled plugins that support InnoDB storage engine and partitioned tables. Additionally, we have enabled support for very large tables (up to 1.844×10^{19} rows), secured communication based on the SSL library, more characters sets, and embedded server. We have moved a location of the configuration files into the `/etc/mysql` directory. The table data and replication logs will be stored on a separate partition in `/data/mysql` directory (see above). Server will be running under the *mysql* user account which was created in previous steps. Remaining parameters are described in the official MySQL documentation [37].

If the configure script finishes without errors, it is possible to build and install MySQL server and clients applications by entering the following commands:

```
make
make install
```

At this moment, the installation is finished, however some further actions are required before the the server can be started. Directories for database data, logs, and configuration must be created and owner of these directories must be set to the *mysql* account.

```
mkdir /etc/mysql
mkdir /var/lib/mysql
chown mysql.mysql /var/lib/mysql
mkdir /data/mysql
chown mysql.mysql /data/mysql
```

Then, global configuration file for the server and for various client applications (`my.cnf`) must be provided. Source archive with MySQL distribution contains several templates. Since very high load of the database servers is expected, the template `my_huge.cnf` has been used. At first, this template is copied into configuration directory `/etc/mysql`:

```
cp support_files/my_huge.cnf /etc/mysql/my.cnf
```

We have modified several predefined values in order to improve the performance. The configuration file is divided into several sections. In the *[server]* section, variables that modify a behaviour of the MySQL server are defined. We have removed the *skip-networking* option to enable communication over the TCP/IP. With this option, the communication between client and server is based on local Unix sockets which is much faster than the TCP/IP communication, however clients must be running on the same machine as the server. Then, we have used the *datadir* variable to define a directly that should be used for storing database data. Since we have decided to use the MyISAM storage engine, we have added the *skip-innodb* option that prevents the InnoDB from loading at start of the server. To improve a performance of the MyISAM engine, we have included the *skip-external-locking* option. With this option, the server does not lock files with MyISAM. When using this option, one must guarantee that the MySQL server is the only process that is modifying these files. To enable concurrency support, we have configured the *thread_ concurrency* to double of the processor cores. By setting the *max_ connections* variable, we have defined a maximal number of clients that can be connected to the server at one time. We have also used the *long_ query_ time* and *log_ slow_ queries* variables to enable logging of the slow queries. Knowledge of the slow queries is very important for detecting of the incorrectly indexed tables. The server section of the configuration file also serves for setting up of the replication which is described in Section 3.3 of this chapter.

The behaviour of the SQL shell *mysql* can be altered in the *[mysql]* section of the configuration file. Here, we have added the *no-auto-rehash* option that disables completion of column, table, and database names which makes the *mysql* client to start faster. If required, the name completion can be enabled by typing the REHASH command on the SQL console. Additionally, we have provided a *port* option that defines to which port should the client connect. For the security reasons, we have removed the *password* parameter, thus each user (with the exception of the anonymous user) connecting to the database must enter the password.

In the *[mysqldump]* section, the behaviour of the database backup tool *mysqldump* is defined. We have added the *quick* flag that forces the program to retrieve rows for a table from the server a row at a time which speeds up dumping of large tables. Additionally, we have increase a maximal size of the packet that can be received from the server to 16 MB by adding the *max_ allowed_ packet* variable.

When the global configuration is defined, it is possible to initialize system databaze `mysql` that contains information about database users and user privileges. To create it, type

```
mysql_install_db --user="mysql"
```

Finally, MySQL server can be launched:

```
mysqld_safe --user="mysql"
```

After a first start of the server, one should immediately set the password for MySQL administrator account which is called *root*. This can be achieved by the utility *mysqladmin*:

```
mysqladmin -u root password <password>
mysqladmin -u root -h pccodb0X.cern.ch password <password>
```

To simplify starting and stopping of the server, we have registered the *mysqld* process as a known system service. To do so, we have extracted the start up script from the binary RPM package of the MySQL 5.0.77. Then, we have copied the script into the `/etc/init.d` directory and used the *chkconfig* utility to add the *mysqld* to list of services that should be started when system enters runlevels 3 (full multiuser mode) and 5 (multiuser mode with X Window system). The initialization script can also be used to manually start, stop, or restart the MySQL server:

```
/etc/init.d/mysqld {start|stop|restart}
```

3.3 Database replication

The *database replication* is a mechanism in which changes made to databases on a master server are propagated into databases on slave servers. In MySQL, the replication is implemented by several processes that are writing and reading log files containing changes made to the database tables. The replication is an asynchronous process, the slave servers do not have to be connected to the master server permanently.

At the COMPASS experiment, the physical database servers *pccodb11* and *pcodb12* are synchronized by the master–master replication. Furthermore, the server *pccodb11* is replicated to server *compass02* that is located in the CERN IT center. This topology serves the following purposes:

1. *high availability*; the database service is powered by two physical servers that are synchronized by the master–master replication. If one of the servers crashes, the service is still available. When the recovery of a crashed server is completed, it uses the replication synchronize itself.
2. *backups*; during regular backups, the database tables need to be locked. Thus the backup is executed on one of the servers; the replication is temporarily paused on this server. When the backup is finished, tables are unlocked and replication resumed. The process is transparent to clients, they communicate with the other server that is unaffected by the backup process and table locking.
3. *data distribution to long distances*; one physical server is also replicated into the CERN computing center. From there, it is replicated into other database servers that are running at home institutes of the members of the COMPASS collaboration. This configuration is known as a *chain replication* and can be regarded as a form of a geographical backup.
4. *load balancing and scalability*; with help of the MySQL Proxy software, it is possible to use the replication to implement load balancing. The proxy software would forward all queries that modify data to one server, the queries that only retrieve data would be distributed between slaves. By adding more slaves, it would be possible increase a performance of the database service, if needed.

On the master server, a *binary log* is created provided that the *log-bin* option is set in the configuration file `my.cnf`. All events that modify table data or structure are written into this log together with a timestamp of modification. The replication is based on three processes, one works on the master server, the remaining two on the slave server. On the master server, the *Binlog dump* thread is running; its purpose is to read the contents of the binary log and send the updates from this log to the slave server. On the slave server, the *I/O thread* connects to its master server, receives the updates of the binary log, and writes them into the *relay log*. Finally, the slave *SQL thread* reads the modifications stored in this relay log and executes them. The master server creates a separate Binlog dump thread for each slave that participates in the replication. Each of these slaves has its own I/O and SQL threads. To see whether these threads are running, one can use the SHOW PROCESSLIST statement on the SQL console.

The MySQL supports three formats of replication: *statement based replication* SBR, *row based replication* RBR, and *mixed–format logging*. In version 5.1.45 of the MySQL software, that has been installed on the new server, the SBR is the default format. In this format, the entire SQL statements that modify table data or table structure are replicated. The main advantage of this format lies in the fact that less data is written into the binary log: if an update or a delete affects multiple rows, only this one statement is logged. Since the log contains all the statements that modify data, it can be used as an *incremental backup*. On the other hand, this format cannot replicate all types of statements that modify the data, especially the non–deterministic behaviour cannot be replicated. The RBR is the safest replication format, it can replicate all types of the modifications. In contrast to the SBR format that replicates the statements, the RBR format replicates the results of the statements, i.e. the changed rows. Unfortunately, the RBR requires more disk space and additionally, the executed statements cannot be reconstructed from the binary log. The mixed format combines the advantages of the

SBR and RBR formats. By default, the SBR format is used, only when an unsafe update is executed, the format is switched to the RBR.

Configuration of the master–master replication

In the new database architecture, we have configured the servers to use the master–master configuration. In this configuration, the server *pccodb11* acts as a replication master of a slave server *pccodb12*. At the same time, the server *pccodb11* acts as a replication slave of a master server *pccodb12*. The configuration of the master–master replication can be divided into several steps. In the first step, we have configured the *pccodb11* server as a replication master by adding multiple parameters into the server configuration file `/etc/mysql/my.cnf`:

```
server-id       = 1
log-bin=/data/mysql/mysql-pccodb11-bin
expire-logs-days = 10
max_binlog_size = 1024M
binlog-ignore-db=DATE2006_log
#other databases to ignore
binlog-do-db=DATE2009_log
#other databases to include into binary log
```

Each server in the replication environment must be identified by its unique *id* that is defined by the *server-id* variable. Then, the binary logging is enabled by defining the value of the *log-bin* variable; it specifies the path to the binary log. The log should be kept for 10 days and after each 1 GB of updates, new log file should be created. Finally, using a combination of *binlog-do-db* and *binlog-ignore-db* parameters, we specify which databases should be included and excluded into the binary logging. Here, we have disabled logging of the changes into the *DATE2006_log* database as it contains historical data which do not change anymore. Note that MySQL also supports a replication of a single table. The server must be restarted in order to start the binary logging. Then, it is possible to create a special database user account that will be used only for the replication by the Binlog Dump thread:

```
GRANT REPLICATION SLAVE ON *.*
      TO 'replicationdb'@'<IP address of pccodb12>'
      IDENTIFIED BY '<password>';
```

The state of the master can be displayed by entering the SHOW MASTER STATUS command (see Listing 5). One should use the output of the command to note down the values of the *File* and *Position* parameters that are necessary during configuration of the replication slave. The output also shows which databases are included or excluded into or from the binary logging.

Listing 5 Status of the replication master

```
SHOW MASTER STATUS;
+--------------------------+-----------+-------------+------------------+
| File                     | Position  | Binlog_Do_DB | Binlog_Ignore_DB |
+--------------------------+-----------+-------------+------------------+
| mysql-pccodb11-bin.000004 | 544465063 | DATE2006_log | DATE2009_log     |
+--------------------------+-----------+-------------+------------------+
```

In the second step, the server *pccodb12* is configured as a replication slave of a replication master *pccodb11*. Again, several parameters needs to be added into the server configuration file `/etc/mysql/my.cnf`:

```
server-id=2
master-host=<IP address of the pccodb11 server>
master-port=3306
master-user=replicationdb
master-password=<password for the replicationdb user>
replicate-ignore-db=DATE2006_log
#other databases to ignore
replication-do-db=DATE2006_log
#other databases to replicate
log-slave-updates
```

We have assigned the unique value 2 to the *server-id* of the *pccodb12* server. Then, we have entered several parameters (IP address, port, user name, and password) that are needed by the slave I/O thread to establish a communication with the master's Binlog Dump thread. The combination of the *replicate-do-db* and *replicate-ignore-db* variables defines databases that should and should not be replicated from the master server. Finally, we have enabled the *log-slave-updates* option that forces to include the replicated statements to the binary log on the slave server. This is required when building the chain replication topology. Then, it is possible to connect the slave I/O thread to the Binlog Dump thread on the master by entering the CHANGE MASTER statement on the SQL console:

```
CHANGE MASTER TO
  MASTER_HOST='IP address of pccodb11',
  MASTER_PORT=3306,
  MASTER_USER='replicationdb',
  MASTER_PASSWORD='<password for the replicationdb user>'
  MASTER_LOG_FILE='mysql-pccodb11-bin.000004',
  MASTER_LOG_POS='544465063';
```

Note, that the values for the MASTER_LOG_FILE and MASTER_LOG_POS parameters correspond to the value of the *File* and the *Position* parameters from the SHOW MASTER STATUS statement (see above). The slave can be started by the START SLAVE command. The slave creates and update the master info log that contains the connection parameters and position in the binary log. To verify that slave has started correctly, one can use the SHOW SLAVE STATUS statement that generates report of the slave status. One should check that the *Slave_IO_Running* and *Slave_SQL_Running* parameters contain the value *Yes* and the *Slave_IO_State* is *Waiting for master to send event.*

At this stage, the *pccodb11* server works as a master and the *pccodb12* server works as a slave. To complete the master–master configuration, the above described procedure needs to be repeated while changing roles of the *pccodb11* and the *pccodb12* servers.

There is a potential threat connected with the multi–master replication and tables that contain an automatically incremented integer column (AUTO_INCREMENT). If new rows are inserted into those tables at the same time, the servers may become desynchronised. To eliminate this issue, a small modification of the server configuration file is required. One needs to add the *auto_increment_increment* parameter and set it to N where N is the number of hosts that participate in the multi–master replication (i.e. two in this case). Additionally, the *auto_increment_offset* option must be added into the configuration file and set to different values from the set $1, 2, \ldots, N$ on all servers.

3.4 Configuration of the MySQL Proxy

MySQL Proxy is a software that analyzes and modifies the communication between the MySQL client applications and the MySQL server. In the default configuration, the proxy only redirects queries issued by clients to the backend database server and it returns the unchanged results to the clients. The behaviour of the proxy software can be programmed by the scripts in the *Lua* language. Then, the proxy can modify (e.g. correct mistakes in statements) or filter queries (e.g. remove the unoptimized queries). Also, the rows can be added, removed, or modified in a result set produced by the MySQL server. The package with the MySQL Proxy installation contains several example Lua scripts. Additionally, the MySQL Proxy supports changing the backend server within a running connection since version 0.6. This feature can be used to implement a slave aware load balancing. In this mode, the proxy forwards all queries that modify table structure or data (e.g. INSERT, UPDATE, DELETE, ALTER, DROP, CREATE statements) to the replication master. The queries that only retrieve results (i.e. SELECT statement) are distributed between the replication slaves. This system is scalable, more slaves can be added in case a higher performance of the database service is needed.

We have used a MySQL Proxy software to implement a fail–over feature. In this configuration, the proxy forwards all communication to one physical server, normally *pccodb11*. If a watchdog process (Nagios in our case, see the following section) detects a failure of this server, it instructs the proxy to change the backend to the remaining *pccodb12* server. We have downloaded the binary package with the MySQL Proxy and extracted it into the file system on the *pccodb10* server. Then, we have used the Bash language to create a start script for the proxy. The script takes one parameter that specifies the backend server. If the value of the parameter equals 1, the proxy forwards the traffic to the *pccodb11* server, if the value of the parameter equals 2, then the proxy forwards the traffic to the *pccodb12* server. Any different value of parameters causes that the script finishes without starting the proxy. Suppose, that the proxy is installed in the `ROOT_DIR`, then the following parameters are required to configure proxy to forward all traffic to the *pccodb11* server:

```
ROOT_DIR=/usr/local/mysql-proxy
LUA_PATH="$ROOT_DIR/share/doc/mysql-proxy/?.lua" \
          $ROOT_DIR/bin/mysql-proxy \
          --proxy-address=$ADDRESS:$PORT\
          --daemon \
          --proxy-backend-addresses=$MASTER1:$PORT
```

The *ADDRESS* and *PORT* variables contain a hostname and a port number under which should be the proxy available. The *MASTER1* variable hold the IP address of the *pccodb11* server that should be set as a backend server. The proxy should be started as a Unix daemon.

The installation package contains the `ro-balance.lua` script that should be passed to the MySQL Proxy, if the slave–aware load balancing is needed. Additionally, the proxy should be started with the *proxy-read-only-backend-addresses* parameter:

```
LUA_PATH="$ROOT_DIR/share/doc/mysql-proxy/?.lua" \
          $ROOT_DIR/bin/mysql-proxy \
    --proxy-address=$ADDRESS:$PORT\
    --daemon \
    --proxy-backend-addresses=$MASTER:$PORT #\
    --proxy-read-only-backend-addresses=$SLAVE1:$PORT \
    --proxy-read-only-backend-addresses=$SLAVE2:$PORT \
    --proxy-lua-script=$ROOT_DIR/share/doc/mysql-proxy/ro-balance.lua
```

The Proxy forwards the queries that modify data to the server specified by the *MASTER* variable and the queries that only retrieve results forwards to replication slaves specified by the *SLAVE1*, *SLAVE2* parameters.

3.5 Backup and monitoring

Together with the MySQL master–master replication, the regular backups and the continuous monitoring are the vital components that contribute to the high availability and the high reliability of the database service. The backup is based on custom script that are regularly executed by the system scheduler *cron*, the monitoring is based on the *Nagios* system.

Database backups

There are three types of backups at the new database architecture: *daily*, *hourly*, and *incremental*. The daily and hourly backups are implemented by a shell script that uses the *mysqldump* tool to backup table data and are regularly executed by the system scheduler *cron* on both database servers *pccodb11* and *pccodb12*. However, as the *mysqldump* tool locks the table, the script is terminated on the server that acts as a backend for the MySQL Proxy. We have created a simple script *getmaster* that is deployed on the proxy server *pccodb10* and returns the address of a current backend server. The daily backup creates a full dump of all databases with the exception of the *DATE_ log* database. This database contains software log that are not critically important, thus only structure of tables in this database is backed up to save time. To reduce time required to load the backup back into the server, the dumps are created with a *–disable-keys* option that adds the Alter Table Disable Keys statement before the dumped data and the Alter Table Enable Keys at the end of the dumped data. The database dumps are compressed by the *gzip* program to save disk space. Additionally, the old compressed backups are deleted after several days. The hourly backup is implemented in a similar fashion. However, the *beamdb* database is not backed up at all on hourly basis. As the *beamdb* is the largest logical database, the hourly backup is created much faster than the daily backup. During the backup process, the database replication is temporarily paused. When the backup is completed, the replication is resumed and the server resynchronizes itself with the backend server.

Since all the SQL statements that modify data are written into the binary log during the database replication, the binary log can be regarded as an audit tool and also as an incremental backup. The contents of the binary log can be browsed using the *mysqlbinlog* utility (as the relay logs on the slave servers have the same format as the binary logs, they can also be inspected using this utility). The *mysqlbinlog* tool converts the binary log into a plain text that can be afterwards imported back into the database:

```
mysqlbinlog binlog.000001 > /tmp/backup.sql
mysqlbinlog binlog.000002 >> /tmp/backup.sql
mysql -u root -p < /tmp/backup.sql
```

Furthermore, it is possible to remove problematic statements (such as an accidental DELETE or DROP statements) from the dump before importing it into the server. The *mysqlbinlog* program support the *–start-datetime* parameter that forces the program to start reading the log from the given timestamp. Thus, in case of accident, it is possible to recover almost all data by merging information from the daily, hourly, and incremental backups.

The databases are also replicated to the server located in the CERN computing center and consequently, this server is replicated into several computing centers of the COMPASS member's home institutes. The chain replication thus serves as a geographical backup. Should any severe accident occur in the COMPASS experiment hall, the database contents can still be recovered from the remote servers.

Monitoring system

The computers participating in the data acquisition system of the COMPASS experiment are continuously monitored by the *Ganglia* system. Ganglia measures the available resources such as a disk space or CPU usage and displays the state of machines in a graphical form that is easily comprehensible by members of the shift crew. However, we have decided to use the *Nagios* monitoring system to watch over the database system. The Nagios also monitors available resources on the remote host and presents the results in a graphical web interface. In contrast to Ganglia,a Nagios can monitor the state of the remote services. Furthermore, the Nagios system is able to perform a predefined action in case an accident is detected. Nagios can also notify a system operator by an e-mail or a text message.

Nagios is very flexible system, it is possible to extend it by the plugins. A *Nagios plugin* is a small application or a script that monitors a state of some service or resource. Nagios plugin should return an integer value with the following meaning:

1. *STATE_ OK* is returned if the service is working as expected,
2. *STATE_ WARNING* means that the state of the service should be investigated,
3. *STATE_ CRITICAL* should be returned if the service behaves abnormally or does not run at all,
4. *STATE_ UNKNOWN* indicates that the plugin is not able to verify the state of the service.

The Nagios plugin can also print several lines describing the state of the service or the resource on the standard output. The Nagios system periodically executes the plugins and displays these messages in a graphical web interface. Additionally, if the state of some service changes (typically from the *STATE_ OK* to the *STATE_ CRITICAL*), Nagios notifies the operator and executes the predefined action (typically restarts the crashed service).

Several most commonly used plugins are distributed by the Nagios developers in the *nagios-plugins* package. Many other plugins can be download from the Nagios Exchange website, [38]. Finally, one can use almost any programming language to develop a custom plugin. We have decided to use the default plugins to monitor CPU usage, free space on disk partitions, state of network interface, and state of the system scheduler *cron* on the database servers. From the Nagios Exchange, we have downloaded plugins that monitor state of the MySQL server and state of the database replication. We have also developed a custom plugin in a Bash language that monitors a temperature of the CPU cores.

The custom plugin called *check_ cpu_ temperature* uses the *sensors* utility that reads and prints information provided by various hardware sensors. The database servers are equipped by two quad core Intel Xeon processors. In order to access the temperature sensor, the *coretemp* kernel module is required to be installed and loaded. The module can be installed from the

RPM package provided by the *Exta Packages for Enterprise Linux* package repository, [32]. The sensors prints several lines of text, the lines that correspond to the temperature of a CPU core have the following format:

```
Core $C:       +$T °C  (high =  +100 °C)
```

The *$C* variable is replaced by the core number, the *$T* variable is replaced by the measured temperature. The cores are numbered from 0 to 7, the script uses the `for` cycle to process information of all cores:

```
MAX=0
for i in ‘seq 0 $CORES‘
do
  temp=‘/usr/bin/sensors | grep "Core $i" |
                         cut  -d "+" -f 2 |
                         tr -cd ’[[:digit:],[.],[,]]’‘
  REPORT=$REPORT" Core $i: "$temp"C"
  if [ $temp -gt $MAX ] ;
  then
    let MAX=$temp;
  fi
done
```

The *$CORES* variable contains number of process cores. Using the *seq* tool, a set of numbers $\{0, 1, \ldots, \$CORES\}$ is generated. In each iteration, a temperature for the corresponding CPU core is extracted from the output of the sensors tool with a set of tools that are chained together by the *pipe* `|` operator. At first in $i-th iteration, the regular expression parser *grep* is used to select the line that contains the "`Core \$i string`". Then, this line is passed to the *cut* program that selects second column of the output divided by the + separator, i.e. it produces the "`$T °C  (high =`" string that is passed to the *tr* program which transforms or removes characters from the given string. In this example, the *tr* returns only digits. This means that the whole chain of programs extracts the value of the temperature of the $i-th processor core. The value is appended to the *$REPORT* variable that will be printed on the standard output and later displayed in the web interface of the Nagios. In the Bash language, the square brackets are used for tests, the *gt* operator tests if the left argument is greater than its right argument. Thus, when the cycle processes all cores, the variable *$MAX* contains a temperature of the warmest core.

When the cycle is completed, the script uses the *$MAX* variable to decide which value should be returned. Variables *$warn* and *$crit* contain a threshold values for signaling a warning and a critical state. We have set the warning threshold to 80 °C and the critical threshold to 90 °C. At around 100 °C the server should be automatically halted to prevent a damage from the overheating. We have observed that under the normal conditions, the core temperatures remain around 45 °C.

```
if [ $MAX -lt $crit ]; then
  if [ $MAX -ge $warn ]; then
    echo "WARNING - Max temperature is high ($MAX C)"
    echo $REPORT
    exit $STATE_WARNING;
  fi
fi
```

Two nested tests are evaluated to decide whether a measured maximal temperature is lesser than the critical threshold (the *lt* operator) and at the same time, it is greater or equal than the warning threshold (the *ge* operator). If both conditions are fulfilled, the script prints a warning message with a highest temperature and list of core temperatures. Nagios displays this text in a web interface. The script exits with *STATE_ WARNING* return value. The code for signaling the *STATE_ CRITICAL* and *STATE_ OK* is similar to this block. If the *$MAX* variable equals zero, the sensors were not able to detect the core temperature. In this case, the *STATE_ UNKNOWN* code is returned.

Since the Nagios package has not been in official repositories of the Scientific Linux CERN distribution, we have installed the Nagios by the source codes compilation. We have decided to install Nagios and Nagios plugins on the proxy server *pccodb10*, on the database servers *pccodb11* and *pccodb12* we have installed only plugins. Since Nagios displays the state of the system using the dynamic web pages created in the PHP language, a web server needs to be configured before the Nagios installation. Binary package for the Apache web server and PHP language have been available in the SLC repository, we have installed them using the *yum* tool. For the safety reasons, it is recommended to create a dedicated Unix user account and group that will be used to execute Nagios commands and plugins:

```
useradd -m nagios
passwd nagios
groupadd nagcmd
usermod -a -G nagcmd nagios
usermod -a -G nagcmd apache
```

At first, the *nagios* account is created, then a password for this account is set. Then a new group *nagcmd* is added into the system and *nagios* and *apache* user accounts are added into the new group using the *usermod* tool that modifies user accounts. When the accounts are prepared, it is possible to decompress the source package with Nagios sources. The *configure* script detects all required components and configures the build process. If the configuration succeeds, the Nagios can be built from source codes by using the *make all* tool. After the compilation, it is possible to use the *make install* command to install Nagios binaries, *make install-init* to install the initialization script, *make install-config* to install default configuration files, and *make install-commandmode* to configure directory for external commands. Then, it is necessary to enter name and e-mail of the operator into the `contacts.cfg` configuration file. Then, the configuration file for the web interface needs to be installed into the Apache `conf.d` directory by entering the *make install-webconf* command. Also, the user account for the web interface of the Nagios needs to be created:

```
htpasswd -c /usr/local/nagios/etc/htpasswd.users nagiosadmin
```

In the next step, the Nagios plugins must be installed. The plugins are distributed in a source package that can be compiled and installed by a traditional configure, make, and make install procedure. The plugins must be installed on all machines that are required to be monitored by Nagios. Finally, it is possible to use the *chkconfig* utility to register Nagios as a known service and add it to a list of service that should be automatically started after a boot of the operating system.

After the installation, it is necessary to configure which machines and which services and resources should Nagios monitor. The Nagios configuration is split into several files. The general configuration of the system including path to the other configuration files or logging and debug options is stored in the `nagios.cfg` file. The configuration file `contacts.cfg` has already been edited during the installation. Then, for each monitored host, two configuration files should be provided: `hostname.cfg` and `hostnameservice.cfg` (hostname should be replaced by actual hostname of the machine). The `hostname.cfg` file contains IP address, hostname, role, and description of the host. In Listings 6, the configuration of the *pccodb11* server is written.

Listing 6 Nagios configuration: definition of host

```
define host{
        use             pccodb
        host_name       pccodb11
        alias           COMPASS DB server 1
        address         #ip address of the server
}
```

The configuration file `hostnameservices.cfg` for the *pccodb10* and *pccodb12* servers is similar. The file defines a list of services and resources that should monitored on the corresponding host. Each service in this file can be characterized by several parameters: its name, description, category, check command and parameters of the check command, check interval, or retry interval. In Listing 7, the service that checks the availability of the MySQL server software is described.

Listing 7 Nagios configuration: definition of service

```
define service{
        use                     generic-service
        host_name               pccodb11
        service_description     MySQL server
        max_check_attempts      3
        check_interval          2
        retry_interval          1
        check_command           check_nrpe!check_mysql
        notification_options    w,c,r
        event_handler           restart-proxy1
}
```

The service should be checked on the *pccodb11* server. The description of the service will be used in the web interface of Nagios. The service should be checked every 2 minutes. In case a problem with the service is detected, the service should be rechecked every minute. After three problems in a row, the *event_handler* should be executed. The *restart-proxy1* handler changes the backend server of the MySQL Proxy to the *pccodb12* server. The w, c, r values of the *notification_options* parameter specify that the operator should be notified each time the server enters a warning state (w), a critical state (c), or recovers back into an ok state (r).

The check commands and event handlers are defined in the `commands.cfg` file. Each command or handler is characterized by its name and its command line. Listing 8 describes the *check_proxy* command that verifies the state of the MySQL proxy process:

Since the MySQL Proxy runs at the same host *pccodb10* as the Nagios, the corresponding Nagios plugin check_procs can be executed locally on *pccodb10*. The check_procs plugin checks how many processes with the name specified in the C argument are running. The value $1:$ of the c argument means that the plugin should return a critical status if less than 1 process with the name *mysql-proxy* is running; i.e. when the *mysql-proxy* is not running at all on the *pccodb10* server.

Listing 8 Nagios configuration: definition of command

```
define command{
    command_name    check_proxy
    command_line    $USER1$/check_procs -C mysql-proxy -c 1:
}
```

Nagios can also monitor services and resources on the remote hosts, e.g. on *pccodb11* and *pccodb12*. If the service listens on some port, the Nagios can try to communicate to this service (e.g. *http* server). However some services (e.g. *cron*) and most resources cannot be checked by this direct approach. For this reason, Nagios provides two means of indirect communication. First method is based on the *check_by_ssh* plugin which connects to the remote server using the *secure shell* (ssh), executes the required check plugin locally, and returns the status to the Nagios. However, this method requires that the Secure Shell Server is installed, configured, and running on all remote hosts. Furthermore, if many services are monitored on many remote host, this method can burden the CPU.

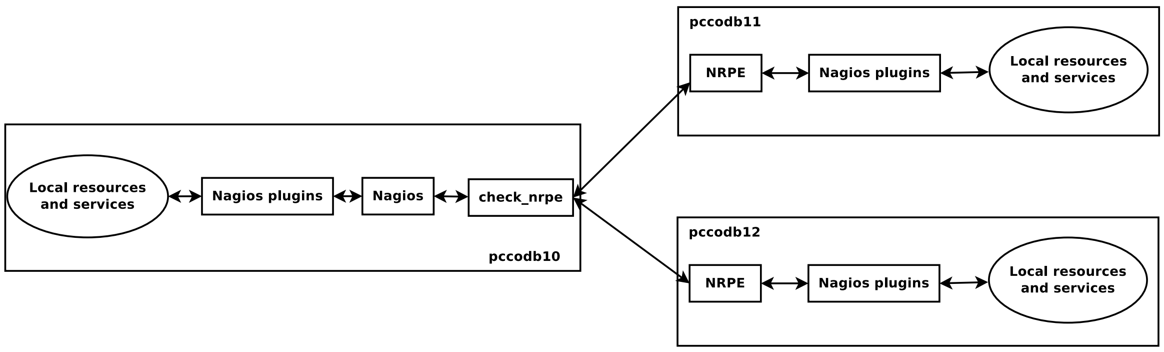


Figure 4.5: Monitoring remote resources and services using the Nagios Remote Plugin Executor. Rectangles represent Nagios processes, elipses represent monitored services and resources.

The second method of the indirect checks is based on the *Nagios Remote Plugin Executor* (NRPE). The NRPE is an agent that is installed on the remote hosts. It listens on the TCP port 5666. Nagios uses the *check_nrpe* plugin to send check command to NRPE, NRPE executes it and returns the corresponding status of the requested service.

We have installed the NRPE on the *pccodb11* and *pccodb12* servers by building the source package. Note that the NRPE requires that Nagios plugins are already installed on the given host. Again, we have used the *configure* script to prepare the compilation process and *make all* to start the compilation. Then, we have installed the *nrpe* agent by entering the *make install-daemon* command and configuration file by entering the *make install-daemon-config* command. To enable a network communication, we have registered the *nrpe* as a service under the *Extended Internet Daemon* (xinetd) server by entering *make install-xinetd* command. Then, we needed to insert the information about the *nrpe* into the `/etc/services` file:

```
nrpe 5666/tcp #NRPE
```

This means that the *xinetd* server passes all request on the TCP port 5666 to the *nrpe* agent. Finally, for the security reasons, we have entered the IP address of the proxy server (i.e. *pccodb10*) to the variable *only_from* in the configuration file `/etc/xinet.d/nrpe`, thus the *nrpe* agent can receive commands only from this machine. To start the *nrpe* agent, one needs to restart the xinetd server.

In the above example, the definition of the service that monitors state of the MySQL server has been described. According to this example, the check command uses the *nrpe* agent. However, the *check_nrpe* plugin must be installed on the monitoring host *pccodb10.* The source package is available, it can be installed by the usual *configure*, *make*, *make install-plugin* sequence of commands. Then, the *check_nrpe* command needs to be defined in the `commands.cfg` file according to Listing 9.

Listing 9 Nagios configuration: definition of check_nrpe command

```
define command{
  command_name    check_nrpe
  command_line    $USER1$/check_nrpe -H $HOSTADDRESS$ -c $ARG1$
}
```

In the command line, the host address and additional arguments are passed to the *check_nrpe* plugin. Nagios retrieves the host address from the host configuration file (e.g. `pccodb11.cfg`), the arguments are defined in the service definition after the exclamation mark: in this case, the plugin takes only one parameter with value *check_mysql* that specifies which command should executed by the *nrpe* agent on the remote host. This command must be defined in the `nrpe.cfg` configuration file on the remote hosts *pccodb11* and *pccodb12.* The command definition should contain a *Nagios* plugin that should be executed and optional arguments that should be passed to this plugin.

```
command[check_mysql] =
  /usr/local/nagios/libexec/check_mysql -u nrpe -p nrpe
```

The *check_mysql* plugin tries to connect to the MySQL server on the localhost with user name *nrpe* and password *nrpe.* Then it executes the Show Status statement that prints several variables that describe a current state of the server. The *check_mysql* plugin extracts part of this information such as uptime of the server, number of queries, or number of active connection and prints it on the standard output. The NRPE returns this information to the Nagios process that displays it on the web interface.

The Nagios web interface can be accessed on the Compass internal web server under the address `http://pccodb00/nagios` Although a shift crew still uses the Ganglia monitoring system, the Nagios has already identified several problems of the MySQL replication. Thanks to the reporting capability, the problems have been quickly resolved and the data acquisition has not been affected.

3.6 Data migration and verification

After completing the configuration of the MySQL and MySQL Proxy on new database servers, it was possible to transfer database structure and data from the old to the new servers. Many changes have been implemented in the new version of the MySQL since the version 4.0.77. Additionally, 32–bit version of the operating system had been installed on the old servers, while we have installed 64–bit operating system on the new servers. Some of these changes could potentially affect the backward compatibility, thus it was essential to verify that the data remained unchanged on the new servers. To prevent the clients from modifications of the data, we had to temporarily disable the virtual address *pccodb00*. The process has been thoroughly planned and tested in order to reduce the downtime of the database service, yet it took almost 12 hours to complete the process.

At first, we have used the *mysqldump* tool to create backup of the all databases on the old *pccodb02* server. A database dump created by the mysqldump tool is a text file that contains SQL statements CREATE DATABASE and CREATE TABLE that create empty databases and tables followed by the INSERT statements that populates the tables with data. When importing the dump file into the database server, these instructions are used to reconstruct the structure and the data. We have used the *mysqldump* tool with several parameters:

- The *--quick* (*-q*) option makes the *mysqldump* to receive table rows from server a row at time. This parameter should be used especially when dumping of large tables.
- The *--extended-insert* (*-e*) parameter instructs the *mysqldump* tool to generate INSERT statements that should include multiple values. The bulk insert reduces overhead and size of the dumped files and therefore increases the speed of insertion of the dump into the new servers.
- The *--skip-dump-date* option disables dumping of a current timestamp into the dump files. We have used this option because the dumps of the same tables created at different times are different which would affect the verification process. For the same reason, we have also used the *--skip-comments* parameter that disables dumping additional metainformation about the system into the dump files.

We have found out that several tables that contained large VARCHCHAR or BLOB columns could not be properly dumped. To solve the issue, we had to set the *max_allowed_packet* variable that affects the maximal size of a packet that can be received to 16 MB. Although, the *mysqldump* can create a dump of the entire database, we have decided to create a dumps on the table level to simplify the process of data verification. For the same reason, we have used the *--no-data* and *--no-create-info* parameters to distribute a table structure and table data of each table into two files. The dump files have been copied over network using the *scp* tool to the *pccodb11* server. Then, we have used the *mysql* client application to import the dumps into new database server:

```
mysql -u root -p -h localhost < table.frm.sql
mysql -u root -p -h localhost < table.data.sql
```

We have redirected the standard input from the keyboard to the dump files using the *redirection operator* <. At first, table structure is created, then the table is populated with data. Note that the *pccodb12* server resynchronizes itself thanks to the database replication. After importing all tables, a data verification had to be performed. However, several databases should not be compared: the *information_schema* is a database containing metainformation about other databases managed by the server, this database has been implemented in version 5 of the MySQL server, thus it does not exists in the old servers. The *mysql* database is a system database with information about users and privileges, its format changed between versions 4 and 5 of the MySQL server. Finally, the *phpmyadmin* database stores data of the web database management tool called *phpMyAdmin*; since we have installed newer version of the tool on the new server, the format of the database has changed as well. We have created and compared four dumps of each database table:

- dump from the new server created by 64–bit mysql client in version 5.1.45

- dump from the new server created by 32–bit mysql client in version 4.1.22
- dump from the old server created by 64–bit mysql client in version 5.1.45
- dump from the old server created by 32–bit mysql client in version 4.1.22

This time, we have used the *mysql* client application to execute the SELECT * FROM statement that prints all rows from the given table on the standard output; we have redirected the standard output to the disk file using the *redirection operator* >. The output produced by the *mysql* client does not contain the *Insert* statement, thus it is more compact than output produced by the *mysqldump* tool. Still the output of the largest database tables (e.g. table *ECAL_MON* with monitoring data of the electromagnetic calorimeter, table *messages* with debug messages generated by the DATE software) contained several gigabytes of data, therefore a fast file comparison method had to be employed. As a first step, we have decided to compare the dumps with a *md5sum* tool which calculates a 128 b *message digest* (a hash) of a given file or message using the *MD5* algorithm, [29]. It is clear that the MD5 is not an injective function; however probability of a hash collision (i.e. a case when two different messages produce the same hash) is vanishingly small and can be neglected. Some databases contain several hundreds of tables (e.g. *DATE_log*), thus we needed to automate the process of creating table dumps and calculating the hashes. The MySQL server creates a separate directory for each logical database and stores all tables from given logical database into the corresponding directory. Furthermore, each MyISAM table is represented by several files in this directory: FRM file contains table structure, MYD file contains table data, and MYI file contains table index. We have used this knowledge to develop a shell script that iterates over all files in a given database directory and for each FRM files, it creates a dump and calculates the check sum.

```
#!/bin/bash
HOST="$1"
DB="$2"
DUMP="/tmp/dump.txt"
LOGFILE="/tmp/report-$DB.log"
DATADIR=/data/mysql
```

This script takes two parameters: address of database server and name of logical database. In the Bash language, the value of the *n*–th parameter is stored in a special variable *$n* The variable *$0* contains the name of the script. At first, the value of the command line parameters is assigned to the *HOST* and the *DB* variables. The *DUMP* variable contains a path to the dump file, the *LOGFILE* contains path to the file with md5 checksums. Finally, the *DATADIR* variable holds a location of the data directory of the MySQL server.

```
touch $LOGFILE
WD=`pwd`
cd $DATADIR/$DB
```

The *touch* tool updates the access and modification times of given file to the current time. If the given file does not exist, it is created by the tool. The *pwd* utility prints current working directory. Since the *pwd* is enclosed in the quotes operators `, its output is assigned to the *WD* variable instead of being printed on the standard output. Then, the working directory is changed to the corresponding database directory using the *cd* tool.

```
for X in *.MYD
```

```
    do
      TABLE=‘basename $X .MYD‘
      echo "[‘date‘] $DB.$TABLE" >> $LOGFILE
      sh /tmp/select.sh $HOST $DB $TABLE
      LOCALMD5=‘md5sum $DUMP | cut -f 1 -d " "‘
      echo $LOCALMD5 >> $LOGFILE
      rm -f $DUMP
     done
  cd $WD
```

The Bash language supports iterating over files in a directory. In this particular example, only files with MYD extension that correspond to the database tables are processed. The file name of the currently processed file is assigned to the *X* variable. The *basename* utility is used to strip directory prefix and given suffix (usually file extension) from the given file name, thus the *TABLE* variable holds the actual name of the database table. The *echo* utility displays a line of text on the standard output. The line contains current timestamp produced by the *date* utility, the database name, and the table name. This line is appended to the end of the log file thanks to the another *redirection operator* `>>` (the `>` also redirects a standard output, however it rewrites the destination file). Then, a custom script `select.sh` is executed; it connects to the server *HOST* and selects all rows from table *TABLE* in database *DB* and writes them into the file *DUMP*. Finally, the md5 checksum of the dumped file is calculated. The *md5sum* tool prints the checksum and the file name on the standard output. We have used the `|` operator to redirect the standard output of the *md5sum* process to the standard input of the *cut* process that prints select parts of given lines. In this example, the line is divided into several fields delimited by the space character (the *d* parameter) and first field (the *f* parameter) is selected. The md5 checksum is appended to the log file and the dump file is removed by the *rm* tool. Then, the cycle is repeated for the following MYD files. After processing all tables, the script returns to the original working directory.

The `|` operator can be substituted by the subsequent calls of the other redirection operators `>` and `<`. Suppose *a* and *b* processes. Then the `a | b` call that forwards standard output of the process *a* to the standard input of the process *b* can be replaced by the following calls:

```
a > /tmp/file
b < /tmp/file
```

At first, the output of the process *a* is redirected into a temporary file using the `>` operator. Then, the content of the temporary file is redirected to the standard input of the process *b* using the `<` operator. However, since the `|` operator transfers data between processes using a memory FIFO instead of creating a temporary files, it is a much faster method. Furthermore, the *b* process does not have to wait until the process *a* finishes when the `|` operator is used.

Using the MD5 checksums, we have verified that a majority of tables was imported correctly into a new database servers. Only several tables produced different checksums from the old and the new server. Unfortunately, the *md5sum* can only decide whether two files are the same. To identify the exact lines in which the dump files differ, we needed a more sophisticated tool. We have decided to use a *diff* tool that compares given files line by line. Unfortunately, this tool requires more resources (time, memory, and CPU power) to process the files than the *md5sum* does. This tool has shown that the dumps differed in the lines containing decimal numbers.

We have found out that the definition of the data type DECIMAL(M, N) has been changed between versions 4.1.22 and 5.1.45. This type represents a subset of rational numbers; the parameter M defines the maximal number of significant digits, the parameter N defines the number of digits that follow the decimal point. Originally, in the older versions of the MySQL server, the type had been represented as a character string and each digit as well as a sign had been stored as a character. For positive decimal numbers, the plus sign could be replaced by an additional digit which extended a range for positive number by one order. For example, the type DECIMAL(4,2) had represented all decimal numbers from the interval $I_1 = [-99.99, 999.99]$ stored with a precision to hundredths. However, the definition changed in version 5.0.3 of the server to comply with the SQL standard, [33], which dictates that the DECIMAL(M, N) type defines a subset of rational numbers with up to $M - N$ digits before the decimal point and up to N digits following the decimal point. Thus, the type DECIMAL(4, 2) covers rational numbers from the interval $I_2 = [-99.99, 99.99]$ stored with a precision to hundredths. It can be clearly seen that the interval I_2 is a sub–interval of the I_1, therefore the DECIMAL(M, N) can contain more values in the older versions of the MySQL server that did not follow the SQL standard.

During the migration, the values from the interval $I_1 \setminus I_2$ were truncated which breached the integrity of the tables with DECIMAL columns. We have used the DROP TABLE statement to remove the affected tables from the database on new servers. Then, we have manually modified the files with the dumped structure of the tables, we have extended a range of the DECIMAL columns. We have created the modified tables and populated them with the data from the dump files. Finally, we have created and compared dumps of the affected tables again. This time, the method based on the *md5sum* tool has proved that the migration had succeeded.

4 Maintenance of the new database architecture

After the importing and verification of data in the new server, it was possible to connect the client applications to the new database service. Several minor problems appeared, we have investigated and resolved them. Then, we have been using the Nagios tools and MySQL logs to monitor the performance and stability of the database service. As part of the maintenance, we have collaborated in development of a new database application that visualizes the load of the various components of the data acquisition system and we have also use the EXPLAIN statement to evaluate and improve the execution plans of the most frequent queries.

Starting of the operation of the new database architecture

In order to connect the client applications to the new database service, we needed to set the *pccodb11* server as a backend server of the MySQL Proxy, start the MySQL proxy, and finally reconfigure the *pccodb00* virtual address to point to the proxy server *pccodb10*. To set the virtual address, we have used the *ip* tool that serves for manipulations of routing, tunnels, and network devices. The tool supports the *addr* command that sets up the IP address on the network device:

```
ip addr add vvv.xxx.yyy.zzz/16 dev eth0
```

The *add* parameter of the *addr* command adds a new IP address for the given device, *eth0* which corresponds to the ethernet card in this case. We have appended this command to the `rc.local` initialization script on the *pccodb10* server, thus the virtual address is automatically configured every time the server starts.

4.1 Developing new data acquisition monitoring application

With a collaboration with the data acquisition experts, we have developed a new application that monitors the load of various components of the data acquisition system ranging from the frontend electronics to the triggers and event builders. We have been asked to propose a database that would be used to store the measured data. The data would be gathered and inserted into the database by the *Cinderella* process that implements the online filter functionality. The frontend that displays the data would be implemented as a graphical tool based on the *ROOT* framework by the COMPASS DAQ experts. The *ROOT* is a framework that focuses on data analysis, data storage, and data visualization, [7]. ROOT is based on the C++ language, it also contains the C++ interpreter called *Cint*. Additionally, the *Detector Control System* of the experiment gradually incorporates the tools that visualize these monitoring data.

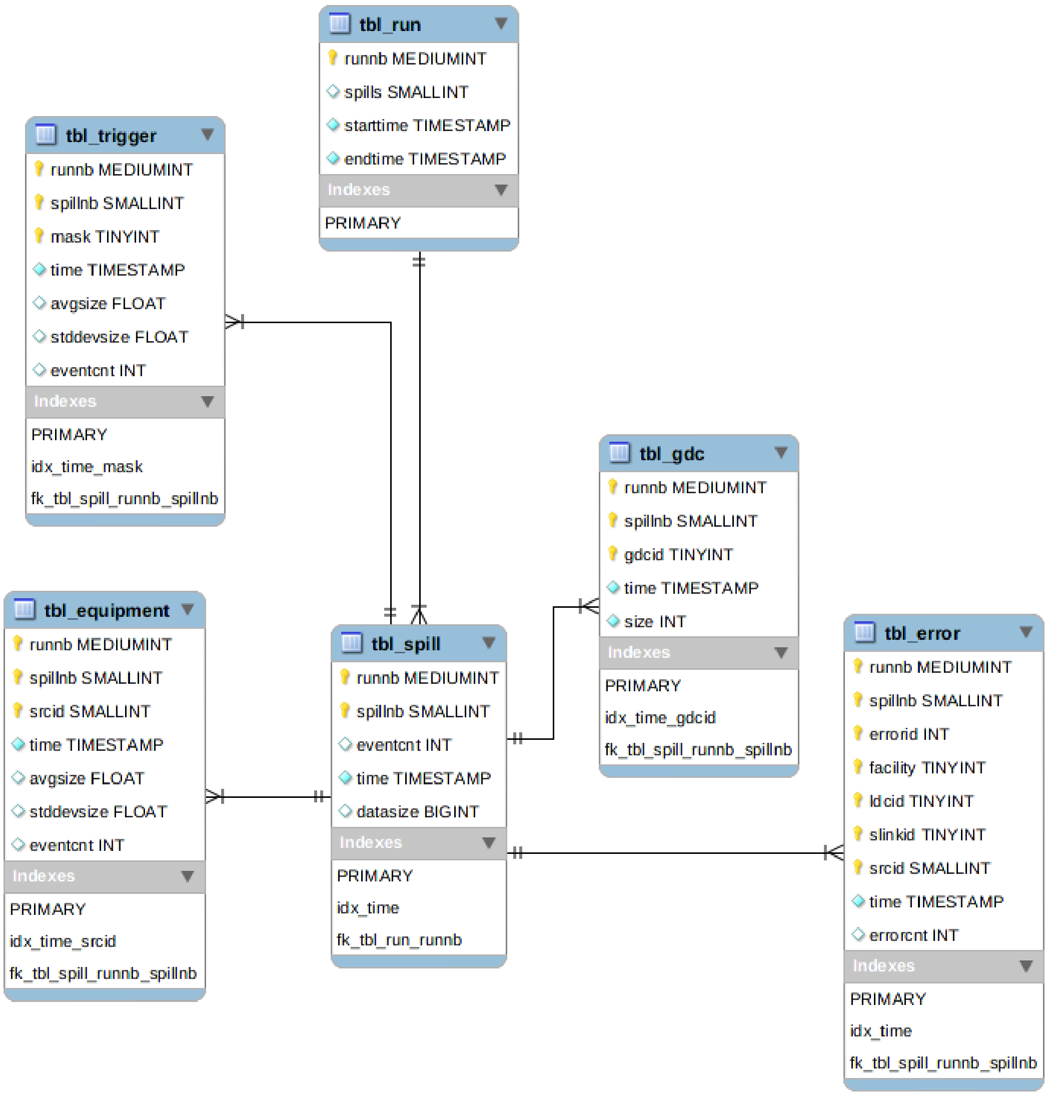


Figure 4.6: Schema of the *daqmon* database

We have proposed a structure of the *daqmon* database using the MySQL Workbench graphical tool. According to the proposal, the database consists of the following tables:

1. *tbl_run* table contains start time, stop time, and number of spills for each run.
2. *tbl_spill* table stores numbers of events registered for each spill.
3. *tbl_gdc* table holds a total size of events processed by the event builders at given time stamps.

4. *tbl_trigger* table stores statistics about occurrence of given trigger masks and spills over time represented by the run number, spill number, and time stamp.
5. *tbl_equipment* table contains statistics about usage of different subdetectors over time represented by the run number, spill number, and time stamp.
6. *tbl_error* table contains information about errors including error type, time stamp, or facility.

By mixing data from different tables, it possible to retrieve information about devices that produced the most of data, compare load of the event builders, or detect sources of hardware problems.

The proposed database structure has been approved after some discussion and the database *daqmon* has been installed on the database servers and included into the replication. A dedicated user account *daqmon* has been created and configured for accessing the database. Furthermore, the *cinderella* user account has been given grants to insert, update, and select rows from the tables in the *daqmon* database. During testing of the new database, the replication has been interrupted. The problem was caused by the online filter process that was inserting data into two databases, *daqmon* and *beamdb*, at the same time. To save time, the process did not use the USE statement to change the currently active database. It is possible to insert data into a table in an inactive database provided that the INSERT statement contains both database and table names. Unfortunately, the replication requires that all records are inserted into table in the active database. Therefore, the online filter has been corrected to always switch the active database to solve the problem.

After approximately one month in the operation of the *daqmon* database, a data acquisition expert has requested a change in the structure and data of the *tbl_trigger* table. A new column for storage of the inverted trigger mask should be added into the table structure, furthermore existing trigger masks should be renumbered. We have used the *mysqldump* tool to extract all data from the *tbl_trigger* table. We have employed the *--fields-terminated-by* and *--tab* parameters to produce CSV file with table data. We have prepared a Perl script that processes this CSV file line by line recalculating the trigger mask according to requested rules and adding the inverted mask and stores the modified rows into another file. Then, the table is dropped from the database, recreated with altered structure, and populated with the modified rows.

4.2 Evaluation of the query execution plan

During configuration of the MySQL servers, we have enabled logging of the slow queries. Under the term of *slow query*, we understand a query that is being evaluated longer than a certain time limit defined by the variable *long_query_time*. Slow queries are appended into a text file specified by the variable *log_slow_queries*. Knowledge of slow queries is very important as they can degrade the performance or even overload the database server.

Execution times of queries can be improved by adding appropriate indexes to the tables. The MySQL software supports the EXPLAIN statement that evaluates execution plan of the given SQL query. The EXPLAIN tool shows which index (if any) is used during the query evaluation, how many rows are searched, if a temporary file is created, or if an additional pass is required to sort the result. This information should be used for designing correct table indexes. The EXPLAIN tool displays the results as a table with the following columns:

- The *id* column contains a sequential number of the SELECT statement within the examined query.

- The *select_ type* column informs about a type of the SELECT statement, e.g. the *Simple* correspond to simple queries that do not use subqueries or unions, the *Subquery* type corresponds to a first SELECT in a subquery, or the *Primary* type corresponds to the outermost SELECT statement in the query.
- The *table* column identifies the table that is being searched.
- The *type* column explains how the tables are joined. In the worst case, if this column contains the *All* value, it means that a full table scan is performed for each combination of rows from the previous table. On the other hand, in the best case, this column contains the *const* value and at most one matching row is found in the table and read at the start of the query.
- The *possible_ keys* column lists the indexes that can be used by the MySQL to find the requested rows in the table. If the column contains the *Null* value, no index can be used.
- The *key* and *key_ length* columns contain information about the key and its length that MySQL uses for retrieving the requested rows. From the key length, one can deduce a number of parts of multicolumn index that are actually used.
- The *ref* column explains which columns are compared to the index (listed in the *key* column) during retrieval of rows from the table.
- The *rows* column contains estimated number of rows that needs to be examined during query execution.
- The *Extra* column provides additional information about the way in which the query is evaluated. For example, the *Using temporary* value in this column means that a temporary table must be created to contain the query result, the *Using filesort* means that additional pass is required to return the sorted result, or the *Impossible WHERE* means that the WHERE clause of the SELECT statement is always false and thus, it cannot return any rows. The *Extra* column can contain multiple values.

Additional columns are shown if the EXTENDED keyword is used with the EXPLAIN statement. The PARTITIONS keywords should be used when examining queries over the partitioned tables. Complete description of the output including all possible SELECT types and all possible values in the *Extra* column can be looked up in the MySQL documentation, [37]. The usage of the EXPLAIN statement will be described on the *daqmon* database.

During development of the *daqmon* database, we have used the EXPLAIN tool to propose indexes for tables. We will demonstrate the usage of the EXPLAIN tool on the table *tbl_ trigger* that contains information about occurrences of trigger masks in time. The structure of the *tbl_ trigger* table is shown in Listing 10.
Suppose that the monitoring application built on the *daqmon* database should display the trigger mask, the timestamp, and the average size of all the records with given run number (e.g. 85626) ordered by the time. The corresponding rows can be retrieved from the table using the SQL statement from 11.
The output of the EXPLAIN command is summarized in the table 4.7.

id	*select_ type*	*table*	*type*	*key*	*rows*	*Extra*
1	SIMPLE	tbl_trigger	ALL	NULL	1127528	Using where; Using filesort

Table 4.7: The result of the EXPLAIN command on an non–optimized table

The result of the EXPLAIN statement revealed several problems: the type *All* means that all the 1127528 rows in the table must be searched, none key/index can be used. Moreover, an

Listing 10 Structure of the table tbl_trigger

```
CREATE  TABLE IF NOT EXISTS ‘daqmon‘.‘tbl_trigger‘(
 ‘runnb‘ MEDIUMINT NOT NULL COMMENT ’Run number’ ,
 ‘spillnb‘ SMALLINT NOT NULL COMMENT ’Spill number’ ,
 ‘mask‘ TINYINT NOT NULL COMMENT ’Trigger mask’ ,
 ‘time‘ TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
                                  COMMENT ’Timestamp’,
 ‘avgsize‘ FLOAT NULL
      COMMENT ’Avg. event size for the  mask in spill’ ,
 ‘stddevsize‘ FLOAT NULL COMMENT ’Standard deviation
                    of the event size for the mask in spill’ ,
 ‘eventcnt‘INT NULL
        COMMENT ’Number of times the mask appeared in spill’,
 PRIMARY KEY(‘mask‘, ‘runnb‘, ‘spillnb‘),
 INDEX idx_time_mask(‘time‘, ‘mask‘)) ENGINE = MyISAM;
```

Listing 11 Test query on the tbl_trigger table

```
SELECT mask, time, avgsize FROM tbl_trigger WHERE runnb=85626
       ORDER BY time;
```

additional pass is required to sort the result. The type *Simple* of the query means that nor union nor subqueries are used during evaluation of the query. The primary key of the table *(mask, runnb, spillnb)* cannot be used because the *runnb* is not its prefix. If the columns in the primary key are reorganized into the following order *(runnb, spillnb, mask)*, the primary key could be used to retrieve the desired rows. This assumption can be confirmed by the EXPLAIN command (see table 4.8). This time, only 2383 records are searched, though the file sorting is still performed.

id	*select_type*	*table*	*type*	*key*	*rows*	*Extra*
1	SIMPLE	tbl_trigger	ref	idx_runnb_spillnb_mask	2383	Using where; Using filesort

Table 4.8: The result of the EXPLAIN command on the table with the optimized index

Under certain circumstances, it is possible to satisfy the ORDER BY clause using the index, thus eliminating the need of the file sorting. According to the documentation [37], this is valid for the queries with the following structure:

```
SELECT * FROM table WHERE keypart1=constant ORDER BY keypart1;
```

Unfortunately, the examined query does not have this structure because the key that retrieves the rows (*Primary Key*) is different from the key that is used in the ORDER BY clause (the *idx_time_mask*).

The *runnb* in the WHERE clause can be replaced by the time interval between the start and the end of the run. The information about the start and the end of the run is stored in the table *tbl_run* described in Listing 12.

Listing 12 Structure of the tbl_run table

```
CREATE TABLE IF NOT EXISTS ‘daqmon‘.‘tbl_run‘ (
 ‘runnb‘ MEDIUMINT UNSIGNED NOT NULL COMMENT ’Run number’,
 ‘spills‘ SMALLINT DEFAULT NULL COMMENT ’Number of spills in run’,
 ‘starttime‘ TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
                 COMMENT ’Time when run started’,
 ‘endtime‘ TIMESTAMP NULL DEFAULT NULL
                 COMMENT ’Time when run ended’,
 PRIMARY KEY (‘runnb‘)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Listing 13 Query on the tbl_run table

```
SELECT starttime FROM tbl\_run WHERE runnb=X;
```

The start time of the run X is returned with the query from Listing 13.
In a similar fashion, one can also obtain the time when the given run ended. The *runnb* is the PRIMARY KEY of the table, therefore at most one record with the given run number can exist in the table. This means that only one record needs to be searched to return the start/end time of the given run. By substituting the run number with the corresponding time interval in the original query, we get the following query:

Listing 14 Optimized test query on the tbl_trigger table

```
SELECT mask, time, avgsize
FROM tbl_trigger
WHERE time>=(SELECT starttime from tbl_run WHERE runnb=85626)
  AND time<=(SELECT   endtime from tbl_run WHERE runnb=85626)
ORDER BY TIME;
```

.

The results of the EXPLAIN (see table 4.9) statement confirm that both subqueries are indeed searching only 1 row in the *tbl_run* table as was expected. Additionally, the file sort is not needed anymore and the query is executed faster. However, the speed improvement is not very significant in this particular case. The result of the query contains approximately 2 000 rows and the file sort can be done in the memory buffer so it is reasonably fast. In case the size of memory buffer is exceeded, a temporary table must be created and sorted on the disk and the file sorting is slow. The maximal size of the memory buffer is controlled by the variable *sort_buffer_size*. To sum it up, the file sorting should be avoided, if possible.
We have been also asked to analyze the most frequently used queries over the *ecal_mon* table. The queries are regularly issued by the *Detector Control System* every 15 minutes. The *ecal_mon* table in the *beamdb* database contains information about state of blocks that form the COMPASS electromagnetic calorimeter. With over one billion rows, it is the largest table in the database, therefore proper indexing of the table is essential for the smooth operation of the database service. Using the EXPLAIN tool, we have verified that the table indexes are correctly used during evaluation of the queries. Additionally, the EXPLAIN tool contained the *Select tables optimized away* value in the *Extra* column for several queries. This means that the query contains some aggregate function such as MIN or MAX that can be resolved using the table

id	*select_type*	*table*	*type*	*key*	*rows*	*Extra*
1	SIMPLE	tbl_trigger	range	idx_time_mask	1932	Using where;
2	SUBQUERY	tbl_run	const	PRIMARY	1	
3	SUBQUERY	tbl_run	const	PRIMARY	1	

Table 4.9: The result of the EXPLAIN command on the modified query

index, therefore no rows are browsed and only one row is returned.

5 Results of migration and outlook

The original database architecture was powered by two physical servers that were mirrored by the replication. As a result of the combination of high load and insufficient hardware, the service suffered from performance problem that lead to several crashes. We have proposed a new database architecture based on a newer hardware as well as on more recent version of software. Additionally, the new database architecture includes a proxy server that also serves for monitoring. The continuous monitoring, the regular backups, and database replication should contribute to high availability and reliability of the service.

The process of migration to the new database architecture has been successfully completed in May 2010, just before start of the data taking. During June 2010, several problems with locking of the *ecal_mon* table appeared. After decreasing a number of inserts into the table, the problem disappeared. The only forced shutdown of the database service was caused by an unexpected power cut in the COMPASS experiment hall. After this shutdown, the uptime exceeded 5 months (from July till winter shutdown in December).

The most serious problem appeared in May 2012, just a few hours before end of the winter shutdown and start of the data taking period. The database server *pccodb11* has crashed, probably as a result of hardware failure. After restart of the crashed server *pccodb11*, the replication slave on the *pccodb12* server stopped working. The exactly same problem appeared also on the *compass02* server in the CERN computing center. We have used the SHOW SLAVE STATUS statement to investigate the problem and found that the slave *I/O thread* that is responsible for receiving events from the binary log on the master server and storing them into the relay log is in the *Not running* state. Furthermore, the *Last_IO_Error* variable contained the *Got fatal error 1236 from master when reading data from binary log: 'Client requested master to start replication from impossible position'* message.

Thus, the master server failed to write all events into its binary log before the crash and after the restart, the slave *I/O thread* tried to receive these unwritten messages. After start of the *mysqld* process on the *pccodb11* server, a new file with binary log has been created. We have used the CHANGE MASTER statement to try to force the slave process on the *pccodb11* server to skip the unwritten events by reading from the new log. After restarting the slave process, it crashed again, this time with the *SQL thread* that is responsible for reading the events from the relay log and executing them in the *Not running* state and with error message set to *'Duplicate entry for key PRIMARY'*. This error could probably be repaired by stopping slave, setting up the variable SQL_SLAVE_SKIP_COUNTER to number of bad events, and restarting the slaves.

However, to ensure the full synchronization, the replication process had to be started over. At first, the *pccodb10* server has been manually shut down to prevent Nagios and database clients from interfering with recovery process. Then, on both server, the replication slaves have been stopped using the STOP SLAVE statement. Then the MYSQLD processes have been turned off on both servers. With servers not running, it has been possible to resynchronize data directories of both servers. After the operation, both servers have been started again. Using the SHOW MASTER STATUS, the current binary logs and positions in these logs have been detected. Then, the CHANGE MASTER statement has been used to inform the slave processes about the current binary logs. Finally, after start of the slave processes, the proxy server *pccodb10* has been turned on and the database service recovered.

Immediately after the recovery of the crashed server, the database service could be restarted. Unfortunately, as the replication process was stopped, the redundancy of database service was lost. Therefore, the servers needed to be resynchronized quickly to prevent loss of data in case the *pccodb11* server crashed again.

During the data taking period (i.e. approximately from beginning of June to end of November), the size of the directory with the database data increased from 114 GB to 194 GB, the *beamdb* and *DATE_log* databases contain approximately 80 % of the data. The *ecal_mon* table is with more than one billion records the largest table from all the databases. Consequently, time required to complete a daily backup increased from approximately 20 minutes in June to more than one hour in November. Even though the backup is created on a slave server and does not influence the master server, we have decided to create a new databases *beamdb2011*, *DATE2011_log*, and *DATE2011* for data from the data taking in the year 2011. To save disk space, the old databases could be converted into the ARCHIVE storage engine that does not allow modifications but it compresses data.

During the year 2010, we have also created several new databases and tables. The *daqmon* database that serves as a backend for application that monitors performance of the various components of the data acquisition system is described in a dedicated section. Additionally, we have also created the *hcal_mon* table in the *beamdb* database. The table has the same structure as the *ecal_mon* table, it stores information about state of the hadronic calorimeter. The information can be visualized in the *Detector Control System*.

5.1 Design of further improvements of the database architecture

We have developed the new database architecture with scalability in mind. Although the performance seems to be sufficient, it is relatively easy to increase it.

First method of increasing the performance is based on the MySQL Proxy software. Currently, the proxy redirects all queries to one backend database server (usually the *pccodb11*). However, by adding more replication slaves and enabling the load balancing mode of the proxy as described in Section 3.4, it would be possible to easily scale the system performance. Alternatively, the *MySQL Load Balancer* could be used for the same purpose. MySQL Load Balancer is an application based on the MySQL Proxy that also provides read only load balancing over a number of MySQL servers [36]. The load balancer consists of two components: *proxy* and *monitor*. The proxy component handles the client connections and query distribution. The queries that modify data are sent to the master backend server, queries that only retrieve data are distributed between the slave backend servers. The monitor component periodically verifies the state of the backend servers. The MySQL Load Balancer uses the information about state of servers to update a list of available replication slaves. If a monitor detects that some slave lags behind the master or a replication process is crashed on some slave, then the affected slave is temporarily removed from the list of available backends. When the lag behind the master decreases or when the replication process is recovered, then the slave is returned into the list.

We have shown that crash of one server does not severely affect availability of the database service. Unfortunately, in case a replication is broken as a result of damaged binary log, the servers need to be resynchronized which can cause several hours of downtime of the service. To prevent this problem, more slaves should be added into the architecture and the master–master topology should be replaced by master–multiple slaves topology. If the master would experience crash, one slave would become a new master, the former master would resynchronized with the second slave which would be temporarily unavailable. However, the database service could still be available with the new master. This topology could be combined with the load balancing.

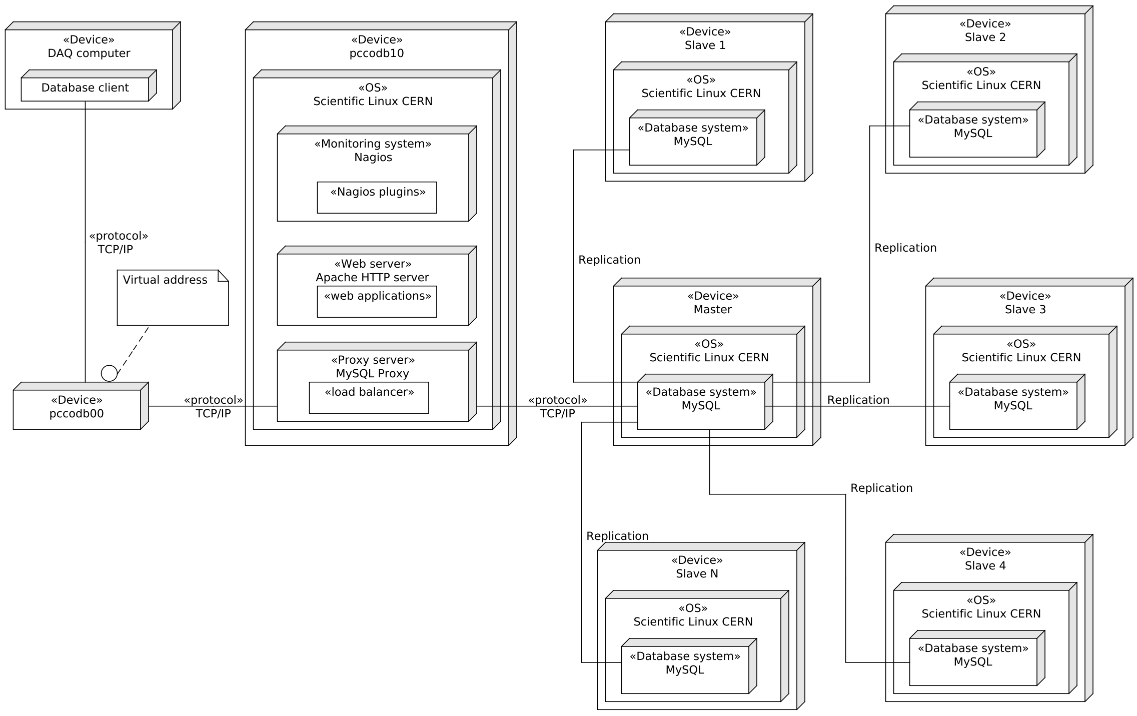


Figure 4.7: Improved database architecture

Furthermore, in an unstable development version 5.6 of the MySQL software, several improvements of the replication technology have been implemented, [27]. At first, the *Global Transactions Identifiers* (GTID) are introduced into the replication. These identifiers simplify tracking of progress of replication between the master and the slave servers. Also, two new utilities have been developed: the *mysqlfailover* utility continuously monitors the replication topology and in the case it detects a failure of the replication master, it promotes the most updated slave to the master role. The utility uses the GTIDs, it also ensures that no transaction is lost during fail over process. Database administrators can also use the other utility *mysqlrpadmin* to disconnect the master server for maintenance purposes; the utility ensures that the most updated slave is promoted to the master. The utility also provides a slave discovery and replication monitoring.

Partitioned tables

The other performance optimization method is based on using some new features of the MySQL server, especially the partitioning. *Partitioning* enables distribution of parts of a single table into multiple files; i.e. parts of the table are saved as separate tables. Partitioned tables have been introduced in version 5.1 of the MySQL server. MySQL supports the *horizontal partitioning*, i.e. the the table is distributed into partitions by rows (in the *vertical partitioning*, the table is distributed into partitions by columns).

During the compilation of the source codes of the MySQL software (see Section 3.2), we have passed the *--with-partitions* option to the *configure* script to enable the partitioned tables. The support for partitioning is available as a plugin for the MySQL server. To verify if the plugin is active, one can use the SHOW PLUGINS statement.

Division of rows into the appropriate partitions is based on the value of the *partitioning function*. Several types of partitioning exist in the MySQL software. Depending on the type, the partitioning function takes as a parameter either a column value, function of one or more column values, or set of column values. Also, the partitioning function is based on the selected type of partitioning; it can be a hashing function, matching against set of values or ranges, or modulus. The partitioning function returns a partition number into which the row should be inserted; the function must be nonrandom and nonconstant. The following types are available:

- In the RANGE partitioning, the user defines a division of possible values of some table column into several continuos, non overlapping intervals. The partitioning function takes as a parameter a value of the given column and according to this value, it inserts the row into the corresponding partition.
- The LIST partitioning works in a similar fashion as a RANGE partitioning. Instead of dividing the possible column values into intervals, the database administrator defines several distinct sets that cover all expected values of given column. The partitioning function takes a value in given column and according to this value, it inserts the row into the matching partition.
- If the HASH partitioning is selected, the database administrator only defines desired number of partitions and a column of the integer data type. The hashing function that takes a value of the given column (or a value of nonrandom nonconstant function acting on the column value) as its parameter and acts as a partitioning function ensures that the rows are evenly distributed into the partitions.
- In the KEY partitioning, the internal hashing function that operates on the supplied list of one or more column. If none column is supplied, the hashing is performed on the primary key of the table.

The MySQL server also supports the composite partitioning (subpartitioning), i.e. a table partition can be also divided into subpartitions.

Configuration	*employees1*	*employees2*
Core2 Duo CPU T9600 @ 2.80GHz	0.92 s	0.26 s
QEmu	32.64 s	13.48 s

Table 4.10: Partition pruning in MySQL 5.1.42

By properly designing a division into partitions, it is possible to dramatically improve some queries. The idea is simple, during the query evaluation only the partitions that can contain the matching rows are browsed; this technique is known as a *partition pruning*. To verify whether a partition pruning is used during the query evaluation, one can use the EXPLAIN statement with the PARTITIONS keyword. To test the partition pruning, we have created a simple table *employees2* with the following structure:

```
CREATE TABLE employees1 (
  id INT NOT NULL,
  salary int(11) NOT NULL
);
```

```
CREATE TABLE employees2 (
  id INT NOT NULL,
  salary int(11) NOT NULL
) PARTITION BY RANGE (salary)(
  PARTITION p0 VALUES LESS THAN (25000),
  PARTITION p1 VALUES LESS THAN (50000),
  PARTITION p2 VALUES LESS THAN (75000),
  PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

we have divided it into four partition according to a value in the salary column. Note that by using the MAXVALUE keyword, the partition *p3* contains all employees with the salary higher than 75 000. The table *employee1* has been created with the same structure, however it has not been partitioned. We have prepared a Perl script that fills these tables with 10 million rows, then we have measured time required to calculate number of employees with salary from the interval [26 000, 49 000]. The test have been performed in the *qemu* virtual system and the hardware described in Section 2.2. The test results are summarized in Table 4.10. On the physical hardware, the execution of query is almost four times faster on the partitioned table. This is caused by the fact that only the partition *p1* that contains approximately 1/4 of rows is searched whereas all 10 million rows must be browsed in the nonpartioned table.

The range partitioning could be used for the *messages* table in the *DATE_log* database. The table contains debug and information messages produced by the DATE software package. Very often, the data acquisition experts need to know the behaviour of the system in a certain time period. Thus it would be possible to define range partitioning based on the values from the *timestamp* column. In fact, this behaviour is emulated by a special *cron* job; the job creates a new table for messages every day and puts older message into archive tables.

As the support for partitioning has been introduced in MySQL 5.1, we have decided not to use it yet. However, in newer versions of MySQL server (5.5. 5.6) improvements of the partitioning have been implemented. Additionally, the support for delayed replication have been added into the newer versions and finally, optimization for multicore systems have been added.

The proposed improvements need to be discussed with COMPASS data acquisition experts during the next frontend electronics meeting. If the proposal is approved and new servers are provided, it can be implemented during the planned technical stop of the CERN accelerators in 2013.

Chapter 5

Remote control room for the COMPASS experiment

In the first chapter of this work, it has been explained that the data acquisition system should also provide monitoring and control facilities. At larger experiments in the high energy physics, additional systems are required for control and monitoring of detectors, voltage systems, gas systems, and other equipment. The COMPASS experiment is operated from several workstations located in a control room situated directly in the experimental hall. From a software point of view, the control and monitoring at COMPASS is mainly based on the DATE package and the *Detector Control System*.

One very important advantage arises from the location of the control room. As it is situated directly in the hall with spectrometer, it is possible to regularly perform safety checks of the gas system and server room. Also, members of the shift crew can easily control access of experts into a restricted area with the spectrometer. On the other hand, the detector and data acquisition experts often need to travel to the control room to analyze and solve problems in their domain. As the hall is situated in forest, approximately 500 m away from the centre of the French part of the CERN campus, it is inconvenient to get on foot to the night shift. Luckily, a shift special shuttle has recently started to ply to the experimental hall. Moreover, it is not possible for members of the shift crew to have a warm meal during the shift. The environment is not very ergonomic, as the hall is very noisy and enclosed in the concrete shielding, thus the control room is lit by cold, artificial lightning. Finally, at the end of the data taking in December 2010, the radioprotection team detected that the level of radiation in the control room approaches the safety limits. A higher intensity of the beam is required for future scientific programs of the COMPASS experiment, therefore in order to protect the personnel, it has been decided to implement a remote control room by a technical coordinator.

Technical coordinator has ordered new workstations and LCD panels that should be deployed in the new control room. The new control room would be installed in the computer room on the ground floor of the office building used by the COMPASS collaboration situated in the French part of the CERN laboratory, approximately 500 m away from the COMPASS spectrometer. The network outlets in this room have been connected into the COMPASS internal network by the CERN IT division. The new control room should provide the same functionality as the old one. The regular checks of the server room and various gas mixing systems should be replaced by the IP cameras. However, it is still to be decided which systems should be monitored by these cameras.

In first section of this chapter, several possible methods of implementation of the remote access are analyzed and compared. In the following section, the proposed design of the remote control room is presented and applications used by the shift crew are introduced. Finally, the preparations for the unattended installation of the new workstations in the remote control room are explained in details. This section should serve as a manual for data acquisition experts of the COMPASS experiment in case a new workstation needs to be included into the control room.

The author of the thesis has supervised a team of undergraduate students who participated in installation and configuration of workstations in the remote control room, the results are summarized in [5].

1 Analysis and comparison of methods of the remote access

Before starting of installation of the operating system on the new machines, we needed to propose a method that should be used for implementation of the remote control and monitoring facility. We have proposed, analyzed, and compared two methods.

The first method would be based on sharing of the remote desktop via the *Virtual Network Computing* (VNC) system. The VNC client sends mouse and keyboard events over the network to the VNC server that is running on the host that shares its screen. The server accepts the events and returns the client screen updates. VNC is a multiplatform system, the clients can run on different operating system than the server, e.g. it is possible to control the remote linux desktop from the VNC client running on MS Windows system. The VNC would be used to provide a remote access from computers in the new control room to the desktops of computers in the original control room, i.e. the VNC clients would be installed on new machines in the remote control room, the VNC server would be installed on machines in the original control room in the experimental hall. Therefore, only one package with the server software would have to be installed on old servers and *de facto* default options could be used during the installation of the SLC operating system. This would greatly reduced the time required to set up the remote control room and we have considered it as one of the most important advantages of the desktop sharing method. Additionally, if a connection between VNC client and server is broken, the operation of control and monitoring software running on the VNC server is not affected and client can reconnect to the the server when the connection is recovered. On the other hand, as the VNC gives a full access to the remote desktop and the remote desktop is still accessible locally, a conflict between remote and local user could happen. We consider the danger of conflicts to be the most serious disadvantage of this method. Furthermore, if a remote machine would experience a crash, a manual intervention in the original control room would be required to restore the affected machine back into an operational state. Moreover, the *remote framebuffer* protocol that is used in communication between VNC server and clients is pixel based, therefore it is very demanding as far as network bandwidth is concerned. However, this disadvantage could be eliminated by using some other technology for desktop sharing that is based on exchange of graphics primitives instead of pixels such as a *Remote Desktop Protocol* RDP. Unfortunately, the RDP is a proprietary software and server part exists only for MS Windows operating system. Finally, a potential problem could occur if a member of the shift crew would accidentally close the window with the VNC client. Although starting of the VNC client is not very difficult, we must keep in mind that not all shift members are skilled computer users.

As the network outlets in the new control room have been connected into the COMPASS internal network, a second option of implementation of the remote control and monitoring could be used. This method would be based on integration of the new machines into the COMPASS network and installation the control and monitoring software on these machines. Clearly, this method would require more time to install and configure the machines. On the other hand, the new remote control room would be an equal clone of the old remote control room, both systems could be used in parallel to monitor the system (however, at most one user could actually control the system at one time); the user management is already implemented in control and monitoring software. If a machine in the original control room crashes, the operation can still continue from the new remote control room. As no additional software would be installed, this method should not cause any difficulties for members of the shift crew.

We have discussed these two methods with a technical coordinator of the experiment and a data acquisition experts. We have agreed that the advantages of the second method compensate the main disadvantage of the more complex installation and configuration process as it should be performed only once. In theory, this method could be used to implement the remote control and monitoring from almost any computer (e.g. workstations situated at home institutes of the Compass members) provided that it is connected into the Compass network. However, only remote monitoring should be performed from outside of the CERN. If a remote control of the experiment and data acquisition would be allowed, the detector experts could lose motivation to visit CERN and no one would be available for repairs of broken components.

2 Proposed design of the layout of the remote control room

Eight new workstations have been ordered for use in the remote control room. Before start of installation of these workstations, we needed to define users that should participate in the control and monitoring process and define deployment of the control and monitoring applications on the new workstations.

2.1 Definition of user roles

As a result of analysis of the existing control room of the Compass experiment, we have proposed several types of users of the designed remote control room: a *shift assistant*, a *shift leader*, a *data acquisition expert*, and a *detector expert*. A shift assistant and a shift leader form a shift crew that must be constantly physically present in the control room during the data taking period. The user privileges are summarized on Figure 5.1 and bellow:

- A shift assistant should use various components of the software package Date to monitor the data taking process and the *Detector Control System* to monitor the state of detectors. In the original control room, a shift assistant also regularly performed a safety visit of the gas barracks. In the remote control room, this safety visit is replaced by the IP cameras installed in these barracks. A shift assistant should report detected incidents to a shift leader and log the incidents into the electronic logbook.
- A shift leader should use the the human interface of the *dateControl* process (see bellow) to start and stop the data taking process. Together with a shift assistant, a shift leader should continuously monitor state of the system. A shift leader should be able to repair minor or documented problems with both data acquisition system and detectors. More serious or undocumented problems should be reported to a corresponding expert and entered into the electronic logbook. Finally, a shift leader should allow an access to the restricted zone of the experiment, if asked by the run coordinator.
- Data acquisition experts can modify configuration of the trigger and the data acquisition systems. They also repair serious problems with these systems.
- Detector experts can modify configuration of the detectors that form the Compass spectrometer. They also repair serious problems with the detectors.

During proposal of the control and monitoring software for the new data acquisition architecture we have extended this use case by adding a visitor and an administrator roles (see Chapter 6 of this work).

2.2 Proposed deployment of control and monitoring applications

After definition of user roles, we needed to propose a deployment of the control and monitoring applications on the workstations in the remote control room. The proposed deployment diagram is shown in Figure 5.2 and the corresponding applications are introduced in this section.

The application that are used in the control room can be divided into two categories: applications that control and monitor process of the data acquisition and applications that control and monitor detectors, beam line, and other hardware equipment. The first group is mainly represented by various subsystems of the Date package, while the second group includes *Detector Control System*, or CERN TV.

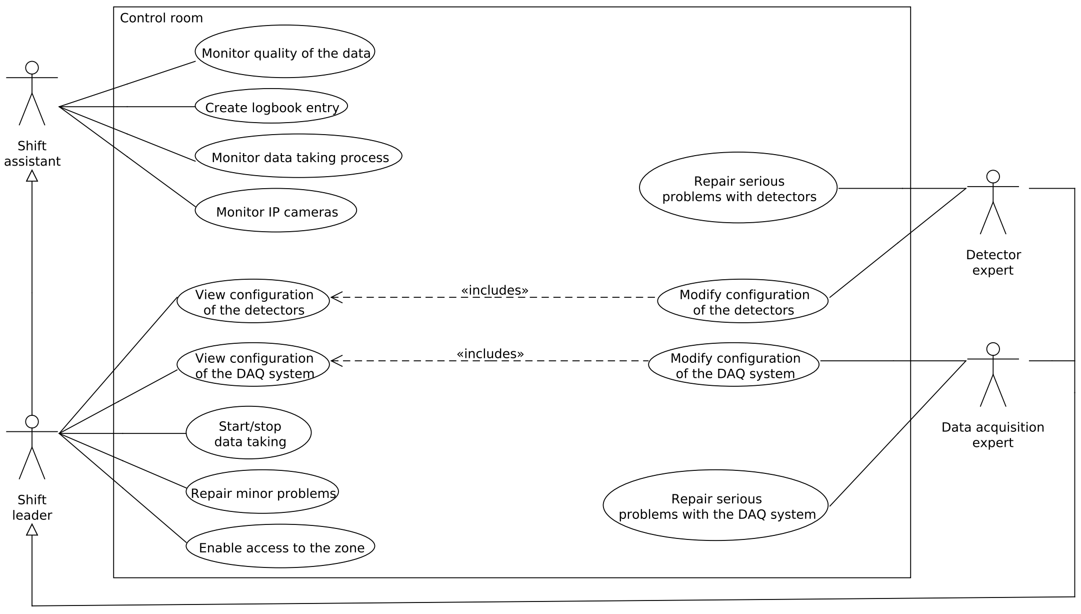

Control room
Shift assistant
Monitor quality of the data
Create logbook entry
Monitor data taking process
Monitor IP cameras
Repair serious problems with detectors
Detector expert
View configuration of the detectors
«includes»
Modify configuration of the detectors
View configuration of the DAQ system
«includes»
Modify configuration of the DAQ system
Data acquisition expert
Start/stop data taking
Shift leader
Repair minor problems
Repair serious problems with the DAQ system
Enable access to the zone

Figure 5.1: Users of the remote control room

The Date is a software package designed to perform data acquisition tasks in network environment. The package is described in Section 2.2.3, this part focuses on the Date components that are used by members of the shift crew to control and monitor the data acquisition.

The *dateControlHI* is a graphical application that provides the control functionality. It allows operators to start, stop, and configure the data taking process. It is possible to include and exclude event builders and readout buffers into the system and also configure detectors handled by the readout buffers. During the data taking, the application also displays additional windows containing status of spillbuffer PCI cards, logic engine, or status of the trigger control system. The application is also interfaced into the logbook system, it can be used to manage information about shifts and to add comments. Multiple instances of the application can be present in the system, however only one at most can actually control the system at the same time, the other instances may be used for monitoring. According to Table 5.1, the application should be deployed on the *pccorc31* workstation. One larger 24“ and one smaller 22” panels should be connected to this workstation.

The *infoBrowser* works as a graphical frontend over the *DATE_log* database that contains debug and information messages produced by the Date processes. An operator can interactively filter messages that should be displayed. The application can work in an online mode when it shows messages as they are stored in the database.

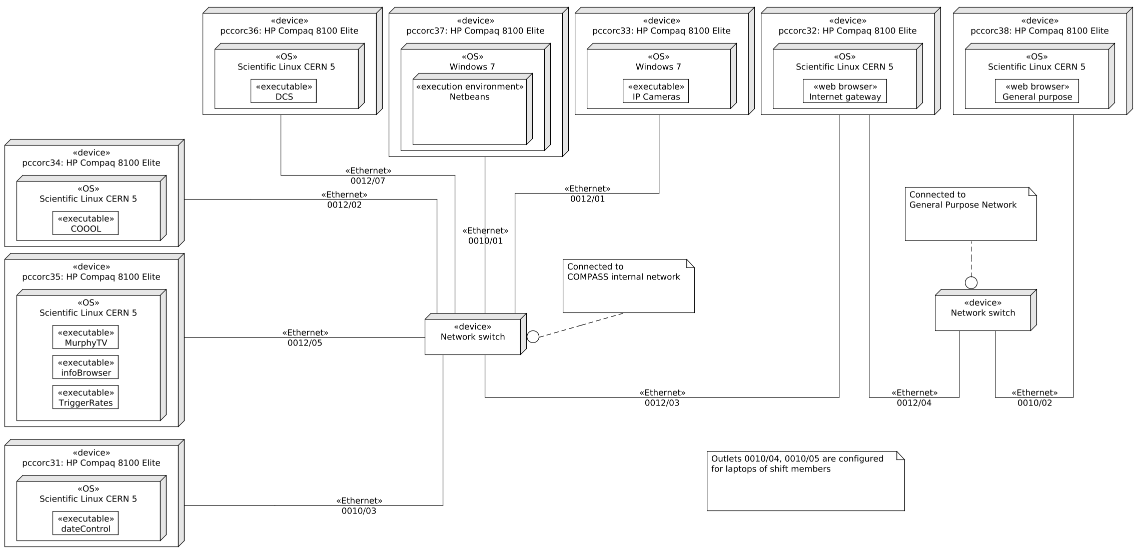


Figure 5.2: Deployment diagram of the remote control room

The *infoBrowser* should run on the same workstation as the *MurphyTV* application that monitors frontend electronics of detectors. During the data taking, *MurphyTV* reads and verifies event headers and displays errors related to quality of data such missing or damaged headers or inconsistent event numbers in the events. The shift crew should use the information provided by the *MurphyTV* to identify a source of problem and to try to correct it by reloading the corresponding module with frontend electronics.

On the third panel connected to the same workstation, the *trigger rates display* should be deployed. It is a web application that plots the current rates of various triggers (and vetoes) that form the resulting trigger signal. The application runs on the web server on the *pccodb10* server, it should be accessed through virtual address *pccodb00*. The information about trigger rates are now also integrated into the Detector Control System that can generate alarm in the case some trigger rate differs from its nominal value.

MurphyTV, *infoBrowser*, and *trigger rates display* should be always running on *pccorc35* workstation. As these applications provide vital information about health of the data acquisition system, each of them should occupy entire LCD panel. The panels should be mounted on wall in order to be visible from the entire control room.

The *COOOL* (Compass Object Oriented Online) is an application that uses remaining CPU power of an event builder to perform analysis of part of data. The results of analysis are displayed in the graphical frontend that runs on the run control machine in form of the Root histogram. During the shift, the *COOOL* histograms should be regularly compared with nominal values (that are superimposed into the plots) to discover potential problems such as high noise or dead channels. If a problem is detected, the shift crew should try to reload the corresponding frontend electronics, put a comment into the logbook, and notify the detector expert. Typically, faulty detector channels produce errors both in *MurphyTV* and in the *COOOL* histograms. If the channels contribute to the trigger signal, the corresponding trigger rate may also deviate from the nominal value. According to our proposal, the *COOOL* should be always running on large 24" panel connected to the *pccorc34* workstation. Smaller 22" panel is also connected to the same workstation, it should be used for web browser with an electronic logbook.

The *electronic logbook* is an application developed and integrated into the DATE system at the COMPASS experiment. Members of the shift crew should use the logbook to track all nonstandard events such problems and interventions that occurred during the shift. The logbook also contains summary information about runs. The information include start and stop time of the run, settings of the trigger control system, status of the spectrometer magnets, status of the beam line, or list of detectors included during the run. These parameters are inserted into the logbook automatically by the DATE processes. The contents of the logbook can be browsed in a web application built in the PHP language. The content of logbook is stored in the *runlb* database handled by the online database service *pccodb00* (see Chapter 4). The database is also replicated to the *compass02* server that is located outside of the COMPASS internal network. Therefore, it the logbook is accessible even by experts that are not physically present at experiment.

The slow control of the spectrometer, voltage and gas systems is provided by the *Detector Control System* (*DCS*). The system can be divided into three layers. The hardware devices controlled by the *DCS* are represented by the *device layer*. The *frontend layer* consists of software drivers for the hardware elements and provides communication protocols between the devices and the *supervisory layer*. The supervisory layer is based on a commercial package *PVSS-II*; it provides graphical interface for monitoring of the hardware elements. This layer can also produce visual and acoustic alarms in the case some of the monitored channels deviates from the expected behaviour (e.g. a voltage system trips). Additionally, the layer can also visualize time development of the monitored parameters, thus it is possible to observe slow changes of variables. Supervisory layer of the *DCS* should be deployed on the *pccorc36* workstation equipped with a single 24" LCD panel.

Cesar is an application used to control and monitor state of the spectrometer magnets and the beam line. Cesar can also be used to control access to the experimental zone during the data taking period. *Cesar* is built on the *Netbeans* platform, it should be installed on the *pccorc37* workstation.

Information provided by the accelerator control room are provided as a TV channel *SPS Page 1* broadcast by a CERN short circuit TV. The channel displays cycle of the SPS accelerator, beam intensity, beam symmetry, or number of particles delivered to the COMPASS target. The SPS operators can also used the channel to display short messages about planned works and unexpected problems on the accelerator. The TV program also generates acoustic signal during extraction phase of the SPS cycle. The information are also available in form of the web page or live stream, thus it is possible to replace a television with a web browser or media player running on one of the run control machines.

A workstation with two LCD panels should be dedicated to displaying the live video recorded by the Internet Protocol cameras. These cameras should replace a need of regular safety checks of gas systems and server room with readout buffers and event builders. The software delivered with the cameras requires the Windows operating system, thus we have decided to installed Windows Vista on the *pccorc33* workstation. The software enables rotation and zooming of the cameras. However, the exact location of the cameras is still to be decided.

We have also decided to installed Windows Vista on the *pccorc38* workstation that is connected into the general purpose network. This workstation should be used by members of the shift crew for generic tasks such as sending e–mails or finding information on the Internet.

3 Kickstart based unattended installation

Eight new *HP COMPAQ 8100 Elite* workstations have been delivered together with sixteen LCD panels and several dedicated PCI Express graphics cards. The new machines had no operating system preinstalled. The roles that we have assigned to the new workstations are summarized on deployment diagram in Figure 5.2; we have decided to install 32–bit version of Scientific Linux CERN 5 on 6 of those workstations and Windows Vista on remaining two machines. Six workstations should be connected into the COMPASS internal network, one general purpose computer should be connected into a *General Purpose Network* (GPN), i.e. Internet. The last machine should work as a gateway, it should contain two network interfaces, one connected into the COMPASS network, the other into GPN. The purpose of various control and monitoring applications is described in more details in the following section of this chapter.

Before operating system installation, we needed to register the new machines into the central LANDB database. This database contains all device that are connected to the CERN network. Owner of the device must enter hostname, hardware address of the network card, manufacturer, model name, and serial number of the device, type and version of the operating system, location, and other relevant information including short description of purpose of the device. After filling these information into a web form, the LANDB system assigns the IP address, address of the default gateway, name servers, and time servers that should be associated with the registered device.

As we needed to install the SLC system on multiple machines, we have decided to use the unattended installation. The *Anaconda* installer program included in the SLC system supports such type of installation based on the *kickstart* method. The method is based on the plain text “kickstart” file that contains installation instructions and parameters. At the start of the installation process, the kickstart file is passed to the Anaconda program which parses it and performs the installation according to the stored instructions without user interaction. As an additional benefit, the kickstart can be reused later, after the initial installation. This can be useful in the case that the machine crashes and needs to be quickly reinstalled into the original state.

Several ways of preparing the kickstart file are available. As the kickstart is a plain text file with relatively simple syntax described in the Red Hat Enterprise Linux Installation manual [40], it is possible to create it manually from a scratch. This is probably the most time consuming and the most difficult method. It is preferable to edit some example kickstart file. To create a sample kickstart file that could be used as a starting point for further modification, it is possible to install the system normally using the interactive installation because the Anaconda creates the kickstart file that describes the installations steps requested by user; the resulting kickstart file is stored in the `/root` directory. To edit it, one can use any text editor. However, graphical tool called *Kickstart Configurator* also exists.

To simplify a deployment of multiple similar machines (such as run control computers), the concept of the *kickstart templates* is created. Templates are used to represent a group of similar computers. Therefore a template file contains parameters and options that are common to all machines in the given group. The options in which the machines differ (typically network configuration) are represented by variables that are substituted by corresponding values when the actual kickstart files are generated from the template. Template method is very flexible as it support the *include* keyword. Thus it is possible to split a template into several *subtemplates*. A subtemplate can be used for more than one group of computers, e.g. run control computers and gateway computers may use the same subtemplate with information about disk partitioning. Furthermore, the kickstart installation method is integrated into the CERN *Automated*

Installation Management Systems (AIMS) facility.

To use the AIMS facility, one needs to create a corresponding master template file (and possibly also several subtemplates) for a group of computers that need to be installed. Then list of computers within a group must be provided in a text file; each line of this text file contains a hostname of the machine and name of the master template, e.g. the line

```
pccorc32 RC.tmpl
```

denotes that the kickstart file for the computer *pccorc32* should be created from the template stored in the `RC.tmpl` file. Then, the corresponding kickstart files for all computers in the group are generated by the *update-kickstart* utility that acts as a frontend of the template system. The tool queries the LanDB with the hostname of the machine to retrieve the connection parameters, it parses the template file(s) and replaces the variables with the corresponding values retrieved from the LanDB and finally, it produces the kickstart file.

Then, the *kickstart4host* script may be used to generate the installation media. The script calls the *aims2client* utility that takes the kickstart file and prepares the network based installation. By using the *kopts* parameter of the *aims2client* utility, it is possible to specify options that should be passed to the linux kernel during the installation. We have used this possibility to force installation in the 1024x768 resolution by adding the *vga=0x317* option as the installation process would freeze with the default resolution. By passing the *name* option, one can specify which version of operating system should be used. We have decided to install 32–bit version of SLC 5.7 that corresponds to the name *slc57_i386*.

The kickstart files and *update-kickstart* and *kickstart4host* utilities are stored on the distributed file system *afs*. For the security reason, only authorized users can edit the kickstart files and templates. Additionally, the utilities can be started only from the gateway computers *pccogw0x*.

3.1 Operating system installation

To start the installation, one needs to enable booting from network in the configuration utility of the *Basic Input Output System* (BIOS). Then, after the restart, linux kernel is booted over the network using the *Preboot Execution Environment* PXE. The kernel initializes the hardware and prepares the installation program Anaconda. The kickstart file published by the AIMS facility is retrieved and passed to the Anaconda. Anaconda performs the installation according to the instruction. After the installation, the machine is unregistered from the PXE thus after the next restart, system starts normally from the local hard drive.

The content of the kickstart file for the *pccorc31* machine that should be used for running the control application of the data acquisition system *Date* will be described. Kickstart files for other linux machines are similar as they are generated from the same template. The kickstart file can be divided into several sections. In the first section, the parameters of the installation process are defined. At first, language and keyboard layout is set to English, then the network interface is configured according to parameters retrieved from the LanDB. Then, the following line in the kickstart file

```
nfs --server linuxsoft --dir /cern/slc57/i386
```

defines that the RPM packages should be downloaded from the directory that corresponds to 32–bit SLC on the network file server `http://linuxsoft.cern.ch`. In the second part of the kickstart file, the partitioning schema of the hard disk is designed:

```
clearpart --all --drives=sda
part /boot --size 512 --ondisk sda
part swap --size 2048 --ondisk sda --fstype swap
part / --size 8192 --ondisk sda --fstype ext3 --grow
```

At first, all existing partitions are cleared on the primary hard drive (*/dev/sda*), then three new partitions are created. The first new partition is designed to contain the */boot* directory, i.e. boot loader, linux kernel, and image of initial ram drive. Second partition should be used as a `swap` space. Finally, the last partition should serve as the root (`/`) of the directory tree. We have decided to format the partition using the journaling file system *linux extended* (ext3). The parameter *grow* specify that the partition should use the all remaining space of the hard disk.

After the hard disk partitioning, the time zone is set. The system is configured to automatically start the *X window system* after boot with the *KDE* set as the default desktop environment:

```
xconfig --defaultdesktop=kde --startxonboot
```

Note that some machines such as event builders or readout buffers do not need graphic desktop environment, thus is to possible to use the *skipx* option to do not install the X window system. Then, the password for the user root is defined. For security reasons, the password is encrypted. Also, the *shadow* file is enabled. This configuration file stores password of system users in encrypted form. On the other hand, we have decided to disable firewall as the machine should work in the internal network and should not be exposed directly to the Internet. In the next step, the boot loader is installed:

```
bootloader --append="selinux=0" --location mbr
```

We have decided to install it into the *Master Boot Record* of the hard disk. We have used the *append* parameter to enter additional options that should be passed by the boot loader to the linux kernel. The option *selinux=0* disables the functionality of the *Security Enhanced linux*. In the following section of the kickstart file, list of RPM packages that should be installed is defined. It is possible to enter a single packages as well as whole package group, the name of the package group is prefixed by the @ character, e.g. the *@kde-desktop* group contains all packages that provide the graphical desktop environment KDE. Although the KDE should be used as a default desktop environment (see above), we have also installed the GNOME environment as it might be preferred by some shift members. We have also selected to install other package groups that contain packages that could be used during shifts such as a web browsers, text editors, or publishing tools. Additionally, several development libraries such as Tk toolkit or MySQL client libraries required by the DATE package have been selected for installation. To the beginning of the package list, we have added the *resolvedeps* option that should guarantee that Anaconda installs selected packages together with all dependency packages.

3.2 Configuration of the operating system

After the installation of packages, the system is rebooted and Anaconda executes commands that are included in the *postinstallation* section of the kickstart file. At first, address of the gateway is appended to the system configuration file `/etc/sysconfig/network`. Then, the system time is synchronized with the time server over the *Network Time Protocol* NTP. In the following step, the group *daq*, user accounts *daq* and *objsrvvy* that should be used by members of shift crew and DATE processes are created. Contents of several directories is retrieved from the repository on the COMPASS file server *pccofs01*. These directories contain preprepared configuration files such as a list of repositories for the *yum* package manager. From these repositories, several additional packages including CERN libraries, GNU scientific library, MySQL driver for the Perl and TCL language, Ganglia monitoring system, or plotting program gnuplot are installed. The ownership of the synchronized directories is set to the root user using the *chmod* tool. The online directory is created in the root `/` directory, network file system is mounted under this directory, and corresponding entry is appended into the `/etc/fstab` configuration file:

```
echo "pccofs00:/online  /online nfs     intr,hard" >> /etc/fstab
```

The `/etc/fstab` file contains static information about various file system and mount points. Each file system is represented by a single line the file that consists of up to 6 columns. In the first column, the block device or network file system as in this case that should be mounted is specified. In the second column, the address of mount point is defined. The file system used on the mounted device is described in the third column. The fourth column contains options associated with the corresponding file system. The *intr* option allows signals to interrupt the operation in case a major timeout appears; if the *hard* option is used, then the "server not responding" message is printed on console in case a major timeout appears, additionally the timed out operation should be retried indefinitely. Complete list of option can be looked up in the manual page (cf. *man 5 nfs*). The fifth and six columns contain information for the *dump* and *fsck* tools. If these column are not present, the tools assume that the file system does not need to be dumped (by the *dump* tool) or checked (by the *fsck* tool).

In the last part of the kickstart file, the script that should be executed after the first boot of the system is prepared and stored in the `/etc/rc.d/init.d/` directory. At first, the script sets up the environment variable *HOST* with hostname of the machine. Then, the public ssh keys of the machine are copied to the shared network file system and in turn, ssh keys of the other machines in the *Compass* internal network are retrieved from the network file system. Additionally, the `memtest86+` binary file is copied from the network to the `/boot` directory and corresponding menu entry is added to the configuration of the Grub loader. In the next step, the *updatedb* program creates a database for the *locate* program that is used for fast finding of files by names. As the *updatedb* scans the entire directory tree, it takes several minutes to build the database. When the database is completed, several symbolic links are created in the root directory `/` to simplify accessing directories with the DATE installation and CERN libraries.

In the following step, the script uses the *chkconfig* tool to enable or disable starting of several system services. The *ntp* service that serves for time synchronization and the *gmond* service that sends monitoring data to the Ganglia system are enabled. On the other hand, the *yum-autoupdate* service that automatically updates RPM packages, the *afs* service that provides connection to the distributed afs file system, the *bluetooth* service, the *iptables* service, and other service are disabled. Then, the system is configured to use the printer located in the control room and the printing service *cups* is enabled. Then, the script removes itself in order not be executed after the next boot and finally, it reboots the system.

Last line of the kickstart file contains instruction to remove the machine from the PXE, thus the installation process does not start over after the following start of the system.

Configuration of the X window server

After the operating system installation, multiple LCD panels needed to be connected to some of the machines. Namely, the *pccorc35* machine designed to display *infoBrowser*, *MurphyTv*, and trigger rates (see Table 5.1) should use three panels. We have installed two dedicated

Ati Radeon 5450 graphic cards into the PCI Express slots of the machine. The graphical tool used to configure display failed to work with more than one graphics card. Therefore, we have decided to install proprietary graphic driver and edit the configuration file of the X window server `xorg.conf` manually. The configuration file consists of sections that correspond to monitors, screens, graphic cards, and other devices including keyboard and mouse that are used as a building blocks of the multiple display configuration. At first, the *Server layout* is defined:

```
Section "ServerLayout"
        Identifier     "Multihead layout"
        Screen      0  "Screen0" 0 0
        Screen         "Screen1" LeftOf "Screen0"
        Screen         "Screen2" LeftOf "Screen1"
        InputDevice    "Keyboard0" "CoreKeyboard"
        InputDevice    "Mouse0" "CorePointer"
EndSection
```

The layout consists of three screens arranged horizontally from right to left and one keyboard and a mouse. In this particular case, one screen corresponds to one physical monitor. Each screen is represented by its own section in the configuration file:

```
Section "Screen"
        Identifier "Screen0"
        Device     "Device0"
        Monitor    "Monitor0"
        DefaultDepth     24
        SubSection "Display"
                Viewport   0 0
                Depth     24
                Modes     "1920x1200"
        EndSubSection
EndSection
```

To each screen, a device (i.e. graphic card) and a monitor is assigned. The screen should use the $1920x1200$ resolution with 24–bit color depth. The section for screens *Screen1* and *Screen2* as defined in the *ServerLayout* section is similar. The graphics card is described in the *Device* section:

```
Section "Device"
        Identifier  "Device0"
        Driver      "fglrx"
        BusID       "PCI:32:0:0"
        Screen          0
EndSection
```

The device should be handled by the proprietary *fglrx* driver. The *BusID* option defines the address of the PCI Express bus on which the card is connected. The corresponding address can be found in output of the *lscpci* utility. The output of the card should be displayed on the screen 0. Graphic card can have multiple connectors, the other connector of this card is defined as device *Device1* and displayed on the screen 1. The second graphics card is connected to address `PCI:1:0:0`. Only one connector of this card is used, it is defined as *Device2* and displayed on screen 2. The LCD panels are defined in the *Monitor* section:

```
Section "Monitor"
        Identifier   "Monitor0"
        Option      "VendorName" "ATI Proprietary Driver"
        Option      "ModelName" "Generic Autodetecting Monitor"
        Option      "DPMS" "true"
        HorizSync   30-80
        VertRefresh 50-100
EndSection
```

The monitor section has been automatically configured by the graphical utility provided with the proprietary drivers. Again, section for the *Monitor1* and *Monitor2* are similar. We have also enabled *Composite* extension of the X server. The extension provides a buffer than can be used by applications for off–screen rendering of windows. The composite manager merges the windows into an image that represents the entire screen and renders it on the physical display.

```
Section "Extensions"
        Option      "Composite" "Enable"
EndSection
```

Additionally, we have set the AIGLX (Accelerated Indirect GL X) option on to enable GL accelerated effects on the desktop.

```
Section "ServerFlags"
        Option      "AIGLX" "on"
EndSection
```

It is also possible to configure X window system to span one screen over multiple physical monitors (i.e. it is possible to drag window from one monitor to the other) instead of defining a separate screens for each monitor.

Finally, the `xorg.conf` file can also contain sections dedicated to configuration of various input devices such as keyboards, mice, or touch pads. The following section describes a keyboard:

```
Section "InputDevice"
        Identifier  "Keyboard0"
        Driver      "kbd"
        Option      "XkbModel" "pc105"
        Option      "XkbLayout" "us"
EndSection
```

The keyboard is handled by the standard *kbd* driver. The device is recognized as a generic keyboard with 105 keys. However, the *XkbLayout* is the most interesting option as it can be used to define multiple keyboard layouts; however it this particular case, only the English layout is defined.

4 Summary

We have been asked to implement a remote control room for the Compass experiment. We have evaluated possible scenarios and decided to integrate the new run control machines into the Compass network. We have defined users of the remote control room and proposed deployment of the control and monitoring applications of the new workstations. Then we have prepared kickstart files that have been used to install and configure these workstations according to the proposal. The purpose of the installed workstation is summarized in Table 5.1. We have successfully started the data taking from the remote control room. Thus, from functionality point of view, the control room can be put into an operation. However, air conditioning of the room still needs to be installed. By implementing remote control and monitoring, substantial amount of money that would otherwise have to be invested into the additional shielding of the spectrometer has been saved.

Workstation	*System*	*Purpose*	*Network*
`pccorc31`	SLC 5	dateControl	COMPASS
`pccorc32`	SLC 5	general purpose	COMPASS and GPN
`pccorc33`	Windows 7	IP cameras	COMPASS
`pccorc34`	SLC 5	COOOL, logbook	COMPASS
`pccorc35`	SLC 5	infoBrowser, MurphyTv, trigger rates display	COMPASS
`pccorc36`	SLC 5	Detector Control System	COMPASS
`pccorc37`	SLC 5	Cesar	COMPASS
`pccorc38`	Windows 7	Internet access	GPN

Table 5.1: Workstations in a remote control room

Although, the remote control and monitoring of the experiment is in principle possible from any device that can connect to the COMPASS network, we do not recommend remote control from outside of the CERN because without experts on site, no one would be able to fix hardware problems of the detectors.

Because official support of the SLC 4 ended in December 2010, we also used the kickstart files to perform the migration of the majority of servers participating in the data acquisition on version 5 of the Scientific Linux CERN during the winter shutdown in 2010. In cooperation with data acquisition experts, we prepared several kickstart templates for different roles such as event builders, readout buffers, or file servers.

Chapter 6

Analysis and proposal of the software for new data acquisition architecture

The current data acquisition system of the COMPASS experiment can be divided into several layers. The *frontend electronics* perform readout and digitization of the detector channels. The data coming from multiple channels are gathered by the *concentrator modules* CATCH and GeSiCA. Then, the data produced during the beam extraction are buffered in the *spillbuffers* which are PCI cards equipped in the *readout buffer servers.* Readout buffers are connected to the *event building servers* through the Gigabit Ethernet. This architecture is described in more details in Section 2.1 and in [23].

The system is in operation since the technical run 2001 that was followed by the first physics run in 2002. During the operation, several new detectors and detector channels have been added into the system. Also, the beam intensity and trigger rates increased. Consequently, total amount of recorded data increased from $260\,TB$ in 2002 up to approximately $2\,PB$ in 2010. To compensate the higher data rates, the hardware of event builders and readout buffers was exchanged several times; Table 6.1 summarizes evolution of hardware of the readout buffers, similar evolution of the hardware can be also seen for event building machines.

Year	*Number of new ROBs*	*Processor*	*RAM*
2002	16	2× Intel Pentium III 866 MHz/1 266 MHz	1 GB
2004	19	2× Intel Pentium III 1 000 MHz	1 GB
2006	10	2× Intel Pentium 4 Xeon 3 600 MHz	4 GB

Table 6.1: Evolution of the hardware configuration of the readout buffers according to [1, 23]

Adding new hardware requires not only money but also manpower needed to install, configure, and test new servers. Since the older hardware is still used, several generations of servers coexist in the data acquisition. Consequently, some effort is also needed to distribute the load of servers according to their age and performance. Unfortunately, the spillbuffer cards are based on the PCI bus that is today considered as deprecated, therefore simple upgrade of hardware is not option for the nearest future.

Two upgrade scenarios are being considered. The first scenario proposes to develop the PCI Express version of the spillbuffer cards. This scenario would require replacement of readout buffers with new servers equipped with the PCI Express bus. On the other hand, with the exception of the kernel module for the spillbuffer card, no software development would be involved. According to the second upgrade scenario, the readout buffers and event builders should be replaced with a custom made cards based on the *FPGA* technology. This would decrease number of components and increase reliability of the system. As an additional benefit, the existing readout buffers and event builders could be reused as an online filter farm. On the other hand, this scenario would involve development of the new control and monitoring software for this new hardware architecture. This hardware architecture is being developed at the Technical University in Munich, the software for this architecture is being developed in Czech Technical University in Prague.

This chapter focuses on software development for the new data acquisition architecture. At first, the hardware part of the architecture is briefly introduced. Then, the results of the requirements analysis are summarized. The proposal of the control and monitoring software that fulfills these requirements is described in the following section. The author of this thesis has supervised two undergraduate students who implemented the proposal, [4, 19]. Finally, the performance of this implemented version of the proposed system is evaluated.

1 New data acquisition hardware

The main idea of the upgrade is to replace the readout and event building network with a custom made hardware based on the *Field Programmable Gate Array* technology, [18]. The first layers of the data acquisition system, i.e. the frontend electronics and concentrator modules, would remain unchanged. However, data from the concentrator modules would be transferred into the first layer of the FPGA cards instead of into the spillbuffers. The FPGA cards would be used in two layers. First layer would perform multiplexing of data from multiple sources and transfer the multiplexed data to the second layer of cards that would perform the event building. Only change of configuration would be needed to change the functionality of a card; there would be no need to modify the firmware.

The card is equipped with 4 GB of the RAM, 16 serial links that provide bandwidth of 3.25 GB/s. Also, a *MICO32* softcore processor is included on the cards. This processor would be used to power the application that would communicate with the monitoring and run control software. The card is designed as a module for the *ATCA* carrier card. Each ATCA card can hold four FPGA modules. The first prototype of the FPGA card was developed in summer 2011, the carrier card is in development as in spring 2012. The complete functionality would be implemented in eight carrier cards. Six cards would perform data multiplexing to effectively utilize the serial links, the remaining two cards would perform the event building. The complete events would be sent to the online farm.

Three independent layers of data transmission channels participate in the new data acquisition architecture:

1. The time control system (TCS) serves for the time synchronization of various components of the system. It distributes absolute time, trigger, and event identification over the optical fiber network.
2. Event building network is based on the FPGA modules, each module acts as an unblocking switch.
3. The flow control network is based on the Ethernet. This network provides information about status and health of the various components of the system as well as about quality of the data. This network is also used by the control software for sending commands and configuration requests to the hardware. The control software communicates with an embedded application that is powered by the softcore processor on the FPGA card.

A new software for monitoring status of the various components and for controlling the operation of the system would be required, the following chapter analyses requirements that the software should fulfill.

2 Requirements analysis

At first, the functionality of the software needed to be defined. The existing system uses the Date package that performs all data acquisition tasks ranging from event building to run control. In the new architecture, the functionality provided by *readout*, *recorder*, and *eventBuilder* Date processes (eee Section 2.2.3) is implemented directly in the FPGA cards. Therefore, the software for the new data acquisition system should focus especially on functionality connected to control and monitoring including configuration of hardware and information reporting. Additionally, the software should also include a user friendly interface that would be used by members of the shift crew to operate the system, e.g. to start or stop data taking process. The data acquisition expert should use this interface to configure the hardware and to store the configuration into a database. The software should pass the commands issued by the operators to the hardware. In turn, the software should also collect information about health of the hardware including buffer usage, free memory, or data rates. These information should be used to implement the load balancing and also presented in a graphical form easily comprehensible by shift members. Besides the health of the hardware, the software should also monitor the quality and integrity of data, e.g. to verify the event headers or calculate check sums of data. Furthermore, the software should also collect message generated by all its subsystems and stores them into a database. Some graphical frontend working over this database should also be developed to simplify browsing of the stored messages.

The majority of the above mentioned functionality is already supported by the Date package, therefore we have evaluated a possibility of using the package on the new data acquisition architecture, [14]. We have decided not to use the Date software for the new system because of the following reasons:

- DATE software requires to be installed on the linux operating system running on the x86–compatible hardware. However, in the new architecture, part of the software processes should be deployed on the embedded linux running on the *MICO32* softcore processor. Therefore, a considerable effort would be required to port the corresponding Date processes on the different hardware platform.
- Date is a very flexible system, it can be used by full scale experiments with hundreds of nodes as well as by small laboratory experiments with single processing node. The package can work in the collider mode as well as in the fix target mode. As a result, Date is very complex system. We prefer to use much simpler software for the new data acquisition architecture.
- By developing a custom software, a potential dependency on the third party software would be diminished.
- During several years in operation at Compass experiment, several undocumented modifications have been implemented into the system, therefore starting with a new system would be a good opportunity to complete the documentation and knowledge base about the system.

On the other hand, it is essential that the new software uses the same data format as defined by the Date. This is probably the most important requirement on the new software, it guarantees that the events assembled by the new system remain in the format compatible with software for reconstruction and physical analysis. The DATE architecture should also be used as a source of inspiration during development. Additionally, several DATE processes could possibly be reused with small modifications, e.g. the COOOL program that performs analysis of part of data could be deployed on the online farm. Additionally, the *MurphyTV* application that monitors quality of the data could be reused as the event format remains unchanged. Also, the electronic logbook should be integrated into the new system.

The software should be distributed on several distributed nodes, the connection is to be based on the Ethernet technology. We have discussed the possibility of using several communication libraries and decided to use the *Distributed Information Management* (DIM) software package that is already used in the frontend layer of the *Detector Control System* of the experiment. Therefore, usage of the library should simplify integration of the new monitoring subsystem into the DCS.

In previous chapter, the installation of the remote control room has been described. The new software should be able to control and monitor the data acquisition hardware from this remote location. Additionally, the software should handle multiple user roles, e.g. shift members should be able to start and stop data taking, while the experts should have the full control over the software. Also, multiple users should be able to use the monitoring facilities simultaneously, while only one user at most should control the system in each time. Furthermore, the application providing the user interface should support multiple platforms including GNU/Linux with X window server and MS Windows.

On the other hand, the data acquisition system does not have to be controlled in the real time. This simplifies the software development because there is no need to use any special real time library, nor real time operating system.

2.1 Evaluation of the DIM library

The monitoring software should be able to update status information about hardware at rate of at least 100 Hz. Therefore, we needed to verify if a system built on the DIM communication package is capable of exchanging messages at this frequency. Additionally, we needed to compare performance of the C++ and Java interfaces of the library.

Definition of the test

For this purposes we have proposed a simple test case scenario. In this scenario, a publisher publish one monitored information service and one command service. The subscriber subscribes to this monitored service and sends a command to the publisher. When publisher receives this command, it updates value of the monitored service which triggers subscriber to get the update information. When subscriber receives the updated information, it sends another command and the cycle is repeated. The time required to complete one million of iterations is measured for C++ and Java versions of the test case. We have repeated the tests for different sizes of message provided by the monitored service, therefore the results should be also used to define the optimal size of message with information about health of hardware.

At first, the source code of the C++ version of the publisher will be analyzed. In the main function, the new command service is created and the DIM server is started:
By calling the start static method of the `DimServer` class, the DIM server is registered at the *DIM Name Server* DNS and started. The parameter of the method defines a unique name of the server. The server is handled in the separated thread, thus the main thread can enter the

Listing 15 Main function of a sample DIM server

```
int main(int argc, char **args){
  Command cmnd;
  DimServer::start("BENCH");
  while(true)
    pause();
}
```

infinite loop in which it does nothing. The `Command` class extends the `DimCommand` class of the DIM library. In the following listing, the code of constructor of the `Command` class is displayed:

Listing 16 Custom DIM command

```
Command(): DimCommand("BENCHSRV/CMD", "I"){
  p = new packet;
  for(int i = 0; i < SIZE; i++)
    p->buffer[i] = 0;
  service = new DimService("BENCHSRV/SRV", "I", p, sizeof(packet));
}
```

At first, the constructor of the ancestor (i.e. the `DimCommand` class) is called, it takes two parameters. First parameter defines a name of the DIM command that is passed to the DNS; the other parameter describes format of the command. The letter *I* denotes that the command is represented as an integer value. Then, the constructor creates new information service. Several versions of the `DimService` constructor exist, this version takes four parameter: name of the service, format of the service, pointer to memory location that stores the service data, and size of the service data. The service format is used during exchange of information between different platforms, it defines contents of the service data in the following form: *T:N[;T:N]*[;T]*. The *T* character defines type of item, e.g. *I* is used for the integers, *C* for characters, *F* for floats. *N* defines number of items of given type. Complete description of the format can be looked up in the DIM manual, [31]. In this particular case, an integer array defined as

```
typedef struct packet{
  int buffer[SIZE];
} PACKET;
```

is used as a data structure that holds the service data. The array is initialized to contain zeros in all items. The first item should be used as a counter of exchanged messages. The `Command` class contains method `commandHandler` overloaded from the base class:

Listing 17 Custom command handler

```
void commandHandler(){
    int cmd = getInt();
    p->buffer[0]++;
    if(cmd == 0) p->buffer[0] = 0;
    service->updateService();
}
```

The method is called whenever a new DIM command is received from any DIM subscriber. Several methods that can receive the command value are available. In this particular case, the `getInt` method is used to receive the integer value of the command. As defined in the proposal of the test case scenario, the content of the information service is updated and the update is announced by calling the `updateService` method. If a subscriber sent a command with value 0, the counter of iteration is reset back to zero. This completes description of the publisher.

In the main function of the subscriber application, the DIM client is created. Again, the client code is handled in the separated thread, thus the main thread enter infinite loop in which it does nothing:

Listing 18 Main function of the DIM client

```
int main(int argc, char **args){
    Client client;
    while(1) {
        pause();
    }
    return 0;
}
```

The `Client` extends the `DimClient` class of the DIM library that implements several static methods related to the DIM clients/subscribers such as sending DIM commands. In the constructor of the `Client` class, a data structure for storing the service data is prepared and several variables that should track progress of the test are initialized:

Listing 19 Constructor of the DIM client

```
Client(): info("BENCHSRV/SRV", -1, this) {
  start = true;
  p = new packet;
  progress = 0;
}
```

Also the instance of the `DimInfo` class is constructed; this class implements subscription and reception of the DIM service. The `Client` class contains the `infoHandler` method overloaded from the `DimClient` class:
The method is called when the client subscribes to the service and then each time the publisher calls the corresponding `updateService` method (see above). The `getInfo` method of the `DimClient` class can be used inside the handler to return a pointer to the instance of the `DimInfo` class that provides information about currently handled service. The `DimClient` can handle multiple services at the same time, the `getInfo` method makes it possible to distinguish these services. The `DimInfo` class provides the `getData` method that returns a void pointer (`void*`) to the service data. Then, the value of the `start` logical variable is used to determine if the handler has been called for the first time, If this is the case, then the command 0 is sent to the publisher. This command forces the publisher to reset the counter of iterations back to zero (see above). Also, the current timestamp is assigned to the `starttime` variable. If the handler has not been called for the first time, the command 2 is sent to the publisher which triggers the publisher to update the information service *BENCHSRV/SRV*.

Finally, if the value of the first item of the integer array reaches 1 000 000, the current time stamp is taken again, elapsed time, number of exchanged messages per second, and transfer speed

Listing 20 Service reception using the infoHandler method

```
void infoHandler(){
  DimInfo *curr = getInfo();
  if(curr == &info) {
   p = (packet*)curr->getData();
   if(start) {
      start = false;
      DimClient::sendCommand("BENCHSRV/COMMAND", 0);
      starttime = time(0);
      i = 0;
   } else {
      DimClient::sendCommand("BENCHSRV/COMMAND", 2);
   }
```

is calculated and printed on the standard output. The the subscriber program is terminated using the `exit` function defined in the `unistd.h` header.

Results of the test

The test has been evaluated on the local 100-MBit/s network, it has been repeated for sizes of the service data ranging from 4 B till 256 kB. The publisher has been deployed on a workstation powered by quad core processor Intel Core Quad running at 2.5 GHz, the subscriber has been deployed on a laptop powered by dual core processor Intel Core 2 Duo running at 2.8 GHz. Both machines have been equipped with 4 GB of RAM. For smaller sizes of messages, the system has been able to exchange approximately 3 500 messages per second. With increasing message size, the network utilization is also increases and approaches the limit of 12 500 kBit/s given by the hardware. The results seem to be in good accordance with a similar benchmark performed by C. Gaspar, [31].

Message size	Data flow	Msg per seconds	Message size	Data flow	Msg per second
4 B	14 kB/s	3 700	2 kB	3 899 kB/s	1 900
8 B	29 kB/s	3 700	4 kB	7 246 kB/s	1 800
16 B	58 kB/s	3 700	8 kB	7 407 kB/s	900
32 B	116 kB/s	3 700	16 kB	8 840 kB/s	600
64 B	233 kB/s	3 700	32 kB	10 390 kB/s	300
128 B	456 kB/s	3 700	64 kB	10 667 kB/s	170
256 B	923 kB/s	3 700	128 kB	11 035 kB/s	90
512 B	1 582 kB/s	3 200	256 kB	11 179 kB/s	40
1 kB	3 690 kB/s	3 700			

Table 6.2: Performance of the C++ interface of the DIM library

Java version uses the native C code through the *Java Native Interfaces* (JNI) calls. As can be seen from the Figure, the performance hit caused by JNI calls is about 20 % for smaller messages; for larger messages, the performance hit can be neglected. Unfortunately, the Java interface of the library is not complete, therefore we have decided to use the C++ version for development. The requested number of messages exchanged per seconds should be feasible with the DIM library.

3 Proposal of the run control and monitoring software

We have used the results of the analysis of requirements summarized in the previous section to design a proposal of the run control and monitoring software. According to the proposal, the software consists of the following types of main processes: DIM name server, master, slaves, user interfaces, message logger, message browser, and database server.

The software should be deployed on approximately 30 distributed nodes of various types: FPGA cards, online computers, database servers, or even laptops and personal computers. All these nodes should be connected into the same network, the communication should be based on the DIM library. The library uses the name server DNS that serves for connecting clients/-subscribers to servers/publishers. The DNS process should be deployed on the x86–compatible server.

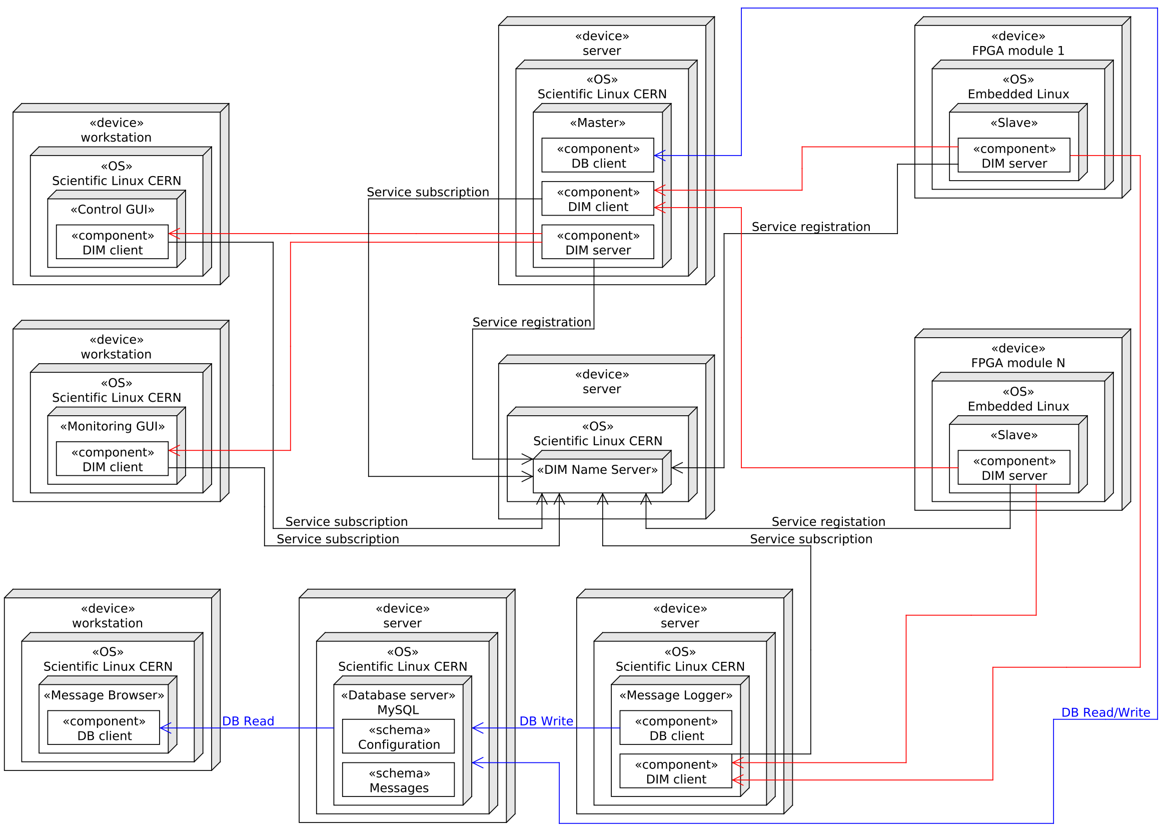


Figure 6.1: Software architecture of the new data acquisition system. Solid lines represent exchange of commands and data, dashed lines represent communication with DNS, dotted lines represent communication with database.

The *slave processes* should provide communication with the hardware. The slaves should be deployed on all kind of nodes including both types of FPGA cards (i.e. data multiplexors and event builders) and also on computers in the online farm. Slaves should work as DIM servers. On one hand, they should receive the commands and use these commands to configure and control the corresponding hardware, e.g. to start or stop operation of the FPGA cards. On the other hand, the slaves should also collect information about status and health of the underlying hardware and publish these information as the DIM services. These information should depend on type of the underlying hardware and should include load of processor, free amount of memory, or usage of network interfaces. As the slave should be deployed on various hardware architecture, it should be implemented in the C language for the portability reasons. However, tests of the DIM library under the embedded linux that should be running on softcore processors on the FPGA cards still needs to be performed and evaluated.

The *master process* should work as a mediator between the human operators and slave processes. At the same time, the master should act as a DIM client and a DIM server. As a DIM client, the master should collect information about the health of the hardware by subscribing to the corresponding DIM services of the slave processes. Additionally, the master should distribute configuration information and commands to the slave processes. On the other hand, as a DIM server, the master process should receive commands issued by the applications with user interface and forward them to the slave processes. It should also publish report about state of other nodes based on the information obtained from the slaves.

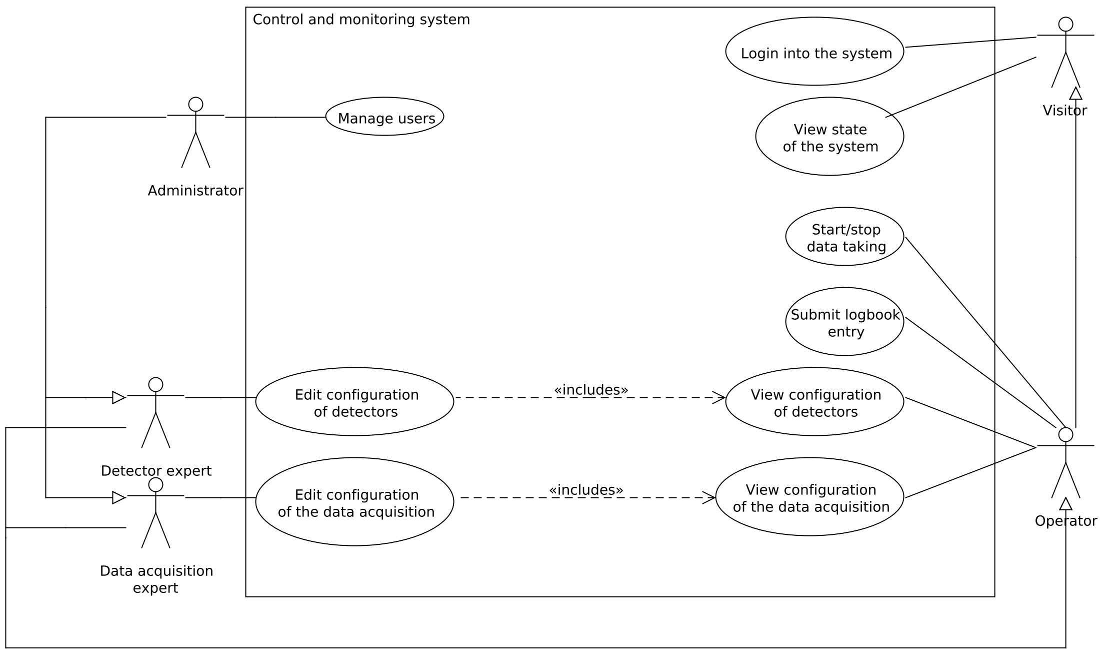


Figure 6.2: Use case diagram for the new data acquisition system

The master process should be deployed on the standard, x86–compatible server powered by the Scientific Linux CERN operating system, therefore, it is possible to use some higher level library or framework during implementation. The master should also access the online database. For compatibility with existing data acquisition system, we have decided to use the MySQL database software. The database should contain information about nodes, configuration of the system, users of the system, or software logs. The master process should use the DIM services to distribute data from database to the other processes participating in the system. Therefore, only the master process requires the database access and there is no need to use database client libraries on the embedded linux.

Additionally, the responsibilities of the master process should also include user management.

The master should receive commands issued from applications that provide user interface and should guarantee that at most one user can take the control over the system at the same time. However, multiple users should be able to monitor the behaviour of the system simultaneously. Multiple types of user accounts should be defined:

- The *visitor* account should be used for monitoring of the system as it can only view the status of the various hardware and software components of the data acquisition system and also of the detectors.
- The *operator* account should be used by members of the shift crew during the normal operation. This account can start or stop data taking, submit comments into the electronic logbook, and inspect configuration of the data acquisition system and of the detectors. Operators also inherit privileges of the visitor account, i.e. they can monitor the system.
- Accounts in the *data acquisition expert* group should be used to modify configuration of the data acquisition system.
- Accounts in the *detector expert* group should be used to modify configuration of the detectors.
- Finally, the *administrator* account should be used for modification of information about users and user privileges.

The information about users and user privileges should be also stored in the online configuration database. The master process should use information from this database to handle authentication and authorization of users. The operator account should be permitted to login to the system only from the workstations located in the control room. On the other hand, the visitor account should be allowed to login from any location to enable experts to monitor the system even from their home institutes.

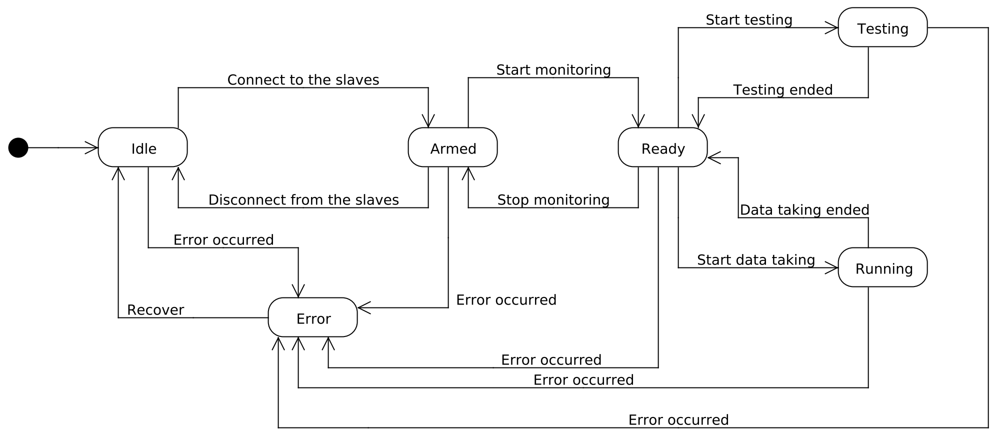


Figure 6.3: State machine diagram of the master process (preliminary version)

The control and monitoring software is a complex system. We have decided to simplify the development process by describing the behavior of the system with the *finite state machines*. The finite state machine that describes the state of the master process (and state of the whole data acquisition system in turn) should include at least the following states:

- The system should be started in the *Idle* state. This state should enable modifications of the system configurations such as adding or removing nodes. In this state, the connection to the slave processes is not established.

- In the *Armed* state, all the processes participating in the control and the monitoring software are started and the connection to the slave processes is being established.
- The system in the *Ready* state is prepared for starting of the operation, all processes are launched and communication with slave processes is established. The master processes periodically receives information about health of the system provided by the slave processes.
- In the *Testing* state, the trigger control system controller is working, the data are being taken but marked as *Test.* The errors in data consistency are tolerated in this state. It is possible to add or remove detector channels into the system in this state.
- In the *Running* state, the data are being taken and marked as *Good.* However, errors related to data consistency are not allowed in this state.
- The system enters the *Error* state in case some problem in hardware or software is detected. The recovery strategy still needs to be discussed, the software should contain some kind of automatic recovery feature.

The final design of the finite state machine should probably also contain some transitional states such as *Starting slaves* or *Stopping slaves.*

Also the behaviour of the slave processes should be defined by the state machine that consists of similar states:

- The slave process should be started in the *Idle* state in which it does not publish any information.
- In the *Ready* state, the slave processes regularly update information about status of the underlying hardware and waits for the commands sent by the master process.
- In the *Testing* state, the data are being taken and marked as *Test.* The slave processes watch quality of data and tolerate errors related to the data quality.
- In the *Running* state, the data are being taken and marked as *Good.* However, if a slave process detects problems with quality of data, it does not tolerate it.
- The slave process enters the *Error* state when a serious problem with hardware, software, or data quality is detected.

The state of the master process should depend on states of the slave process in the following way: the master process can enter the *Ready*, *Test*, or *Running* state only if all the slave processes enter the same state. On the other hand, if any slave process enters the *Error* state, then the master process (and the entire system in turn) and all other slave processes enter the *Error* state too.

Several software libraries or frameworks provide implementation of the finite state machines. The *State Machine Interface* (SMI++) is a framework developed for the DELPHI experiment at CERN, [13]. The framework consists of the *State Manager Language* that describes the real world object by the state machines and the *State manager* that controls the states of these state machines. Communication between the State manager and state machines is based on the DIM services. Additionally, the support for the state machines has been introduced in the version 4.6 of the Qt framework. The framework contains classes that correspond to state machines, states and final state, and state transitions, [39].

The master process and all slave processes also communicate with the *Message logger* process. This process collects software messages generated by other processes, buffers them, and stores them in the online database. The messages should inserted in the database table be with

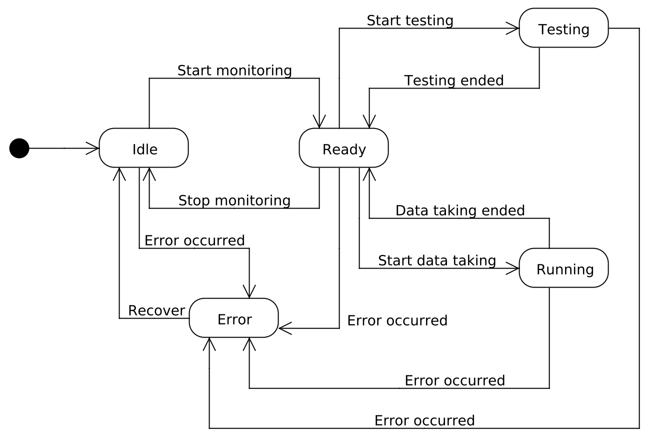
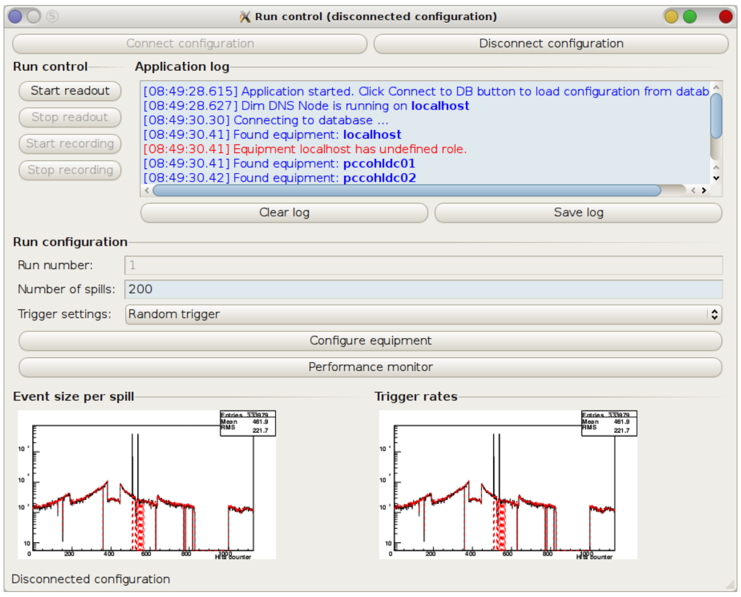


Figure 6.4: State machine diagram of the slave process (preliminary version)

information about severity, sender, run number, spill number, event number, and timestamp. The Message logger should be deployed on x86–compatible server powered by the SLC operating system. Obviously, this process requires the database access. A library that provides other nodes access to the Message logger should be provided, preferably in the C language for the portability reasons. The Message logger should replace the *infoLogger* facility provided by the DATE package. The Message logger should be compatible with the format of the messages table in the *DATE_ log* database defined by the DATE package.

The contents of the table with messages stored by the Message logger process should be viewed by the *Message browser* process. The process should provide a graphical interface that should allow users to interactively filter the messages by the timestamp, severity, or sender. The browser should also display message online, as soon as they are stored in the corresponding table. The application should implement the *Model–View–Controller* design pattern [9] with database table being the model, list view in the graphical user interface being the view, and set of filters being the controller. The Message browser should replace the functionality of the *InfoBrowser* application of the DATE package. The Message browser should not depend on the DIM library, therefore it could be deployed on any machine that has access to the online database. Multiple instances of the Message browser should work simultaneously in the system.
According to the proposal, the *user interface* of the control and monitoring software should be implemented as a stand alone application. The user interface should act as a DIM client. It should send DIM commands to the master process which would process these commands and delegate them to the hardware via the slave processes. On the other hand, the user interface should subscribe to the services published by the master that provide summary information about health of the hardware. Multiple user interfaces should be allowed to monitor the system simultaneously, however the master process should guarantee that at most one user interface controls the system at the same time. The application providing the user interface should be used routinely on the workstations in the control room powered by the SLC operating system during shifts. However, detector and data acquisition experts should be allowed to use the application from their laptops, at least for monitoring purposes. Therefore, the application should support also MS Windows platform. The user interface should be modular, i.e. users should be able to customize the layout to their needs by showing or hiding various components such as a trigger rate display, run control panel, or configuration panel. The application should



```
[08:49:28.615] Application started. Click Connect to DB button to load configuration from datab
[08:49:28.627] Dim DNS Node is running on localhost
[08:49:30.30] Connecting to database ...
[08:49:30.41] Found equipment: localhost
[08:49:30.41] Equipment localhost has undefined role.
[08:49:30.41] Found equipment: pccohldc01
[08:49:30.42] Found equipment: pccohldc02
```

Run number: 1
Number of spills: 200
Trigger settings: Random trigger

Figure 6.5: Proposal of the graphical user interface

also reflect privilege of the currently logged user, e.g. the visitors should have access only to the monitoring components, while the run control panel should be displayed to operators and configuration panels to the corresponding data acquisition and detector experts.

We have decided to develop the graphical user interface in the Qt framework. The framework is known mainly as a rich library of graphical widget ranging from simple buttons or check boxes to complex tables or graphical canvases. Additionally, the framework also contains classes for network programming, memory management, database access, multithreading, or finite state machines. Furthermore, the framework extends the object model of the C++ language with introspection and signal and slot mechanism. The graphical editor of user interfaces Qt designer and complete *Integrated Development Environment* QtCreator are also included in the framework. The framework supports all the major platforms including GNU/Linux with X window system, MS Windows, and Mac OS. The framework should be used to develop the user interface, Message logger, Message browser, and master process. Only the slave process that should be deployed on the embedded linux powered by the softcore processor cannot be implemented in the Qt framework as it does not support this platform. Additionally, the transport protocol used for communication between nodes should not depend on the Qt framework.

3.1 Transport protocol

The communication between nodes that participate in the control and monitoring software is based on exchange of messages between DIM publishers and DIM subscribers. In cooperation with data acquisition experts, we have defined a custom *communication protocol* that is described by the format of message. Each message consists of a *header*, a *body*, and a *trailer*. The message trailer should contain total size of the message, version of the communication protocol, unique identifiers of sender and receiver, message number, and timestamp. The size of the header should be 32 Bytes. The payload of message should contain useful data such as information about health of hardware. The size of payload must be a multiple of 4 Bytes, it can also be empty. The trailer should consists of message number, check sum of the message and some reserved bytes. The size of the trailer should be 16 Bytes. Format of the message is completely described in Table 6.3.

Format of the message			
Header			
1.	Data size	4 bytes	Total size of the message in 32b words = header size+payload size+trailer size
2.	Version	4 bytes	Version of the protocol
3.	Sender ID	4 bytes	Unique ID of the message's sender
4.	Message number	4 bytes	Number of the message
5.	Receiver ID	4 bytes	Unique ID of the message's receiver
6.	Message ID	4 bytes	ID of the message
7.-8.	Time	8 bytes	Time stamp
Payload			
9.	Body	(0-N) × 4 bytes	Body of the message (can be empty)
Trailer			
10.	Reserved	4 bytes	0x00000000
11.	Reserved	4 bytes	0x00000000
12.	Message number	4 bytes	Number of the message (the same as in the header)
13.	Check sum	4 bytes	Check sum of the message

Table 6.3: Message format

As this protocol should be used by all applications participating in the system, it should be implemented as a function library that should be linked to these applications. As this library should also be used by slave processes running on embedded linux, it should be developed in the C language for portability reasons. The library should contain functions that allow construction and parsing of the messages. Additionally, the library should support a data type that describes character array with known size and that should be used for manipulation with messages. The functions that allows appending another character array or character to existing character array, return prefix, center, or suffix of the character array, clears the array, or returns the size of the array should be also implemented in the library. The data type should provide the similar functionality as the `QByteArray` class of the Qt framework [39] that cannot be used because of the portability reasons.

4 Results and outlook

The proposal of the control and monitoring system introduced in the previous section has been finalized during summer 2011. Especially, the definition of the finite state machines that describe behavior of the master and the slave processes has been completed. During autumn 2011, the first minimal version of the software that fulfills the requirements given by the proposal has been implemented. Implementation details can be found in works [4, 19]. The current status of the software development has been presented to the Compass member on the collaboration meeting in November 2011, [6].

The first series of performance and stability tests of the software have been carried out during winter shutdown of the experiments on workstations in the new control room and on event building machines: message logger, application with the graphical user interface, and the database server have been deployed in the control room, DNS, master process, and slave processes have been deployed on event builders. All machines included in the tests have been connected to the COMPASS internal network by Gigabit Ethernet that provides theoretical bandwidth of up to 128 MB/s.

The purpose of these tests has been to evaluate stability and performance of the software. During the tests, performance have been compared for different number of slaves and different sizes of the message payload (see Table 6.3). We have proved, that for messages larger than 1 kB, the system is able to almost fully saturate the network. The theoretical bandwidth cannot be reached because of the overhead of the TCP/IP that includes headers of packets and overhead caused by components of the network. For this size, the system is capable of exchanging approximately 90 000 messages each second. This is promising result as it is required that the system updates information about status of the hardware each 10 ms. For smaller messages, the performance drop seems to be caused mainly by communication with the DNS.

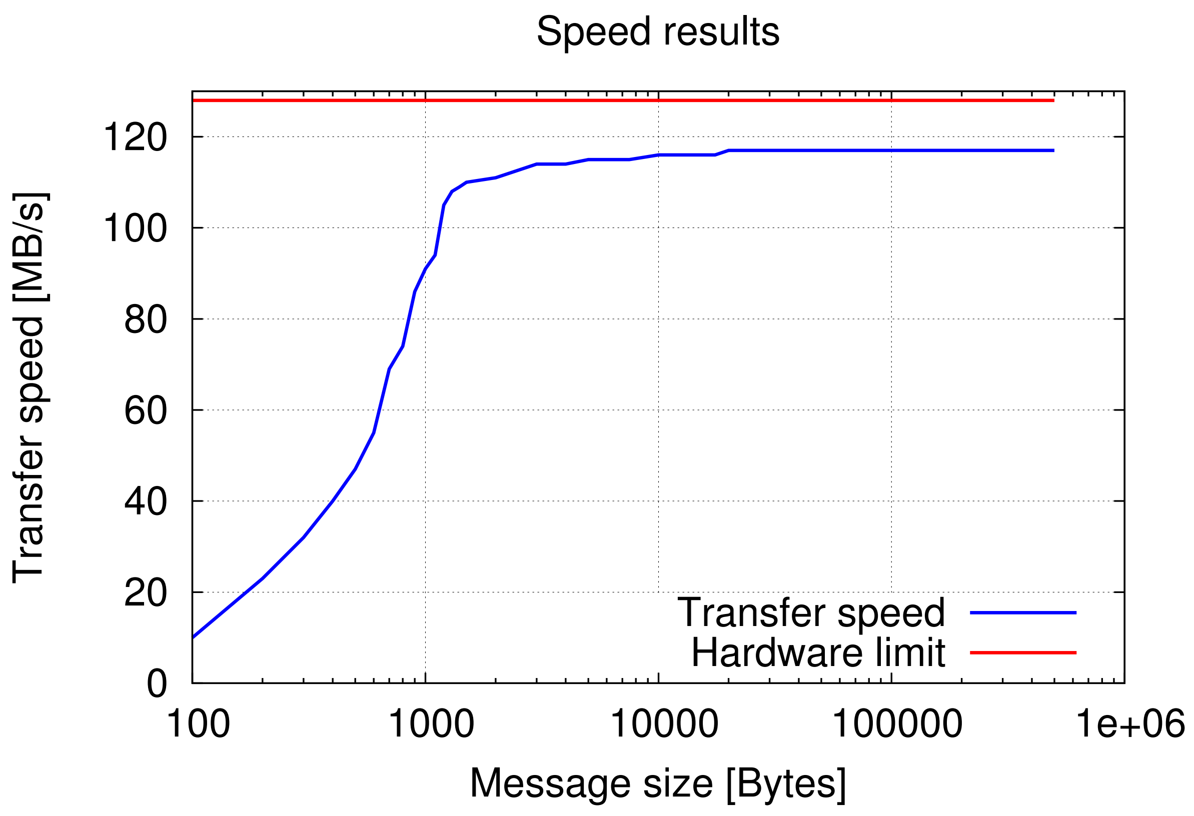


Figure 6.6: Test results: Transfer speed

In the second series of tests, we have studied the stability of software in time. In this test scenario, the master has been exchanging messages with up to 10 slaves over period of 20 hours. The results show that the transfer speed remained constant, additionally, no memory leaks were detected during the test. The observed spikes in the graph still need to be investigated, we suppose that they are caused by synchronization of the system time with time server over the network time protocol NTP.

Currently, the functionality of the slave process is being extended to support monitoring of the health and status of HGeSiCA modules. HGeSiCA is a VME card that gathers data from four frontend cards into a subevent. It receives information about event identification from the trigger control system and adds it to the subevent. The task of the slave process is

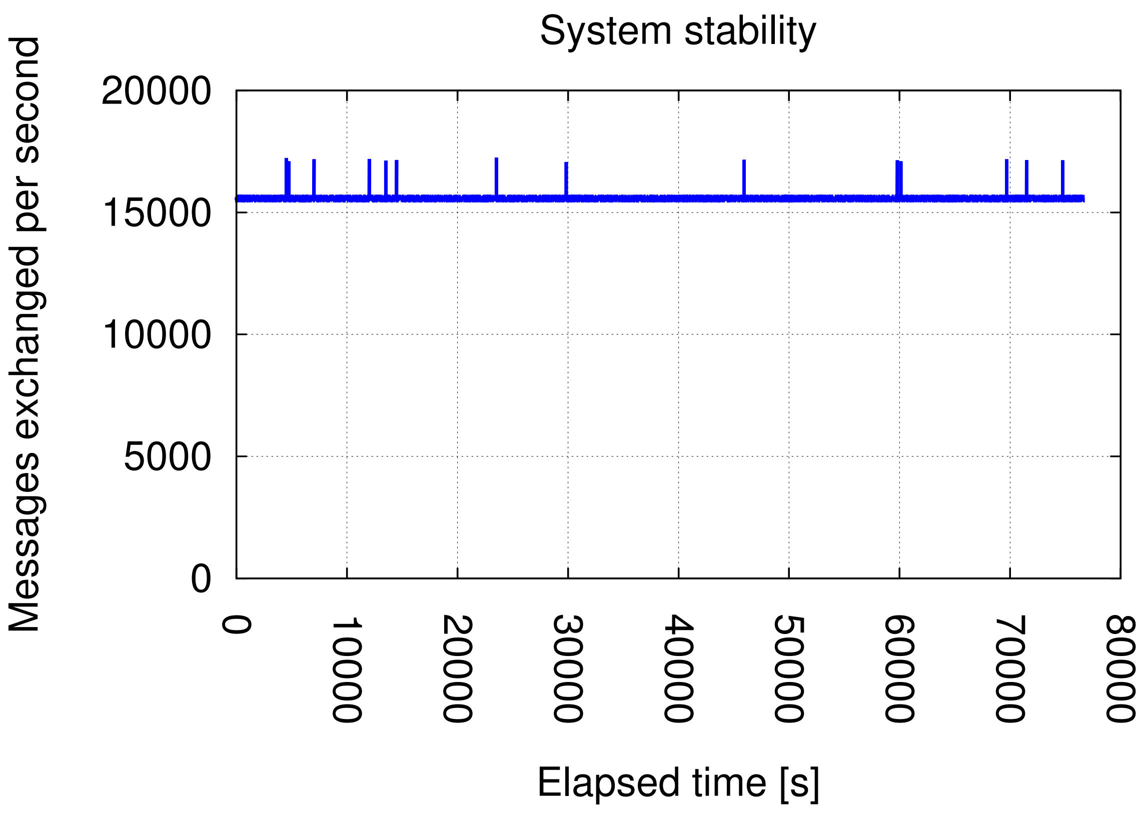


Figure 6.7: Test results: stability of the system

to continuously read contents of the status registers of the HGeSiCA module. These registers include information about last processed event, amount of processed data, configuration of the module (i.e. active ports), and also error mask.

The final version of the firmware for the FPGA cards is under development in spring 2012. When the firmware is prepared, the slave processes will need to be ported on the embedded linux powered by the softcore processor. The slave processes will need to access data about status of the cards and will need to send commands and configuration information to these cards. The first tests with the FPGA cards are scheduled for autumn 2012. The goal is to have the system fully functional in the year 2013. The shutdown of all particle accelerator at CERN is expected for the year 2013, therefore the year should be used for testing of the hardware and the software parts of the system. If the tests are successful, the system will be deployed at the COMPASS experiment starting from the year 2014.

Conclussion and contribution of the thesis

The aim of this thesis was to analyze selected parts of the software support of the Compass experiment at CERN and to propose new architecture of these parts. We have decided to focus on the data acquisition system of the experiment. In Chapter 2, the existing data acquisition system has been analyzed. The existing system is based on custom electronics that perform detector readout and network of servers that perform event building. From the software point of view, the data acquisition tasks are powered by the Date package.

The Chapter 4 focused on the MySQL database that is used by the Date package to store configuration information and logs. Also, the information about conditions of the spectrometer and the beam line during the data taking process are stored in the database. During the year 2009, the database service experienced several crashes related to the increased load. We have analyzed the database architecture and concluded that the performance problems had been caused by combination of outdated hardware and software. We have evaluated performance of newer version of the MySQL software and compared it to older version. We have proposed and implemented new database architecture. The new architecture is powered by two physical servers that are synchronized by the master–master replication. Third server is used as a proxy server. The system is continuously monitored by the Nagios system, database is regularly backed up. The implemented database architecture is able to handle increased load. However, in order to increase availability and simplify a crash recovery, we recommend to increase the redundancy by adding more physical servers. This could also be used to implement load balancing.

The radioprotection limits might be exceeded in the control room in the planned scientific programs as a result of increased beam intensity. Therefore, we have been asked to implement the remote control room for the Compass experiment. In Chapter 5, we have discussed several possible implementations of the remote access. Then, we have described the kickstart method that has been used to install the system on the workstations in the new control room. We have successfully configured the Date to work from the remote control room, consequently we have saved considerable amount of money that would otherwise have to be invested into additional shielding of the spectrometer. The remote control room is now prepared for deployment. Although the method described in Chapter 5 can be used to configure any computer connected to the Compass network for remote access, we recommend to use only remote monitoring. The remote control should be performed only from the control room for the safety reasons.

Finally, Chapter 6 is dedicated to development of a control and monitoring software for the new data acquisition architecture. The hardware part of the architecture is being developed at the Technical University in Munich; it is based on the custom FPGA cards that should replace event building network that is based on deprecated hardware. At first, we have analyzed the requirements posed on the software. Based on these requirements, we have prepared and presented proposal of the new control and monitoring software. The software is to be deployed on heterogeneous, distributed nodes. We have decided to build the software on the DIM communication library. The proposal defines roles that should be implemented in the architecture, behavior of the actors is described by the state machines. The implementation of the proposal is described in master's degree thesis [4, 19]. The software architecture has been tested during winter shutdown of the experiment, we have confirmed that the software meets the expected performance requirements and runs stably. In the following steps, the software needs to be ported on the FPGA hardware. It is expected to have fully functional system prepared for the testing under the real conditions in 2013. If these test are successful, the new architecture will be deployed for data taking starting from the year 2014.

As part of the thesis, the author has acted as a supervisor specialist of undergraduate students who implemented the proposed control and monitoring software for the new data acquisition architecture of the Compass experiment. The students have successfully defended their master's degree thesis in June 2012, [4, 19].

List of Figures

List of Tables

List of Publications

Papers

1. C. Adolph, V. Jarý, et al. (the COMPASS collaboration): *COMPASS-II proposal*, CERN-SPSC-2010-014; SPSC-P-340 (May 2010)
2. C. Adolph, V. Jarý, et al. (the COMPASS collaboration): *Exclusive ρ^0 muoproduction on transversely polarised protons and deuterons*, accepted in Nuclear Physics B, arXiv:1207.4301
3. C. Adolph, V. Jarý, et al. (the COMPASS collaboration): *Experimental investigation of transverse spin asymmetries in muon-p SIDIS processes: Collins asymmetries*, submitted to Physics Letters B, arXiv:1205.5121.
4. C. Adolph, V. Jarý, et al. (the COMPASS collaboration): *Experimental investigation of transverse spin asymmetries in muon-p SIDIS processes: Sivers asymmetries*, submitted to Physics Letters B, arXiv:1205.5122.
5. C. Adolph, V. Jarý, et al. (the COMPASS collaboration): *Measurement of the Cross Section for High-p_T Hadron Production in Scattering of 160 GeV/c Muons off Nucleons*, submitted to Physics Review Letters, arXiv:1207.2022.
6. M. Bodlak, V. Jary, I. Konorov, A. Mann, J. Novy, S. Paul, M. Virius: *Developing control and monitoring software for data acquisition system*, submitted to the Acta Polytechnica.
7. M. Bodlák, V. Jarý, I. Konorov, A. Mann, J. Nový, S. Paul, M. Virius: *Software Development for the COMPASS Experiment*, In: 38th Software Development, Ostrava: VŠB – Technická univerzita Ostrava, 2012; ISBN 978-80-248-2669-1. pp. 10–17.
8. M. Bodlák, V. Jarý, T. Liška, F. Marek, J. Nový, M. Plajner: *Remote Control Room For COMPASS Experiment*, In: 37th Software Development, Ostrava: VŠB – Technická univerzita Ostrava, 2011, ISBN 978-80-248-2425-3. pp. 1–9.
9. L. Fleková, V. Jarý, T. Liška: *Mass Data Processing Optimization on High Energy Physics Experiments*, In: 4th International Conference on Advanced Computer Theory and Engineering, Dubai, 2010, ISBN 978-07-918-5993-3.
10. L. Fleková, V. Jarý, T. Liška, M. Virius: *Proposal and results of COMPASS database upgrade*, In: Stochastic and Physical Monitoring Systems, Děčín, 2010, ISBN 978-80-01-04641-8. pp. 45–50.
11. L. Fleková, V. Jarý, T. Liška, M. Virius: *Využití databází v rámci fyzikálního experimentu COMPASS*, In: 36th Softwaru Development, Ostrava: VŠB – Technická univerzita Ostrava, 2010, ISBN 978-80-248-2225-9. pp. 68–75.
12. V. Jarý: *COMPASS Database Upgrade*, In: Doktorandské dny 2010, Praha: ČVUT, 2010, ISBN 978-80-01-04664-9. pp. 95–104.
13. V. Jarý: *Detector simulation with Geant4*, In: Doktorandské dny 2009, Praha: ČVUT, 2009, ISBN 978-80-01-04436-0. pp. 71–80.

14. V. Jarý: *Towards a New Data Acquisition Software for the COMPASS Experiment*, In: Doktorandské dny 2011, Praha: ČVUT, 2011, ISBN 978-80-01-04907-5. pp. 95–104.
15. V. Jarý, T. Liška, M. Virius: *Developing a New DAQ Software For the COMPASS Experiment*, In: 37th Software Development, Ostrava: VŠB – Technická univerzita Ostrava, 2011, ISBN 978-80-248-2425-3. pp. 35–41.

Conference talks

1. M. Bodlák, V. Jarý, J. Nový: *Nový řídicí a dohledový systém pro experiment COMPASS*, In: Installfest 2012, Prague 2012.
2. M. Bodlák, V. Jarý, J. Nový: *Software for the new COMPASS data acquisition system*, In: COMPASS collaboration meeting, Geneva 2011.
3. G. Deferne, V. Jarý, et al. (the OSQAR collaboration): *The OSQAR Experiments at CERN to probe the QED & Astroparticle Physics*, In: Advanced Studies Institute, Symmetries and Spin, Prague 2010
4. L. Fleková, V. Jarý, T. Liška: *New COMPASS database architecture*, In: COMPASS Frontend Electronics meeting, December 2010, Geneva.
5. L. Fleková, V. Jarý, T. Liška: *Proposal on the COMPASS database upgrade*, In: COMPASS Frontend Electronics meeting, March 2010, Geneva.
6. L. Fleková, V. Jarý, T. Liška, M. Virius: *Report on COMPASS database upgrade*, In: Advanced Studies Institute, Symmetries and Spin, Prague 2010.
7. V. Jarý: *Databáze ve fyzice vysokých energií*, In: Installfest 2011, Prague 2011.
8. V. Jarý: *DATE evaluation*, In: COMPASS Frontend Electronics meeting, April 2011, Geneva.
9. V. Jarý: *Development of New DAQ System For the COMPASS and PANDA experiments*, In: Workshop Devoted to 100th Anniversary of the discovery of low temperature superconductivity, Pec pod Sněžkou 2011.
10. V. Jarý: *Highly available and reliable database for the COMPASS experiment*, In: Advanced Studies Institute, Symmetries and Spin, Prague 2012.
11. V. Jarý: *New DAQ System For the COMPASS Experiment*, In: Advanced Studies Institute, Symmetries and Spin, Prague 2011.
12. V. Jarý: *Software for the new COMPASS DAQ*, In: COMPASS Frontend Electronics meeting, November 2011, Geneva.

Bibliography

[1] P. Abbon et al. (the COMPASS collaboration): *The COMPASS experiment at CERN.* In: Nucl. Instrum. Methods Phys. Res., A 577, 3 (2007) pp. 455–518

[2] Ch. Adolph, . . . , V. Jarý et al. (the COMPASS collaboration): *COMPASS-II proposal.* CERN-SPSC-2010-014; SPSC-P-340 (May 2010)

[3] T. Anticic et al. (the ALICE collaboration): *ALICE DAQ and ECS User's Guide.* CERN, ALICE internal note, ALICE-INT-2005-015, 2005.

[4] M. Bodlák: *COMPASS DAQ – Database architecture and support utilities.* Prague, Czech Technical University in Prague, June 2012

[5] M. Bodlák, V. Jarý, T. Liška, F. Marek, J. Nový, M. Plajner: *Remote Control Room For COMPASS Experiment.* In: 37th Software Development, Ostrava: VŠB – Technická univerzita Ostrava, 2011, ISBN 978-80-248-2425-3. pp. 1–9.

[6] M. Bodlák, V. Jarý, J. Nový: *Software for the new* COMPASS *data acquisition system.* In: COMPASS collaboration meeting, Geneva, Switzerland, 18 November 2011
Also available at: `http://wwwcompass.cern.ch/compass/collaboration/2011/co_1111/`

[7] R. Brun, F. Rademakers: *ROOT - An Object Oriented Data Analysis Framework.* In: Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) pp. 81–86.

[8] P. Charpentier, M. Dönszelmann, C. Gaspar: *DIM, a Portable, Light Weight Package for Information Publishing, Data Transfer and Inter-process Communication.* In International Conference on Computing in High Energy and Nuclear Physics 2000, Padova, Italy 1-11 February 2000

[9] E. Gamma: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series, 1995, ISBN 0201633612

[10] J. Gehrke, R. Ramakrishnan: *Database Management Systems, Third Edition.* McGraw-Hill, August 2002, ISBN 978-00-724-6563-1

[11] L. Fleková, V. Jarý, T. Liška: *Proposal on the COMPASS database upgrade.* In: COMPASS Frontend Electronics Meeting, Geneva, Switzerland, 26 March 2010

[12] L. Fleková, V. Jarý, T. Liška, M. Virius: *Využití databází v rámci fyzikálního experimentu COMPASS.* In: Konference Tvorba softwaru 2010, Ostrava: VŠB - Technická univerzita Ostrava, 2010, ISBN 978-80-248-2225-9 pp. 68–75.

[13] B. Franek, C. Gaspar. *SMI++ Object Oriented Framework for Designing and Implementing Distributed Control Systems.* In 10th IEEE Real-Time Conference 1997, Baunne, France, 22-26 September 1997

[14] V. Jarý. *DATE evaluation.* In: COMPASS DAQ meeting, Geneva, Switzerland, 29 March 2011

[15] V. Jarý: *Live distributions.* Prague: Czech Technical University in Prague, June 2008. Master's Degree project

[16] V. Jarý, T. Liška, M. Virius. *Developing a New DAQ Software For the COMPASS Experiment.* In: 37th Software Development, Ostrava: VŠB – Technická univerzita Ostrava, 2011, ISBN 978-80-248-2425-3. pp. 35–41.

[17] A. Král, T. Liška, M. Virius: *Experiment COMPASS a počítače.* In Československý časopis pro fyziku 5. Prague, Czech Republic, 05/2005. pp. 472.

[18] A. Mann, F. Goslich, I. Konorov, S. Paul. *An AdvancedTCA Based Data Concentrator and Event Building Architecture.* In 17th IEEE-NPSS Real-Time Conference 2010, Lisboa, Portugal, 24–28 May 2010

[19] J. Nový: *COMPASS DAQ – Basic Control System.* Prague, Czech Technical University in Prague, June 2012

[20] I. Konorov: *private communication.* September 2010.

[21] M. Matsumoto, T. Nishimura: *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator.* 1998. ACM Transactions on Modeling and Computer Simulation 8 (1): 3–30

[22] T. Nagel: *Cinderella: an Online Filter for the COMPASS Experiment.* München: Technische universität München, January 2009.

[23] L. Schmitt et al.: *The DAQ of the COMPASS experiment.* In: 13th IEEE-NPSS Real Time Conference 2003, Montreal, Canada, 18–23 May 2003, pp. 439–444

Online sources

[24] M. Achour et al.: *PHP Manual* [online]. June 2012.
Available at: `http://php.net/manual/en/index.php`

[25] G. Chazarain: *iotop homepage* [online]. June 2012.
Available at: `http://guichaz.free.fr/iotop/`

[26] E. van der Bij, Stefan Haas: *CERN S-LINK homepage* [online]. June 2012.
Available at: `http://hsi.web.cern.ch/hsi/s-link/`

[27] M. Keep: *MySQL 5.6 Replication - Enabling the Next Generation of Web & Cloud Services*
Available at:
`http://dev.mysql.com/tech-resources/articles/mysql-5.6-replication.html`

[28] A. Lentz: *MySQL Storage Engine Architecture* [online]. 2010. Available at:
`http://dev.mysql.com/tech-resources/articles/storage-engine/part_1.html`

[29] R. Rivest: *The MD5 Message–Digest Algorithm* [Internet RFC 1321]. April 1992. Available at: `http://tools.ietf.org/html/rfc1321`

[30] *COMPASS web page* [online]. June 2012. Available at: `http://wwwcompass.cern.ch`

[31] *Distributed Information Management System: User Manual* [online]. June 2012. Available at: `http://dim.web.cern.ch/dim/dim_user.html`

[32] *Extra Packages for Enterprise Linux* [online]. June 2012. Available at: `http://fedoraproject.org/wiki/EPEL`

[33] *ISO/IEC 9075-1:2011 Database languages – SQL* [online]. June 2012. Available at: `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=53681`

[34] *Linuxsoft: Software repository and installation service* [online]. December 2011. Available at: `http://linuxsoft.cern.ch/`

[35] *MySQL Master–Master Replication* [online]. 2010. Available at: `http://www.howtoforge.com/mysql_master_master_replication`

[36] *MySQL Load Balancer Guide* [online]. 2010. Available at: `http://downloads.mysql.com/docs/mysql-load-balancer-en.a4.pdf`

[37] *MySQL 5.1 Reference Manual* [online]. June 2012. Available at: `http://dev.mysql.com/doc/refman/5.1/en/`

[38] *Nagios Exchange* [online]. June 2012. Available at: `http://exchange.nagios.org/`

[39] Trolltech Inc.: *Qt reference documentation* [online]. May 2012. Available at: `http://docs.trolltech.com`

[40] *Redhat Hat Enterprise Linux 5 Installation Guide* Available at: `http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/`

[41] *Using the GNU Compiler Collection* [online]. 2012. Available at: `http://gcc.gnu.org/onlinedocs/`