České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská
Katedra fyzikální elektroniky

BAKALÁŘSKÁ PRÁCE

Jan Tomsa

Praha – 2014

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská
Katedra fyzikální elektroniky

Monitorovací nástroje pro sběr dat fyzikálního experimentu COMPASS v CERN
Monitoring tools for the data acquisition system of the COMPASS experiment at CERN

Bakalářská práce

Autor práce:	**Jan Tomsa**
Vedoucí práce:	**Ing. Vladimír Jarý, Ph.D.**
Konzultant:	**Ing. Josef Nový**
Akademický rok:	**2013/2014**

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA JADERNÁ A FYZIKÁLNĚ INŽENÝRSKÁ

Katedra fyzikální elektroniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	**Jan Tomsa**
Obor:	**Inženýrská informatika**
Zaměření:	**Informatická fyzika**
Školní rok:	**2013/2014**
Název práce:	**Monitorovací nástroje pro sběr dat fyzikálního experimentu COMPASS v CERN** **Monitoring tools for the data acquisition system of the COMPASS experiment at CERN**
Vedoucí práce:	**Ing. Vladimír Jarý, Ph.D.**
Konzultant:	**Ing. Josef Nový**

Pokyny pro vypracování:

1. Seznamte se se stávajícím a nově vyvíjeným systémem pro sběr dat experimentu COMPASS v CERN.
2. Seznamte se s aktuálním stavem monitorovacích nástrojů pro právě vyvíjený systém
3. Seznamte se s relačními databázemi a frameworkem Qt
4. Definujte vhodné datové struktury pro reprezentaci stavů systému
5. S využitím navržených struktur upravte monitorovací nástroje

Literatura:

1. T. Anticic et al. (the ALICE collaboration): *ALICE DAQ and ECS User's Guide*. CERN, ALICE internal note, ALICE-INT-2005-015, 2005
2. J. Blanchette, M. Summerfield: *C++ GUI Programming with Qt 4*. Prentice Hall PTR, Upper Saddle River, NJ, USA. 2006, ISBN:0131872494
3. M. Bodlák: *COMPASS DAQ – Database architecture and support utilities*. Prague, Czech Technical University in Prague, June 2012
4. J. Gehrke, R. Ramakrishnan: *Database Management Systems*, Third Edition. McGraw-Hill, August 2002, ISBN 978-00-724-6563-1

Datum zadání:	24. říjen 2013
Datum odevzdání:	7. červenec 2014

[illegible] ………………………………

Vedoucí katedry

[illegible] ………………………………

Děkan

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská
-1-

V Praze 24.10.2013

Acknowledgment

I would like thank to my supervisor, Ing. Vladimír Jarý, Ph.D., for leading my bachelor thesis.

Poděkování

Chtěl bych poděkovat Ing. Vladimíru Jarému, Ph.D. za vedení mé bakalářské práce, její jazykovou korekturu a věcné připomínky.

Declaration

I declare that I have carried out this bachelor thesis myself and I have mentioned all used information sources in bibliography.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

V Praze dne 7. 7. 2014 Jan Tomsa

Název práce:

Monitorovací nástroje pro sběr dat fyzikálního experimentu COMPASS v CERN

Autor:	Jan Tomsa
Obor:	Inženýrská informatika
Druh práce:	Bakalářská práce
Vedoucí práce:	Ing. VLadimír Jarý, Ph.D.
Konzultant:	Ing. Josef Nový

Abstrakt

Tato práce se zabývá monitorovacími nástroji pro sběr dat na experimentu COMPASS v CERN. Je to velmi důležitá součást celého softwarového balíku sběru dat. Zajišťuje stálý přehled o průběhu sběru dat. Na začátku proběhne seznámení s CERNem a experimentem COMPASS. Je popsán starý i nový systém sběru dat jak z hardwarového, tak softwarového hlediska. Následuje seznámení s použitými softwarovými technologiemi jako je programovací jazyk C++, framework Qt, komunikační knihovna DIM a MySQL databáze. Práce se potom podrobněji zabývá návrhem a implementací programu Message Browser pomocí frameworku Qt a dalších technologií. Je zmíněn i program Message Logger.

Klíčová slova: CERN, DAQ, COMPASS, Qt, DIM, C++, MySQL

Title:

Monitoring tools for the data acquisition system of the COMPASS experiment at CERN

Author: Jan Tomsa

Abstract

This thesis covers the monitoring tools of the COMPASS experiment at CERN. It is a very important part of the whole software package for the data acquisition. It provides a constant overview over the progress of the data acquisition. The first part introduces CERN and the COMPASS experiment at CERN. Then the thesis describes both the old and the new Data Acquisition System of the COMPASS experiment from both hardware and software point of view. The used software technologies are introduced, especially the Qt framework and the DIM communication library. The following and the most important part is dedicated to the design and implementation of the Message Browser program. The Message Logger is briefly mentioned.

Key words: CERN, DAQ, COMPASS, Qt, DIM, C++, MySQL

Contents

Introduction

During the CERN shutdown during the years 2013 and 2014, both hardware and software part of the Data Acquisition System of the COMPASS experiment is being upgraded. The monitoring tools are not less important than any other part of the project. They are a necessary part of the system, because they ensure an overview over the progress of experiment. If something is not right with the DAQ process, monitoring tools provide immediate feedback. The goal of this thesis is to document, improve and finish the monitoring tools of the COMPASS experiment.

The first chapter is an introduction to CERN, COMPASS and it's Data Acquisition system. Both hardware and software part are mentioned.

The following chapter is dedicated to the description of the software tools used in this project. It serves as an overview of the Qt framework, the DIM communication library, and as a slight introduction to the MySQL database.

The third chapter describes the monitoring tools. It explains their capabilities and how to use them.

The main contribution of this thesis is covered in the last, and the longest chapter that is devoted to a detailed description of the implementation of the monitoring tools.

Chapter 1

CERN & COMPASS

1.1 CERN

CERN, the European Organization for Nuclear Research, is an international physics laboratory, operating one of the largest and most complex instruments to study the basics of matter - fundamental particles. It is located north-west of Geneva (Switzerland) in the Swiss-French borderland, with the main site in Meyrin. CERN was established in 1954 by 12 European countries. Currently, there are 20 countries with membership status and 7 observers.
CERN provides infrastructure and set of 6 accelerators, a decelerator and many detectors for high-energy particle physics experiments. The largest accelerator (LHC - Large Hadron Collider) is designed to produce particle collisions with energy up to 14 TeV. Particles gain energy in the cascade of accelerators and collide at speed close to the speed of light either with fixed targets or with each other. Surrounding detectors record the results of these collisions.

1.2 COMPASS

COMPASS (COmmon Muon Proton Apparatus for Structure and Spectroscopy) is a high-energy particle-physics fixed target experiment situated in the north area of the Super Proton Synchrotron (SPS) particle accelerator at CERN. It's main mission is to study the structure and the spectroscopy of hadrons using high intensity muon and hadron beams. The unique CERN SPS M2 beam line is used as a source of these particles, serving the beam with energy within the range of 50 GeV and 280 GeV.
The experiment was conditionally approved in 1997 by CERN Research

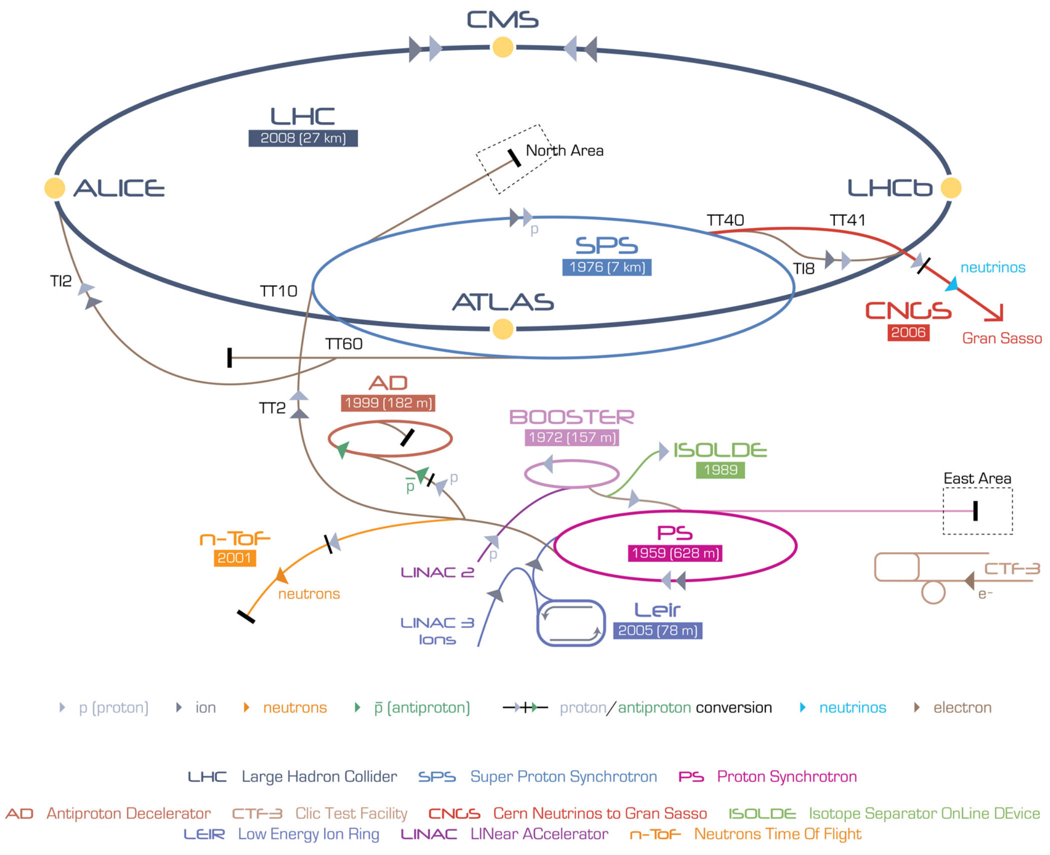


Figure 1.1: Experiment COMPASS is located in the North Area [9]

Board and assembled during the following years. The first period of data taking took place between 2002 and 2004, followed by one year shutdown. Then COMPASS resumed data acquisition with a muon beam (2006-2007), hadron spectroscopy with pion and proton beam (2008-2009) and spin structure function measurements with a polarised proton target (2010-2011).
In 2010, the second phase of the experiment (COMPASS-II) has been approved by the CERN Research Board. The scientific program focuses mainly on the Primakoff scattering, the Drell-Yan effect, and the generalized parton distributions. [8]
COMPASS consists of two main parts - Small Angle Spectrometer (SAS) and Large Angle Spectrometer (LAS). Both parts contain detectors for tracking, particle identification, and calorimetry. Total length exceeds more than 60 meters, therefore the flight time of particles cannot be neglected and the precise time synchronization is ensured by TCS (Trigger Control System).
The particle current from the beamline is not continuous. SPS sends 10

seconds long spills every 30-40 seconds (depending on the SPS cycle). One spill for the hadron beam carries approximately 10^8 particles, $2 \cdot 10^8$ for the muon beam. The record of the flight and interactions of the particle in spectrometer is called event. The average size of every event is roughly 40 kB. The total amount of data recorded during one spill can reach up to 18 GB.

1.3 COMPASS DAQ

1.3.1 Current (Old) COMPASS DAQ

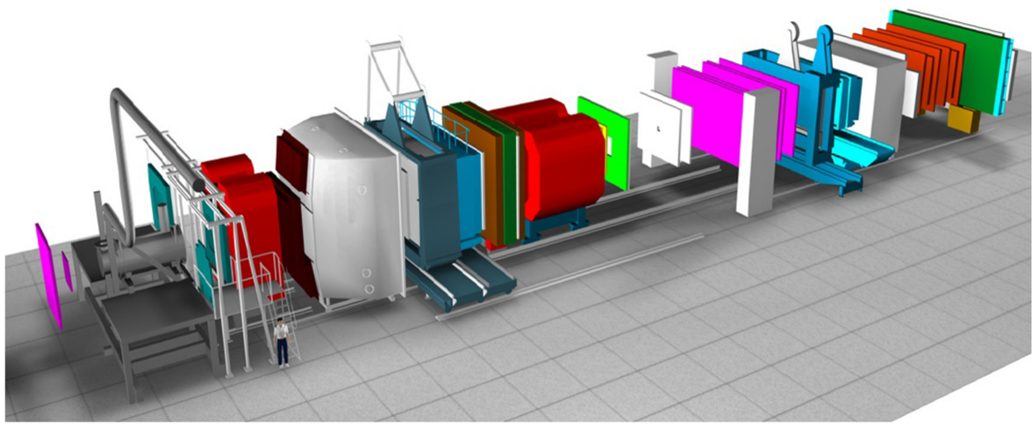


Figure 1.2: An artistic view of the experiment COMPASS [8]

Hardware

The COMPASS DAQ (Data Acquisition) is composed of multiple layers. The first layer, the frontend electronics, is connected directly to the detectors and reads out analogue data and converts it to digital values. COMPASS contains around 250, 000 of these detector channels. In the second layer, 130 CATCH and GeSiCA modules create subevents. The assembly is coordinated by the Trigger Control System (TCS). When the trigger signal arrives, data from multiple channels are readout and concentrated into subevents enriched with a timestamp. The subevents are then carried to 32 readout buffer (ROB) servers via optical S-Link interface. Each of them contains 4 PCI cards equipped with 512 MB of memory, called spillbuffers. Their purpose is to buffer the subevents and to distribute the load during the whole SPS cycle. Subevents are then moved through the Gigabit Network to event building computers that filter and assemble the final events. Events, after being locally stored on hard disks, are transfered to the CERN permanent storage CASTOR. [1] [6] [2]

Software

The old DAQ was using modified DATE software package that was originally developed for the LHC experiment ALICE. DATE (Data Acquisition and Test Environment) data acquisition software has been designed with emphasis on easy scalability. Thus it can run on one or many processors/computers. It requires Intel compatible hardware and Linux operating system. It offers tools to actual readout and event building, run control, messages and information logging, and event sampling.
InfoLogger and InfoBrowser are tools used for monitoring purposes. The InfoLogger retains incoming messages from the system and stores them into a MySQL database. The InfoBrowser is a GUI (Graphical User Interface) program for browsing and displaying those messages gathered in the database.

1.3.2 The New DAQ

Hardware

Due to increasing amount of collected data, the old DAQ hardware slowly becomes insufficient. It has been decided to upgrade the obsolete technologies such as PCI bus with modern hardware. The first (the frontend electronics) and the second layer (CATCH and GeSiCA modules) of the DAQ system will not be changed. However, Readout buffers and event builders are replaced with modern FPGA (Field Programmable Gate Array) cards. The output will be consistent with the old setup in order not to affect any following processing and physical analysis of data. The new system will be easily scalable (for simple future upgrades), more energy efficient, and will offer sufficient performance.

Software

As a reaction to the upgraded hardware, a new DAQ software has to be developed. It must maintain the same (or better) functionality and keep the current data format unchanged (in order not to influence any further parts of data processing - backward compatibility). It is inspired by the DATE software package, however it will be more lightweight and thus easier

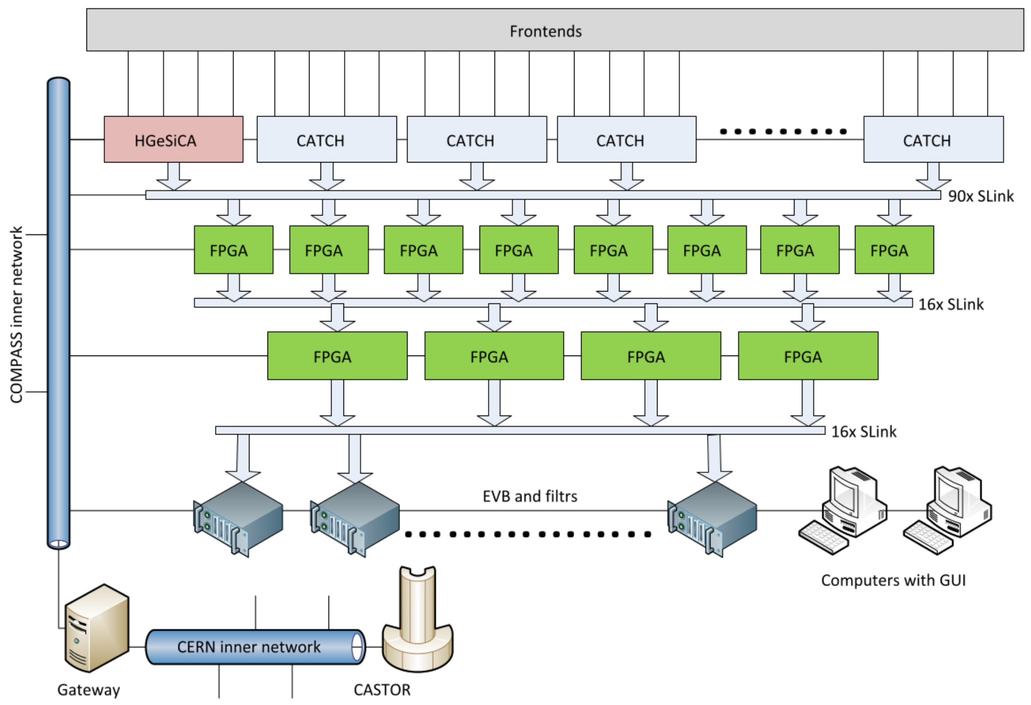


Figure 1.3: The new HW setup of COMPASS DAQ [2]

to maintain. It will be used only for control, configuration, and monitoring purposes. The actual readout and event building will be performed by the FPGA hardware. Five types of applications are currently under development:

- Master process - a Qt console application. It is the most important part, almost all application logic is concentrated here. It serves as a mediator between Slaves and GUI.
- Slave process - an application that controls and monitors a custom hardware. It is controlled and configured by the Master, it informs the Master about the state of the hardware it is deployed on.
- GUI - a Qt GUI application designed for controlling and monitoring with emphasis on modularity. It can be launched in many instances, however only one has the rights to change configuration and to execute control commands, the other serve only for monitoring. It sends commands to the Master. Master resends the monitoring data about the hardware controlled by the Slaves to the GUI.
- Message Logger - a console application that receives informative and error messages and stores them into the MySQL database. It is directly

connected to the Master and to the Slave processes via the DIM service (DIM is presented in section 2.3).

- Message Browser - a GUI application that provides an intuitive access to messages from system (stored in the database) with an addition of online mode (displaying new messages in realtime). Equipped with filtering and sorting capabilities, it is able to run independently from the whole system in case of emergency.

Chapter 2

Software technologies

2.1 C/C++

C and C++ are a general-purpose and one of the most popular programming languages. C was designed by Dennis Ritchie in 1972. C++ first appeared in 1983, written by Bjarne Stroustrup. C++ extends functionality of pure C by adding object oriented features. The latest C++ standard was approved in 2011. [17]

2.2 Qt framework

Qt is a multi-platform application framework extensively used for developing applications with a graphical user interface (GUI), although it can also be used for developing non-GUI command-line programs.
The first version of the Qt was released in 1995. Qt was being developed by a Norwegian company Trolltech until 2008, when it was acquired by Nokia. Three years later, Nokia sold the commercial licensing part of Qt to Digia, but Nokia still remains as the main developer of Qt. Qt is available under a commercial license, GPL v3 and LGPL v2. [12] [11]
Qt is supported on broad spectrum of platforms including:

- Windows
- Apple OS X
- X11 - GNU/Linux, FreeBSD, HP-UX, Solaris . . .
- Wayland
- Android
- iOS

Qt has been also ported to many other operating systems and embedded linuxes - Open Solaris, OS/2, webOS, Tizen, Ubuntu, Amazon Kindle DX, Symbian, Windows Mobile...
Although the Qt Framework is primarily developed as a C++ library, Qt provides several bindings for Java, Pascal, PHP, Python, Ruby, C#, and other languages.
The Qt SDK offers variety of developers tools. Qt Designer is a software used to design widgets and an appearance of graphical user interface (GUI). It can act as a stand-alone application, but it is usually a part of a Qt Creator. The Qt Creator is a complete IDE (Integrated Development Environment) for developing C++ Qt applications. It supports syntax highlighting, code control, revision control systems (GIT, Subversion) or performance testing tools such as Valgrind.

2.2.1 Graphical User Interface

Qt offers an intuitive way of creating GUI. The whole GUI is based on widgets. Widgets are visible elements that are used to display informations or to control the running application. A `QObject`, the base class for all Qt Objects, stays on top of the Qt class hierarchy. `QWidget` is derived from QObject and all widgets are subclasses of QWidget.

Widgets are classified according to their main functionality:

- Buttons - Push Button, Radio Button, Check Box, ...
- Item Widgets - List Widget, Tree Widget, Table Widget ...
- Containers - Group Box, Frame, Widget, Scroll Area ...
- Input Widgets
  - Text input - Line Edit, Text Edit, Plain Text Edit ...
  - Numbers - Spin Box, Double Spin Box
  - Time & Date Edit
  - Scrollbars
  - Sliders
- Display Widgets - Label, LCD Number, Progress Bar, Horizontal/Vertical Lines ...
- Layouts - Vertical, Horizontal, Grid ...
- Spacers

Widgets are very flexible. They can be contained in other widgets (Group Box, Frame ...), every widget can act as a stand-alone window. To avoid complications with fixed-size widgets (during window resizing), it is good practise to place them into layouts. Layouts take care of their size and position in respect to other layouts and widgets within the given layout. The .ui file, where the information about GUI is stored, uses XML structure to describe the hierarchy and settings of widgets in a window. [bodlos, pepa, wiki, stranky qt]

2.2.2 Signals & Slots

The Signals & Slots mechanism is the major feature of Qt. It provides independent inter-object communication between Qt objects. This concept can be demonstrated in a scenerario, where GUI Widgets can send a signal including data about the event which can be received by slot function in another Qt Object. For example clicking a button causes the appropriate widget to emit a signal. Slots are regular functions, that after being linked with a signal, are immediately triggered when the signal is emitted. Signals and slots can be connected and disconnected at any time during the execution of an application by calling `connect()` or `disconnect()` function.

```
connect(sender,SIGNAL(data),receiver,SLOT(data));
disconnect(sender,SIGNAL(data),receiver,SLOT(data));
```

Widgets possess built-in signals and slots, additionally user-defined signals and slots can be implemented. It is possible to establish multiple connections and even connections between signals.

2.3 DIM Library

DIM library (Distributed Information management) is a communication system originally developed for Delphi experiment at CERN. DIM, as many other communication systems, is based on the client/server paradigm, that is further extended by DNS (DIM Name Server).

DIM was designed to meet up to following requirements: [5]

- Efficient Communication Mechanism
  - Asynchronous Communication - no polling at regular intervals required, a process is informed when a change occurs.

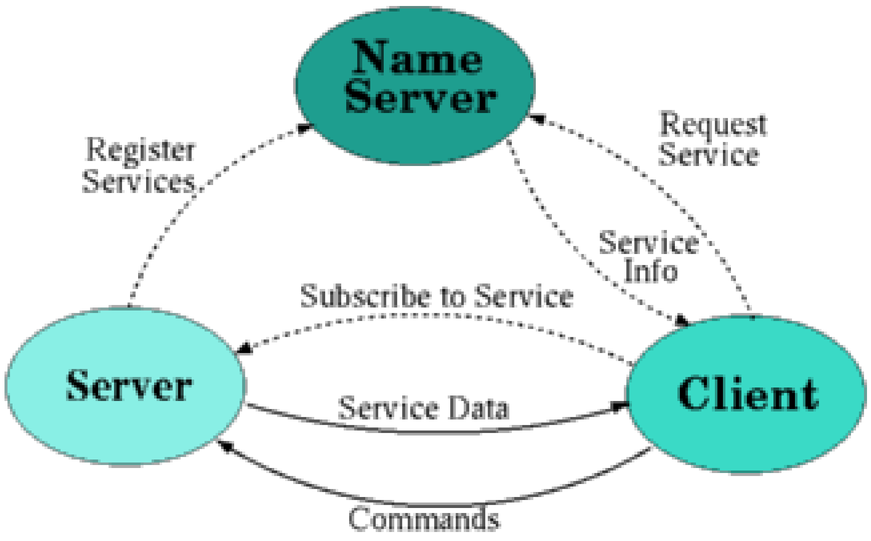


Figure 2.1: Interaction of DIM components [13]

  - 1 to N Communication - one or more processes have to be informed, when something changes.

- Uniformity
  - DIM should be able to handle all communication within the system. A homogeneous system is much easier to program and maintain.
  - The same interface for all types of machines

- Transparency
  - Wherever a process runs, it should be able to communicate with any other process in the system, using a location-independent mechanism.
  - The DIM should allow processes to move from one machine to another, i.e.: to be independent of machine it runs at. It would help to an easier recovery from errors and to balance the load between different machines.

- Reliability and Robustness
  - DIM should provide a mechanism for automatic recovery from error situations or the migration of processes between machines.

A service is the fundamental base of the DIM. Service transfers a set of data (of any type and size) from server to client. Services are requested by clients usually only once at startup and the server updates them automatically in a given period of time or upon a change of conditions. To assure the transparency (i.e.: client does not need to know where a server is running), DNS server was added into the architecture. Servers publish services and register them to the DNS, whereas clients subscribe to these services. The DNS keeps track of all running servers and services they provide. Clients initially ask the DNS whether the service they request is provided. If it is so, the DNS sends them the address of the server that offers the service and then the communication is carried out directly between the server and the clients. If any error or a change of a location of a server occurs, the DNS informs affected clients and servers to re-establish their connection.

The DIM Library includes a tool that allows to monitor current setup - DID - Distributed Information Display. It shows all servers, services they provide, and a list of their clients. [13] [1]

2.4 MySQL

MySQL is an open-source implementation of SQL (Structured Query Language) relational database management system (RDBMS). It was first released in 1995, since then, it has become the second most widely used [db-engines.com] database system. Currently it is developed by ORACLE corporation and it's source codes are available under GNU General Public License. It is written in C++, but there are libraries allowing other programming languages to access the MySQL database. MySQL is available for all common platforms such as Windows, Linux, OS X, Solaris ... [1] [15]
A database is an organized collection of data. [16] Typically, it is a set of mutually related tables. Tables are 2-D structures of rows and columns. Data records are stored in rows, whereas each column is defined by it's data type. MySQL distinguishes between numerical, date/time, and string data types.

2.4.1 Data types

Numerical data types include common integer, fixed-point and floating-point types. Integer types in sizes from 1 to 8 bites can be both in signed and unsigned versions. Boolean type is in fact only an alias for 1-bit integer type (TinyInt). Decimal and Numeric represent the fixed-point data types. Float

and Double (Real) stand for floating-point numerical types.
Date and time related values can be stored in Time, Date, DateTime, or TimeStamp columns.
MySQL recognizes several string types. Char and VarChar are similar types, initially declared with the maximum length of the string they store. The difference is that the char type always stores the full length, the stored value is right-padded with spaces. The VarChar columns contain variable-length strings.
Binary and VarBinary types store binary strings, that is, they have no character set and the comparison and sorting is based on numeric values of the bytes in stored strings. The difference between Binary and VarBinary is the same as between Char and VarChar.
Similar difference is also between Blob and Text types. Blob is binary, whereas Text stores nonbinary character strings. These types are designed for longer texts. According to given maximum required length in the declaration, the actually used type is either TinyText, MediumText, or LongText (or their -Blob alternatives).
MySQL also uses enumeration data type Enum and a Set type designed for representation of a set of string values.

2.4.2 Flags

Beside the specific data type, columns can have additional properties and restrictions.

- `NOT NULL` - the column cannot contain an empty field. `NULL` means that the value is not known or that it has no value.
- `UNIQUE` - it assures that values in the column will not be duplicated; it does not forbid `NULL` values.
- `AUTO INCREMENT` - it let's the database engine to create an unique number for each new row in the column.
- `PRIMARY KEY` - it is the `UNIQUE` atribute combined with the `NOT NULL`; only one column can be marked as a PK

2.5 Transport protocol

The *Transport protocol* is a custom protocol developed for exchanging information between the nodes of the system. It uses the `QByteArray` as a data storage. It also provides methods to parse and create messages. More detailed description of the *Transport protocol* can be found in [1] and [2].

Chapter 3

Monitoring tools

3.1 Message Logger

The Message Logger is a console application created for gathering informative and error messages from the other DAQ nodes. It is using DIM to directly connect to the Master and the Slave processes (the messages from Slaves are not forwarded through the Master). It is using custom Transport protocol (section [2.5]) and the DIM library to communication. These messages are then stored into the MySQL database.
As a DIM client, it subscribes to many DIM information services from the slaves and the master process. It is also registered as a DIM server via the DIM DNS, therefore the master can control whether data taking is enabled or disabled.

3.2 Message Browser

The Message Browser is a GUI (Graphical User Interface) application developed using Qt framework. It's purpose is to display both messages already stored in the MySQL database by the Message Logger and the currently incoming messages. It offers rich and intuitive filtering and sorting options. It is intended as a replacement of the infoBrowser application from the DATE software package [7] used in the old COMPASS DAQ setup.
The Message Browser is based on the standard model-view-controller software architecture. Model is represented by the underlying data structure containing all messages fetched from the database and received via the "online" part. The view is a table in the graphical interface displaying those messages. Controller stands for the filtering mechanism, allowing only the messages meeting the chosen criteria to be displayed.

Upon startup, the application loads messages from the MySQL database from the last month (this time period might be refined later). Then it tries to subscribe to the same services as the Message Logger does. If it succeeds, it begins to receive informative and error messages from system as they supervene. Therefore it is not necessary to poll the database periodically for new messages. It is worth mentioning that it does not overtake the job of the Message Logger of storing the messages into the database. If the Message Browser is unsuccessful in connecting to the DIM services (that probably means the whole system is down), the database-related part is still untouched and working. This approach allows efficient usage of system resources (no unnecessary polling), while ensuring independence from the rest of the system.

3.2.1 Graphical user interface and control

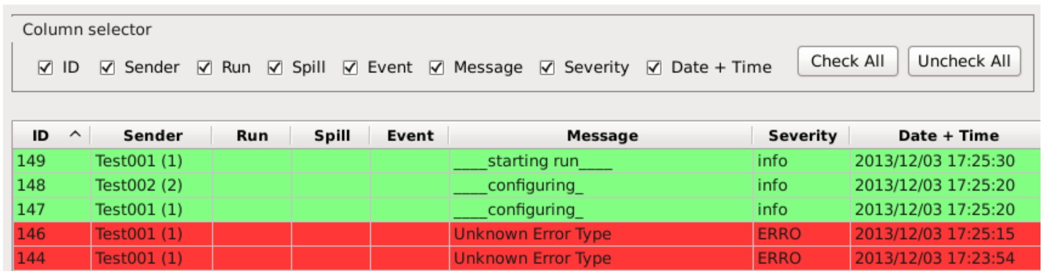
The Graphical User Interface of the Message Browser consists of four main widgets arranged into a grid. In the upper part of the window there are sections called Column selector and Filter setting, the filter panel occupies the right side of the window and the rest of the space is filled with a table containing the actual informative and error messages. The dimension of these widgets is managed by horizontal and vertical layouts (briefly described in the section 2.2.1) thus ensuring resizing along with the main window.

Column selector

The Column selector widget contains eight `CheckBoxes` and two `buttons`. CheckBoxes are assigned to corresponding columns (ID, Sender, Run, Spill, Event, Message, Severity, and Date + Time) in the table with messages. As the name of the widget suggests, checking or unchecking a CheckBox allows the user to show or hide the desired column. The buttons are labeled "Check All" and "Uncheck All". Pressing them causes all CheckBoxes to check or uncheck. It is the same as clicking on them independently, but faster. All actions will take effect immediately.

Filter settings

The Filter settings section is made of three buttons (labeled as "Set initial date", "Default", and "Show/Hide filters") and one date picker. Upon startup, the initial date is set by default to the date a month ago. User can

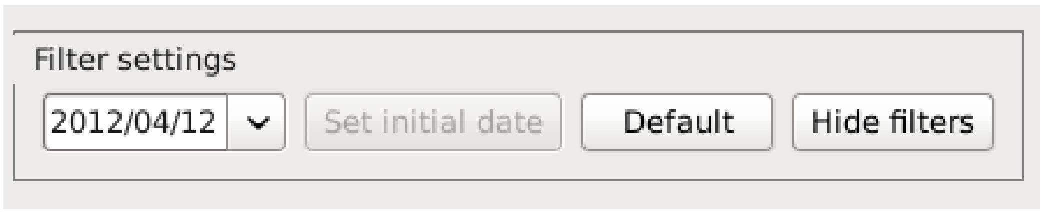

Column selector

☑ ID ☑ Sender ☑ Run ☑ Spill ☑ Event ☑ Message ☑ Severity ☑ Date + Time

ID	Sender	Run	Spill	Event	Message	Severity	Date + Time
149	Test001 (1)				____starting run____	info	2013/12/03 17:25:30
148	Test002 (2)				____configuring_	info	2013/12/03 17:25:20
147	Test001 (1)				____configuring_	info	2013/12/03 17:25:20
146	Test001 (1)				Unknown Error Type	ERRO	2013/12/03 17:25:15
144	Test001 (1)				Unknown Error Type	ERRO	2013/12/03 17:23:54

Figure 3.1: The Column selector widget with a few rows of underlying table

change this date and when the "Set initial date" button is pressed, the Message browser loads all records from the MySQL database since the selected date. The filtering and column settings remains unchanged. The "Show/hide filters" button shows or hides the left side filter panel. When the panel vanishes, the resulting empty space is used by the table with messages. It is useful to use when user doesn't need to make any further adjustments to the filter settings and wants to use the full width of the window for displaying the messages.
The "Default" button is a tool for resetting the application to it's default settings. All filters and column selectors are set to initial values and the Message browser loads the last month of recorded data.


Filter settings

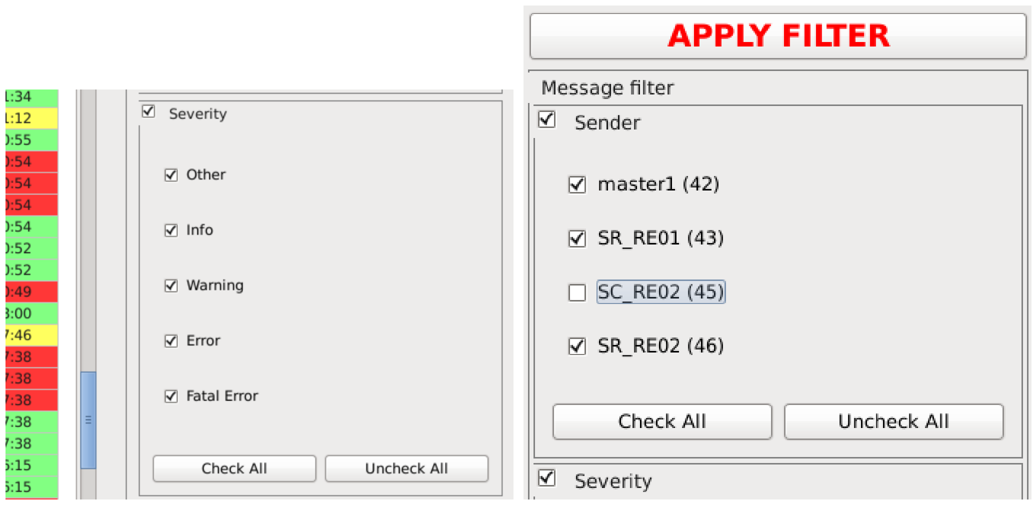
2012/04/12

Figure 3.2: The Filter settings widget

Filter panel

The Filer panel is a widget designed to adjust filter settings. It is composed of a big red "Apply filter" button and seven checkable areas for setting up parameters of the filter. If an area is checked, it expands and reveals detailed settings of the corresponding parameter. Unlike column selector, user must hit the "Apply filter" button to use the new filtering settings.

The Sender group allows filtering the messages by the device that generated them. It contains checkboxes only with the names and ID's (in order to distinguish between devices with the same name) of the source computers whose messages are currently loaded in the memory. Additionally there are "Check all" and "Uncheck all" buttons to speed up the selection of desired senders. If all senders are checked, it has the same resulting effect as if the whole group is unchecked.



(a) The Severity (b) The Sender

Figure 3.3: The Severity and Sender filtering options

The Severity group allows filtering by the severity of the messages. It behaves the same way the Sender group does, except for the fact that there are permanent choices (not dependent on messages currently loaded in the memory). The severity of the messages can be one of - Info, Warning, Error, Fatal Error, and Other. By default, the Info severity is unselected.

Run number, Spill number, and Event number groups are all very similar. There are two Radio buttons to choose between an exact number or a range. If the "exact" option is choosen, only the messages with choosen number of run, spill, or event will be displayed. If the "range" is choosen, "From" and "To" checkboxes are enabled. Depending on which of them the user checks, messages with number higher, lower, or within a given range will be displayed.

The Date&Time area allows filtering based on the date and time the messages were generated. User can filter all messages originated before or after a specified date and time, or within a desired date and time range.

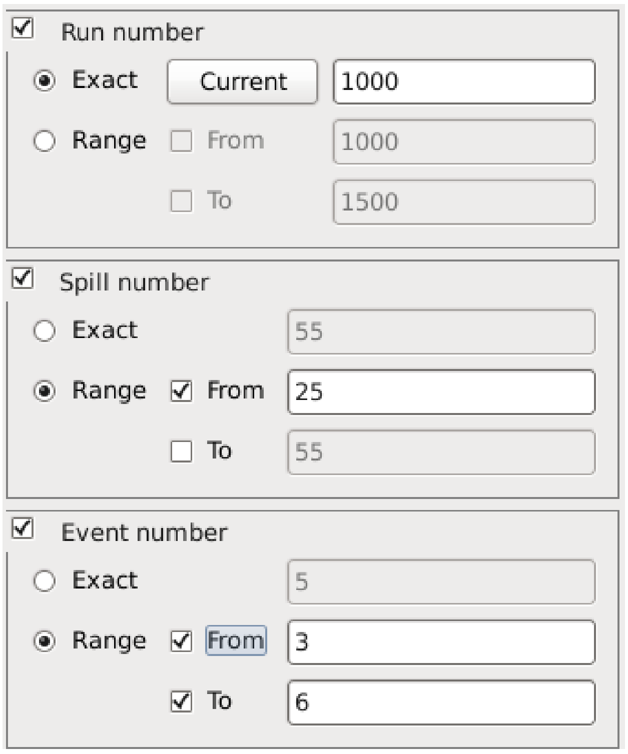


Figure 3.4: The Number filters

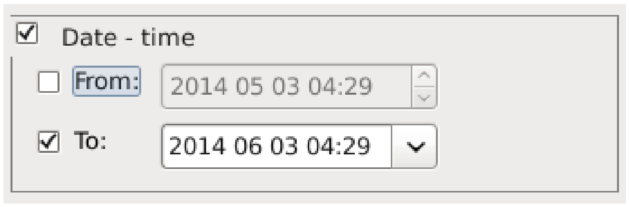
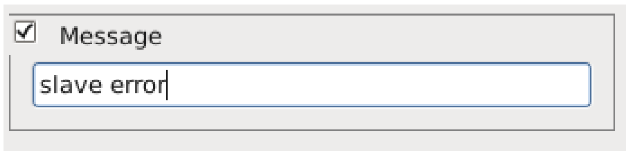


Figure 3.5: The Date and Time filter

The last filtering option displays messages that contain the entered text. The searching is not case sensitive. Pressing the `Enter` key triggers the filtering - no need to click on the "Apply filter" button.

Table view widget

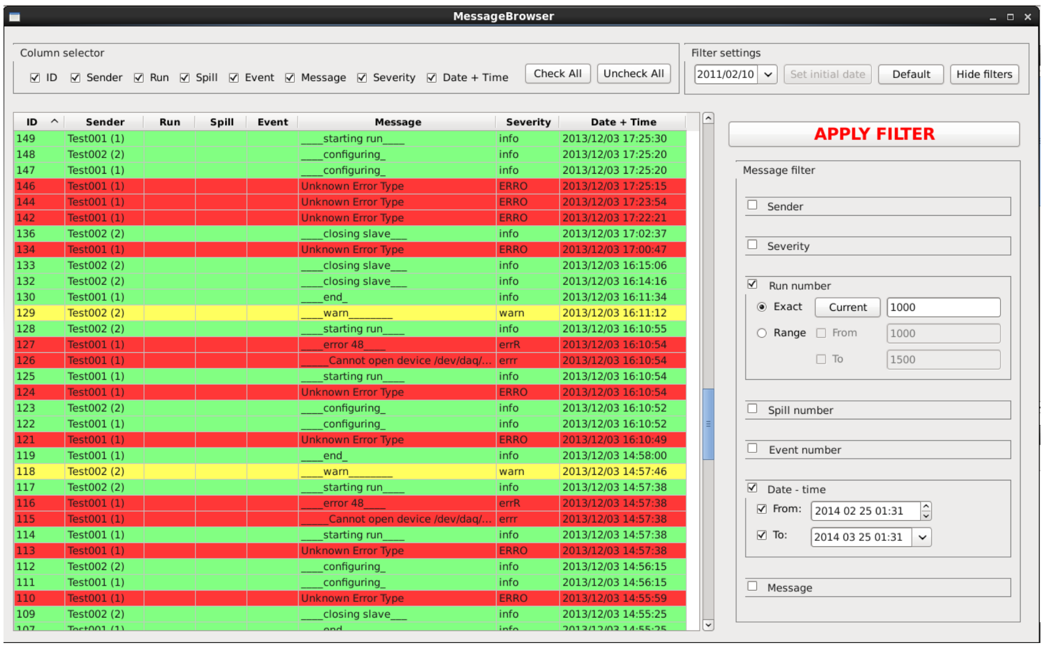
The table view widget is the core of the application. It displays messages loaded from the database and messages received from other nodes. It is a table with eight columns - ID of the message, Sender name and ID, Run, Spill


☑ Message

slave error

Figure 3.6: The Message text filter

and Event numbers, text message, severity, and Date + Time. Every message occupies a single row. It is possible to sort the entries by any column in both ascending and descending order. When a header of a column is clicked the first time, the content of the table is ordered by this column in the ascending order. The second click causes the data to order in the descending order. The width of columns can be easily adjusted to match the needs of the user. The color of the rows is determined by the severity of the message to provide an information about the occurrence of errors with a quick glance at the screen. Informative messages appear with a green background, errors are red, fatal errors blue, and the other messages are without any color highlighting.



Message filter

☐ Sender

☐ Severity

☑ Run number
● Exact [Current] 1000
○ Range ☐ From 1000
☐ To 1500

☐ Spill number

☐ Event number

☑ Date - time
☑ From: 2014 02 25 01:31
☑ To: 2014 03 25 01:31

☐ Message

ID	Sender	Run	Spill	Event	Message	Severity	Date + Time
149	Test001 (1)				____starting run____	info	2013/12/03 17:25:30
148	Test002 (2)				____configuring_	info	2013/12/03 17:25:20
147	Test001 (1)				____configuring_	info	2013/12/03 17:25:20
146	Test001 (1)				Unknown Error Type	ERRO	2013/12/03 17:25:15
144	Test001 (1)				Unknown Error Type	ERRO	2013/12/03 17:23:54
142	Test001 (1)				Unknown Error Type	ERRO	2013/12/03 17:22:21
136	Test002 (2)				____closing slave___	info	2013/12/03 17:02:37
134	Test001 (1)				Unknown Error Type	ERRO	2013/12/03 17:00:47
133	Test002 (2)				____closing slave___	info	2013/12/03 16:15:06
132	Test002 (2)				____closing slave___	info	2013/12/03 16:14:16
130	Test001 (1)				____end_	info	2013/12/03 16:11:34
129	Test002 (2)				____warn________	warn	2013/12/03 16:11:12
128	Test002 (2)				____starting run____	info	2013/12/03 16:10:55
127	Test001 (1)				____error 48____	errR	2013/12/03 16:10:54
126	Test001 (1)				_____Cannot open device /dev/daq/...	errr	2013/12/03 16:10:54
125	Test001 (1)				____starting run____	info	2013/12/03 16:10:54
124	Test001 (1)				Unknown Error Type	ERRO	2013/12/03 16:10:54
123	Test002 (2)				____configuring_	info	2013/12/03 16:10:52
122	Test001 (1)				____configuring_	info	2013/12/03 16:10:52
121	Test001 (1)				Unknown Error Type	ERRO	2013/12/03 16:10:49
119	Test001 (1)				____end_	info	2013/12/03 14:58:00
118	Test002 (2)				____warn________	warn	2013/12/03 14:57:46
117	Test002 (2)				____starting run____	info	2013/12/03 14:57:38
116	Test001 (1)				____error 48____	errR	2013/12/03 14:57:38
115	Test001 (1)				_____Cannot open device /dev/daq/...	errr	2013/12/03 14:57:38
114	Test001 (1)				____starting run____	info	2013/12/03 14:57:38
113	Test001 (1)				Unknown Error Type	ERRO	2013/12/03 14:57:38
112	Test002 (2)				____configuring_	info	2013/12/03 14:56:15
111	Test001 (1)				____configuring_	info	2013/12/03 14:56:15
110	Test001 (1)				Unknown Error Type	ERRO	2013/12/03 14:55:59
109	Test002 (2)				____closing slave___	info	2013/12/03 14:55:25

Figure 3.7: The Message Browser application

Chapter 4

Implementation

4.1 Message Browser

4.1.1 Model

The implementation of the Message Browser uses a concept of the model-view-controller software architecture. The *model* part is represented by a newly designed *msgDataModel* class. Unlike the depreciated *MSGBrowserModel* class, which was built upon a `QSqlQueryModel`, the new one subclasses Qt's `QAbstractTableModel`. The `QSqlQueryModel` provides an easy access to the MySQL database, on the other hand it doesn't easily allow to display data from other sources, which is necessary due to the "Online mode" functionality requirement. The `QAbstractTableModel` seemed a better choice, because it is prepared to be a model for table views and a custom underlying data structure is required as a temporary storage for messages.
The private section contains three important member variables:

```
private:
    QList<QList<QVariant> > *m_data;
    uint m_rows;
    uint m_cols;
```

The `m_data` variable is a dynamic two-dimensional structure of nested `QLists` of `QVariant`. It is used as the temporary storage for messages. The `m_rows` and `m_cols` variables contain the count of rows and columns that are currently stored in `m_data`. `QVariant` acts like a union for the most common Qt data types. [12, QVariant]

In order to subclass the `QAbstractTableModel` class, `data()`, `rowCount()`,

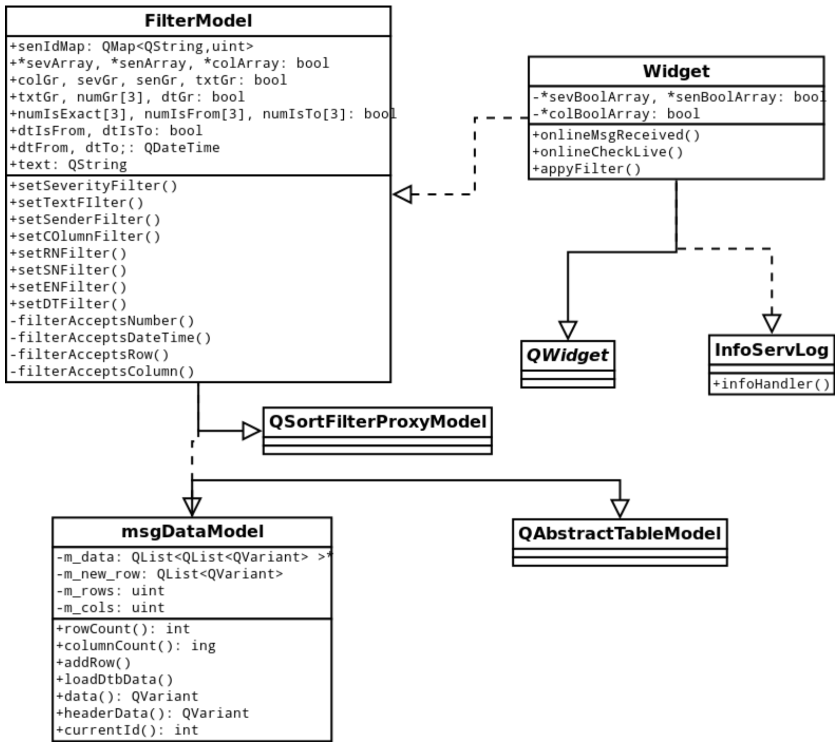


Figure 4.1: The class diagram of the Message Browser

and `columnCount()` methods have to be reimplemented. The `data()` function:

```
QVariant data(const QModelIndex &index, int role)
                                              const;
```

is used by the *view* part of the application to retrieve everything it displays. The view passes the `index` and `role` parameters. The index specifies coordinates of the cell of interest, the role describes the type of information required. The role can have many values, but the `Qt::DisplayRole` is the most important. In case the role is set to `Qt::DisplayRole`, the `data()` function returns the actual data from `m_data` in a form of `QVariant` which the view converts to a string value. The text information of messages is stripped of whitespace and the time-stamp is converted into a convenient date and time format.

```
switch (role){
  case Qt::DisplayRole:{
     col=index.column();
```

```
    switch (col){
    case COL_TEXT:{     //getting rid of whitespaces
       qv=m_data->at(index.row()).at(COL_TEXT);
       text=qv.toString().trimmed();
       return QVariant(text);
    }
    case COL_STAMP:{
       qv=m_data->at(index.row()).at(COL_STAMP);
       return qv.toDateTime().
             .toString("yyyy/MM/dd hh:mm:ss");
    }
    default:
       qv=m_data->at(index.row()).at(col);
       return qv;}
  }
```

For this application, the `Qt::BackgroundRole` is also important. It is used to retrieve information about the background color of the involved cell. The color for the whole row is determined by the severity of the message. The *Fatal error* has a blue color, *Error* is red, *Warning* yellow, *Informative* messages are green, and the rest is without any color.

```
case Qt::BackgroundRole:{
   qv = m_data->at(index.row()).at(COL_SEV);
   sev=qv.toString().toLower().left(3);
   if (sev == "fer"){    //fatal error
       return QVariant(QColor(0,0,255,125));
     }
   else if (sev == "err"){  //error
       return QVariant(QColor(255,0,0,200));
     }
   else if (sev == "war"){  //warning
       return QVariant(QColor(255,255,0,160));
     }
   else if (sev == "inf"){  //information
       return QVariant(QColor(0,255,0,125));
     }
   else{       //everything else
       return QVariant(QColor("white")); }
}
default: return QVariant();}
```

Because this *model* provides resizable data structure (every new incoming message should be added as a new row to the m_data member variable), a general insertRows() method has been reimplemented. It is important to call beginInsertRows() before and endInsertRows() right after the insertion of the new row into the model data structure. These methods notify the connected *views* about the changes. Because only one row is added at the time, simple addRow() public method was added. This method receives a QList<QVariant> containing a new message and stores it into a private member variable m_new_row and then calls insertRows() to carry out the actual adding of the new message.

```
bool msgDataModel::insertRows(int row,int count=1,...
    ... const QModelIndex &parent=QModelIndex())
{
    beginInsertRows(parent,row,row+count-1);
      m_data->append(m_new_row);
      m_rows=m_data->length();
    endInsertRows();
    m_new_row.clear();
    return true;
}
void msgDataModel::addRow(QList<QVariant> ql){
    m_new_row=ql;
    insertRows(m_rows);
}
```

The loadDtbData() is the last important method of the msgDataModel class. This method is not only called at the startup of the application, but also when user selects a new initial date, or presses the Default button. This method completely changes the data in the *model*. Therefore it is necessary to notify all components that depend on the model. Calling beginResetModel() on the beginning and endResetModel() in the end will do so for us. At first, it clears all previously stored data and then it loads new data from the database. The connection to the MySQL database is provided by the database library custom-made for the DAQ project [bodlos diplomka]. A MySQL query is prepared according to the initial date provided as a parameter, and then executed. The result is then appended row by row to the main data structure. The rows are appended directly to the m_data, there is no need to explicitly call the addRows() method, as it would unnecessarily call insertRows() and the connected methods for notifying of other components - they have already been notified by the beginResetModel() method. The name of the source device of a currently stored message is enriched by it's database Id.

```
void msgDataModel::loadDtbData(QString date){
  beginResetModel();

  m_cols=8;   //number of columns
  m_data->clear();
  QString query="SELECT a.Id, b.Name, a.run_number,
  a.spill_number, a.event_number, a.text,
  a.severity, a.stamp, a.process_id
  FROM Message_log a, Process b
  WHERE a.process_id = b.Id AND a.stamp >= :date
  ORDER BY a.Id ASC";

  QSqlQuery que;
  que.prepare(query);
  que.bindValue(":date", date);
  que.exec();

  while (que.next()){
    for (uint j=0;j<m_cols; j++){
      if (j==COL_NAME){     //source of mmsg + (Id)
        m_new_row.append(QVariant(que.value(j).
         .toString()+" ("+que.value(COL_PROCID).
         .toString() +")"));
      } else {
        m_new_row.append(que.value(j));
      }
    }
    m_data->append(m_new_row);
    m_new_row.clear();
  }
  m_rows=m_data->length();

  endResetModel();
}
```

4.1.2 Filtering

The `FilterModel` class is subclassed from the `QSortFilterProxyModel` class. This class acts as a mediator between the data model and a view displaying those data. It provides sorting and filtering capabilities. The QSortFilter-ProxyModel acts as a wrapper for the original model. The model transforms the structure of a source model by mapping the model indices it supplies to new indices, corresponding to different locations, for views to use. This approach allows a given source model to be restructured as far as views are concerned without requiring any transformations on the underlying data, and without duplicating the data in memory. [12, QSortFilterProxyModel]

The `QTableView` used for displaying data has a `sortingEnabled` property. When it is set to `true`, the sorting capabilites of the *QSortFilterProxyModel* are activated. When user clicks the horizontal header of the view, the view calls the `sort()` method of the underlying model. By repeated clicking, the sorting alters between ascending and descending order. The model then provides the data sorted in the desired way.

For filtering purposes, two major methods from the *QSortFilterProxyModel* super class - the `FilterAcceptsRow()` and the `FilterAcceptsColumn()` - have been reimplemented. They return `true` (and thus the row/column will be displayed in the `QTableView`) in case that given row or column matches the user-defined filter criterion.

Column filter

The column filter allows users to select (via checkboxes) columns which should be displayed. The behavior is described in section 3.2.1. The checkboxes are created dynamically upon startup and their labels are read from the header data of the `msgDataModel`. Toggling of each checkbox is connected to the `columnSelectorChanged()` slot method of the `widget` class. This method fills a `colBoolArray` boolean array with information about which checkboxes are checked, it also takes care of proper resizing of the remaining columns. The boolean array is then passed to the `filter model` via it's interface. It is important to call `fmodel->invalidate()` to notify the filter about changed criterion (`fmodel` is the instance of the `FilterModel` class that takes care of filtering and sorting). The `FilterAcceptsColumn()` method then allows filtering of columns according to this data. The `colArray` points to the original `colBoolArray`, and the method returns `true` (column will be displayed) if the related checkbox is checked.

```
fmodel->setColumnFilter(colBoolArray);
```

```
bool FilterModel::filterAcceptsColumn(
  int source_column,
  const QModelIndex) const
{
    return (colArray[source_column]);
}
```

Severity filter

The severity filter is used to filter messages depending on their severity. The severity can be one of *Info, Warning, Error, Fatal error, or Other.* This filter is quite similar to the column filter. The severity checkboxes are created upon startup and inserted into a checkable `QGroupBox` *groupSev.* The toggle event of the group and checkboxes is connected to `notifyFilter()` slot method of `widget` class that enables the "Apply filter" button. When the filter is applied, the status of the checkboxes is read-out into a `sevBoolArray` boolean array. This array is then passed to the `filter model` via it's `setSenderFilter()` method.

Sender filter

This filter filters messages by the name of the device that generated them. The functionality is similar to the Severity filter - a set of dynamically created checkboxes layed out in a GroupBox. At the beginning of the sender initialization, the names and Id's of senders are loaded from the MySQL Database. The `getSendersInfo()` method loads only those senders that generated the messages currently stored in the `data model`. The `n_name` is a `QVector<QString>` variable that stores the names and Id's of the loaded senders. The `n_onlineIdToName` is a `QMap<uint,QString>` variable that maps the database Id's of senders to their names. It is used in the "Online part" when a new message arrives (further discussed in section 4.1.3). The `n_ids` maps the sender's rank to it's name - it is used while evaluating the `filterAcceptsRow()` method. When the `GroupBox groupSen` or the individual checkboxes are checked or unchecked, it executes the `notifyFilter()` slot. When the "Apply button" is pressed, the information which checkboxes are checked is passed to the `filter model` via the `setSenderFilter()` method.

```
void  Widget::getSendersInfo(){
 QString query="SELECT DISTINCT b.Name, a.process_id
                FROM Message_log a, Process b
                WHERE a.process_id = b.Id
                      AND a.stamp >=:date";
 QSqlQuery que;
 que.prepare(query);
 que.bindValue(":date", initDateString);
 que.exec();

 uint i=0;
 while (que.next()){
   n_names.append(que.value(0).toString()+" ("+
      +que.value(1).toString()+")");
   n_onlineIdToName.insert(que.value(1).toInt(),
      n_names.at(i));    //for incomming  messages
   n_ids.insert(n_names.at(i),i);  //for filtermodel
   i++;
 }
 nodesCnt=n_names.size();
}
```

Number filters

Number filters provide filtering of messages based on their scope in the experiment. The filters can be set in several ways. User can filter by an exact number, or limit the values by a lower or upper boundary, or an exact range of values can be defined. When Run number group, Spill number group, or Event number group is (de)activated, or when anything inside these groups is changed, a signal that triggers `notifyFilter()` slot is emitted. It enables the "Apply filter" button. When the filter is applied, the settings of the Number filters is passed to the `filter model` via `setRNFilter()`, `setSNFilter()`, and `setENFilter()`.

```
void FilterModel::setENFilter(bool e, bool isex,
                int ex, bool isfrom, int from,
                bool isto, int to)
{
    numGr[FEVE] = e;
    numIsExact[FEVE] = isex;
    numIsFrom[FEVE] = isfrom;
    numIsTo[FEVE] = isto;
    numValsExact[FEVE] = ex;
    numValsFrom[FEVE] = from;
    numValsTo[FEVE] = to;
}
```

The `setENFilter()` method passes seven arguments that are assigned to seven corresponding member variables of the `filter model`. `numGr` stores information if the group is activated (`true`). `numIsExact` is `true` if the given group is set to "exact", `false` stands for "range". `numIsFrom` and `numIsTo` indicate whether "From" and "To" checkboxes are checked. `numValExaxt`, `numValFrom`, and `numValTo` hold the actual user-filled integer values from the text fields. These variables are arrays of either three booleans or integers. They hold information for all three Number filters. The index `FRUN` stands for run numbers, index `FSPI` for spill numbers, and index `FEVE` for event numbers.
These variables are passed to the `filterAcceptsNumber()` method that is part of the `filterAcceptsRow()` method.

```
bool FilterModel::filterAcceptsNumber(int num,
        bool e, bool isex, int ex, bool isfrom,
        int from, bool isto, int to) const
{
    if (!e) return true;
    if (isex) return (num == ex);
    if (isfrom && isto) return
      ((from <= num) && (num <= to));
    if (isfrom && !isto) return (from <= num);
    if (!isfrom && isto) return (num <= to);
    return false;
}
```

The `filterAcceptsNumber()` returns `true` if the filter for the given group is disabled, or if the filter is activated and the number meets the required criteria. Otherwise `false` is returned and thus given message is rejected and not displayed.

Date & Time filter

The Date and Time filter mechanism is almost identical to the Number filters. It filters the message according to the Date and Time of their origin. The corresponding `filterAcceptsDateTime()` method works the same way as in the Number filters section (but it compares Date and Time values (`QDateTime`), not integers). The Date & Time only lacks the "exact" option.

Text filter

The text filter is for filtering messages according to the text they contain. When the text group is selected, a text is entered or changed, the filter is notified via the connected signal. When the "Apply button" is pressed, the `setTextFilter()` method is called. It updates the `filter model` and it starts rejecting those messages that do not contain the entered text.

Applying the filters

When the "Apply filter" button is pressed, the current settings of all filters is read-out and passed to the `FilterModel` class via it's setters. The `filterAcceptsRow()` method then uses this information to determine whether given row will be displayed. It returns `true` if the row passes filters for every category.

```
bool FilterModel::filterAcceptsRow(int source_row,
      const QModelIndex &source_parent) const
{
 bool acceptNums = true;
 bool acceptDt;
 .
 .
 .

 for(uint i = 0; i < 3; i++)
 {
    acceptNums = ((acceptNums) &&
      (filterAcceptsNumber(nums[i],numGr[i],
      numIsExact[i],numValsExact[i],numIsFrom[i],
      numValsFrom[i],numIsTo[i],numValsTo[i])));
 }

 acceptDt=filterAcceptsDateTime(dt,dtGr,dtIsFrom,
         dtIsTo,dtFrom,dtTo);

 return (!sevGr || sevArray[sevNum]) &&
 (!txtGr ||
    errText.contains(text, Qt::CaseInsensitive)) &&
 (!senGr || senArray[senIdMap[sender]]) &&
 (acceptNums) &&
 (acceptDt);  }
```

4.1.3 The Online mode

The so-called "Online mode" is a brand new feature of the Message Browser. It allows for displaying new messages in real-time at the moment they are generated, without querying the MySQL database. It uses the DIM library to subscribe to the same services as the Message Logger does. It utilizes the `DimStampedInfo` and `DimBrowser`.

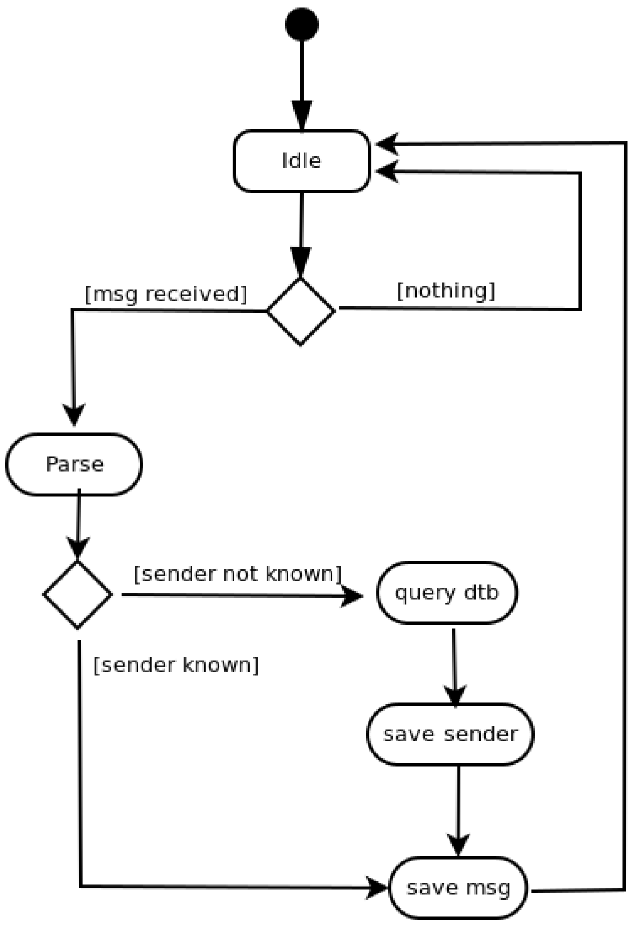


Figure 4.2: The activity diagram of the Online mode

Subscribing

The `onlineCheckLive()` method takes care of subscription to information services that can possibly generate and send a new messages. In order to obtain a list of these services, it is necessary to take an advantage of the

`DimBrowser` class. The `getServices()` method returns the count of the services that match the given name (in this case, it returns the count of all services named "INFO_SERVICE_*", where the asterisk stands for any string). An instance of `InfoServLog` class is created for every "INFO_SERVICE" to handle. The `infLog` is a variable of `QVector<InfoServLog*>` type that holds pointers to instances of the `InfoServLog` class. The `connectedInfLog` string contains the names of all info services subscribed. When a new instance of `InfoServLog` is created, it's `newMessage()` signal is connected to the `onlineMsgReceived()` slot in order to save the received message to the data structure. This `onlineCheckLive()` method is triggered by a times every five seconds to periodically check for changes in the Information Service providers.

```cpp
void Widget::onlineCheckLive(){
 count = br.getServices("INFO_SERVICE_*");
 if(count>0)count--;
 if(count>infCount)
 {
  infCount=0;
  InfoServLog *Il;
  while (br.getNextService(name, format) != 0)
  {
   if(!connectedInfLog.contains(name))
   {
    Il=new InfoServLog(name,ready);
    infLog.append(Il);
    connect(Il,SIGNAL(newMessage(QList<QVariant>)),
    this,SLOT(onlineMsgReceived(QList<QVariant>)));
    infCount++;
    connectedInfLog.append("||");
    connectedInfLog.append(name);
   }
  }
  qDebug() << "Subscribed to " << count;
 }
}
```

Message handling

The `InfoServLog` class is subclassed from the DIM `DimStampedInfo` class in order to receive and handle new messages enriched with a timestamp.

Every time a new message arrives, the `infoHandler()` method is executed. It extracts the timestamp (`getTimestamp()`) and the data (`getString()`) from the message. The data is then parsed via the *Transport Protocol* [2]. The `msgBody` is split into pieces that are used to assemble the message in a format used in the `msgDataMode` class. However, the ID of the message is not known yet - the `data model` is out of scope, an arbitrary value of *-1* is inserted as a space-filler. The likely ID will replace it in the next phase. The `sender` is represented by it's database Id, not by the typical combination of it's name and Id.

```
void InfoServLog::infoHandler()
{
 TransportProtocol tp;
 if (ready)//if info taking is enabled
 {    //gets message with message size
  QByteArray data(getString(),getSize());
  QByteArray sender, receiver, msgNumber, msgBody;
  QDateTime dt;
  uint sec(getTimestamp());
  dt = QDateTime::fromTime_t(sec);
  tp.parseMsg(data, &sender, &receiver, &msgNumber,
              &msgBody); //parses received message
  QList<QByteArray>list = msgBody.split('|');

  if(msgNumber.toUInt()>0 && msgNumber.toUInt()<1000
                          && list.size()>=5)
  {
   QList<QVariant> nMsg;
   nMsg.append(QVariant(-1));        //ID of msg
   nMsg.append(QVariant(tp.ByteToNumber(sender)));
   nMsg.append(QVariant(
          tp.ByteToNumber(list[MSG_RUN])));//run
   nMsg.append(QVariant(
          tp.ByteToNumber(list[MSG_SPI])));//spill
   nMsg.append(QVariant(
          tp.ByteToNumber(list[MSG_EVE])));//event
   nMsg.append(QVariant(list[MSG_MSG]));//message
   nMsg.append(QVariant(list[MSG_SEV]));//severity
   nMsg.append(QVariant(dt.toString(
            "yyyy/MM/dd hh:mm:ss"))); //timestamp

   emit newMessage(nMsg); //sent to the next phase
   qDebug() << "Info came.";
  }else
  {
   qDebug()<<msgNumber.toUInt()<<"  "<<list.size();
   qDebug()<<"Invalid info";
  }
 }
}
```

The assembled message is received by the `onlineMsgReceived()` slot. The Id of the message is immediately set to it's probable value (previous Id + 1). The real Id is assigned by the MySQL database when the Message Logger stores the message into the database. The Id of sender is read from the message and assigned to the integer `ind` variable. If the program has been in touch with this sender Id before (i.e. the `QMap n_onlineIdToName` contains this Id), the sender part in the message is replaced by the mapped value (combination of the sender's name and Id). Nevertheless, if the Id of the sender is not recognized (meaning that the local copy of the data does not contain any messages from this sender yet), the Id has to be assigned (mapped) to the sender's proper name. The name is loaded from the MySQL database. `n_names`, `n_onlineIdToName`, and `n_ids` are updated with this new name. Then a new checkbox for the new sender is created in the sender area to extend sender filtering options. Finally, the new message is inserted into the `data model` data structure.

```
void Widget::onlineMsgReceived(QList<QVariant>nMsg){
 nMsg[0]=QVariant(dmodel->currentId()+1);

 QString newName;
 int ind=nMsg[COL_NAME].toInt();  //~1, sender col
 if (!n_onlineIdToName.contains(ind)){
  //dtb query
  QString query="SELECT b.Name
              FROM Process b
              WHERE :currInd = b.Id";

  QSqlQuery que;
  que.prepare(query);
  que.bindValue(":currInd", ind);
  que.exec();
  if (que.next()){
    newName=que.value(0).toString();
  } else {
    newName="UNKNOWN";
  }
  //updating structures
  int ii=n_names.size();
  n_names.append(newName+" ("+
        +QString::number(ind)+")");
  n_onlineIdToName.insert(ind,n_names.at(ii));
  n_ids.insert(n_names.at(ii),ii);
  //new checkbox
  nodesCnt++;
  nodesCh.append(new QCheckBox(n_names[ii],
                    ui->senderArea));
  senderLayout->addWidget(nodesCh[ii]);
  nodesCh[ii]->setChecked(true);
  senBoolArray.append(true);

  connect(nodesCh[ii], SIGNAL(toggled(bool)),
           this, SLOT(notifyFilter()));
  fmodel->setSenderFilter(ui->groupSend->isChecked(),
         n_ids, senBoolArray.data());

 }
 nMsg[1]=QVariant(n_onlineIdToName[ind]);

 this->dmodel->addRow(nMsg);   //new msg stored
}
```

4.2 Message Logger

The core of the *Message Logger* is very similar to the *Online part* of the *Message Browser*. The main difference is that it has a direct access to the MySQL database and instead of just displaying the received messages, it stores them into the database. The `main()` function takes care of subscribing to the informative services and keeping up-to-date list of services that are still active. It uses the `DimBrowser` class to get a count of active services and their names. The `main()` function contains an infinite loop that keeps checking whether there are any new services and subscribes to them, if necessary.

```
int main(int argc, char *argv[])
{
. . .

 if (db.createCMADConnection()) {
  cout<<"Database connection established."<<endl;
 }else {
  cout<<"Database connection not established."<<endl;
  qDebug() << db.dberror();
 }

. . .

while (true)
 {
   count = br.getServices("INFO_SERVICE_*");
   if(count>0)count--;
   cout<<count<<endl;
   if(count>infCount)
   {
     infCount=0;
     while (br.getNextService(name, format) != 0)
     {
       if(!connectedInfLog.contains(name))
       {
         infLog.append(new InfoServLog(name, &db,
                                        ready));
         infCount++;
         connectedInfLog.append("||");
         connectedInfLog.append(name);
       }
     }
     cout << "Subscribed to " << count  << endl;
   }
   sleep(1);
  }
return 0;
}
```

Every time a message is received, an `infoHandler()` of the `InfoServLog` class is called. It works almost the same way as in the online part of the *Message Browser*. The difference is that the messages are stored into the database.

```
void InfoServLog::infoHandler()
{
 TransportProtocol tp;
 if (ready)//if info taking is enabled
 {
 .
 .
 .
   tp.parseMsg(data, &sender, &receiver, &msgNumber,
              &msgBody); //parses received message
   QList<QByteArray>list = msgBody.split('|');
  if(msgNumber.toUInt()>0 && msgNumber.toUInt()<1000
                         && list.size()>=5)
  {
     if (!db->newLogEntry(tp.ByteToNumber(sender),
         list[MSG_SEV], list[MSG_MSG],
        (list[MSG_RUN]).toUInt(),
        (list[MSG_SPI]).toUInt(),
        (list[MSG_EVE]).toUInt()))
     {
       cout<<"Writing into database failed."<<endl;
       qDebug() << db->dberror();
     }
     cout << "Info came." << endl;
     }
     else
     {
       cout << "Invalid info" << endl;
     }
  }
}
```

Conclusion

The planned shutdown of the CERN during the years 2013 and 2014 was utilized to upgrade the hardware and software of the Data Acquisition System of the COMPASS experimnet at CERN. The harware part was equipped with the programmable FPGA cards. The software was completely reimplemented. The finish of upgrade of the monitoring tools has been covered in this thesis.

All objectives of this thesis have been fulfilled. In the beginning, the CERN, the COMPASS experiment at CERN and both the old and the new Data Acquisition Systems were fully introduced. The Data Acquisition System was approached from both the hardware and software direction.

In the following part, all important software technologies (such as the Qt framework, MySQL database, and the DIM communication library) were sufficiently explained to the level of requirements of this thesis.

The improvements for the Message Browser, such as the new `data model` and the `online mode` have been designed and successfully implemented. Many other adjustments in the design, functionality and user experience have been completed.

Further testing and fine-tuning is planned during the summer 2014 to achieve perfect stability and robustness that is required during the run. The start of data taking is planned on the fall 2014.

Bibliography

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

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

[3] *Blanchette J., Summerield M.:* **C++ GUI Programming with Qt 4 (2nd ed.)**
Prentice Hall, Qt preview for Symbian S60, February 2008.

[4] *Adolph Ch. et al. (The COMPASS Collaboration):* **COMPASS-II Proposal**
CERN-SPSC-2010-014; SPSC-P-340, May 2010.

[5] *C. Gaspar, M. Dönszelmann, Ph. Charpentier:* **DIM, a Portable, Light Weight Package for Information Publishing,Data Transfer and Inter-process Communication**)
CERN, European Organisation for Nuclear Research, CH 1211 Geneva 23, Switzerland

[6] *M. Bodlak, V. Frolov, V. Jary, S. Hube,r I. Konorov, D. Levit, A. Mann, J. Novy, S. Paul, and M. Virius:* **New data acquisition system for the COMPASS experiment**)
Topical Workshop on Electronic for Particle Physics, September 2012

[7] *Anticic T. et al.:* **(ALICE DAQ Project): ALICE DAQ and ECS User's Guide**)
CERN, EDMS 616039, January 2006

[8] **http://wwwcompass.cern.ch/**
The COMPASS official website

[9] **http://cern.ch/**
European Organization for Nuclear Research

[10] **http://en.wikipedia.org/wiki/CERN**
Wikipedia, the Free Encyclopedia

[11] **http://en.wikipedia.org/wiki/Qt_(software)**
Wikipedia, the Free Encyclopedia

[12] **http://qt-project.org**
The Qt developers site

[13] **http://dim.web.cern.ch/**
The DIM website

[14] **http://db-engines.com/en/ranking**
Knowledge Base of Relational and NoSQL Database Management Systems [June 2014]

[15] **http://www.mysql.com/**
MySQL database management system. [June 2014]

[16] **http://en.wikipedia.org/wiki/Database**
Wikipedia, the Free Encyclopedia [June 2014]

[17] **http://en.wikipedia.org/wiki/c++**
Wikipedia, the Free Encyclopedia

Appendix A

The content of the enclosed CD

- The electronic version of this document
- The source codes for the *Message Browser*
- The source codes for the *Message Logger*