

WARSAW UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering

Ph.D. THESIS

Monika Joanna Jakubowska, M.Sc. Eng.

**Investigation and Prototyping of a New Data Monitoring
Scheme for the ALICE Experiment (CERN)**

CERN-THESIS-2019-386
04/09/2020



Supervisor

Professor Lech Grzesiak, Ph.D., D.Sc.

Co-Supervisor

Łukasz Kamil Graczykowski, Ph.D.

Warsaw 2019

Acknowledgments

I am indebted to my Supervisor, Prof. Lech Grzesiak, for his support and unlimited patience. He is the person who believed in me, gave me the hope and was always ready to help me with everything I needed.

Enormous support I got from my friends Dr. Małgorzata Janik and Dr. Łukasz Graczykowski. Malgosia gave me an idea, the possibility and a lot of support and without her this thesis would not see the daylight. Łukasz became my auxiliary supervisor and he organized me a test necessary to work. I am also grateful for his support and great work he put into this thesis.

I would like to express my gratitude to the whole ALICE Collaboration. Special thanks should go to Barthélémy von Haller, who was helping me with my work from the very beginning, for his tutoring and guidance. I would also like to thank Pierre Vande Vyvre and Vasco Barroso, the ALICE Data Acquisition leaders at the time, for their support and generosity, Roberto Divia for constant express support, Jeremi Niedziela and Adam Węgrzynek, my countrymen, for smart talks and support, Filippo Costa, Sylvain Chapeland and Adriana Telesca for their hospitality. I also thank Julie Hadre and Peggy Pithioud for help in organizing my stay at CERN. I owe additional special thanks to Dr. Yiota Foka for her help during the last stage of this thesis.

I also wish to give my deepest thanks to Dr. Janusz Oleniacz from the Faculty of Civil Engineering, Warsaw University of Technology for help with testing environment, constant support and patience. Thanks to him I did not give up in crisis situations.

I would like to express my gratitude to Dr. Krzysztof Duszczyk for his inspiring teaching and expertise. He was a supervisor of my Engineer and Master of Science theses. He was the one who convinced me to go to doctoral studies. Without him I would never be where I am.

I would also like to thank all my colleagues from Division of Electric Drives for the friendly atmosphere, countless conversations, jokes and going out for a beer together.

Last but certainly not least, I would like to thank my mom, my beloved boyfriend Piotrek and all of my friends for their belief in me and keeping up my spirits. My mother is the person who show me the path in my life.

Abstract

Experiments that are held at the LHC at CERN, the European Organization for Nuclear Research, are known for a huge amount of data which are recorded at a very high rate. Moreover, the experiments are required to maintain a high data taking efficiency while the beam at the Large Hadron Collider (LHC) is present. This requires dedicated tools which enable to monitor online the quality of the recorded data, allowing for rapid interventions in case of any problems. Such actions may be for example the need for re-calibration of certain detectors, exclusion of individual detectors from data taking, and many more.

The goal of this work is to develop a concept of streaming data generated by the ALICE detector in real time to end users. The scope of the project is to investigate and create a prototype (a set of computer programs), demonstrating the advantages of the proposed solution over the existing framework. Therefore, the tasks required to accomplish the goals set for this project are the following: survey of existing computing infrastructure and QA solutions, preparing requirements for extension of the infrastructure for the new solution, implementation of the newly developed framework and preparation of libraries and API for end-users. The proposed solution is based on the use of proxy servers. The ZeroMQ framework was used for data transmission, which allows the use of different environments and programming languages on individual machines. Prototypes use multi-threaded architecture and were written in C / C++. The work includes short performance tests of the resulting solution.

The target environment for the created prototype will be the so-called "Run 3" – the next running period of the LHC, which will start in the first quarter of the year 2021. The test environment is "Run 2", which has been operating until now, which was launched in the emulation mode on machines at the Warsaw University of Technology. The proposed solution can be easily used in a similar physical experiment, as well as in other areas of the industry, wherever real-time data transfer between different platforms is required using various programming languages.

Keywords: *Big Data, ZMQ, CERN, ALICE, data monitoring*

Streszczenie

Opracowanie i badanie prototypu nowego systemu monitorowania danych z eksperymentu ALICE (CERN)

Eksperymenty przeprowadzane w ośrodku naukowo-badawczym CERN charakteryzują się ogromną ilością pozyskiwanych danych oraz niesamowitą prędkością zbierania tych danych. Z uwagi na fakt, że czas realnych eksperymentów jest często krótszy niż czas potrzebny na przygotowanie urządzeń do zbierania danych, potrzebne są narzędzia, które umożliwią sprawne monitorowanie poprawności działania w celu natychmiastowej kalibracji infrastruktury (m.in. czujników).

Celem niniejszej pracy doktorskiej jest opracowanie koncepcji przesyłu danych, generowanych przez detektor ALICE, w czasie rzeczywistym do użytkowników końcowych. Zakresem pracy jest zarówno opracowanie koncepcji jak i wykonanie prototypu (zestawu programów komputerowych), który umożliwi swobodne korzystanie z nowego rozwiązania. W skład wymaganych zadań wchodzi: wykorzystanie istniejącej infrastruktury oraz zaimplementowanych tam rozwiązań, opracowanie planu rozbudowy o dodatkowe maszyny, integracja nowego rozwiązania oraz wykonanie bibliotek dla użytkowników końcowych. Zaproponowane rozwiązanie opiera się na wykorzystaniu serwerów pośredniczących (ang. proxy). Do przesyłu danych wykorzystany został framework ZeroMQ, co umożliwia zastosowanie różnych środowisk oraz języków programowania na poszczególnych maszynach. Prototypy wykorzystują architekturę wielowątkową i zostały napisane w języku C/C++.

Środowiskiem docelowym dla wykonanego prototypu będzie tzw. „Run 3”, czyli następne uruchomienie Wielkiego Zderzacza Hadronów, które rozpocznie się w pierwszym kwartale 2021 roku. Środowiskiem testowym była emulacja systemu akwizycji i procesowania danych działającego w ALICE w trakcie LHC Run 2, która została na potrzeby tej pracy uruchomiona na maszynach znajdujących się na Politechnice Warszawskiej. Zaproponowane rozwiązanie można bez problemu wykorzystać w podobnym eksperymencie fizycznym, jak również w innych obszarach gospodarki, wszędzie tam, gdzie wymagany jest przesył danych w czasie rzeczywistym między różnymi platformami.

Słowa kluczowe: *Big Data, ZMQ, CERN, ALICE, monitorowanie danych*

Contents

1	Introduction	11
1.1	Problem statement	13
2	A Large Ion Collider Experiment	14
2.1	CERN	14
2.2	Accelerator complex	16
2.3	ALICE Experiment	19
2.4	Big data in ALICE experiment	22
2.5	ALICE Data Acquisition Environment	23
2.5.1	DAQ in Run 2	23
2.5.2	ALICE O ²	25
2.6	ALICE monitoring systems	26
2.6.1	Self-monitoring	27
2.6.2	Manual logging system	27
2.6.3	Data Quality Monitoring	28
2.6.4	Data visualization	29
3	Design and prototype of a new monitoring system	31
3.1	Existing monitoring library	33
3.2	Test environment (DATE)	36
3.3	COnfigurable LDC Emulator (COLE)	40
3.4	Used technology	42
3.4.1	ZeroMQ	42
3.4.2	DATE database	47
3.5	Programs	49
3.5.1	Source programs	49
3.5.2	Proxy programs	53
3.5.3	User library	59

CONTENTS

3.5.4	Client program example	63
3.6	Programs configuration	65
4	Results of benchmark tests	69
4.1	Testing tools	69
4.2	Test machines	71
4.3	Overview of results	73
4.4	Tests of programs source code correctness	87
5	Summary and conclusions	90
	Appendices	100
A	Source Program source code	102
B	Proxy Program source code	110
C	Libraries source code	139
D	Client Program source code	162

Chapter 1

Introduction

Gathering a large amount of data (so-called “big data”) has become more and more popular because of the potential that their analysis might improve our lives in almost every respect, from preventing diseases through combating crime to studying the origins of our world. In order to achieve these goals there is still a lot of work ahead of us. Developing more effective methods of transporting and filtering "big data" can have a large impact on future research in the fields of science and industry. This work aims at creating a novel approach in this area and the ALICE experiment with its great amount of raw data (tens of Gbit/s) is a perfect environment to test such innovative ideas.

As a result of this project a monitoring library, for users who want to monitor experimental data directly from the ALICE computers, is created. ALICE, being a big scientific project, requires monitoring (both online and offline) of the quality of recorded data. For that purpose monitoring programs are made. Thanks to them people can see collected data immediately just after they are taken and sometimes before they are saved. If an invalidity in the quality of the data is spotted, adjustments or even stopping of data recording is required. This solution is devoted for users like operation crew or detector experts. For general public the event visualization is made [1].

Because of a large variety of users platforms and programming languages picked by them, all sending is done by using asynchronous, cross-platform messaging library. All programs are written in C and C++ languages and are based on multi threading model. As a final result static and dynamic libraries are created to allow users of different platforms to use it. This solution has to be very flexible: there is a possibility to change the number of proxy machines as well as number of data collector computers or connected clients. Programs are divided into three parts: sending programs, proxy programs and a user library.

The library will comprise a set of programs designed to get data collected by the ALICE detectors, identify its characteristics, decide if anyone wants to obtain a received sample and send it to the appropriate users. All these tasks are made in real-time. The general design is based on the proxy idea which came across as the best solution (Fig. 1.1).

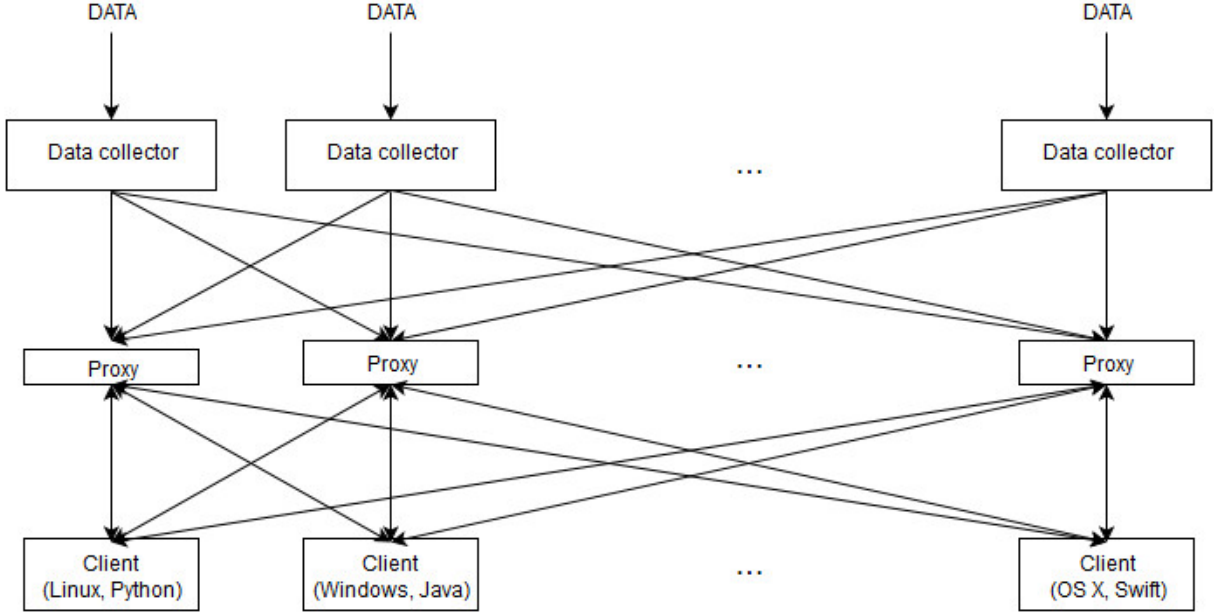


Figure 1.1: Overall idea scheme.

Owing to the flexibility of this idea, it can be successfully used in other similar physics experiments. Moreover, a chunk of code could be applied elsewhere in other fields, for instance in the industry.

Everywhere in the world a lot of machines are collecting data. Sometimes it is very hard to have a remote access to them and in most cases data is saved on some local memory storage machines. By using the monitoring program created as a part of this thesis it is possible to get data from every data collecting machine and deliver it to every client machine regardless of operating systems, architecture or programming languages. It can be used on big Windows-based servers as well as small custom detectors with embedded Linux on-board. One potential example is measuring the properties of electrical network.

This doctoral thesis is made in cooperation with CERN. The presented work has been prepared within the ALICE Collaboration and is an integral part of the ALICE computing framework [2].

1.1 Problem statement

The motivation to create this project was the need to change the currently running monitoring library (see section 3.1) to something that will overcome its limitations. First of all, the architecture of this solution assumes that users connect directly to the machines of the experiment, which can cause the occupation of some of the resources of these machines and, as a result, slowdown the whole experiment.

The most important limitation so far was that the current solution does not support taking very big samples of data. In the existing monitoring package there is only one monitor type option (see section 3.1), which does not have limitations with data size, but using it can be dangerous because in a situation where the client cannot keep up with the throughput of the data flow, the system can even suspend the whole experiment from data taking. This option will be used in the new monitoring scheme, but with extreme caution.

The next limitation is that different threads of one client receive data which can be duplicated. There is a need to specify which connections are from the same client and prevent from sending duplicated data to them. The proposed solution uses a “key” which determines programs that will not receive same data. The existing solution allows to declare a program name, but this variable is used only for debugging or fine tuning.

Another disadvantage can be the fact that using an existing library, the monitoring program can receive data only from one source at a time, there is no option to specify more than one data source. In the case that a user needs data from more than one experiment machines more instances of the program must be run and the collected data have to be compared manually.

The new concept is designed to resolve all these inconveniences and perform at least as good as the current solution exists at the moment. It should also be possible to work in the conditions of the new version of the ALICE experiment (during Run 3 of LHC (see sections 2.3 and 2.5.2)). The solution is expected to add new hardware (proxy servers) and to write a set of program prototypes. The data flow can be obtained from the existing solution. New functions should have their old format counterpart so that existing monitoring programs should still run without any changes required. For benchmarks a test environment should be installed using an emulation of the real Run 2 ALICE experiment environment.

Chapter 2

A Large Ion Collider Experiment

2.1 CERN

CERN is a European research organization founded in 1954. It took its name from the acronym of its first provisional French official title proposed in 1952 “Conseil Européen pour la Recherche Nucléaire” or English “European Council for Nuclear Research”. Nowadays, the official full form is the “European Organization for Nuclear Research”, but the acronym has not been changed [3]. The word “nuclear” currently has only a historical meaning. At the time of inception, physics research concentrated on understanding the internal structure of the atom, but today scientists are getting much further and their work focuses on probing the fundamental structure of the Universe. They try to explain all things that do not coincide with the so-called Standard Model [4] ex. the origin of mass, possible presence of super-symmetric particles [5], dark matter [6] and dark energy [7], as well as understanding of the observed imbalance between matter and antimatter. They also study the state of matter called “quark-gluon plasma” [8–11] which is considered to have existed few moments after the Big Bang and probably is present in the cores of neutron stars [12].

CERN brings together scientists from all over the World. It all started with 12 founding states which signed the convention in 1953. In the next few years more countries were subsequently joining the organization including Poland which joined in 1991. Today CERN has 22 member states, three associate members in the pre-stage to membership and four associate members [13]. There are about 3250 staff members and over 13,000 associated members of the personnel, over 11,500 of which are the users [14].

Since 1957 when the first accelerator the 600 MeV Synchrocyclotron (SC) [15, 16] had been built at CERN, with the increase of scientific knowledge and further needs,

CERN was constructing larger and more complex scientific instruments. The biggest and most famous particle accelerator today is called the Large Hadron Collider (LHC) [17] (see section 2.2). The LHC is placed near Geneva, on the border of two founder states: France and Switzerland. The CERN headquarters are located in Meyrin, north of Geneva, Switzerland. Currently the official organization's logo (shown in fig. 2.1) consists of its name and interlaced rings, which are a simplified representation of the accelerator chain and the particle tracks [18].



Figure 2.1: CERN Official Logo. From Ref. [19].

In addition to delivering high quality physics results, CERN has been very active in development of technology needed for those studies. Many of the developed technologies found applications in other fields like computer science, medicine, and more. One of the most widely appreciated inventions comes from 1989 when the World Wide Web (WWW) was invented by a British scientist Tim Berners-Lee working at the CERN project called ENQUIRE [20]. In the field of medicine CERN did the great achievement with the PET (eng. Positron Emission Tomography) technology. Admittedly, PET was not invented at CERN, but CERN was the first place where a picture of a live organism was made – the scan of a mouse. These findings were presented at the IEEE Nuclear Science Symposium in San Francisco in October 1977 [21]. Currently, CERN is involved in the development of Grid Computing. In 2002 the Worldwide LHC Computing Grid (WLCG) was created (see section 2.4) to share its, constantly increasing, amount of data with computer centres around the world [22].

In the 65 years of CERN's history several people connected with the organization were awarded the Nobel prize. First of these was in 1952 when Felix Bloch and Edward Mills Purcell were awarded for "their development of new methods for nuclear magnetic precision measurements and discoveries in connection therewith". Then in 1976 Burton Richter and Samuel Chao Chung Ting got it for "their pioneering work in the discovery of a heavy elementary particle of a new kind". In 1984 Carlo Rubbia and Simon van der

Meer claimed it for “their decisive contributions to the large project, which led to the discovery of the field particles W and Z, communicators of weak interaction”. Next prize went to Georges Charpak in 1992 for “his invention and development of particle detectors, in particular the multi-wire proportional chamber”. The most recent, but not directly connected with CERN, is the prize of François Englert and Peter W. Higgs awarded in 2013 for “the theoretical discovery of a mechanism that contributes to our understanding of the origin of mass of subatomic particles, and which recently was confirmed through the discovery of the predicted fundamental particle, by the ATLAS and CMS experiments at CERN’s Large Hadron Collider” [23]. Their brilliant idea was proven by CERN scientists.

2.2 Accelerator complex

CERN has the biggest accelerator complex in the world. It consists of several machine stages where every next stage is bringing the particle beam to a higher energy. The LHC [17], being the largest and the most powerful, is the last one in the chain.

LHC is placed approximately 100 m under the ground and is approximately 27 km in circumference. It consists of 8 straight and 8 curved parts. In each arc 154 dipole magnets are bending the beam in order to produce its circular shape. Straight sections have different use – injection, beam dump, collisions (within the experiments) etc. One of them consists of quadrupole magnets that are focusing and electromagnetic resonators that are accelerating the beam. The whole complex is shown in figure 2.2.

In order to achieve the required beam energy superconducting magnets were used. They use niobium-titanium cables that become superconducting below a temperature of 10 K. In that case dipoles need a huge cryogenic system with superfluid helium to keep their temperature in superconducting state. Because helium undergoes a phase transition to the “superfluid” state at about 2.17 K and scientists wanted to produce a magnetic field of 8.33 T even a lower temperature of 1.9 K (-271,3 °C) was needed. This value is lower than the temperature of the outer space (2.7 K, 270.5 °C), so the LHC became one of the coldest place in the Universe [17].

LHC accelerates two beams of particles in opposite ways. Each beam flights in its own vacuum tube. Along the LHC there are 4 points where the beams cross (so-called collision points) located in octants 1, 2, 5 and 8. Experiments are performed using two beams of

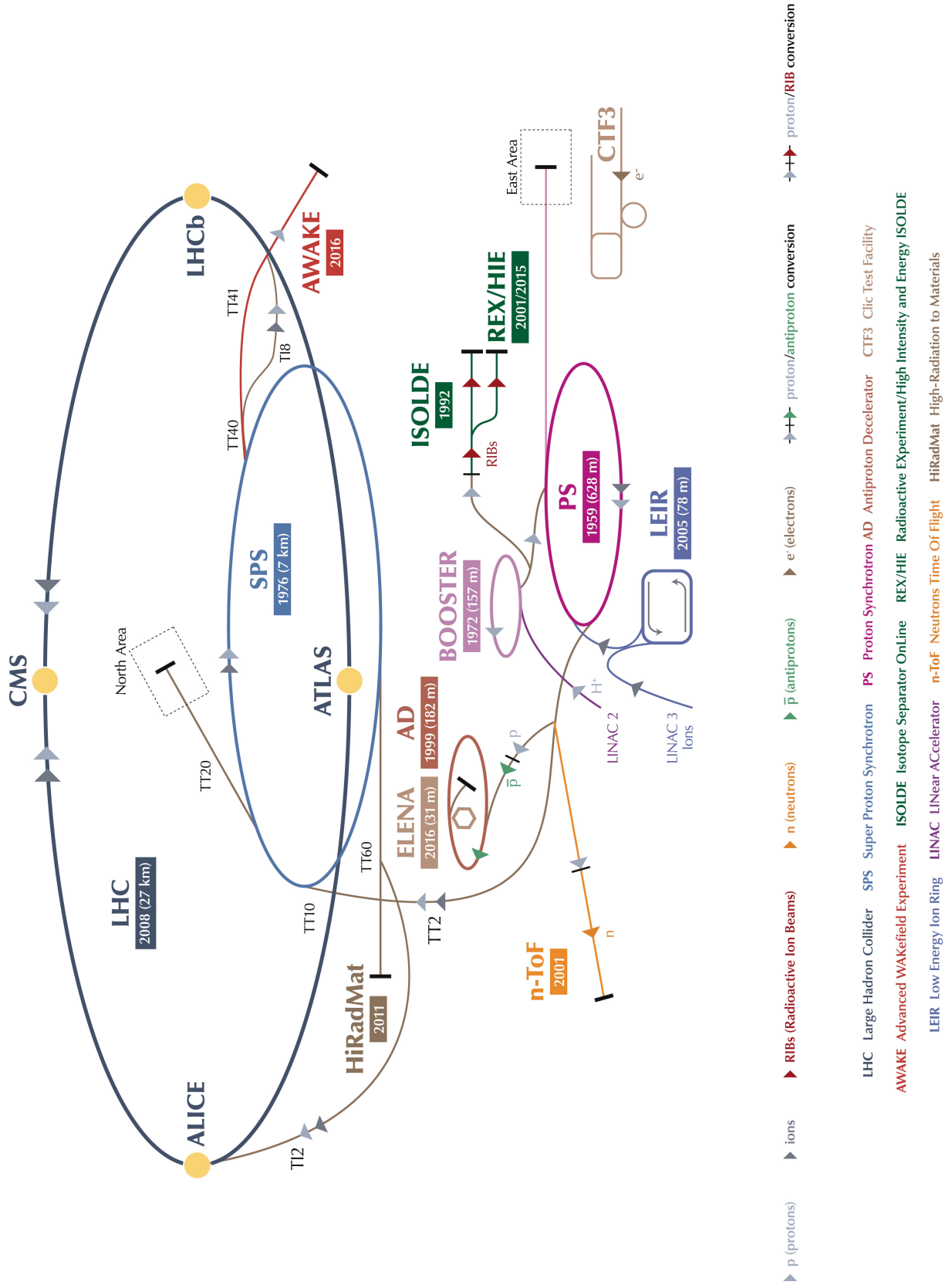


Figure 2.2: The CERN accelerator complex. From Ref. [24]

protons or heavy ions. Both of them are hadrons hence the name of the LHC. Protons are produced by stripping electrons from hydrogen atoms, which are taken from a bottle filled with hydrogen, on the other hand lead ions are produced by heating up a highly purified lead sample.

The way of particles in the accelerator complex is quite long. Their path begins at the point where the particles are obtained and slightly varies depending on their type – for the lead ions is as follows: LINAC 3, LEIR, PS, SPS, LHC, but for protons goes as: LINAC 2, PS Booster, PS, SPS, LHC. The speed that particles reach at the end is always the same and total 99,9999991% speed of light, but the energy varies. Protons achieve energy of 6.5 TeV (collision energy is 13 TeV), whereas heavy ions, which have 82 protons and 125 neutrons, 2.76 TeV/u (energy per nucleon), so the lead-ion beams can have a maximum collision energy of almost 1150 TeV [17].

At CERN there are seven experiments which use detectors to analyze particles produced by collisions in the accelerator. Each of them was created for different purpose and is operated by a collaboration of scientists from all over the world. The four biggest are: ATLAS [25], CMS [26], ALICE [27] and LHCb [28]. They are located in underground caverns on the LHC ring [29]. For further description of ALICE see section 2.3.

2.3 ALICE Experiment

A Large Ion Collider Experiment (ALICE) is one of several CERN projects devoted to research in high energy physics [27]. It is operated by a huge international cooperation which involves over 1500 scientists from more than 150 institutes, all over the World [30]. It is held at so-called Interaction Point 2 (IP2) which is located in Saint-Genis-Pouilly – a small town in Auvergne-Rhône-Alpes region of France, north of the Swiss border.

The main goal of this experiment is to precisely determine properties of the Quark-Gluon Plasma (QGP) [8–11]. Quarks and gluons are the elementary particles that protons and neutrons are made of. It is believed that in a tiny fraction of a second after the Big Bang all matter of the Universe was a huge, extremely dense and hot mixture of, moving with the speed of light, basic bits of matter, dominated by quarks (the fundamental constituents of matter) and gluons (which name comes from the word “glue” as they interact with quarks as a strong nuclear force between them). At that time, when the temperature was around 2000 billion degrees, they were not combined together, but existed separately. After about 10 microseconds they started to bind into bigger molecules. In the ALICE experiment scientists try to recreate conditions similar to those of the very early Universe towards producing the QGP by analyzing high-energy heavy-ion collisions delivered by the LHC. However, as the vast majority of the time at the LHC proton-proton collisions are performed and the main ALICE’s target are heavy-ion collisions, the experiment has been designed to handle both types of data: high rate and small event size “pp” (proton-proton) and relatively low rate (during Run 2 interaction rates ≤ 10 kHz) and big event size nucleus-nucleus. From heavy-ion collisions much more particles are produced than from a proton-proton ones, so it is extremely important to be able to manage huge amount of data which can be produced as their result. The ALICE experiment by now is gathering about 4 GB/s (Pb-Pb running)[31] and there is an estimation to collect so much as 13.2 GB/s after the upgrade [32].

The ALICE experiment operates in certain time slots depending on the generally accepted LHC work long term schedule which is presented in figure 2.3. This scheme is divided into “Runs” and “Long Shutdowns” (LS), which are numbered consecutively beginning from 1, which was the first LHC run started in 2009.

During “Runs” ALICE works 24 hours a day, but the data collection periods are not

that long. One period of continues data taking is also called a "run" (not to be confused with the activities of the LHC for several years) and lasts from several minutes to several hours, followed by a break. These times depend on the daily plan of the ALICE operation as well as on the hardware and software time of proper operation. The group of four people oversee the experiment all day and night. This activity is called a shift. Shift lasts 8 hours so every day people change three times. Additionally during the day a group of so-called “specialists” supervises the operation of the experiment and they are on-call during the night. In the time of “Long Shutdowns” all hardware is improved as well as new ideas are implemented. The long term upgrade program of LHC and four major experiments is planned according to this schedule [33, 34].

During the first and the second runs (2009-2013, 2015-2019) ALICE computing framework was based on two separate modules – online [36] and offline [37].

The online part includes five systems: Trigger System (TRG), Data Acquisition System (DAQ), High-Level Trigger (HLT), Detector Control System (DCS) and Experiment Control System (ECS). The TRG makes a decision, based on triggering detectors, if data are worth being collected or not. The DAQ handles data-flow from the detector up to the storage. The DAQ is the background of this thesis and is described in more detail in section 2.5. The HLT is closely related to DAQ and its main goal is to reduce the volume of physics data by selection and compression. The last two systems are to control the services. DCS is devoted to detectors while ECS is coordinating all the online systems [38]. ECS and DCS are also shown in section 3.2.

The offline part is realized by the ALICE Off-line Project [39]. The core is a framework, called AliRoot [40], designed for simulation, reconstruction and analysis of data collected from detectors which were previously saved on tapes. It uses the ROOT toolkit [41] as a foundation on which the framework and all applications are built. The simulation code is based on the Object Oriented programming paradigm written in C++. It also uses codes like GEANT 3 [42], GEANT 4 [43] or JETSET [44]. For further information please find the project website.

In the next run which is under preparation from December 2018 and is planned to be launched in March 2021, this two-piece model will be replaced by a new Online-Offline computing system which is called ALICE O² (O Square). For more information please see section 2.5.2 and the ALICE Online-Offline Technical Design Report [2].

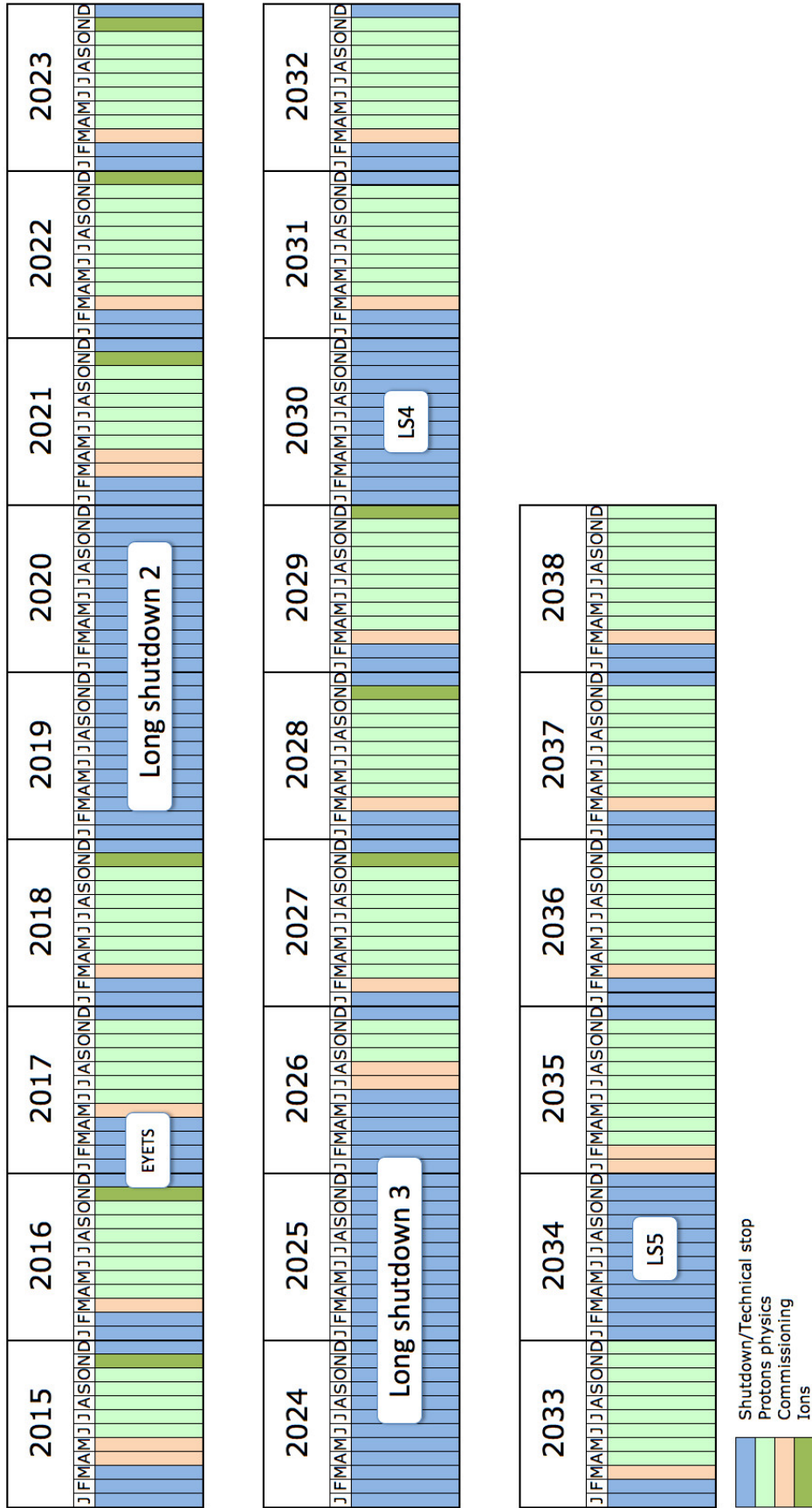


Figure 2.3: LHC long term schedule. From Ref. [35].

2.4 Big data in ALICE experiment

The term “big data” is defined as a data set which is so big and complex that traditional data-processing application software are inadequate to deal with them. Such definition perfectly fits the amount of data which is recorded and analyzed by the LHC experiments.

Data produced by ALICE experiment has a very unique format. On each stage of processing the data format is different, coming from physical equipment as a paged or streamlined event fragments, through full events in experiment machines, as UNIX I/O vectors equipped with additional headers, to the AliRoot format of reconstructed data [36]. On each stage the data size and its amount are so big that needs its own adjusted software to deal with. For more information see section 2.5. For further details about AliRoot, refer to its description in the documentation [45].

Data in the “online” stage of gathering is big, but its amount is quite stable. The “offline” situation is slightly different, because data is growing there in an enormous speed. Additionally, production of this data requires a lot of resources, both in terms of available manpower, equipment, electricity, etc, that every single recorded event is incredibly important. Lost of even a small piece of it is very undesirable. Other LHC experiments are in the same situation. For this reason the WLCG [22, 46, 47] has been created. The base idea of the grid computing was initially proposed in 1999 by Ian Foster and Carl Kesselman and, built on it, WLCG came to life at CERN in 2002. From that moment all recorded data is shared with computer centers around the World. These centers are arranged in “tiers” and giving nearly real-time access to the recorded data to over 10,000 of scientists who are analyzing it every day.

The “online” computer architecture that is planned for Run 3 will change a lot (see sec 2.5.2). Because of the LHC upgrade plans of having higher energies and collision rates, the computing expectations are extremely large. To meet these requirements the idea of using supercomputers, with their huge parallelization ability, was born. Accordingly, all “applications must undergo a transformation to be able to make effective use of it” [48]. Due to specific requirements and high demands on the computing software needed for operating the detector in the LHC Run 3, scientists and engineers started developing a new computing framework already during the LHC Run 2 and before the Long Shutdown 2 (LS2) period.

2.5 ALICE Data Acquisition Environment

The DAQ [36, 38, 49] project in the ALICE is a very complex structure which is designed to handle the full data-flow, starting from collecting data from the detector electronics up to recording on the durable media (tapes).

In order to handle higher data rates during the LHC Run 3 a new Online-Offline (ALICE O²) [2] computing system is under development, which will be significantly different from the system operating during the Run 2 phase of the ALICE operation.

2.5.1 DAQ in Run 2

The full scheme of the DAQ system that was operating during the LHC Run 2 is presented on figure 2.4.

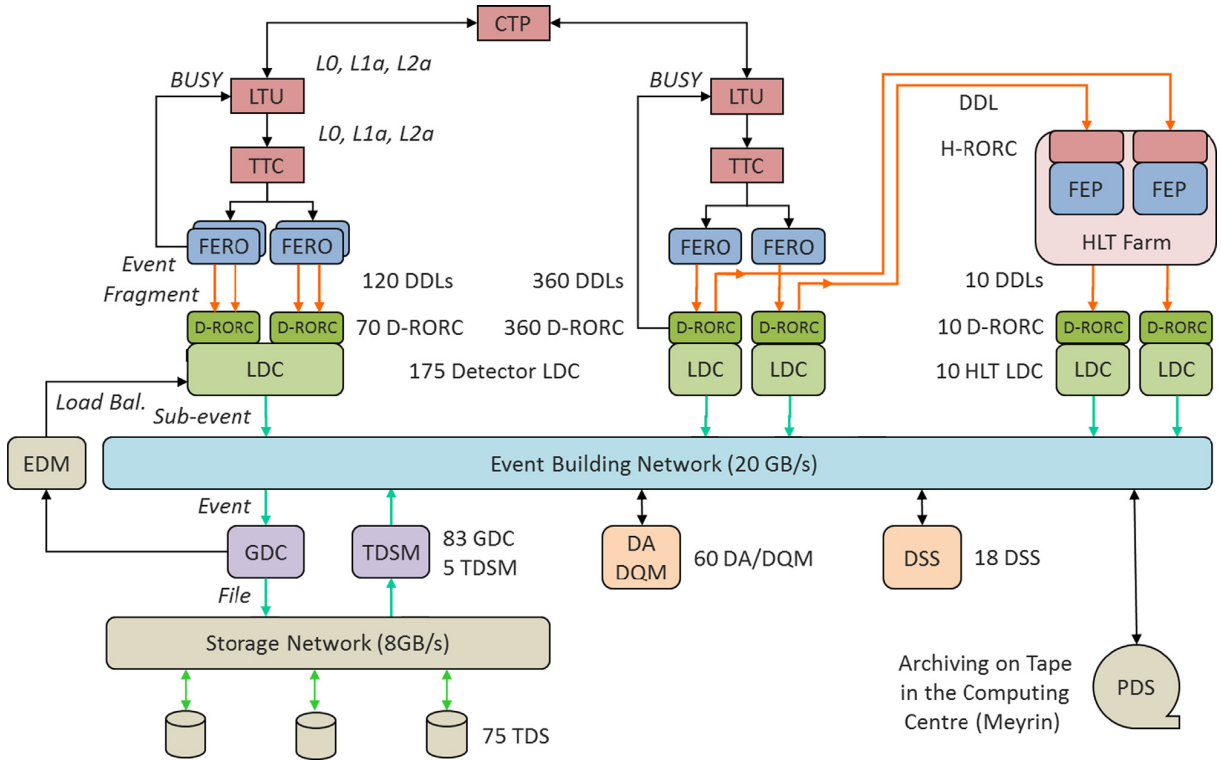


Figure 2.4: ALICE DAQ Architecture for Run 2. From Ref. [38].

Data collection starts when the Central Trigger Processor (CTP) receives the input from the trigger detectors and the LHC clock. Then it decides whether to collect the data resulting from a particular collision and sends a sequence of Trigger Signals (TSs), which includes a time tag [38], through a dedicated Local Trigger Unit (LTU) interfaced

to the Timing, Trigger and Control (TTC) optical broadcast system. This signal comes synchronously to all detectors, composed of Front-End Read-Outs (FEROs), and activates raw physics data (event fragments) extraction.

FEROs are connected to the PCIe or PCI-X boards called DAQ Read-Out Receiver Cards (D-RORCs) using Detector Data Links (DDL). D-RORCs are mounted inside PCs called Local Data Concentrators (LDCs) and perform Direct Memory Access (DMA) transfers of event fragments. In “Run 2” there were about 175 LDCs. Their main task was to perform the first step of the process known as event-building – logical assembling event fragments, coming from the same collision, into sub-events.

Some of the event fragments are copied from D-RORCs to the HLT farm where the first online reconstruction is performed and decisions to accept or reject sub-events are applied. The HLT also reduces the volume of data by compression.

Sub-events from LDCs are shipped to a farm of PCs called Global Data Collectors (GDCs), where the second step of event-building is done - sub-events grouped by the TS are built into a full event. In Run 2 there were about 83 GDC machines. A load-balancing mechanism between GDCs is provided by the Event Destination Manager (EDM) [38].

Coming from GDCs full events are transferred to the Transient Data Storage (TDS) and further via TDS Managers (TDSM) onto Permanent Data Storage (PDS), where they are stored on magnetic tapes. From now on data become available to all ALICE members for the offline analysis. PDS are located in the Computing Centre (CC) while TDS remain at the experimental area.

All data (sub-events as well as full events), commands, messages and statuses are sent by the Event Building Network, which brings together nodes of the system such as Detector LDCs, HLT LDCs, DAQ Services Servers (DSS), TDSM. PDS and servers used for Detector Algorithms (DA) or Data Quality Monitoring (DQM) [36].

Data collected from all LHC experiments is copied by WLCG (see sec. 2.4) to thousands of computers and storage systems in over 170 centers across 41 countries. Because of that it is almost impossible that any of data would be lost. Three of these centers are located in Poland in Warsaw, Cracow and Poznań [46].

The subject of this thesis is focused on the GDC machines introduced above.

2.5.2 ALICE O²

During the LS2 a great deal of hardware equipment will be upgraded. The LHC will significantly increase its luminosity [34]. Therefore, ALICE should be able to inspect all of the interactions in order to fully exploit the scientific potential of the LHC. New conditions are to ensure collisions of Pb-Pb with energy 5.5 TeV and rate of 50 kHz as well as collisions of proton-proton with energy of 14 TeV and rate of 200 kHz [32]. These are much higher values than in Run 2, so that high statistics and high precision measurements are required. To achieve that some of the ALICE parts must undergo an upgrade [32]. These changes will cover the detector hardware such as e.g. Inner Tracking System (ITS) [50], Time Projection Chamber (TPC) [51, 52] or Muon Forward Tracker (MFT) [53, 54] as well as readout, trigger and event reconstruction computing system [55, 56].

As it was mentioned in section 2.3 the concept of ALICE O² [2] is creation of a computing model which will be a combination of existing online and offline systems. These, in turn, are to come with evolution of structures used in Run 1 and Run 2: DAQ, HLT and offline systems. The new model is planned to be used in Run 3 and further in Run 4.

The estimated event size will be around 23 MB, which would result into a global data throughput of approximately 1150 GB/s for the online systems and 460 GB/s to the mass storage [32]. In the existing two-piece solution the HLT replaces some of the raw data with results of the online reconstruction. In the new concept all data will be transferred to the computing system which will take care about everything. The new ALICE Computing Model, presented in figure 2.5, deals with stronger data reduction at the early sections of the data-flow [57]. Data reduction will be done by multiple step reconstruction. The first stage will be performed synchronously parallel with the raw data taking, while the second one, after the local data storage, asynchronously. The quality control points will be in both first and second parts of the data-flow. Both parts will perform also partial calibration and reconstruction. The asynchronous part can use external computing sources to relieve the O² resources.

Data from the detectors should be transferred continuously and the time markers will be used to determine the event pieces. The LHC clock will be used to synchronize and aggregate the data [2].

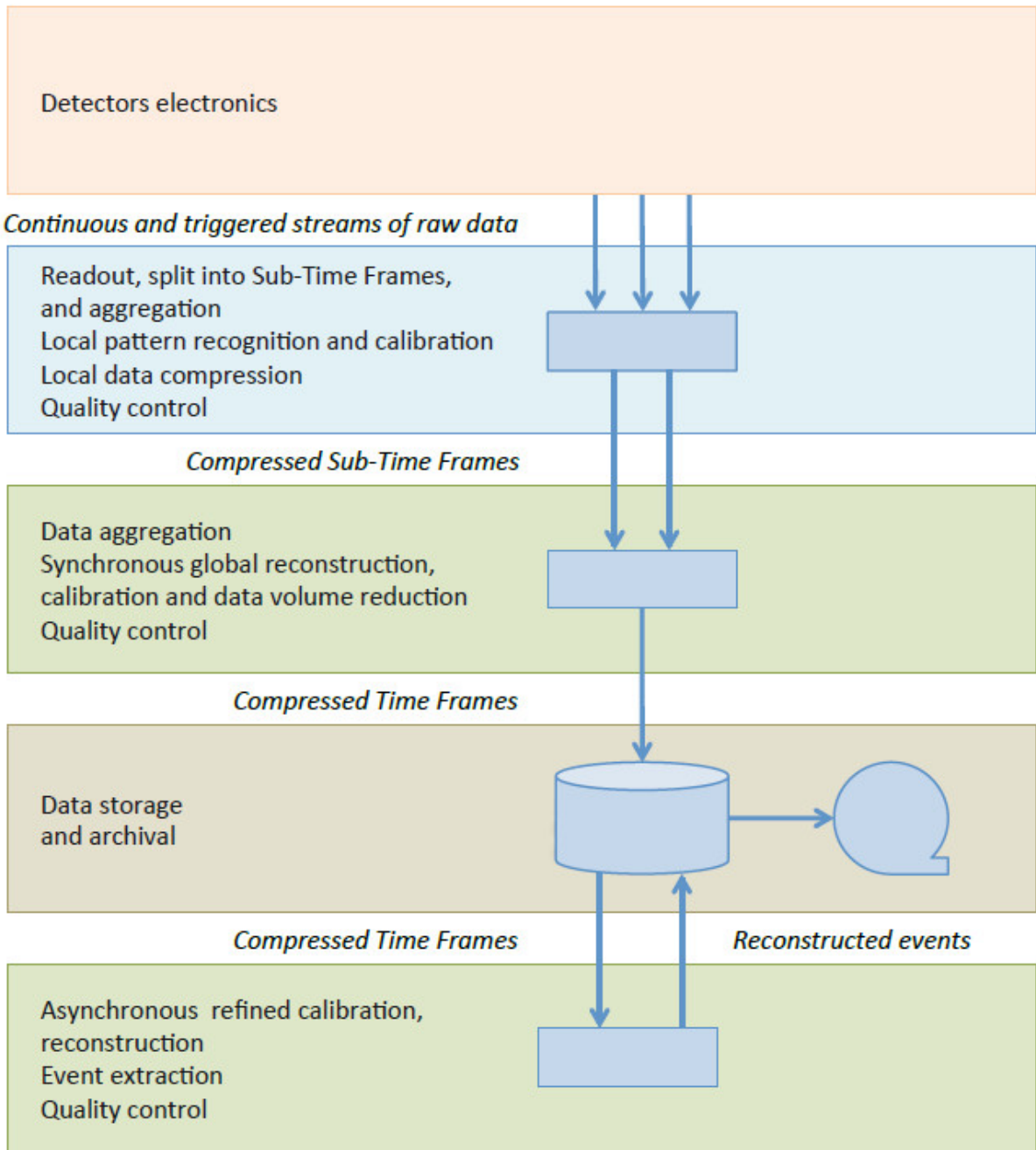


Figure 2.5: Functional flow of the O2 computing system. From Ref. [2]

2.6 ALICE monitoring systems

Huge nuclear physics experiments, as those held at CERN, function on very complex infrastructure. The successful realization their scientific goals depends on their optimization. That is why a continuous 24 hour supervision is needed. Dedicated monitoring systems are required for performing this task efficiently. they require application of methods

of data streaming, networking or data sharing [58].

The monitoring system is useful in many different ways which require different approaches. For this reason it cannot be settled for just one solution, but there is a need to carry out several adjusted implementations.

Monitoring in ALICE is done at various levels, including monitoring of the infrastructure, monitoring of the data flow and subsequent data reconstruction, online data visualization and quality monitoring of raw data.

The core function of DAQ (Data Acquisition) system is to perform the data flow from the active sensors of the detector up to the data storage. It operates on the main stream of data and include some built-in features for data quality and system performance monitoring [38] (see sec. 2.6.3).

2.6.1 Self-monitoring

The ALICE experiment consists of very complex systems which are operated usually by a single person (ex. during the shift). As a consequence a bunch of operations must be held automatically. ALICE DAQ has built-in some self-monitoring tools used for infrastructure monitoring such as Lemon, infoLogger and Orthos.

Lemon (LHC Era Monitoring) [59] is used to monitor around 800 different entities for about 100 different metrics of hardware and software components such as DDL traffic, power values of ambient temperatures. It also can handle some alarms.

infoLogger package [60] is the log system that collects and store log messages from all software DAQ components. It allows to keep trace of runtime execution for later investigation. It is a MySQL-based scheme which store over 500000 messages every day.

Orthos [61] is the alarm system developed to detect and advertise abnormal situations within hundreds of DAQ hardware and software.

2.6.2 Manual logging system

The ALICE electronic logbook [62] is a manual tool designed to read and write records about the experiment. It is directly used by all members of the collaboration. During the shifts every shifter is obligated to upload a report from their shift in order to create an action history and keep track of every single event that took place during the experiment

run. The logs are created also by other stuff as service crew or detector specialists. Every hardware repair or software change must be entered here. The example of a logbook page is shown in Figure 2.6.

Page Browsing

1-20 of 45 (Page 1 of 3)

Fills filters

Local filters

Stable Beams Start: [01/01/2013 00:00:00.]

Runs filters

ECS Start Time: This Year

Beam: Yes

Partition: All Global

Fill Quick Access

Actions

Export...

Statistics

Overview

Fill Info Status	Fill Number	Stable Beams Declared	Filling Scheme Name	Stable Beams Start	Stable Beams End	Stable Beams Duration	Data Taking Duration	Pause Duration	SOR/EOR Duration	Efficiency (%)	# of Runs	Time to First Run
	3564	Yes	50ns_1374s_1278_36_1218_144bpi12inj	14/02/2013 06:37:21	14/02/2013 07:25:27	00:48:06	00:25:42	00:00:01	00:08:25	53.43	2	00:11:49
	3563	Yes	Multi_39b_29_2_6_4bpi13inj	13/02/2013 23:45:21	14/02/2013 04:16:22	04:31:01	04:15:16	00:00:00	00:05:10	94.19	1	00:07:08
	3562	Yes	Multi_39b_29_2_6_4bpi13inj	13/02/2013 18:57:27	13/02/2013 21:09:43	02:12:16	01:59:23	00:00:00	00:05:03	90.26	1	00:06:56
	3560	Yes	50ns_1374s_1278_36_1218_144bpi12inj	13/02/2013 09:01:28	13/02/2013 14:13:58	05:12:30	03:47:06	00:00:00	00:51:22	72.67	12	00:12:13
	3559	Yes	50ns_1374s_1278_36_1218_144bpi12inj	12/02/2013 22:40:14	13/02/2013 06:59:01	08:18:47	07:34:33	00:10:01	00:18:14	91.13	4	00:07:10
	3558	Yes	50ns_1374s_1278_36_1218_144bpi12inj	12/02/2013 14:07:03	12/02/2013 15:16:15	01:09:12	00:51:59	00:00:00	00:09:03	75.12	2	00:06:50
	3557	Yes	50ns_1374s_1278_36_1218_144bpi12inj	12/02/2013 10:32:32	12/02/2013 12:00:42	01:00:40	00:30:22	00:02:04	00:18:35	50.05	4	00:09:08
	3556	Yes	50ns_510b_504_6_414_72bpi11inj	12/02/2013 02:15:31	12/02/2013 07:43:32	05:28:01	04:13:14	00:02:24	00:39:38	77.20	6	00:09:45
	3555	Yes	Multi_130b_46_16_22_36bpi9inj	11/02/2013 22:07:18	11/02/2013 23:54:26	01:47:08	00:37:44	00:00:03	00:13:09	35.22	3	00:39:41
	3544	Yes	200ns_338Pb_338p_15inj24bpi	10/02/2013 03:28:59	10/02/2013 06:05:48	02:36:49	02:02:35	00:07:48	00:14:21	78.17	3	00:08:41
	3543	Yes	200ns_338Pb_338p_15inj24bpi	09/02/2013 18:31:30	10/02/2013 01:06:30	06:35:00	04:47:19	00:12:45	00:46:31	72.74	11	00:07:12
	3542	Yes	200ns_338Pb_338p_15inj24bpi	09/02/2013 10:40:26	09/02/2013 15:03:33	04:23:07	03:25:04	00:13:09	00:29:21	77.94	6	00:07:46
	3541	Yes	200ns_338Pb_338p_15inj24bpi	09/02/2013 02:42:20	09/02/2013 07:47:31	05:05:11	03:39:16	00:20:04	00:34:46	71.85	7	00:06:28
	3540	Yes	200ns_314Pb_272p_14inj24bpi	08/02/2013 16:45:16	08/02/2013 18:50:14	02:04:58	01:03:49	00:01:55	00:27:15	51.07	10	00:06:52

Figure 2.6: ALICE electronic logbook [38].

2.6.3 Data Quality Monitoring

In order to avoid low quality data one needs a feedback on the quality of the data which are going to be recorded. DQM (Data Quality Monitoring) framework [63–65] provides this feedback and help to identify potential issues early.

AMORE (Automatic MONitoring Environment) is a flexible and modular DQM framework used to analyze data samples and visualize it in form of histograms and graphs. It is founded on the ROOT data analysis framework [41]. It uses DATE [66] monitoring library (the raw data stream) to collect data from online part of experiment and AliRoot framework [45] for offline part.

In addition to infrastructure self-monitoring and the AMORE, ALICE also has an online calibration framework which is performing some tasks. It runs a set of programs called DA (Detector Algorithms) [67, 68]. There are two types of DAs: running in dedicated run or in the background.

2.6.4 Data visualization

Data recorded from high energy physics experiments like ALICE is a stream of numbers recorded on magnetic tapes. Data visualization is a very important element, for experts as well as general public, that shows data in human-understandable way and allows to realize how the data looks like. It uses computer graphics technology to visualize it (2D or 3D) [69]. The visualization of a particle collisions may be realized as various histograms and graphs showing various parameters of the collision or as 3D representation of the particle trajectories in space as registered by the detector. In ALICE plots (Fig. 2.7) are generated by the AMORE framework (see sec. 2.6.3) while real tracks (Fig. 2.8) by the EventDisplay framework [1]. EventDisplay is taking raw data and executes online reconstruction of it. ZeroMQ [70, 71] publish-subscribe sockets are used to transfer the data within its internal modules.

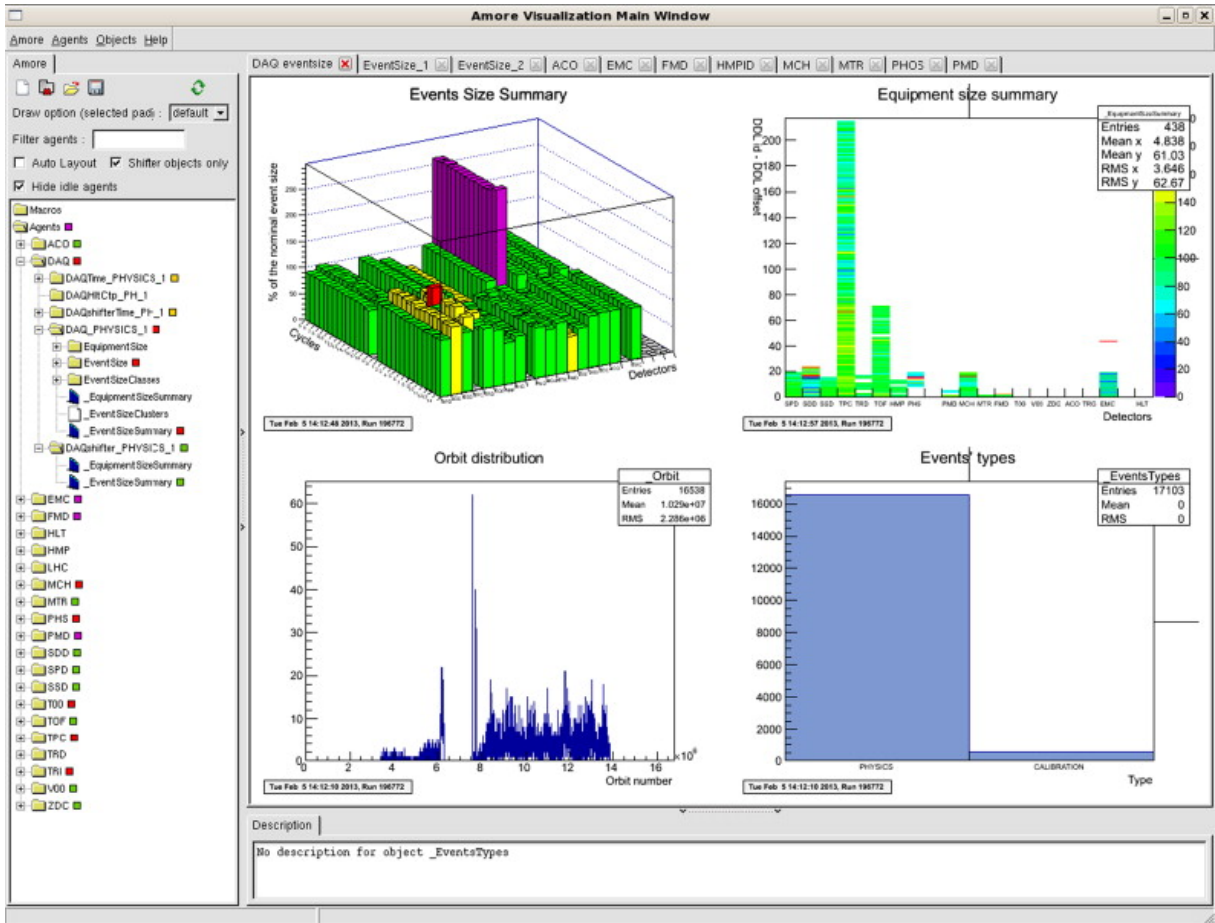


Figure 2.7: AMORE plots visualization [38].

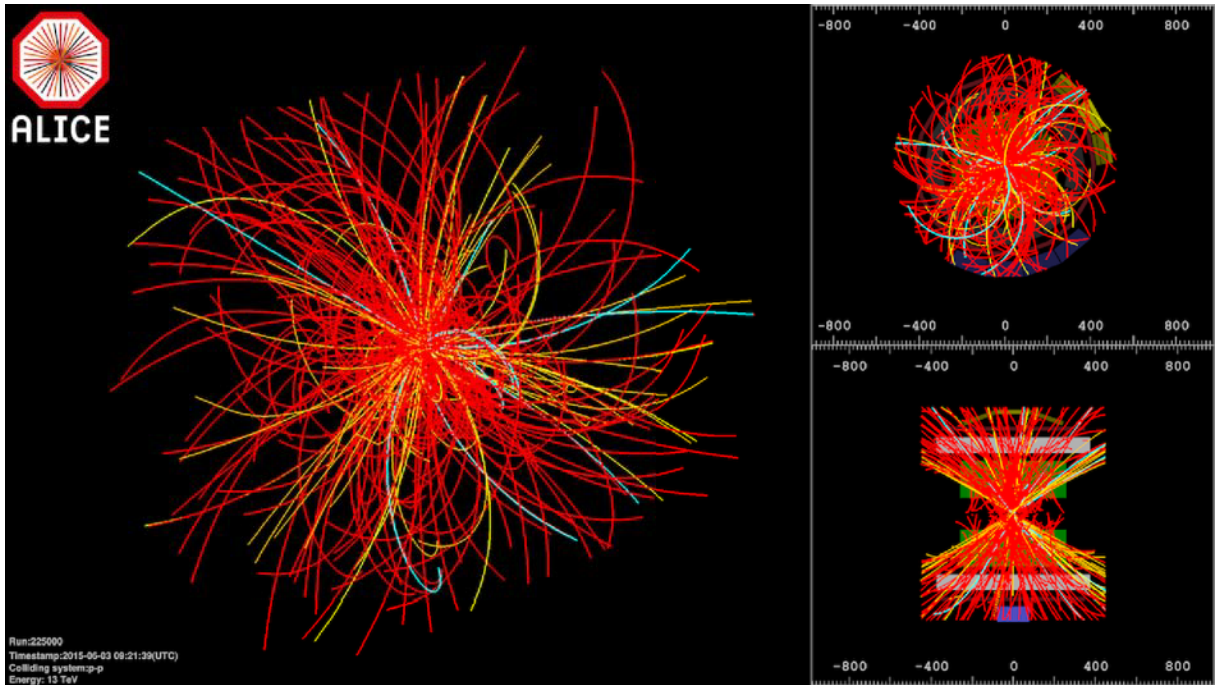


Figure 2.8: EventDisplay collision visualization [1].

Chapter 3

Design and prototype of a new monitoring system

These programs are designed as a prototype for Run 3 of LHC. Due to the fact that some details of the ALICE experiment for Run 3 are still unknown and in the time of creation there was no place to test this solution, all ideas and code are based on Run 2 conditions and should be adjusted later to a new reality. All source code is attached in appendices A, B, C and D.

The main idea is based on using proxy machines between data providers and back-end clients, so that there are three types of machines specified: Source machine (SM), Proxy machine (PM) and Client machine (CM). Every machine which is to be used as a part of a monitoring feature (i.e. SM, PM as well as CM) must have Data Acquisition and Test Environment (DATE) installed (see section 3.2). The whole idea assumes that IP addresses of the SMs are not known, while a list of IP addresses of PMs is placed in the configuration file located in the DATE database. The scheme is shown in figure 3.1.

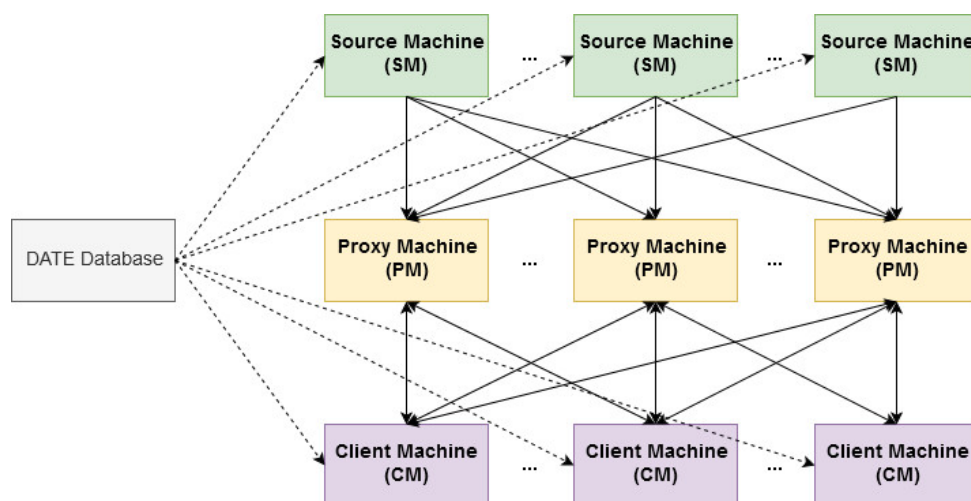


Figure 3.1: Overall hardware design.

On every machine there is a program (one or more instances) which is receiving and/or sending data. They are called Source programs (SPs), Proxy programs (PPs) and Client programs (CPs) for SMs, PMs and CMs respectively. SPs use the existing monitoring library (see section 3.1) as a source of data. The software scheme is presented in figure 3.2.

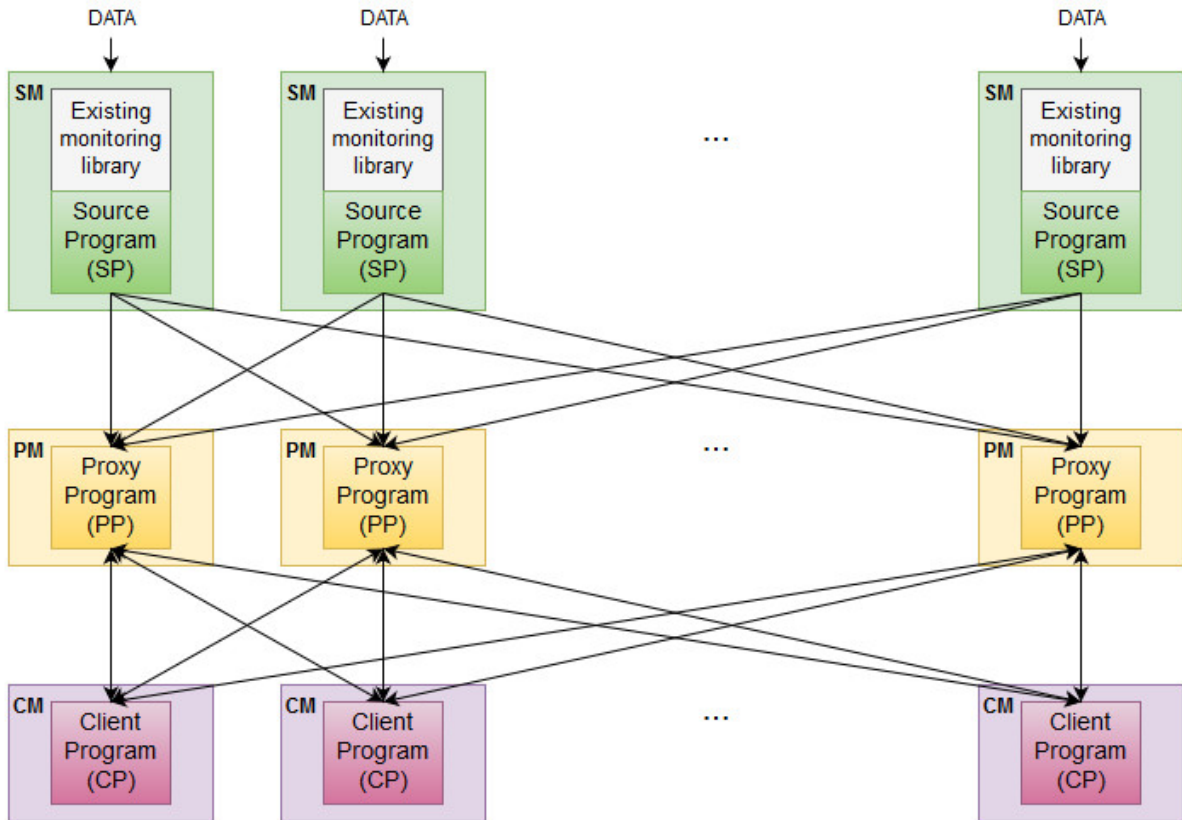


Figure 3.2: Overall software design.

All data from each SP is sent to all PMs. PP gets the data, looks if someone need it and push it there, then disposes. CPs pull all data intended for them. All sending is done by using ZeroMQ messaging library (see section 3.4.1). Distribution of data is divided into three parts: between SP and PP, inside every PP and between PP and CP as further described below.

For Run 2 it is assumed that the role of a SP is played by GDC machines. Events that GDCs produce are called “super events”. Their size is not very big – the largest super event from Pb-Pb collision can have up to 20 MB, but looking at them together with a quite high data rate the data stream is unprecedented. All machines must act quickly in case of doing all well.

The new solution is based on the existing monitoring package, but in the future it can be easily changed to obtain data directly bypassing this library. It allows to take advantage of this software for any other purpose where different data stream is used. However the new solution does not use all functionality of the existing one. It limits only to local monitoring program using an on-line experimental data stream (see section 3.1).

SMs and PMs have Scientific Linux CERN 6 (SLC6) operating system [72] installed (Run 2 conditions) while the system installed on CM is completely discretionary. All programs must have included proxy-based monitoring library. A detailed description of programs operation is provided in section 3.5 which has four subsections. Each of them contains a detailed description of one part.

3.1 Existing monitoring library

Experiments like ALICE require monitoring of experimental data for many purposes ex. statistical analysis to evaluate the quality, detailed analysis or occasional checking of overall status. During the Run 2 the monitoring package (hereinafter Existing monitoring package (EML)) [36] was already used. It exists as a library which can be used by all most commonly programming languages in order to write a monitoring program. It allows to access the live experimental stream (online monitoring) or obtain saved data stored on PDS (offline monitoring). Programs created using this library can be also located on the online or offline/remote host. Existing library has some different functions which helps to tailor own program to specific needs.

Every monitoring program must be composed of some basic parts and should include the DATE event declaration module (event.h) and DATE monitoring declaration module (monitoring.h).

The first step is to declare the source providing the data to monitor. It can be done by using `monitorSetDataSource` entry (listing 3.1.1) from monitoring library.

Listing 3.1.1: C synopsis of declaring data source.

```
int monitorSetDataSource(char* source);
```

It takes one parameter which determines the source of the data collection (more precisely the role name or a TCP/IP host name). As mentioned above it can be a local or

a remote host. Additionally, the source can be specified with the relay host or specific machines in the current run. The syntax is specific. For syntax please refer to the ALICE DAQ and ECS Manual [36].

After declaring the data, the source the program must declare itself to the monitoring scheme. It can be done by using `monitorDeclareMp` entry (listing 3.1.2) from the monitoring library.

Listing 3.1.2: C synopsis of declaring monitoring program.

```
int monitorDeclareMp(char* mpName);
```

It is a very simple function which take one parameter to specify the name of the program. The name can be used for debugging purposes, it does not have any stronger influence on the work of the library.

Next steps are optional and refer to declaring monitor policies as well as wait/nowait policy. Monitor policies can be done by using the `monitorDeclareTable` (listing 3.1.3), `monitroDeclareTableExtended` or `monitorDeclareTableWithAttributes`.

Listing 3.1.3: C synopsis of declaring basic monitor policies.

```
int monitorDeclareTable(char** table);
```

All these functions take a table as an argument. Columns in this table vary for each of the functions. In the basic one there are only two columns which describe event type and monitor type. The default values for these are “all” and “yes” which means that the client wants all types of events as long as there is buffer space. Data from the buffer can be dropped if the higher priority ones appear. When both values are set to “all” the program does not do any filtering as well as does not care about the buffer space. It simply sends all events to the buffer. If the program cannot keep up with the throughput of the data flow the DAQ stops [36]. This is exactly the mentioned “ALL” option which is used in this thesis as a background for a new monitoring program. For the full list of types and their description please refer to the ALICE DAQ and ECS Manual [36].

Before getting an event the behavior of the getting function can be set – the wait option. When it is set to “wait” (represented as TRUE or 1) the program stops on the next step until it receives an event, whereas when “nowait” (represented as FALSE or 0) is set, the program takes empty event and goes on. This option can be done by mon-

itorSetWait, monitorSetNowait, monitorControlWait (listing 3.1.4). The default value is TRUE. When a “nowait” option is chosen the timeout can be specified by using the monitorSetNoWaitNetworkTimeout function.

Listing 3.1.4: C synopsis of setting wait option.

```
int monitorControlWait(int flag);
```

Listing 3.1.5: C synopsis of setting timeout with nowait option.

```
int monitorSetNoWaitNetworkTimeout(int timeout);
```

The last step is to get the available event(s) from the monitoring stream. It can be done by using monitorGetEvent or monitorGetEventDynamic (listing 3.1.6).

Listing 3.1.6: C synopsis of getting event functions.

```
int monitorGetEvent(void *buffer, long size);  
int monitorGetEventDynamic(void **buffer);
```

The difference between them is the buffer type. In the first case function takes two arguments where the first of them is the buffer pointer and the second is the size. In the second case the function takes only one parameter which points to the space reserved from the process heap, so the caller must take care of the memory issues.

This library also includes some other useful entries such as setting timeouts, flushing events or decoding errors, but for more information please refer to the ALICE DAQ and ECS Manual [36].

To finish the work using this library call of the monitorLogout entry (listing 3.1.7) can be used (but it is not mandatory).

Listing 3.1.7: C synopsis of monitoring program logout.

```
int monitorLogout(void);
```

This function takes no argument. After calling it the link is closed and all allocated resources are freed. The link automatically reopens when the monitoring program requests the next event [36].

3.2 Test environment (DATE)

DATE is an ALICE DAQ software framework which was created to perform all data-acquisition activities. It is scalable and can cope with large systems with hundreds of computers as well as on single processor machine. The dataflow is performed as a parallel data streams which are merged and recorded as events [36].

DATE was created for Linux operating systems placed on 32 or 64-bit computer architecture. All active machines must be connected by a network supporting TCP/IP stack and the socket library. The readout program deals with the hardware, which in ALICE currently are DDL and D-RORC (see section 2.5.1), but can be tailored to any type of devices. In the upgrade new types will be supported such as Ethernet [36].

Triggering is performed via the TTC (see section 2.5.1). The readout collects all data and try to identify the original blocks belonging to the same event. Besides the data flow function the DATE environment provides many other features such as i.e. run control, load balancing, event monitoring, electronic logbook, data quality monitoring and many more. For more information please refer to the ALICE DAQ and ECS Manual [36].

The installation and configuration requires several steps to complete. First of them is the installation. All packages are stored in alice-daq repository which is shown in listing 3.2.1.

Listing 3.2.1: alice-daq repository.

```
[main]
[alice-daq]
name=ALICE DAQ software and dependencies - SLC6/64
baseurl=https://yum:daqsofttrpm@alice-daq-yum.web.cern.ch/alice-daq-yum/slc6_64/
enabled=1
protect=1
pgpcheck=0
```

Due to the fact that DATE uses MySQL as a core the MySQL-client and MySQL-devel must be installed. For the DATE server additionally the MySQL-server is also needed. After successful installation of MySQL packages the DATE and ECS must be installed. In SLC6 the yum utility was used. The installation command is shown in listing 3.2.2.

Listing 3.2.2: Installation of DATE.

```
yum install MySQL-client MySQL-devel MySQL-server date ecs
```

After installation some configuration steps are required. For the server machine the database must be created firstly by running a script called “newMySQL.sh” located in the DATE folder. Further steps are the same for the server machine as well as clients. The root must create a new user named alicedaq. After that a new DATE Site must be created. We can use for this purpose the dedicated script called “newDateSite.sh” which asks to enter some data. The only action needed is to press the Enter key several times, except for the DATE user where should be “alicedaq” typed and the creation of a minimal configuration, i.e. 1 detector + 1 LDC + 1 GDC it should be “y” typed. The newly formed folder needs to have owner changed to newly created user alicedaq. Its folder path must be exported to the environmental variable DATE_SITE. The last step is invoking the configuration script named “/date/setup.sh”. After successful installation and configuration of the DATE, we need to do the same for ECS. Same as before the new ECS Site should be installed (newEcsSite.sh script), then the environmental variable ECS_SITE must be set and the ECS setup script needs to be run. At the end the new rule should be added to firewall and a “dateLocalConfig -s” network configuration script should be called. When everything was made successfully the DATE environment is ready to go. All installation commands mentioned before are shown in listing 3.2.3.

Listing 3.2.3: DATE and ECS configuration steps.

```
adduser alicedaq
/date/.commonScripts/newDateSite.sh
chown -R alicedaq.alicedaq /dateSite
export DATE_SITE=/dateSite
. /date/setup.sh
/ecs/newEcsSite.sh
chown -R alicedaq.alicedaq /ecsSite
export ECS_SITE=/ecsSite
/ecs/setup.sh
/date/runControl/start_dim.sh &
/date/infoLogger/infoLoggerServer.sh start
iptables -F; iptables -A INPUT -p tcp -m tcp --dport 22 -j ACCEPT;
dateLocalConfig -s
```

During execution of the scripts the correctness of the partly installation should be confirmed. All scripts are printing information about the procedure, but sometimes problem may not appear in the console. The database creation deserves special attention. After using the “newMySQL.sh” script the user should check manually if the databases and the user ‘daq’ were created successfully.

On the server machine the user `alicedaq` can start an experiment by using a human interface started by a “`DCARUN DAQ_TEST`” command. It starts the Detector Control Agent HMI which is presented in figure 3.3. From this window the Run Control HMI can be opened (see figure 3.4). These two HMIs allow to start the ALICE experiment. The configuration can be changed by using the “`editDb`” (see section 3.6) and “`date-config`” commands.

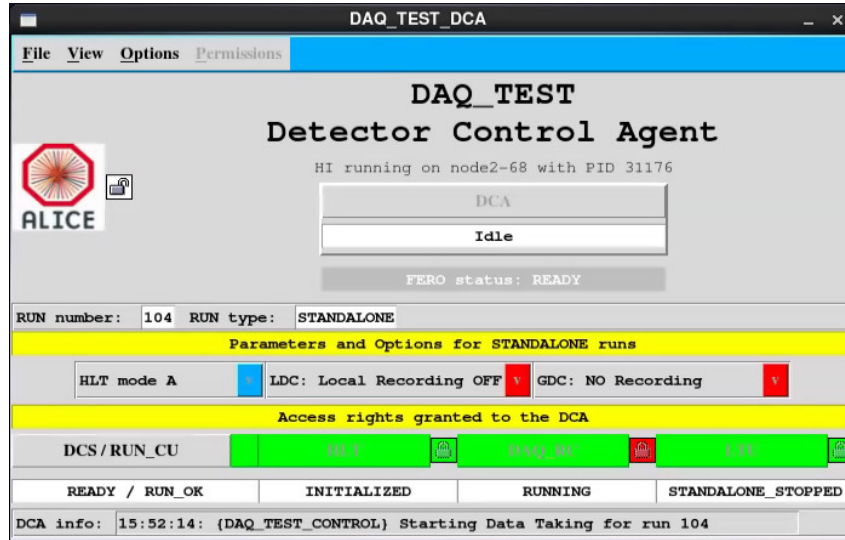


Figure 3.3: Detector Control Agent HMI.

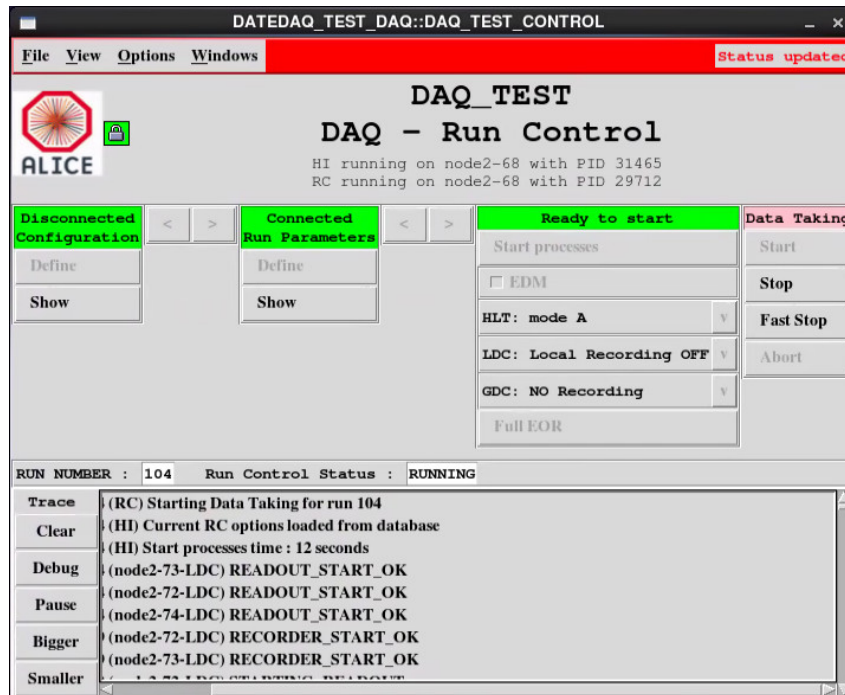


Figure 3.4: Run Control HMI.

3.2. TEST ENVIRONMENT (DATE)

For the user alicedaq the .bashrc script should be supplemented by a DATE configuration which is shown in listing 3.2.4.

Listing 3.2.4: alicedaq .bashrc addition.

```
# User specific aliases and functions
export LD_LIBRARY_PATH=
/opt/date/monitoring/Linux/:/usr/local/lib/:/opt/date/mbp/Linux/:
export DATE_SITE=/dateSite
export DATE_ROOT=/date
source /date/setup.sh
export ECS_ROOT=/ecs
export ECS_ROOT_SITE=/ecsSite
export DUMMY_CTP="YES"
source /ecs/setup.sh
```

All info messages, warnings and errors about the components behavior during the run are collected by the infoLogger package in the MySQL database and can be displayed by the infoBrowser user interface. The HMI is shown in figure 3.5. For full description of this software please refer to ALICE Data Acquisition website [73].

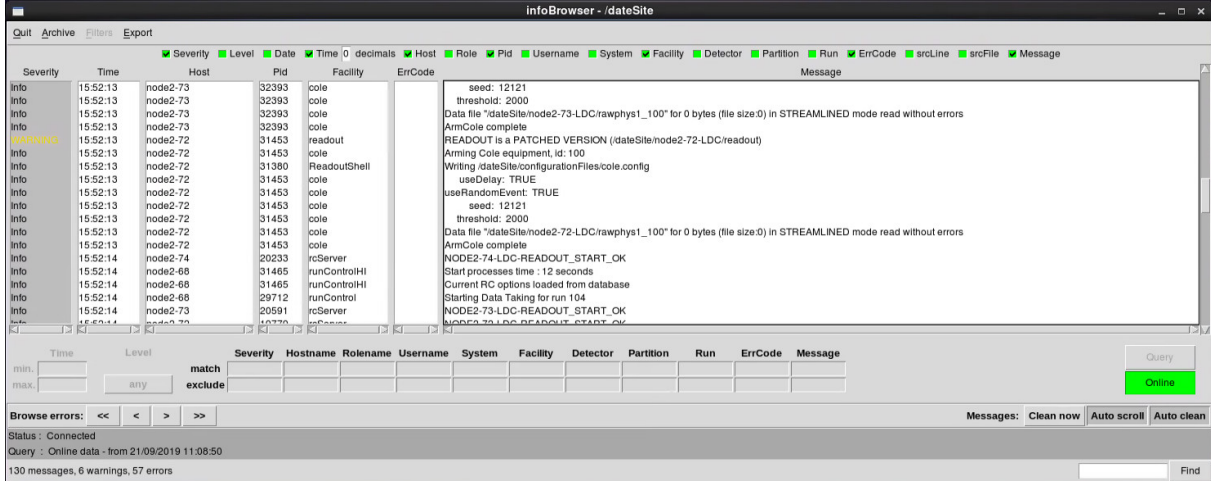


Figure 3.5: infoBrowser HMI.

3.3 COnfigurable LDC Emulator (COLE)

A COnfigurable LDC emulator (COLE) is designed to create ALICE-like events. It is helpful while the environment is not equipped in data producers or a trigger system. In this thesis a COLE was used both for the data production as well as for the triggering system emulation.

The COLE can be used as a standard equipment so the first step to start working with it is to add it to the producer role in the DATE database using “editDb” editor (see section 3.4.2). Therefore the standard equipment named “rand1” and “timer1” should be deactivated and a new COLE equipment should be added. The view of an “Equipment” tab from the “editDb” editor with a new COLE equipment called “cole1” added is shown in figure 3.6. The important parameters here are the “EQUIPMENT_NAME” and the “EqId” number.

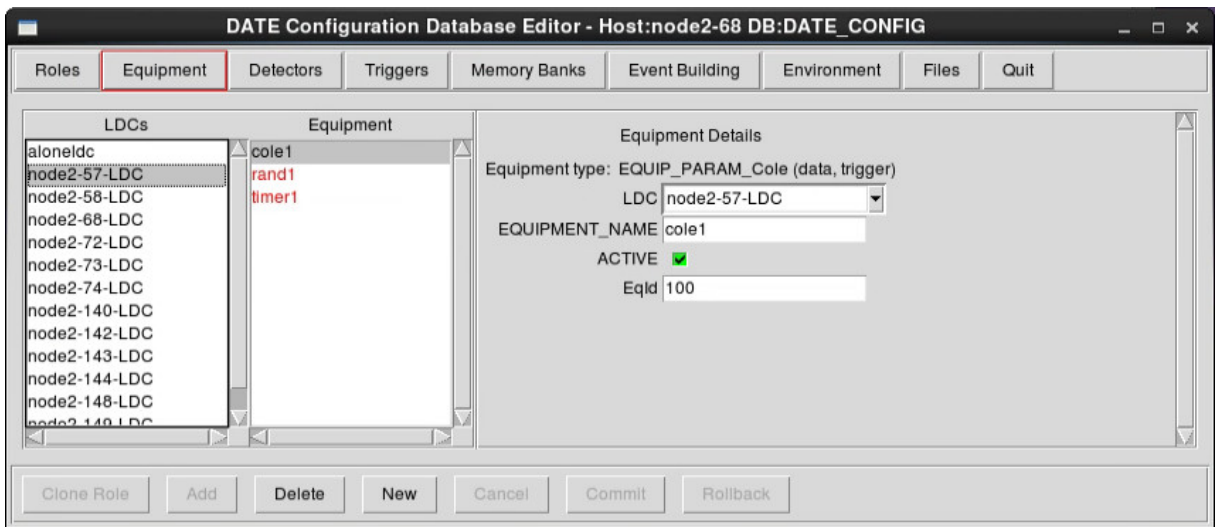


Figure 3.6: editDb “Equipment” tab example.

After COLE equipment was added the configuration file called “equipment.config” should be created. This file should be located in a DATE Site configuration files directory which contains all static parameters used during a run. Therefore, the best practice is to put it inside the DATE database (see section 3.4.2) in the “Files” area. The example of that file is presented in listing 3.3.1. This file contains the detector’s containing LDCs with COLE equipment and its roles. Then all LDCs are listed. For each of them the ID, COLE “EQUIPMENT_NAME” and the “EqId” number are provided.

Listing 3.3.1: "equipment.config" file example.

```
>EQTYPES
>Cole 19 GENDATA TRIGGER
EqId %hd
>LDCS
>node2-144-LDC
node2-144-LDC 9
+ Cole cole1 100
>node2-143-LDC
node2-143-LDC 10
+ Cole cole1 100
```

COLE works based on two configuration files added to the environment: a COLE configuration file and an event payload file. First of them must be named "cole.config" and located in the DATE database (see section 3.4.2) in "Files" area. The COLE configuration file contains three sections: options, events and detectors. Inside the first one some global options can be placed (for more information please refer to the ALICE DATE and ECS manual [36]). In the events section the stream of events created by the DATE system is specified. Each line corresponds to one type of event produced. These lines can be multiplied so the ratio of various event types can be adjusted. Last sections contains detector-specific parameters. One of them specifies the name of a payload file. The example of the COLE configuration file is presented in listing 3.3.2.

Listing 3.3.2: "cole.config" file example.

```
>OPTIONS
useDelay      0
useRandomEvent 0
threshold     500
>EVENTS
DAQ_TEST  CAL  *  *  rawcal1  0
DAQ_TEST  PHY  *  *  rawphys1  0
DAQ_TEST  PHY  *  *  rawphys1  0
DAQ_TEST  PHY  *  *  rawphys1  0
>DETECTORS
DAQ_TEST  0
```

The second file should be placed in the \$DATE_SITE/\$HOSTNAME folder for every role. It contains the raw data for a specific event type – it can be any string of characters. The example of the payload file is shown in listing 3.3.3.

Listing 3.3.3: Payload file example.

```
0000000000
```

There can be multiple payload files associated to multiple events [36]]. The names of that files must be composed of the name placed in COLE configuration file, an underscore and a COLE “Eqid” (for example rawphys1_100).

The last but not least important step to bring the COLE emulator to live is to change the standard readout program on every producer role which has to run a COLE emulator to the COLE readout program. To do that the readout file from a COLE directory must be copied (and replaced) to the role directory inside the DATE Site folder. The command showing the example how to do this is presented in listing 3.3.4.

Listing 3.3.4: Change readout file example.

```
cp /date/cole/Linus/readout /dateSite/node2-144-LDC/readout
```

3.4 Used technology

3.4.1 ZeroMQ

ZeroMQ [70, 74](known also as 0MQ or zmq) is a messaging library, which allows asynchronous communication of some devices through the network protocol. The software which uses zmq can be written in any language and can be launched on any modern platform. ZeroMQ has a socket style API. Communication created by this library is very fast and reliable. Because of its reliability ZeroMQ is used by some huge organizations eg. Cisco, NASA, Microsoft and CERN. In ALICE zmq is currently used, for example, by the EventDisplay software [1] but in this case data is retrieved using EventManager which is a part of the AliRoot framework (see section 2.3).

ZeroMQ allows to choose from a variety of socket types which can be used in their patterns way. The most basic that can be used by the application developer are REQ-REP, PUB-SUB and PUSH-PULL which are described below. There are also some other basic types like PAIR, ROUTER-DEALER, XREP-XREQ, XPUB-SXSUB, but for more information please refer to the zmq website [70].

a) **REQ-REP** (Request – Reply)

This is a classic bi-directional pattern. It acts like a server-client communication – it

contains Requesters (Clients) and a Replier (Server). The server must bind socket for clients connections. The client must create a socket, providing the server's IP address, and connect to it. The server replies to the clients requests. The REQ-REP socket pair is in a lockstep. Both sides use 'send' and 'receive' functions in a loop. The server can reply only once per getting data from client. The client cannot send another request to the server until it gets a response from it. The scheme of REQ-REP pattern is shown in figure 3.7.

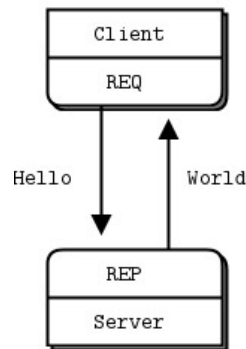


Figure 3.7: REQ-REP ZeroMQ pattern. From Ref. [71].

b) PUB-SUB (Publish – Subscribe)

This is a model of a simplex communication. The server plays a role of a publisher and clients act as subscribers. After the correct subscription is done the server pushes the same data to all subscribed clients. It works like a broadcast. Communication is asynchronous and every Subscriber can subscribe to more than one Publisher to get data. The client will get data until it unsubscribes. The scheme of PUB-SUB pattern is shown in figure 3.8.

c) PUSH-PULL

This is a very useful one-way data distribution pattern. It allows to send data from a pusher to a puller. It can be used to create more complex schemes. A simple Push-Pull scheme is shown in figure 3.9. A pusher can push data to more than one puller, but it always pushes every single message to only one puller. Pullers can pull data from more than one pusher.

The important trait about this type of sockets is that “when a ZMQ_PUSH socket enters an exceptional state due to having reached the high water mark for all downstream nodes, or if there are no downstream nodes at all, then any `zmq_send(3)` operations on

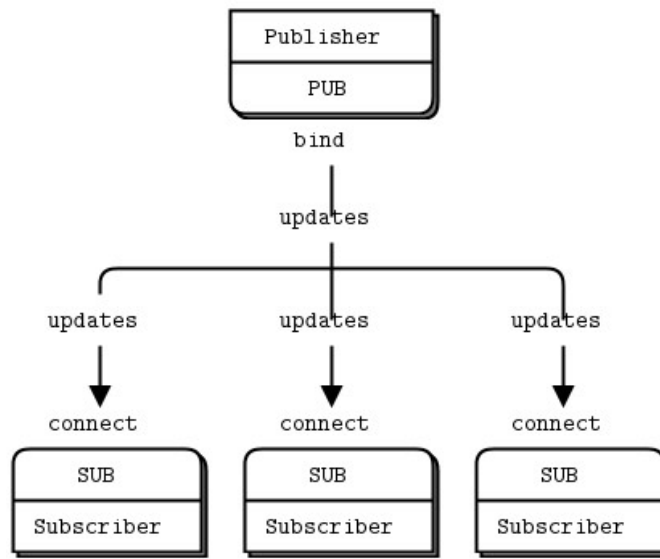


Figure 3.8: PUB-SUB ZeroMQ pattern. From Ref. [71].

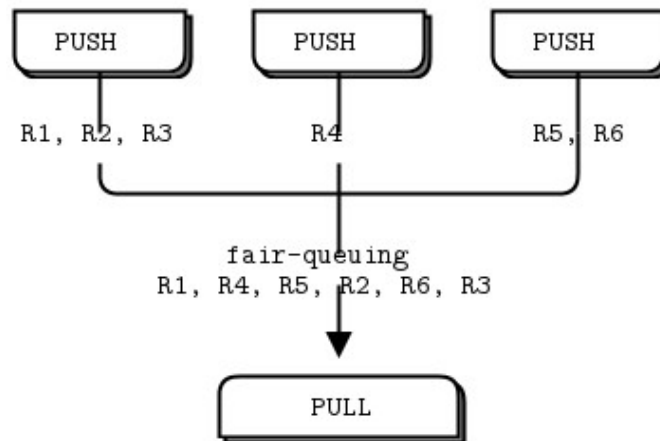


Figure 3.9: PUSH-PULL ZeroMQ pattern. From Ref. [71].

the socket shall block until the exceptional state ends or at least one downstream node becomes available for sending; messages are not discarded” [71].

d) PAIR-PAIR

This pattern is an exclusive pair communication. It is used to connect exclusively two threads using the in-process transport protocol. It should not be confused with the normal socket pair. The communication is bidirectional, but only one peer can be connected to the socket [71]. PAIR-PAIR scheme is shown in figure 3.10.

If the connection reaches a High Water Mark (HWM) the send operations on the sock-

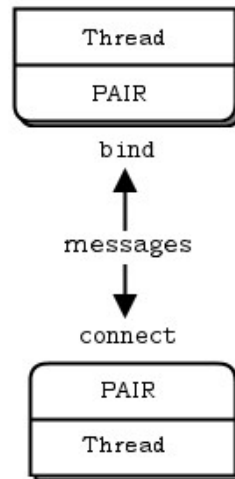


Figure 3.10: PAIR ZeroMQ pattern.

ets will block until the message can be send again. No messages are discarded. The advantage of this type of socket is the simplicity, but the disadvantage is no auto-reconnection functionality. This type of sockets are considered experimental, therefore, some functionality can be missing or broken [75].

Complex schemes

Some basic patterns can be combined to create a complex schemes. An example of a complex communication is shown in the figure 3.11. There a PUSH-PULL pattern is used to create a ventilator-workers-sink scheme. It acts like a supercomputer. The ventilator is dividing data into parts and then is pushing it to a set of workers in as fast as workers can handle data. This is called load balancing. Workers do some job and push data to other devices [71, 74].

In this thesis a mix of that patterns is used. Firstly, the Ventilator pattern is implemented (on every PM) then PMs act like Workers receiving parts of data from Ventilator and then do some job. At the end, Workers push data to some clients. The REQ-REP pattern is used to handle the heartbeat threads.

Bind vs Connect

Every socket, regardless of the type, can be bound or connected depending on current needs. Between connect (`zmq_connect`) and bind (`zmq_bind`) functions there are

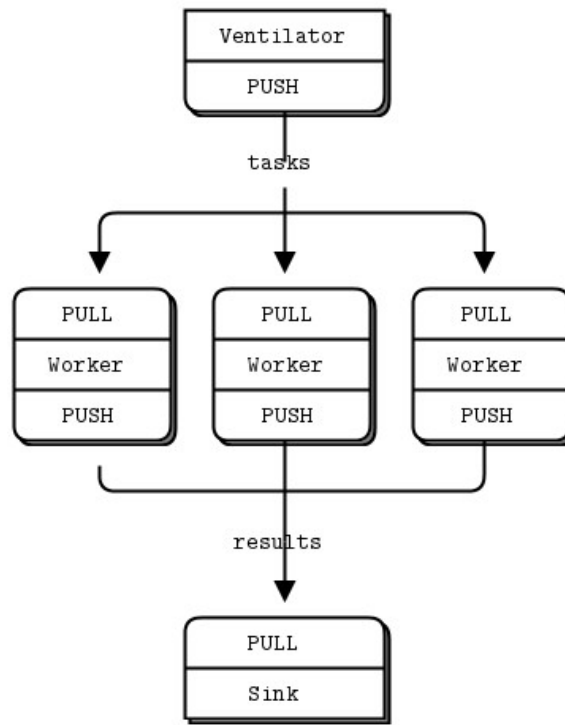


Figure 3.11: Complex Ventilator-Workers-Sink ZeroMQ pattern. From Ref. [71].

two small differences. First of all, binding address must be local, while address to which we can connect can be remote. Therefore, binding should be chosen when the machine's address is not known to others. The second difference is that during binding no queue for packets is created because there is no information how many clients will connect to it, while at the moment of connection the queue is created (one queue per every connection).

Queue (Buffers)

Queue is defined as a pair of buffers – one on the sender side and one on the receiver side. The default size of the buffers is unlimited. This means that the buffer can absorb the whole memory from the machine. The size of the buffer may be adjusted by setting `ZMQ_HWM` (HWM) option by `zmq_setsockopt` function. The value of 0 is default (for `zmq` version up to 3.0.2) and means unlimited buffer size, other value determines the number of outstanding messages in the buffer. ZeroMQ version 3.1.0 and above have the default value of 1000 set. There are two types of `ZMQ_HWM` option: `ZMQ_RCVHWM` and `ZMQ_SNDHWM` which are for receiving buffer and sending buffer, respectively.

Send vs Receive

In the sending (`zmq_msg_send`) and receiving (`zmq_msg_rcv`) functions there is an option available called `ZMQ_NOBLOCK`. In each case it works in a different way. On `zmq_msg_send` method it returns an error (-1 value) when the buffer is full, while on the `zmq_msg_rcv` method this option returns -1 when the buffer is empty. Therefore, the first solution was used in this thesis because it is perfect to prevent from blocking the whole experiment if the client is too slow. The second one can be used on the client side in case that it can do another thing if there is no message to receive.

3.4.2 DATE database

The DATE system is defined by a few information which can be divided into two groups: static and dynamic. Static information is mostly hardware related and is stored in the MySQL database while dynamic is a run-specific configuration which is accessible via the runControl Human Interface [36].

Static data can be edited with a dedicated graphical user interface database editor named “editDb”. It has many “databases” (for more information please refer to ALICE DAQ and ECS Manual [36]), but the most important in the scope of this thesis is the “Files” tab, where a list of files (ASCII or binary) containing other centrally stored parameters that are used in DATE [49] can be seen. The editDb “Files” tab example is presented in figure 3.12.

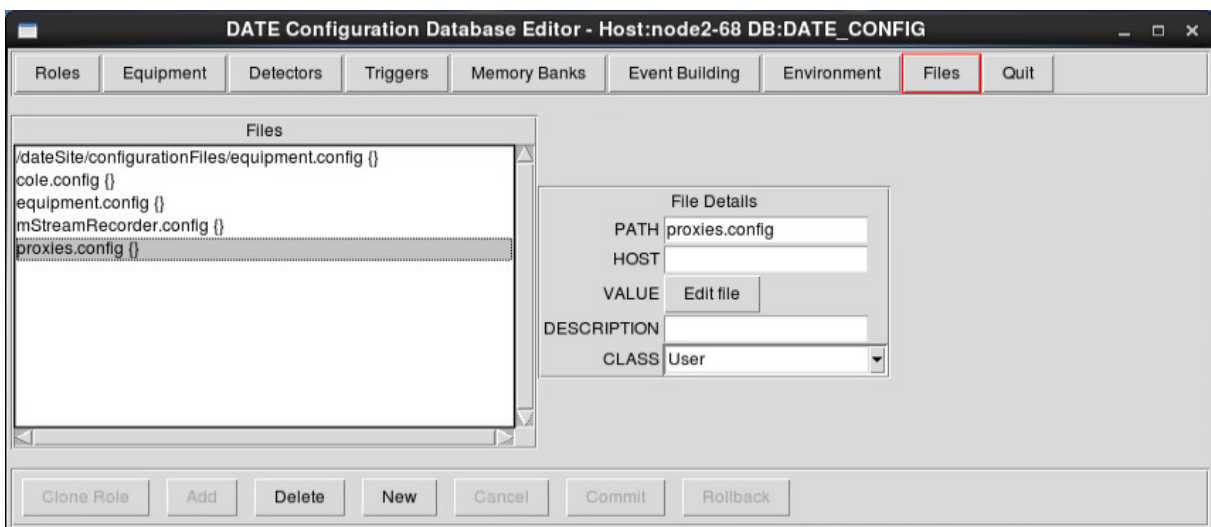


Figure 3.12: editDb “Files” tab example.

Each file has five parameters which identifies it: PATH, HOST, VALUE, DESCRIPTION and CLASS. An unique key is only the PATH-HOST. These files are stored in the DATE site folder.

There is a dedicated API which offers an easy way to access values from these files. To start working with this database a header file called “dateDbFile.h” needs to be included and a “date-config -libs” line added to Makefile rules. To access a desired value it is necessary to do some steps: first open the DATE database, then open a specific configuration file, get the desired value (there is also a possibility to go to the section) and then close the file and the database. The method which gets a value from the database allocates the memory which should be freed manually after use by a caller. An example of a method that uses the file from the DATE database can be found in proxy-based monitoring library which is included in appendix C (e.g. “getPort” method). The example is shown in listing 3.4.1.

Listing 3.4.1: Example of using DATE database API.

```
int error = dbOpen();
if(error != -1)
{
    dbFile db = NULL;
    error = dbFile_open(PROX_CONF_FILE, "", &db);
    if(error == 0 && db != NULL)
    {
        error = dbFile_getNextSection(&db, "Proxies");
        if(error == 0)
        {
            char** pr = NULL;
            error = dbFile_getVal_charArray(&db, "ProxyList", &pr, &amount);
        }
        dbFile_close(&db);
        db = NULL;
    }
    dbClose();
}
```

For the purpose of this thesis inside the DATE database one configuration file called “proxies.config” was created. It is used by the new monitoring package. It contains a list of IP addresses of all PMs (“ProxyList” value) as well as a set of other configurable data like numbers of used ports (i.e. a port for getting data from PM, PP client communication port or a first port for clients connections). For more detailed description please see section 3.6. The access to this file is done by the dedicated API mentioned above.

3.5 Programs

The description of the full idea of the proxy-based monitoring program is split into four separate parts related to the place of the code implementation. The first of them is devoted to the SP machines. In section 3.5.1 the schemes as well as parts of code and its explanation are presented. The second one (section 3.5.2) shows the complete PP idea, scheme, code pieces and its interpretation. The third, featured in section 3.5.3, introduces a user library code and the specification of the included functions usage. Finally, the section 3.5.4 contains an example how to write CP using proxy-based monitoring library.

3.5.1 Source programs

The main goal of every single data sending program called SP is to divide the received data to all PPs as fast as they can handle it. For this purpose the PUSH-PULL type of ZMQ sending pattern, which is further described in section 3.4.1, is used. The scheme of SP data thread is presented in figure 3.13.

SPs are programs located on the machines from which data can be obtained (e.g. LDSs/GDCs (for Run 2), FLPs/EPNs (for Run 3), files). In this prototype SP are placed on GDCs and uses the EML as a source of events. There must be one instance of SP started for every running “role” on each machine. The program can be executed with one argument that specifies the role. The same syntax is used as in EML (see section 3.1). It just takes the first argument from the execution of the program and puts it into the EML function. If no argument is passed then the default value of “:” is set. The default option means that the SP will attach to the first on-line local “role” from the localhost, so if there is more than one role on the machine it should be avoided to leave the default value. Due to this solution the SP can be used as well on a remote machine or can take data from the off-line file. It is just needed to specify a “role” according to the EML parameter syntax. For more information about the syntax please refer to the ALICE DATE and ECS Manual [36].

As the proxy-based monitoring should not have disadvantages of the existing one and have an option to block the experiment in case of getting every single event it produces, SP uses the option “All/All” from the EML, which is the only option that does not have

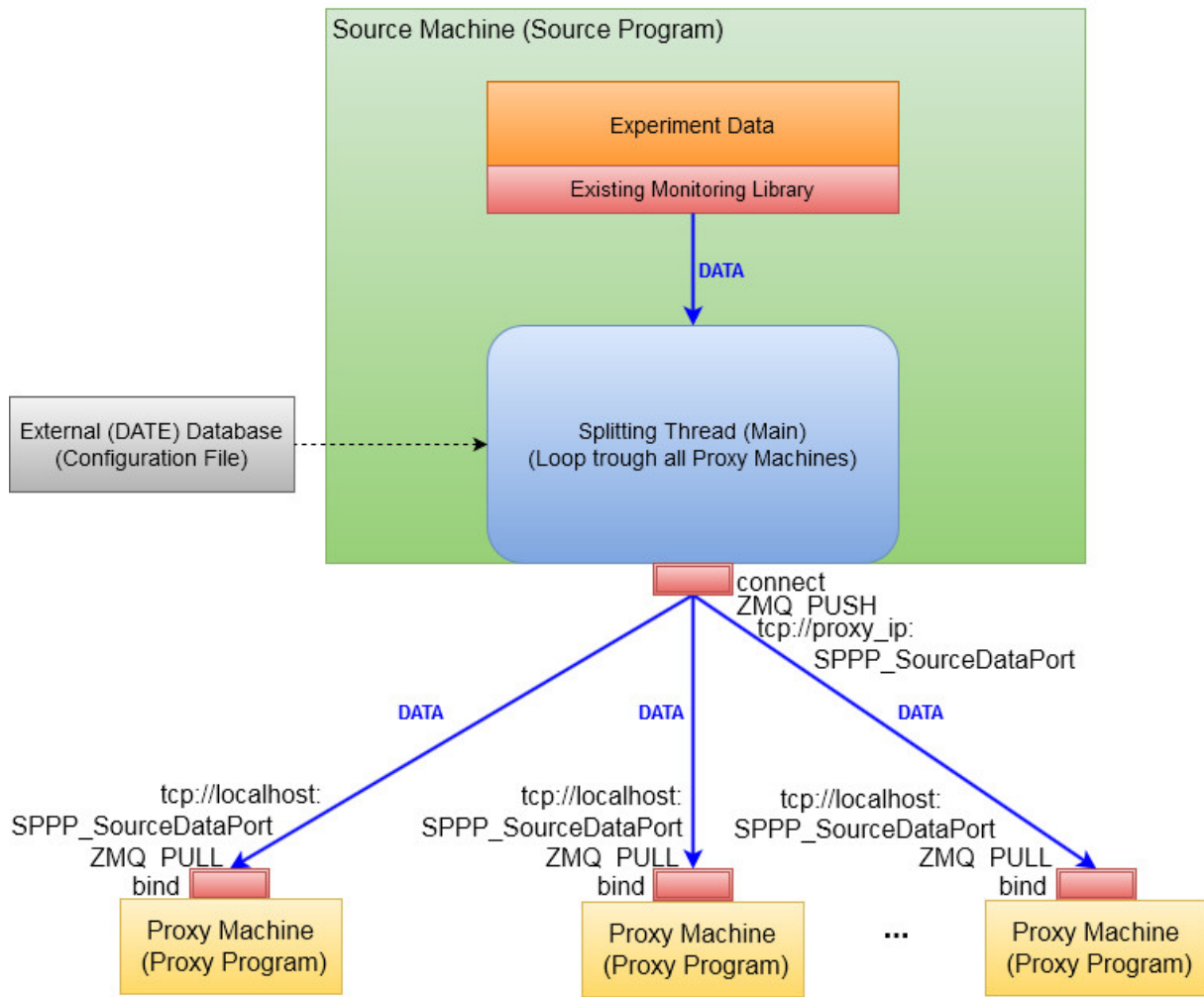


Figure 3.13: The SP data thread scheme.

an event size limit and stops DAQ if program cannot keep up with the throughput of data flow (see section 3.1). This option is used with an extreme care, but it also allows the “Must” option to be executed (see section 3.5.3). A table describing the used monitoring policy is shown in listing 3.5.1.

Listing 3.5.1: Policy table setting.

```
char *table[] = {
    "All events", "ALL",
    NULL
};
status = monitorDeclareTable(table);
```

The assumption that the IP addresses of SMs are not known makes that at least the IP addresses of the PMs must be available somehow. As mentioned in section 3.4.2 a full list is taken by the variable “ProxyList” from the configuration file called “proxies.config”

located in the DATE database. Because of the fact that the port number, on which PM binds data receiving socket, can be obtained from the configuration file and is the same for every PM, there can be only one instance of PP on each PM. On the other hand there can be more than one instance of SP on one SM, which is exactly what is needed. If there is no port specified in the configuration file, the default value is set (see section 3.6). The port must be different (unique) than all other ports used by PMs.

Sending data is preformed by a many-to-many relationship. Every SP has only one socket which has multiple connections to the sockets bound on PMs. Every message is sent only to one PP. There is no duplicates. A separate buffer for every connection (as it is shown in figure 3.14) is created.

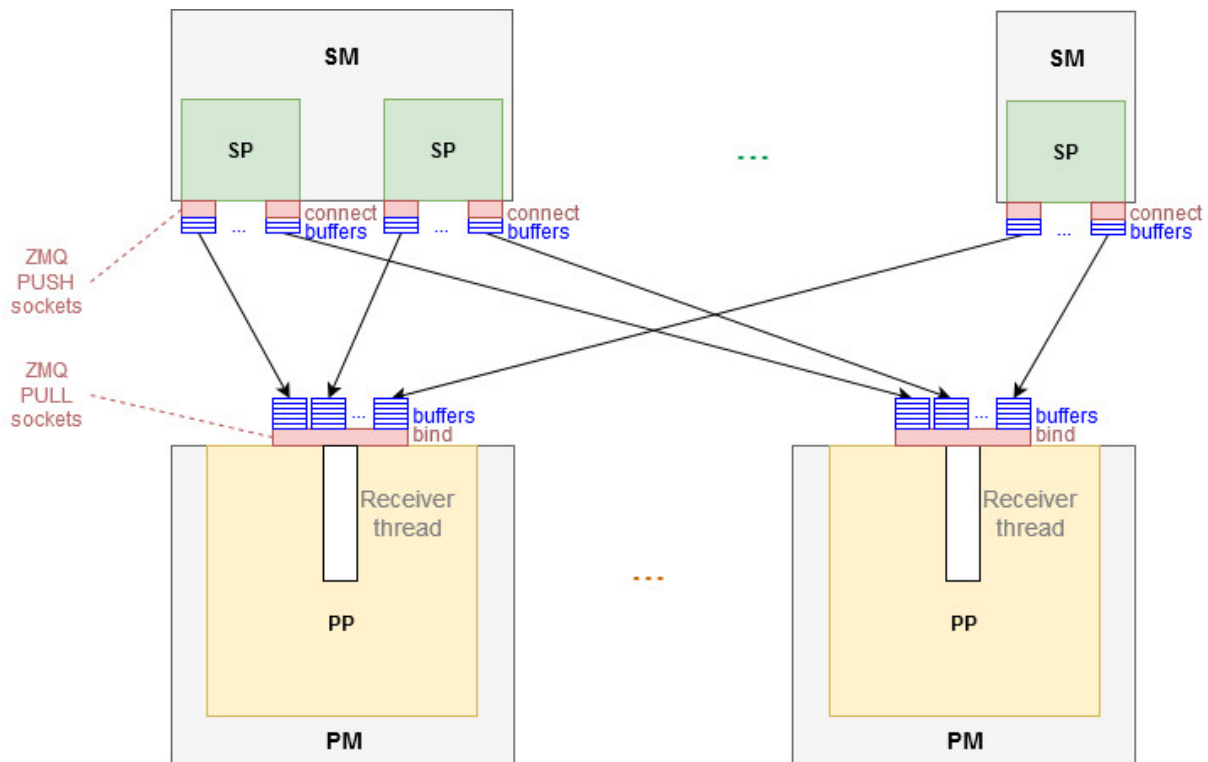


Figure 3.14: Communication between SPs and PPs.

Due to the fact that SM should not use too much of the hardware resources (it is placed on the original machines of the ALICE experiment) a special buffer architecture is configured. To prevent from consuming the whole memory of a SM there is a limit (HWM) created to keep only fixed amount (quite small) of events for each connection. It is specified in the configuration file as “SP_SendHWM” variable. The default value is specified in the table 3.2 (see section 3.6). If HWM was not set memory of SM could fill

up and SM could be stuck or crashed. In this solution if the HWM is reached (for all PPs), the program starts to block all data (this blocks the work of a SP so the whole experiment as well). More information about HWM is available in section 3.4.1. The blocking PUSH-PULL pattern was chosen for a situation when a client requires “Must 100%” of data and is too slow to handle it in the real-time. Then whole experiment must wait for him. Larger buffers are located on the PM side so the SM memory is not consumed by SP.

Every SP tries to send a message to any PP at all costs. The data is dropped only if no PP is “alive”. For this purpose the heartbeat threads were created. For every PM a new thread is started. The scheme of SP heartbeat threads is presented in figure 3.15. Inside this thread a ZMQ REQ-REP pattern is used. To prevent a deadlock a reconnect action is executed every time there is a problem with receiving a heartbeat. After successful communication the “alive” parameter is set to true and when a problem appears to false.

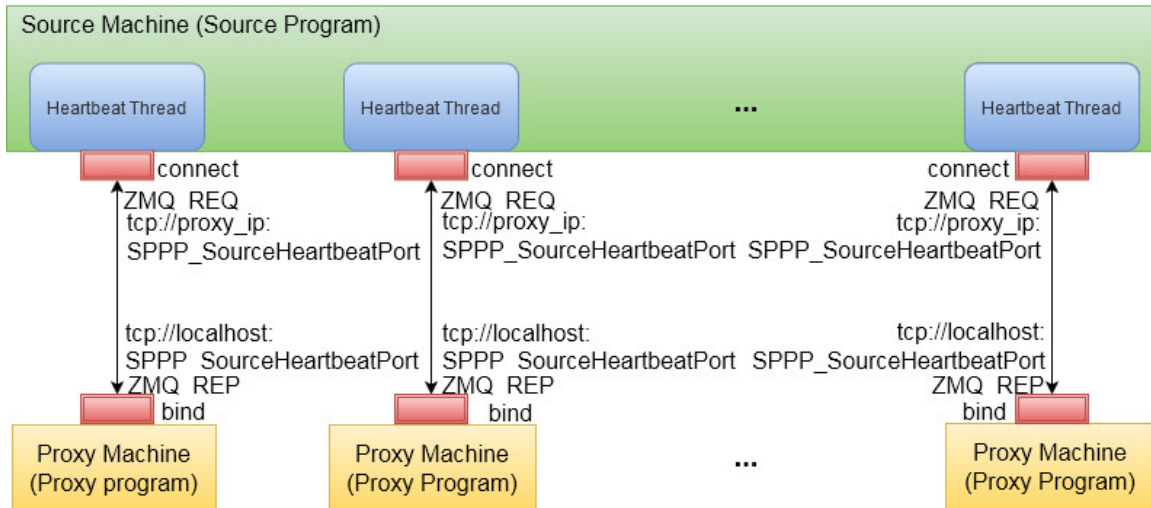


Figure 3.15: The SP heartbeat thread scheme.

The “alive” parameter is also used to keep a proper load-balancing. The SP tries to send data equally to all PPs, but when one of them is dead or too busy to send a heartbeat the SP skips it and sends a message to the next alive PP. By doing this SP does not dispose any message. Only data which was sent successfully, but not correctly processed by a PP is missed. Sending successfully means that ZMQ did not return an error, but due to features of the ZMQ solution, the message can be delivered successfully as well as stuck in the sender buffer, receiver buffer or even somewhere between.

Four configurable variables determine how fast the program handles the communica-

tion problem. “SP_HeartbeatDelay” determines the frequency of checking the PP alive status. “SP_SendTimeout” specifies the timeout how long the main thread waits until it checks again for an alive PP in order to send data. “SP_ReconnectSleep” says how many seconds heartbeat thread waits until it makes the next reconnect attempt, while “SP_HeartbeatTimeout” determines the timeout of sending and receiving heartbeat messages between SP and PP. In case of failure of one of the machines (both SM and PM) SP is able to renew the connection. However, such actions should happen very sporadically. For the fastest start, which is reflected in the best performance the recommended startup order is : first PPs then SPs.

3.5.2 Proxy programs

PPs are more complicated than SPs. There is no single task that is provided by these machines. Every PP can be divided into several parts: one to communicate with SPs, one to communicate with CPs and one realizing some internal functions. The scheme is presented in figure 3.16.

As mentioned in section 3.5.1 there can be only one instance of this program on a single machine. The global ports, such as for the main thread, are taken from a configuration file, placed in the DATE database, called “proxies.config” (for more information see section 3.6). A change of settings in the database while programs are running will not affect the given output. Settings are read from the database only on startup. Every PP has two internal classes (acting as databases) called PROXY_DB and KEY_DB where active clients sessions are stored. It contains session ID as well as ports numbers (for communication as well as data threads), the key and some other parameters.

SP communication

The main task of the SP communication part is to collect messages sent by the SPs and distribute them to active internal threads dedicated to the unique keys of connected clients. This part is made up of two separate threads. One thread is devoted to receiving data from SPs. It is based on ZMQ PULL socket. It binds to a port on a localhost and starts a loop which works as long as the PP is alive. The port is taken by the same variable

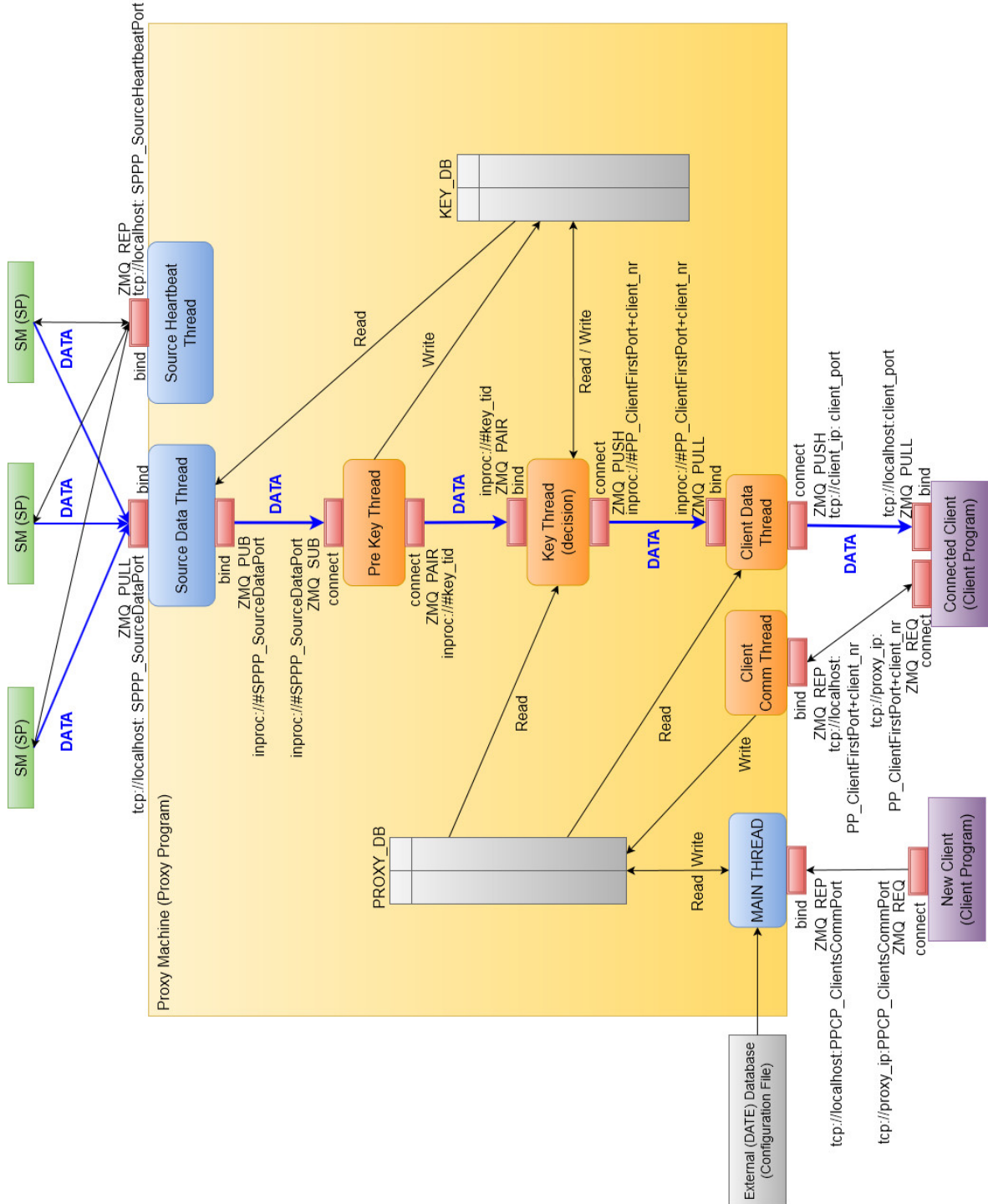


Figure 3.16: The PP scheme.

as in SP – “SPPP_SourceDataPort” from the configuration file in the DATE database. The second thread is a heartbeat thread corresponding to the SP heartbeat thread.

The ZMQ sockets, with buffers for handling some events, are bound on the PM side. The buffers are adjustable by putting the right value in the configuration file or push it as an argument while executing. If there is no HWM specified in the configuration file and there is no argument for the execution of the program, the default value is set (see section 3.6). This size concern one connection, the overall size is multiplied by the amount of connections. This amount is the number of events not bytes.

All received messages are passed along to all internal threads (if any exists). The message is duplicated by the number of active inner threads. If there is no active inner thread the message is dropped. The communication among threads is also based on the ZMQ protocol using PUB-SUB scheme, but this time by in-process transport instead of TCP. In this case it is crucial due to the fact that everything is done within one machine, otherwise there can be problem with too many opened sockets. The socket is bound on the SP communication side. In-process endpoint name is created with an preamble “inproc://#” and a the same variable as in SP – “SPPP_SourceDataPort”.

Internal tasks

Internal tasks are performed inside dynamically created inner threads called “pre-key threads” and “key threads” grouped in pairs. Every pair corresponds to the unique “key” send by clients. Between them there is an in-process connection made by ZMQ PAIR sockets. Every key thread is connected to all clients sharing the same key. The idea of this solution is presented in figure 3.17. The number of inner pairs of threads never exceeds the number of client threads. In extreme cases these values are equal.

Inside the pre-key thread data is received from the ZMQ socket and sent to the key thread. If sending the message fails this thread checks if a corresponding key thread have a “must” flag set. If so it tries to resend a current message, otherwise it drops it. Key thread receives data from a corresponding pre-key thread, then unpacks it, checks its parameters and compares to the specific client’s filters. The function which compares the received event with the client’s filters is shown in the listing 3.5.2. It returns -1 value if the event does not match the client’s preferences, 0 if it matches and client has option

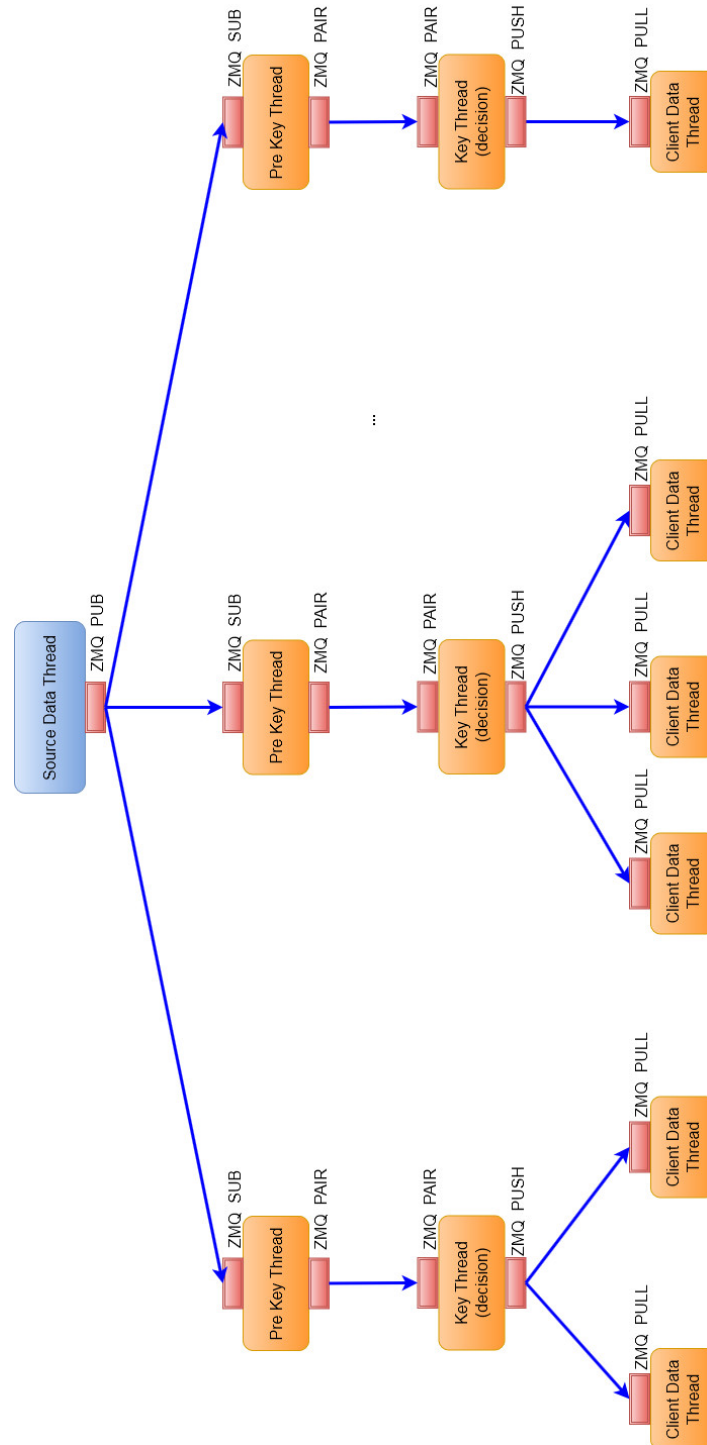


Figure 3.17: The PP dynamic internal threads scheme.

“must all” checked for this filter or (number of preference+1) if the event matches, but the client does not need necessarily all messages of this type.

Listing 3.5.2: Inner thread filters comparison function.

```
int checkPref(struct eventStruct event, int clientnr)
```

Matching event is transmitted to the ZMQ socket of one chosen by above function client (all connected clients use the same key). One message is sent exactly to one client, it is not duplicated even more clients have same matching filters. This solution was created specifically to meet the requirement that several CPs with the same key should not receive duplicated data.

Within every inner thread pair four ZMQ sockets using in-process communication transport are created. First, located in pre-key thread, is a receiving ZMQ_PUB socket which is connected to the source data thread PUB socket. Then two ZMQ_PAIR sockets exists – sending on the pre-key thread side and receiving inside the key thread. The last one is also located in key thread and is a sending ZMQ_PUSH socket to which the thread connects and disconnects by entering the address of the appropriate client’s endpoint.

The client endpoint names for a PUSH socket are taken dynamically from the KEY_DB database by the function called “get_client” presented in the listing 3.5.3 (see appendix B).

Listing 3.5.3: Function getting client threads in-process endpoint names.

```
int get_client(char** &iaddr, int* &icli, int n)
```

It passes a key id and returns the amount of connected clients as well as list of their in-process endpoint names and IDs corresponding to PROXY_DB.

CP communication

This part is devoted to communication between PP and CP. It consists of a static main thread and a bunch of dynamically created thread pairs (data + communication). The maximum number of clients that a PP can support simultaneously is specified using the “PP_MaxClients” variable in the configuration file “proxies.config”. The default value is specified in the table 3.2 (see section 3.6).

The main thread is performing a loop waiting for any CP to connect. When the CP

connects, the main thread is looking in PROXY_DB database if there is a free slot for him. If so the main thread is writing the new CP into a PROXY_DB database, creating a new communication thread and sends the new port number to the CP.

The newly created communication thread is waiting (for time specified by a variable “PP_ClientFirstHeartbeatTimeout”; see section 3.6) for the first heartbeat from the client. If the key of a connected client match the key of a client saved in the PROXY_DB database it saves it in the KEY_DB database, creates a key thread and a new data thread. Later, the communication thread is sending a heartbeat and receiving a client data such as filters as long as the client is connected. When the client changes filters this thread updates the internal database PROXY_DB.

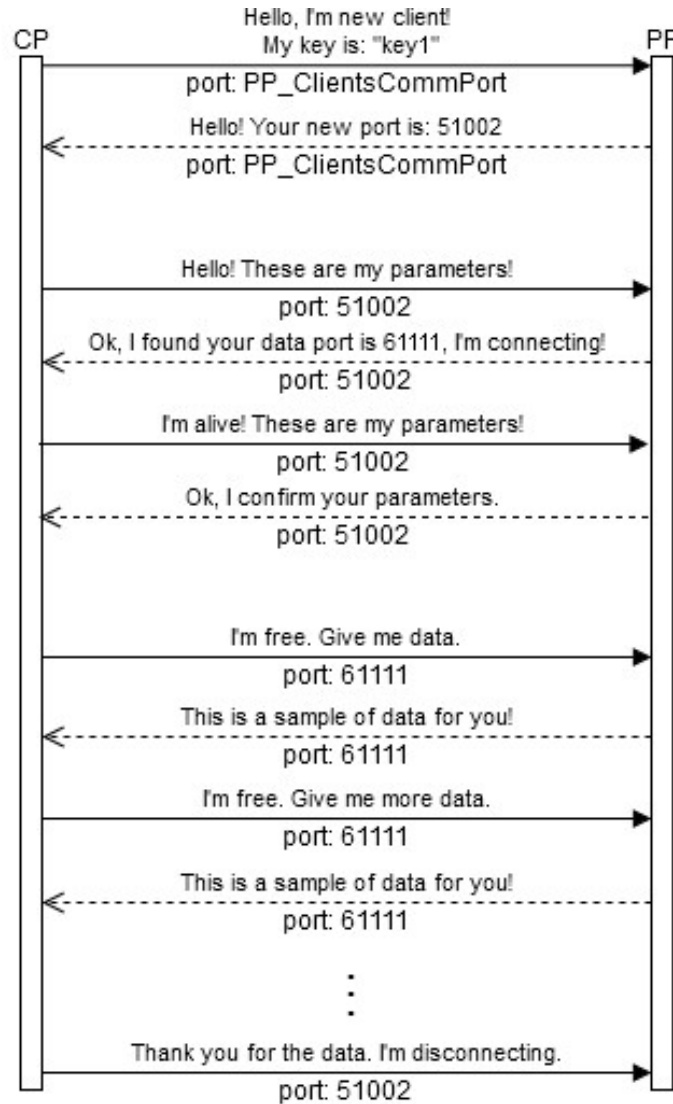


Figure 3.18: Example of connecting new client with CP protocol.

At the same time the data thread connects to the CP ZMQ_PUSH socket (TCP/IP communication protocol) and binds a ZMQ_PULL socket (in-process communication protocol). Then starts a loop which is receiving messages from the corresponding key thread and forwarding it to the CP. There is a buffer with size specified by a “PP_ClientHWM” variable from a configuration file from the DATE Database. The default value is specified in the table 3.2 (see section 3.6). The example of CP communication is presented in figure 3.18.

The data socket is bound on the client side. The client has his own buffers and the memory of the PMs is not consumed. On the other hand the communication socket is bound on the PROXY side in case that the client can disconnect and connect again to it. Because of the REQ-REP type of socket there is no big buffer for this connection.

3.5.3 User library

In the new proxy-based monitoring library there are some new methods to be used by the clients. These functions perform background activities in a way imperceptible to the user. During executing them some sockets are created as well as heartbeat threads for all connected PPs. The scheme is presented in figure 3.19.

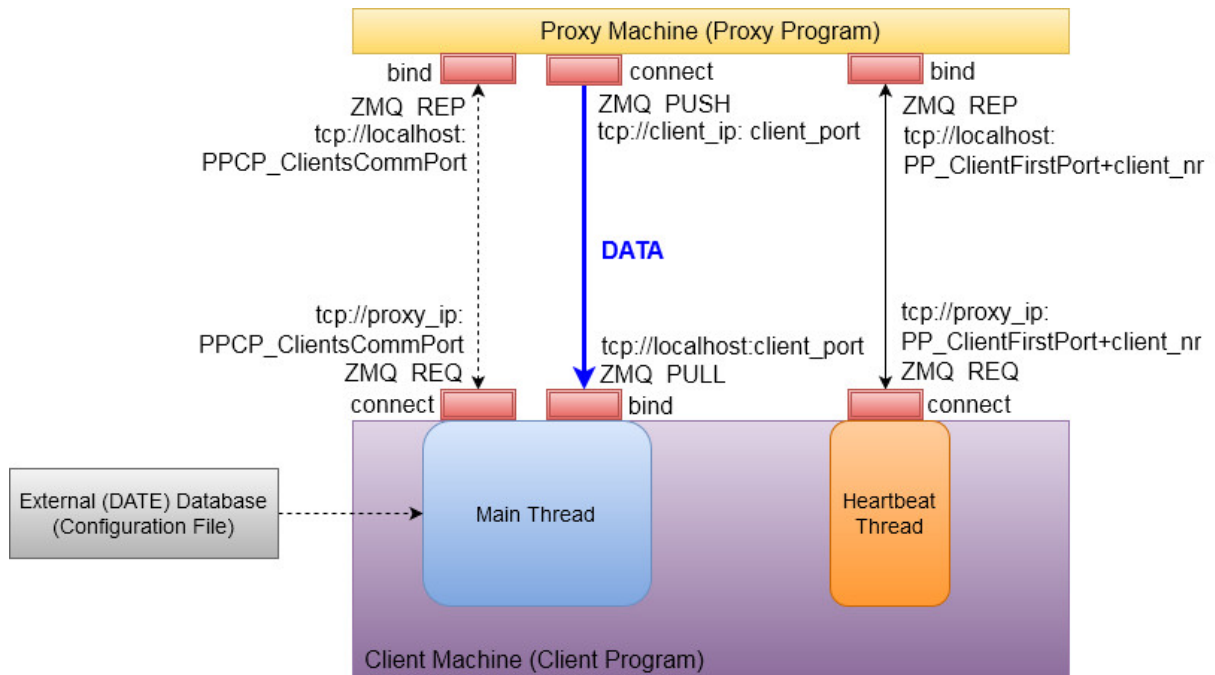


Figure 3.19: The CP scheme.

The list of all public functions (intended for clients use) is presented below (in listings 3.5.4-3.5.14). To maintain backward compatibility, most of these functions also have their old format counterpart, but using that notation cannot fully achieve this library's functionality. Nevertheless, it is recommended to switch to using new functions.

In addition to these functions, the library also contains internal methods used by them and some functions used by other programs from this project, but they are not described here. The source code of all functions is attached in the appendix C.

Listing 3.5.4: mbpCreateContext method.

```
int mbpCreateContext()
```

The mbpCreateContext function (shown in listing 3.5.4) creates a ZMQ context and fills the PROXIES table by a list of PMs in the format of "tcp://ipaddress:". The information about PMs is read from a configuration file (see section 3.6) and does not say anything about their activity. Function takes no argument. It returns a number of PMs on success or -1 on error.

Listing 3.5.5: mbpConnection method.

```
struct mbpConnection* mbpCreateConnection(char* key, char* address)
```

The mbpCreateConnection function (shown in listing 3.5.5) creates a connection structure, allocates memory and fills it with the provided data and default values. It also creates one default preference. The function takes 2 arguments: the key and the address, in format "ipaddress:port", where PMs should sent data. It returns created connection on success or NULL on error. Attention: The key must be unique. A bunch of clients with the same key, even if they are on different machines, are treated as parts of the same program and will not get duplicated data.

Listing 3.5.6: mbpSetPreferences method.

```
int mbpSetPreferences(struct mbpConnection *connection, struct  
    mbpPreference *newPreferences, int PrefNb)
```

This function fills preferences in mbpConnection obtained by mbpCreateConnection function with the provided values. It returns a number of set preferences on success or -1 on failure. If mbpSetPreferences method is not used the default preference, set by the mbpCreateConnection function, is remained.

Listing 3.5.7: mbpConnect method.

```
int mbpConnect(struct mbpConnection *connection, int buffhwm)
```

The mbpConnect function (shown in listing 3.5.7) creates a ZMQ data socket and binds to it. Afterwards it gets a communication port number of the PMs from a configuration file (see section 3.6) and connects to all of them, sending own preferences with a data address and port (which was provided earlier to mbpCreateConnection function), where PMs should sent data later. From communication ports of PMs it receives a unique client heartbeat port. It disconnects from the communication port and creates a heartbeat thread where it connects to the obtained port. Function takes two arguments: a mbpConnection obtained by mbpCreateConnection function and a HWM value (see section 3.4.1). It returns a number of not connected proxy servers (0 if everything is ok, >0 if at least one server is not connected properly).

Listing 3.5.8: mbp Wait methods.

```
void mbpControlWait(struct mbpConnection *connection, bool flag)
void mbpSetWait(struct mbpConnection *connection)
void mbpSetNoWait(struct mbpConnection *connection)
```

This set of functions (shown in listing 3.5.8) set the flag “WAIT” which is responsible for the behavior of the mbpGetData function. The mbpControlWait function takes two arguments: a mbpConnection obtained by mbpCreateConnection function and a boolean specifying a flag value. It just sets the flag according to the provided argument. The mbpSetWait sets the flag to true while mbpSetNoWait sets the flag to false. If the flag parameter is set to true the mbpGetData functions stops and waits (by the time specified by mbpSetWaitTimeout function) until it gets the message, otherwise it checks if there is any message and continue. The default timeout is set to -1 which means that the time is infinite – function waits as long as the message is available. The default value of “WAIT” flag is true. These functions do not return anything.

Listing 3.5.9: mbpSetWaitTimeout method.

```
void mbpSetWaitTimeout(struct mbpConnection* connection, int timeout)
```

The mbpSetNoWaitTimeout function (shown in listing 3.5.9) sets the timeout value for the mbpGetData function. If “WAIT” flag is set to true the mbpGetData function

waits for the specified amount of time (timeout) and when no message is available by this time it goes on. The function takes two arguments: a mbpConnection obtained by mbpCreateConnection function and a timeout value (given in milliseconds). It does not return anything.

Listing 3.5.10: mbpGetData method.

```
int mbpGetData(struct mbpConnection* connection, struct eventStruct
               *event)
```

The mbpGetData function (shown in listing 3.5.10) gets one event from ZMQ buffer from data socket, which should be created earlier by the mbpConnect function. It waits for the available message or not accordingly to the “WAIT” flag status and timeout specified by mbpSetWaitTimeout. The function takes two arguments: a pointer to the structure eventStruct where received data will be copied and a mbpConnection obtained by mbpCreateConnection function. It returns the length of event on success, 0 if there is no event available before the timeout or -1 on failure.

Listing 3.5.11: mbpDisconnect method.

```
int mbpDisconnect(struct mbpConnection* connection)
```

The mbpDisconnect function (shown in listing 3.5.11) unbinds the data socket and closes it. It also disconnects from the client heartbeat socket. The function takes one argument: a mbpConnection obtained by mbpCreateConnection function. It returns 0 on success or -1 on failure.

Listing 3.5.12: mbpDestroyConnection method.

```
void mbpDestroyConnection(struct mbpConnection* connection)
```

The mbpDestroyConnection function (shown in listing 3.5.12) frees memory allocated by the mbpCreateConnection function. It takes one argument: a mbpConnection obtained by mbpCreateConnection function and does not return anything.

Listing 3.5.13: mbpDestroyContext method.

```
int mbpDestroyContext()
```

The mbpDestroyContext function (shown in listing 3.5.13) destroys the context created by the mbpCreateContext function. It takes no argument and does not return anything.

Listing 3.5.14: mbpDestroyContext method.

```
int mbpFlush(struct mbpConnection* connection)
```

The mbpFlush function (shown in listing 3.5.14) flushes all remaining events from a data socket buffer. It takes one argument: a mbpConnection obtained by mbpCreateConnection function and does not return anything.

3.5.4 Client program example

Every client that wants to get monitoring data should write their own special purpose program. In order to do that one can use the provided monitoring by proxy library. The client needs to include the header file called “monitoringbyproxy.h” in their source code and add a shared or static library called “libmbp” (respectively libmbp.so or libmbp.a) during the compilation of the program.

The library methods were described in the previous subsection (3.5.3). Below (in listing 3.5.15) the example how to create a simple program which is getting events from the monitoring program is presented. The same source code is also attached in the appendix D.

In this example the client provides its IP address with a port number (separated by a colon) and sets the HWM to 20. Preferences are specified to receive 100% (with “must” flag set) of calibration events and a 50% (or if he is slower as much as he can handle) of physics events. Then in a loop it gets data (with a timeout of 1000 milliseconds) and print some parameters of a received event.

Listing 3.5.15: Example of client monitoring program written in C.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <zmq.h>
#include "monitoringbyproxy.h"

int main()
{
    int c = mbpCreateContext();
    mbpConnection* cd = mbpCreateConnection("key1", "192.168.1.123");

    struct mbpPreference pd[] = {
        {"50", "PHY", "*", "*", "*"},
        {"must", "CAL", "*", "*", "*"}
    };

    int p = mbpSetPreferences(cd, pd, sizeof(pd)/sizeof(pd[0]));
    p = mbpConnect(cd, 20);

    int rc;
    struct eventStruct* pevent = calloc(sizeof(struct eventStruct), 1);

    if(p != -1)
    {
        while(true)
        {
            rc = mbpGetData(pevent, cd, 1000);
            if(rc != -1)
                printf("Event received: Run: %d, Size: %lu, Type: %d\n",
                    pevent->eventHeader.eventRunNb,
                    pevent->eventHeader.eventSize, pevent->eventHeader.eventType);
        }
    }

    mbpDisconnect(cd);
    mbpDestroyConnection(cd);
    free(pevent);
    mbpDestroyContext();
    return 0;
}
```

3.6 Programs configuration

A configuration file is located in the DATE database. It must be named as “proxies.config”. It has a syntax compatible to the rest of files and an example is presented in listing 3.6.1.

Listing 3.6.1: Example of “proxies.config” file.

```
[Proxies]
ProxyList=137.138.143.215,137.138.143.188

[Ports]
SPPP_SourceHeartbeatPort=49000
SPPP_SourceDataPort=49900

[Limits]
PP_MaxClients=2000
```

The file is divided into three sections: Proxies, Ports and Limits. The first one defines the list of machines that are supposed to play a PM role. For correct work of monitoring, proxies IP variable must be named as “ProxyList” and have a list of plain IP addresses without a prefix, a colon or a port number. Addresses must be separated with a comma. This variable is used by every monitoring program and is the only one from the configuration file as obligatory. Without it none of the programs will start.

The second part contains other global ports parameters which are used by the machines. Each parameter has a prefix, ending with an underline, indicating which program makes use of it (e.g. SPPP_SourceDataPort is used by both SP and PP). Mentioned ports are listed in table 3.1.

The last section contains some parameters which determine limits of monitoring program. The default values of these parameters are listed in table 3.2.

All of parameters in sections “Ports” and “Limits” have its default value hard-coded in case that there is no values in configuration file and programs are executed without any parameters. Unfortunately, incorrect data from the configuration file or from the command line arguments may results in incorrect operation of the program, so the change of the file or executing program must be done with attention.

Table 3.1: Default values of ports parameters

Parameter	Description	Default value
SPPP_SourceHeartbeatPort	port used to create a heartbeat connection between SMs and PMs	49998
SPPP_SourceDataPort	port used to create a data connection between SMs and PMs	49999
PPCP_ClientsCommPort	port to which clients can connect to PMs in order to get their own connection port	50000
PP_ClientFirstPort	first number that can be used for clients connections in PMs	51000

Table 3.2: Default values of limits parameters

Parameter	Description	Default value
MaxEventSize	maximum event size given in bytes, used by all programs to adjust network buffers size	25000000
SP_SendHWM	send buffer limit (HWM) on SP side which keeps events for each connection with PP (given in event number)	1
SP_SendTimeout	the timeout (in milliseconds) how long the SP waits until it checks again for an alive PP in order to send data	1000
SP_HeartbeatTimeout	the timeout (in milliseconds) of sending and receiving heartbeat messages between SP and PP. This timeout should be greater than SP_HeartbeatDelay	100
SP_HeartbeatDelay	the frequency (in seconds) of checking the PP alive status by the SP	1
SP_ReconnectSleep	the delay (in seconds) of SP next reconnect attempt to a PP	10
PP_ReceiveHW	receive buffer limit (HWM) on PP side which keeps events for each connection with SP (given in event number)	20
PP_ReceiveTimeout	the timeout (in milliseconds) how long the PP waits until it goes on inside data receiving thread	1000

PP_SourceHeartbeatTimeout	the timeout (in milliseconds) of receiving heartbeat messages between SP and PP. The send value is one quarter of this value	1000
PP_MaxClients	maximum number of simultaneously connected clients	1000
PP_MaxPreferences	maximum preferences that one client can specify	10
PP_ClientFirstHeartbeatTimeout	the timeout (in milliseconds) how long the PP waits for new client to connect to new port until it ends the thread created for him	5000
PP_SendHWM	send buffer limit (HWM) on PP side which keeps events for a CP (given in event number)	1
CP_NewPortTimeout	the timeout (in milliseconds) how long the CP waits for receiving a new heartbeat port from a PP communication socket	2000
CP_HeartbeatTimeout	the timeout (in milliseconds) how long the CP waits for a heartbeat request from a PP heartbeat socket	5000

Chapter 4

Results of benchmark tests

Benchmark in general is “a point of reference by which something can be measured”[76]. In computer science this term has similar meaning, but with respect to computer performance (related to hardware or software). The point is to use some kind of special algorithm to show the performance in a specified metrics. The real benchmark is when multiple scores (results) are analyzed.

In this thesis in order to perform benchmarks of the created monitoring programs an open source platform developed by the InfluxData was used. Tools that are parts of this platform are described in subsection 4.1, while the results of the tests are presented in subsection 4.3.

4.1 Testing tools

The test environment of this project was built with three out of four from TICK stack projects: Telegraf as a metrics gathering tool, InfluxDB as a storage of collected data and a Chronograf as a visualization tool. The first one of these was installed on all machines which were used for monitoring programs running and the two others were working only on one machine which was used for taking metrics data. An installation locations scheme is shown in figure 4.1.

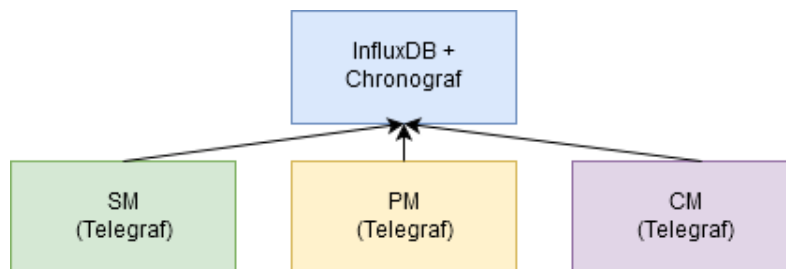


Figure 4.1: TICK projects installation locations.

Telegraf

Telegraf is a plugin-driven agent for collecting, processing, aggregating and writing metrics. It gets data and outputs it to the storage machine. Everything what is needed to get started with it is just to install (in this case version 1.4.2.3 was installed) and configure using “telegraf.conf” file. For the purpose of this project the only change was the endpoint URL for InfluxDB instance. The port number has been left default as 8086. The configuration is presented on listing 4.1.1.

Listing 4.1.1: Telegraf output URL configuration.

```
# Configuration for influxdb server to send metrics to
[[outputs.influxdb]]
urls=["http://node2-69.grid4cern.if.pw.edu.pl:8086"]
```

Metrics were gathered by using “system” input plugin, which allows to collect data such as cpu, memory or network usage.

InfluxDB

InfluxDB is a time-series database for storing, managing and querying data. This tool was used to store all collected metrics. To use this tool some steps are needed. First of all it must be installed properly, then some configuration must be done in “influxd.conf”. After configuration a daemon must be started (just execute “influxd” command or service influxdb start). When the daemon is running some commands on the database can be performed (by executing “influx” command and using syntax similar to SQL like “SHOW DATABASES” etc. – all syntax can be found on influxdata website [77]). The version 1.4.2-1 was used.

Chronograf

Chronograf is an open source web application that is used as a user interface. It is a very complex tool for displaying data collected in a database. The version 1.4.2-1 was used. All diagrams presented in the next section are obtained from it. In this project it was installed on the same machine as an InfluxDB, so the configuration remains default as a localhost. The only thing which was prepared is a dashboard view which can be seen everywhere in the overview section 4.3.

4.2 Test machines

The test environment was installed on the machines located at the Warsaw University of Technology. An OpenStack cluster consists of sixty modern PCs with the following specification: 2 x Intel Xeon CPU E5-2680 v3 @ 2.50 GHz (12 cores, 24 threads), 96 GB RAM, 1 x SSD 480 GB, Ethernet 10 Gbit/s. Every node on the cluster has SCL6 (Scientific Linux CERN 6) installed (kernel release 2.6.3-696.20.1.el6.x86_64) [72]. ZeroMQ is installed in version 4.1.4-0 [75]. For these benchmarks there were 30 of the machines available. The machine use diagram is presented in figure 4.2. For the purpose of benchmarks some machines are multiplied and some remain as single unit.

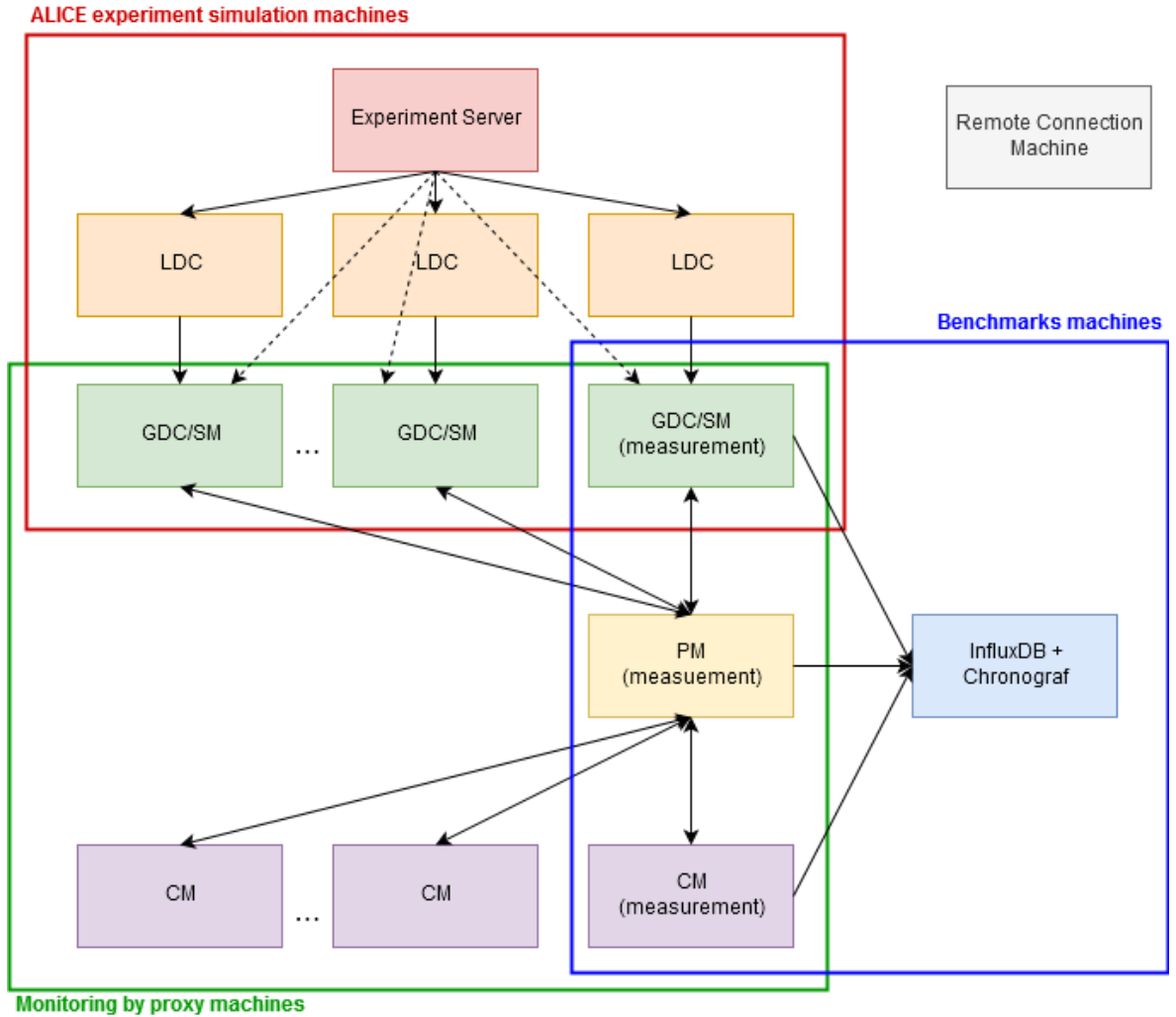


Figure 4.2: The diagram of using test machines.

One single machine was used as a gate so the remote connection did not affect the environment. Some machines were used for the ALICE experiment run (marked with the red frame in figure 4.2). For testing purpose the data stream is performed by a COLE (COnfiguralbe LDC Emulator). It creates ALICE-like events with the payload taken from a file located in DATE Site folder. The prepared test environment at WUT with the full ALICE DAQ framework setup and the COLE emulator is further referred to as the ALICE experiment simulation. The green frame in figure 4.2 indicates the machines that work as a parts of a monitoring by proxy project. In each benchmark test, the number of these machines is adjusted to the individual case. The last group marked with a blue frame are machines used to preform the benchmarks. The number of these machines in each test is constant. One of each machine type SM, PM and CM is used as a source of the measured values. These machines have Telegraf installed. Metrics such as CPU usage, memory usage or network load (input and output) are passed to the machine with InfluxDB and Chronograf installed where they are collected and displayed in the form of graphs. The data, metrics and control flow is shown in figure 4.3.

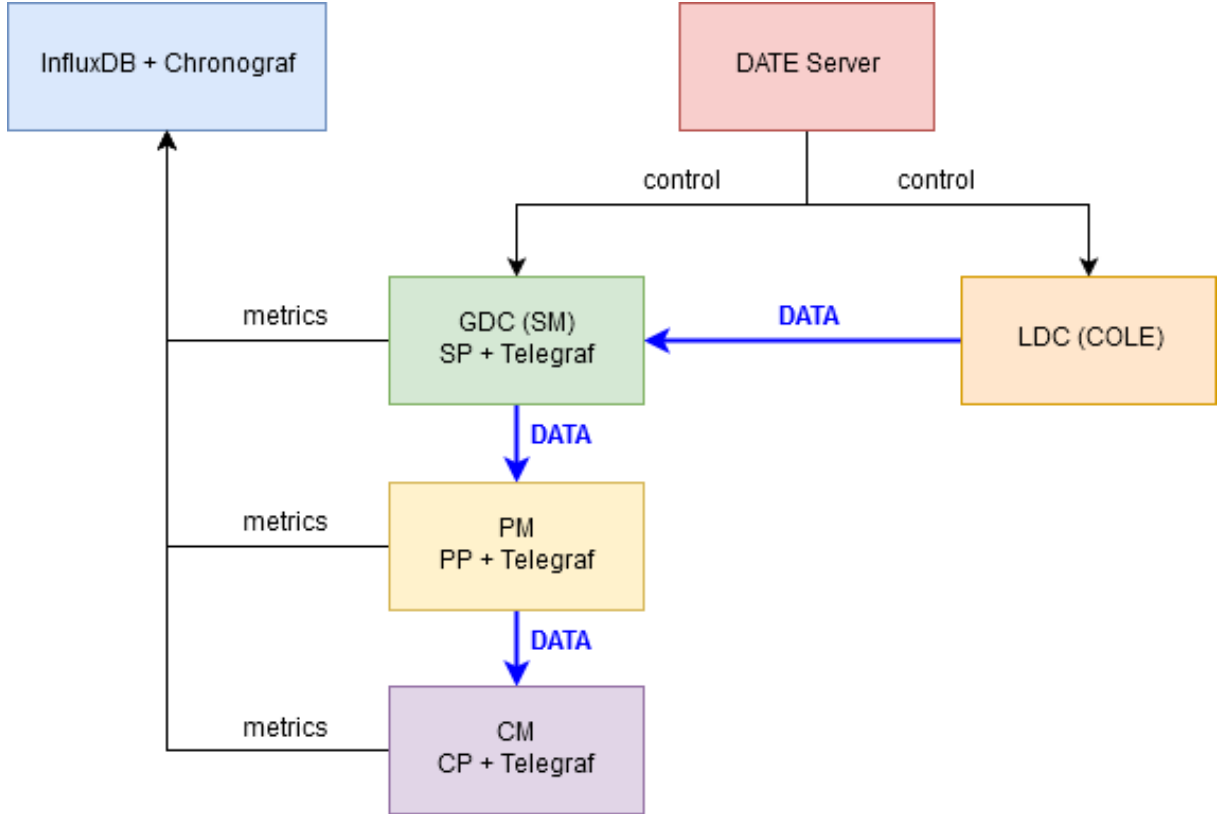


Figure 4.3: The signal flow scheme.

The assumption was that the amount of PMs should be no more than 15% of the event producing machines amount. As in the Run 2 the amount of GDC is 83 the amount of PMs should be up to 12. This means that the monitoring by proxy library should work fine with 7 SMs per PM. As the test environment has limited resources one segment of experiment, consisting of up to 10 SMs, one PM and up to 10 CMs, was performed. This corresponds up to 1/8 of the whole experiment during Run 2 (10 SMs per PM).

For these tests two kind of events are generated – bigger physics events and smaller calibration events in a ratio of 3 to 1. Sub-events are created by three LDCs where every of them have separate payload file for each type of events. Physics files are filled with 6.5 MB of data, while calibration files have only 10 KB. After adding the headers, physics event size reach around 20 MB (exactly 20275436 bytes) as this is the largest size of event for Run 2 and similar to Run 3. Calibration events were 30 KB in size (31124 bytes).

During Run 2 ALICE was generating 4 GB/s of data so for these tests it was assumed that each of 10 SPs should receive minimum 50 MB/s of data. Therefore, every LDC produces up to 200 MB/s of streamlined data and the total maximum data flow is 600 MB/s.

4.3 Overview of results

Four kind of tests were performed. First of them is the basic test made with one of each programs SP, PP and CP. Second test was performed using one SP, one PP and up to 10 CPs. Third one consists of a PP, one CP and up to 10 SPs. The last test was performed on the entire segment consisting of up to 10 SPs, one PP and up to 10 CP. Finally, an additional test was performed with the adjusted number of machines. In all tests CPs do not request any filtering – just take all events with default preferences. Each role was started on a separate machine.

Test results are presented in twelve cells “table” consisted of three columns and four rows. Each column refers to one of the machines and each row refers to one type of measured values. The brief auxiliary scheme which represents the test results is presented in figure 4.4.

From the left, the first column (marked with green frame) shows the data production machine (SM). In the middle (in yellow frame) the proxy machine (PM) is shown. The

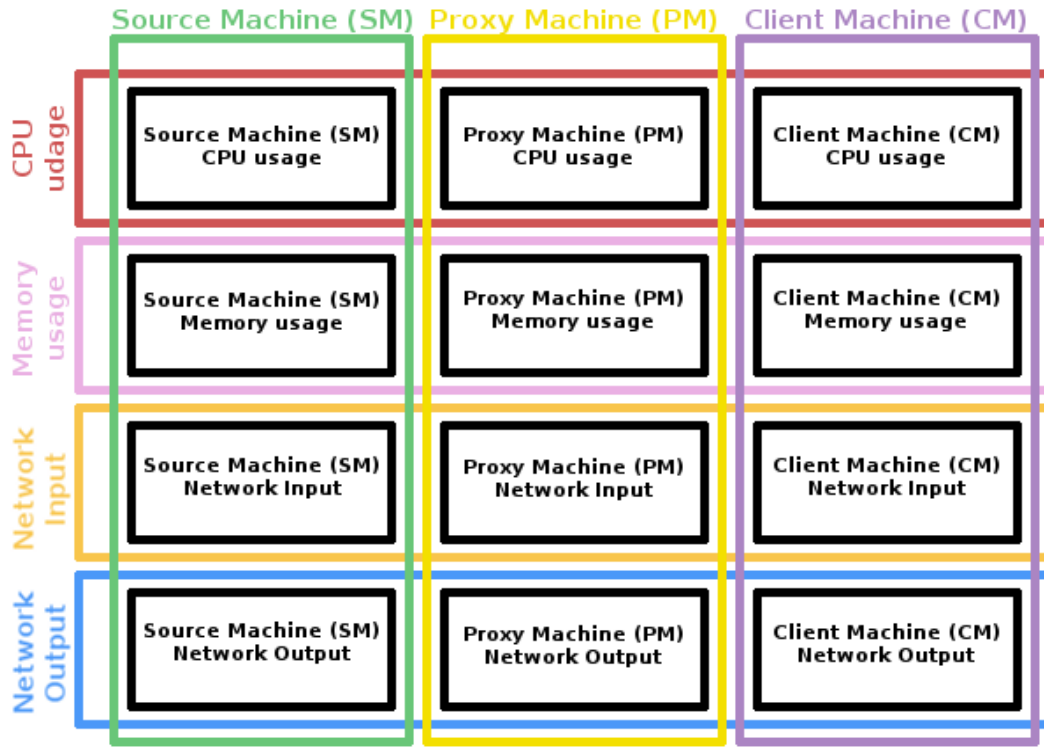


Figure 4.4: Scheme of a test result.

last, right column (surrounded by a violet frame) corresponds to the client machine (CM).

As for the rows, the first (top) row (marked with red frame) shows the percentage of CPU used for the machine. The yellow line the on the graph indicates a system usage while brown one indicates user usage. The second row (in pink frame) presents the memory usage. The third one (surrounded by orange frame) corresponds to the network input load while the last one (in blue frame) the network output usage.

The X axis shows time of the test in hh:mm format. It is common for all charts in one test so that each parameter can be compared on every machine at exactly the same time point. The Y axis corresponds to specific parameter which is currently measured on each chart, it is always the same for the whole row, but the scales may vary.

Test 1 – single programs

This test lasted an hour. At the beginning only the ALICE experiment simulation was running without any monitoring programs. After 10 minutes PP and one SP started. After next 10 minutes one CP connected to the PM. Results are shown in figure 4.5.

4.3. OVERVIEW OF RESULTS

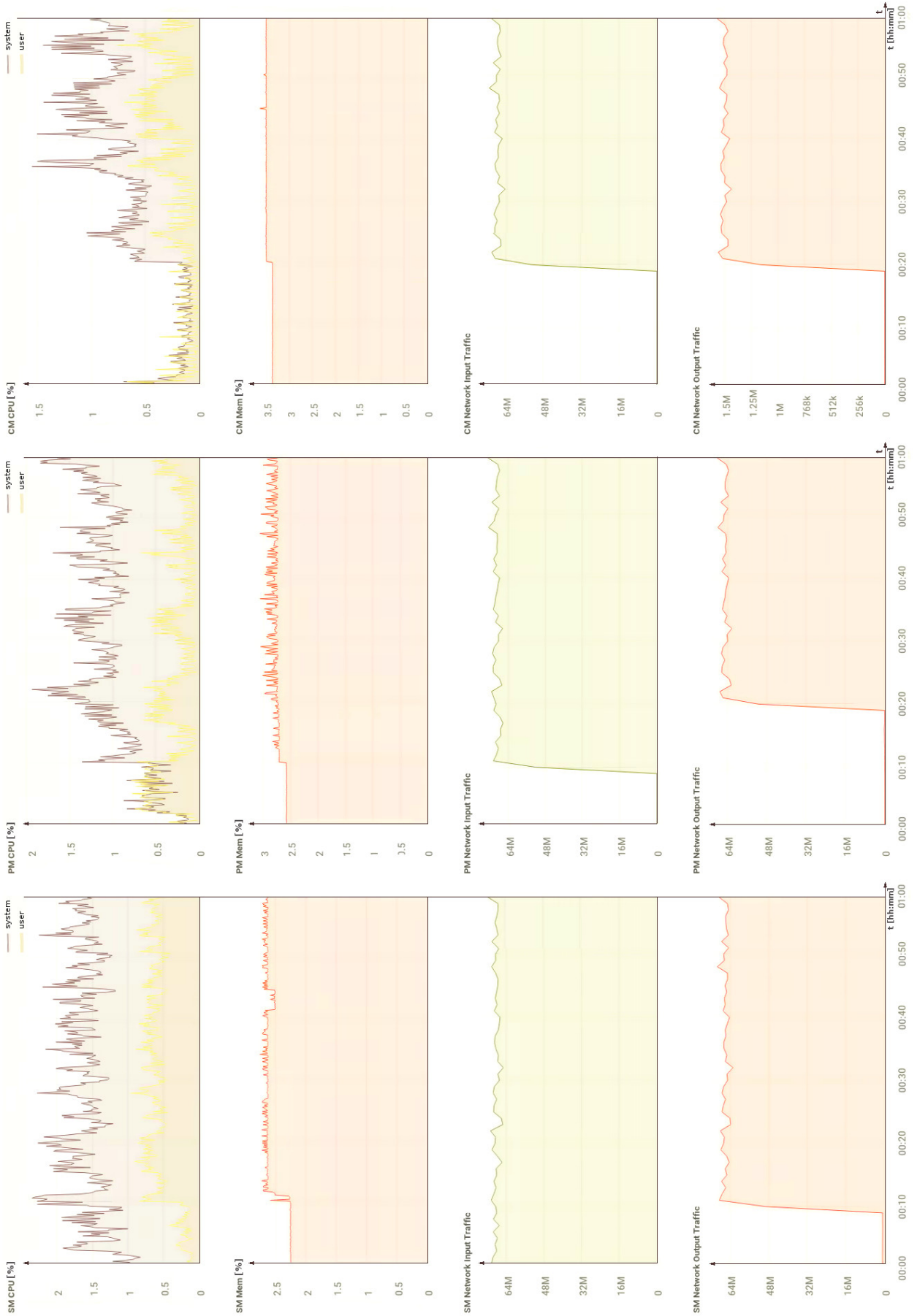


Figure 4.5: Results of test 1.

This test shows that the created program works fine. The SP was producing about 70 MB/s of data (see section 4.2). Throughout the test the data rate was kept at the constant level and all events were received by the connected CP. CPU and memory consumption of all machines raised slightly, but at an acceptable level.

Test 2 – multiple clients

This test was performed in two ways. In the first case, all CPs had connected with the same key, while in the second case each CP had its own unique key. Results are shown in figures 4.6 and 4.7. First test lasted an hour, while the second one two hours. Both began with only the ALICE experiment simulation running. After 10 minutes, one SP and one PP started. Then one CP would connect every 2 minutes. At the end all 10 CPs were running for the rest of the time.

Results of the first case of this tests (when all CPs were using same key) are shown in the figure 4.6. It shows that the more clients with the same key were connected to the server, the less data each of them received. SP was producing about 65 MB/s of data, when only one client was connected, he received the full amount. When the second CP joined data stream split among them causing each of them to receive about 33 MB/s. The next CPs caused data stream divided into three parts. When all 10 CPs were connected each of them was receiving around 6.5 MB/s which corresponds to plus or minus 1/10 of the entire data stream. The CPU and memory usage were at an acceptable level.

The second case of this test (when each CP was using different key) is presented in figure 4.7. One can notice that PP duplicated the data stream and sent copies to each CP. After fifth CP connected the SP sending data rate (and thus receiving data rate of PP) started to drop. After seventh CP connected the data rate of the ALICE experiment simulation started to drop, but it probably coincided and was a consequence of the drop of SP outgoing data rate. Next after the ninth CP connected the sending data rate of PP started to drop slightly. After 20 minutes data rates started to raise again. Unfortunately, even after the next hour, data rate did not reach stable baseline, but approached it with a 10 MB/s fluctuations. The high (around 1.5 MB/s) outgoing data from CP in first 15 minutes was probably caused by the exchange of information to/from PP, opening new sockets etc.

4.3. OVERVIEW OF RESULTS

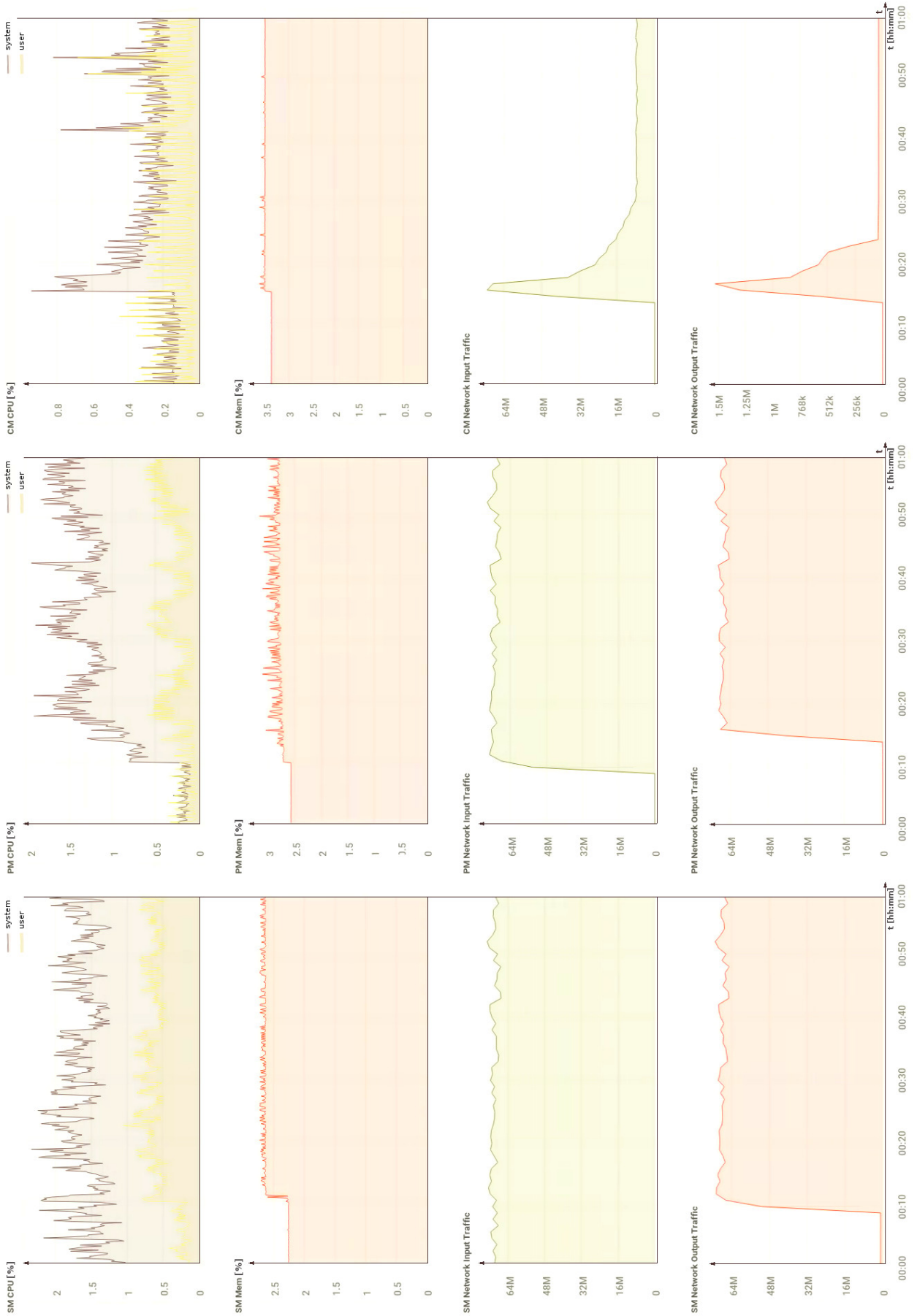


Figure 4.6: Results of case 1 of test 2 (CPs use same key).

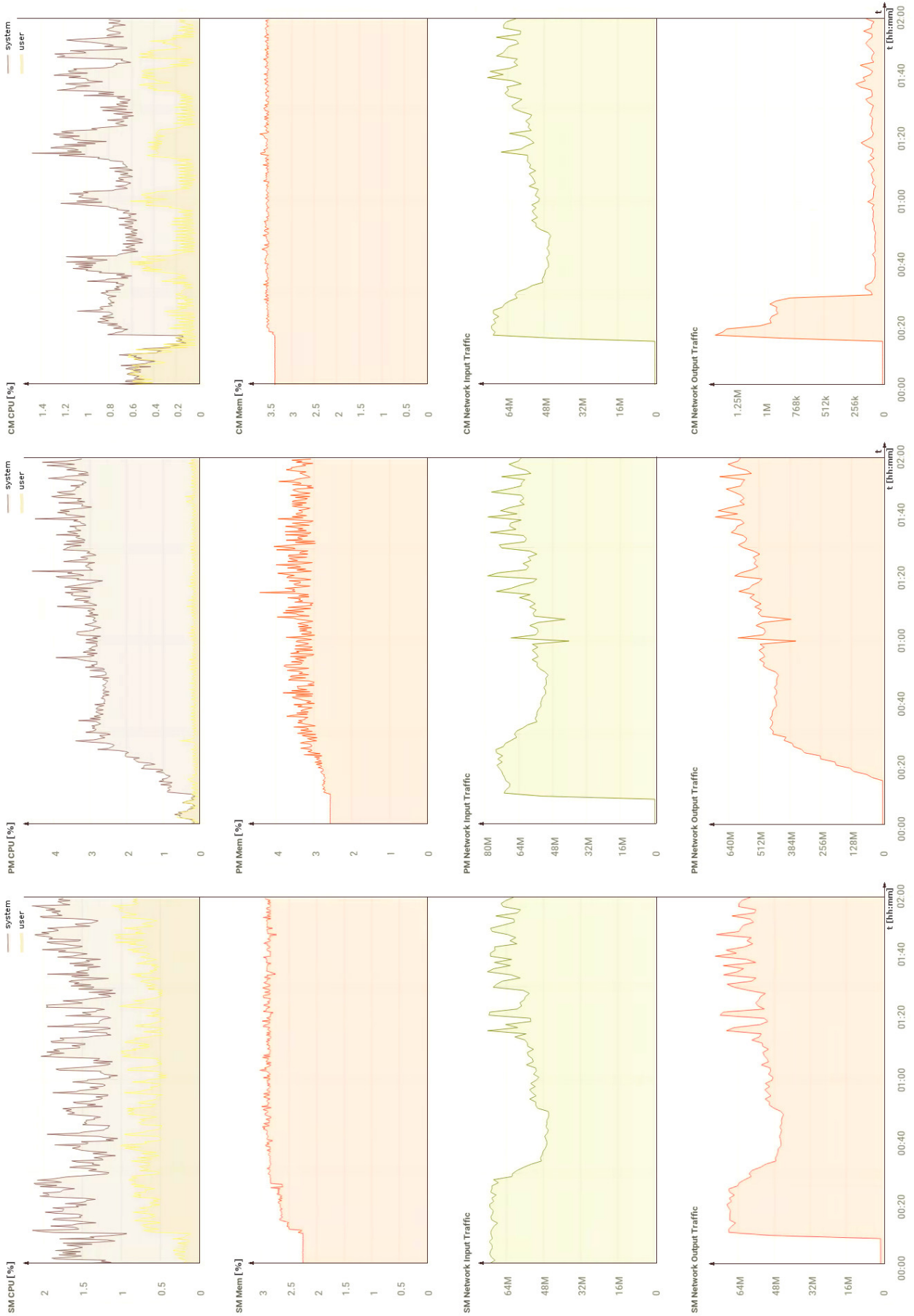


Figure 4.7: Results of case 2 of test 2 (CPs using unique keys).

Test 3 – multiple sources

The third test shows behavior of programs dealing with higher data rate. Test was performed in two variations – without any CP and with one CP connected. Tests lasted an hour and began exactly as previous one with the ALICE experiment simulation running alone. Then one of each type of programs SP, PP and CP (only in second case)) were started. After a few minutes one SP would connect every 2 minutes. For the rest of the time all 10 SPs were sending data to PP. Results of this tests are shown in figures 4.8, 4.9 and 4.10. The SP which has measurement sends around 65 MB/s, but temporary data rate from other machines ranges between 50 MB/s and 75 MB/s.

In first case of test within the first four SPs connections the data rate of each SP was stable at around 65 MB/s. After fifth SP connected the data rate of the ALICE experiment simulation started to drop and it stabilized only after connecting the tenth SP. The data rate incoming to PP was increasing until the seventh SP connected. After that input data rate of a PP was stable at around 355 MB/s instead of near 600 MB/s and output data rate of a SP at 42 MB/s instead of the initial 65 MB/s. The CPU usage of PM was behaving exactly as the input network of this machine, but the memory usage was increasing until the last SP connected. Until the end of the test, the speeds were stable but at a lower level than the initial one. The outgoing data rate of a PP were increasing until the last SP connected as well. This value is associated with sending a heartbeat to each of the SP. In this case inside PP there were only two single threads working – the main which was waiting for a new client, so in fact was doing nothing, because no CP had connected and a thread which was receiving events from SP and dropping them immediately, so the data rate decrease was probably caused by the network bandwidth or its instability rather than program operation.

This test variation was repeated except that SPs were connecting every 5 minutes. Only 9 SPs took part in the test, but receiving full speed of 600 MB/s of data together (every SP was sending around 65-70 MB/s of data). The results are exactly the same, but are more visible. At around 350 MB/s of PP ingoing data rate the data rate of the ALICE experiment simulation started to drop. It was exactly at the moment when sixth SP has connected. More SPs did not raised that value. As in the previous test all values established at lower levels after the last SP connected. The results are shown in figure 4.9.

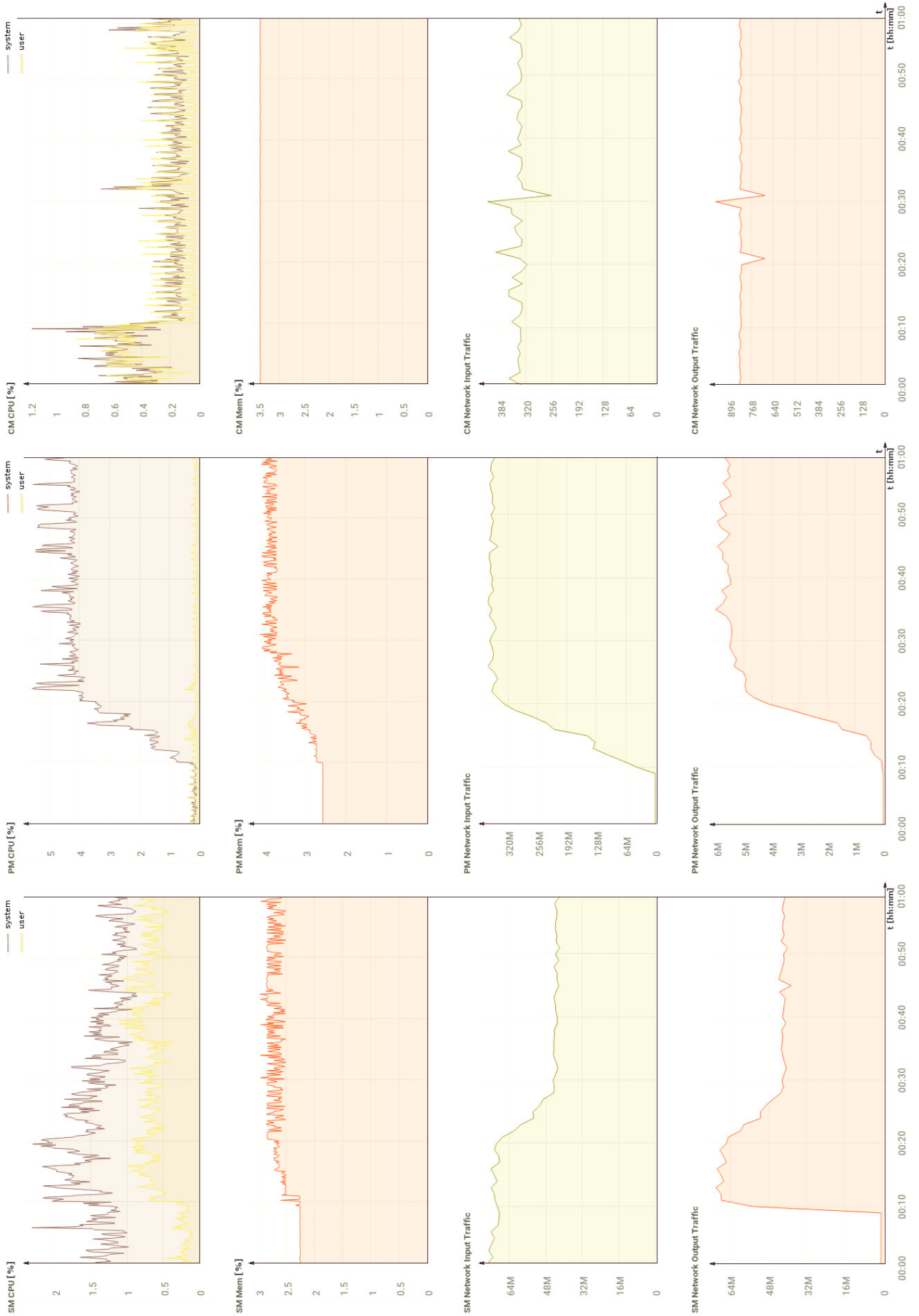


Figure 4.8: Results of case 1 of test 3 (without any CP connected).

4.3. OVERVIEW OF RESULTS

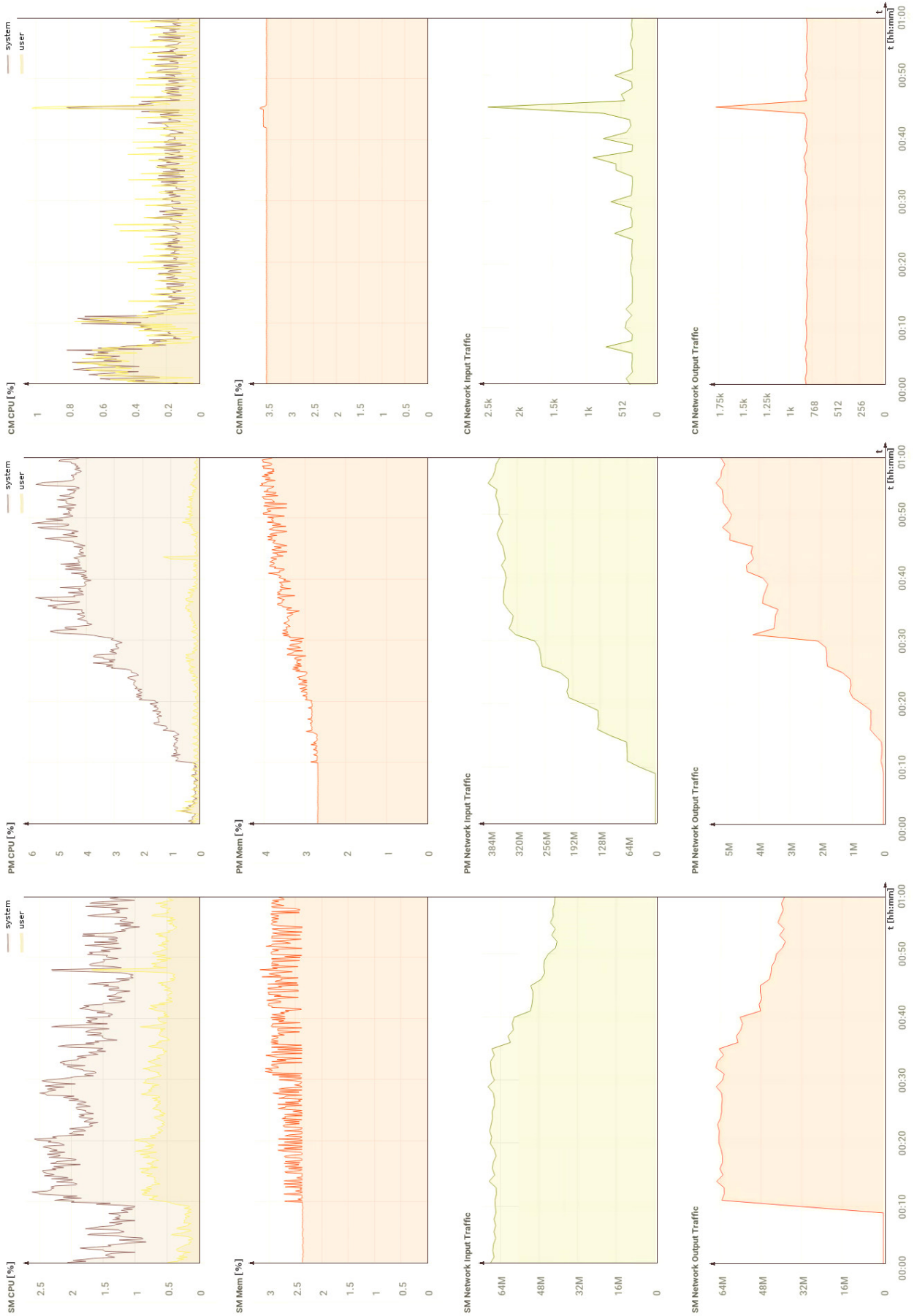


Figure 4.9: Repeated case 1 of test 3 (without any CP connected).

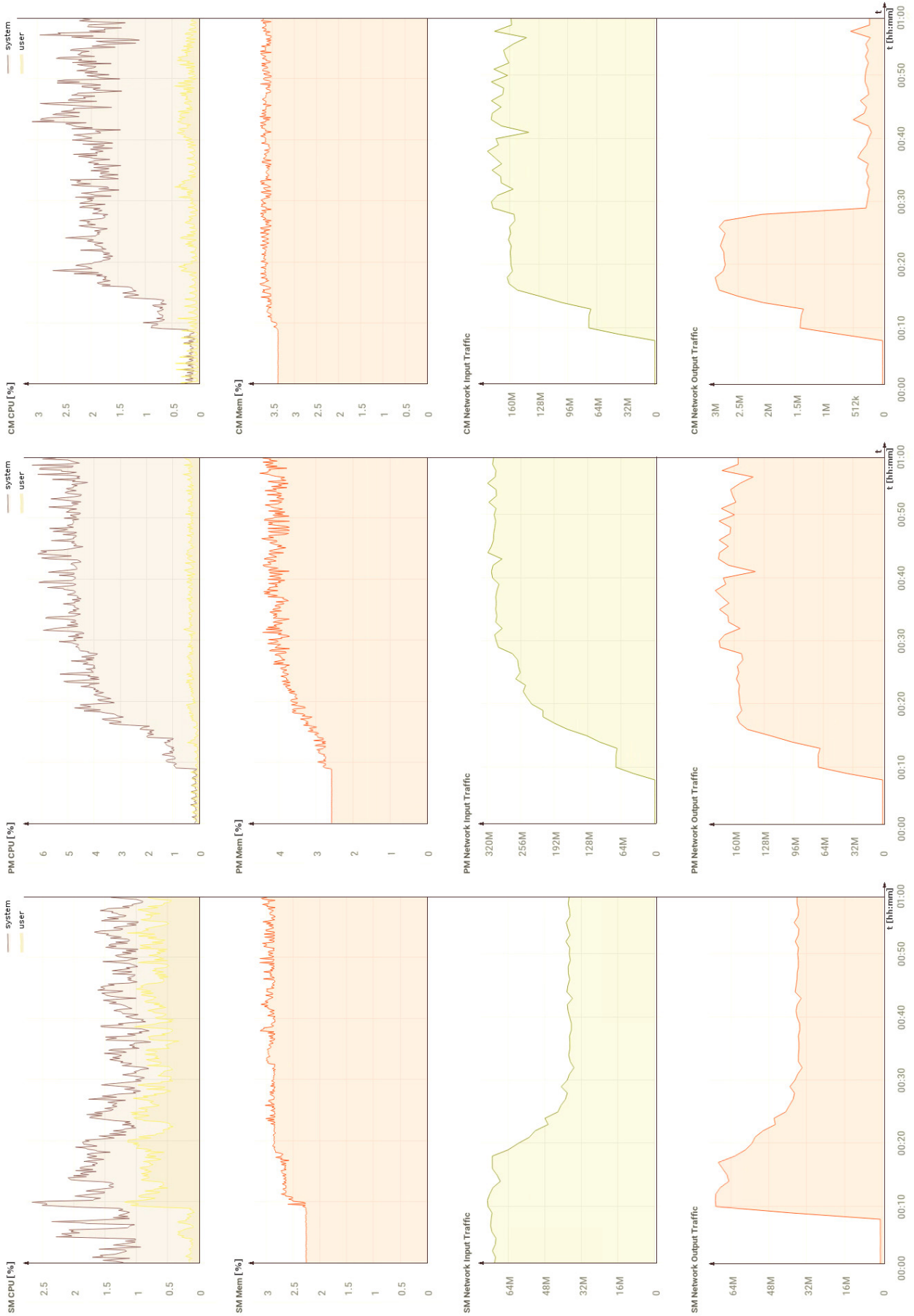


Figure 4.10: Results of case 2 of test 3 (with one CP connected).

In the second version of this test the results are similar, but the data rate values varies. In the first part of the test the same result as in the first test is visible. First ten minutes only the ALICE experiment simulation is working, then for the next 5 minutes only one of a kind of programs is running. The data rate of the ALICE experiment simulation is stable at around 70 MB/s. Full data speed is transported through PP to a CP. Then, every two minutes, next SP were connecting. During connecting first two SPs (3 in total) the ingoing and outgoing data rate of a SP was quite stable at around 70 MB/s, while the ingoing as well as outgoing data rate of a PP was increasing with the difference that the output grew more slowly. When the fourth machine was connected, the data rate of a single SP (and the ALICE experiment simulation) began to drop, while input data rate of a PP was still increasing, but the output has reached a quite constant level of around 160 MB/s. After all SPs connected the data rate of all machines has stabilized, but on much lower values than expected. The PP input data rate remained at level around 310 MB/s (45 MB/s slower than in previous test) and the output data rate of a SP at around 36 MB/s (6 MB/s slower than in previous test).

Test 4 – full architecture

Last test shows performance of the full set of programs. It was performed as a combination of two previous tests – case 1 of test 3 and case 2 of test 2 (with CPs using unique keys). It lasted two hours and began as every other previous test – from only the ALICE experiment simulation running. After that the same scenario as during the first case of the third test was performed using all 10 SPs. Next step from test 2 were carried out – every 2 minutes one more client was connecting. Results are shown in figure 4.11.

Full architecture test shows that a problem somewhere has occurred. Nodes from testing grid are equipped with a 10 Gbit/s network card interface. 10 Gbit/s corresponds to 1.25 GB/s. So theoretically, the traffic on each card can reach this maximum. However, this is only the theoretical maximum data transfer speed for two-way communication under ideal conditions. In addition to the type of network adapter, transmission speed is also affected by factors such as network devices, network cables, hard drive speed and bus speed. In real conditions it is almost impossible to reach that value. For example, recently two scientists from CERN and University of Michigan achieved the best TCP

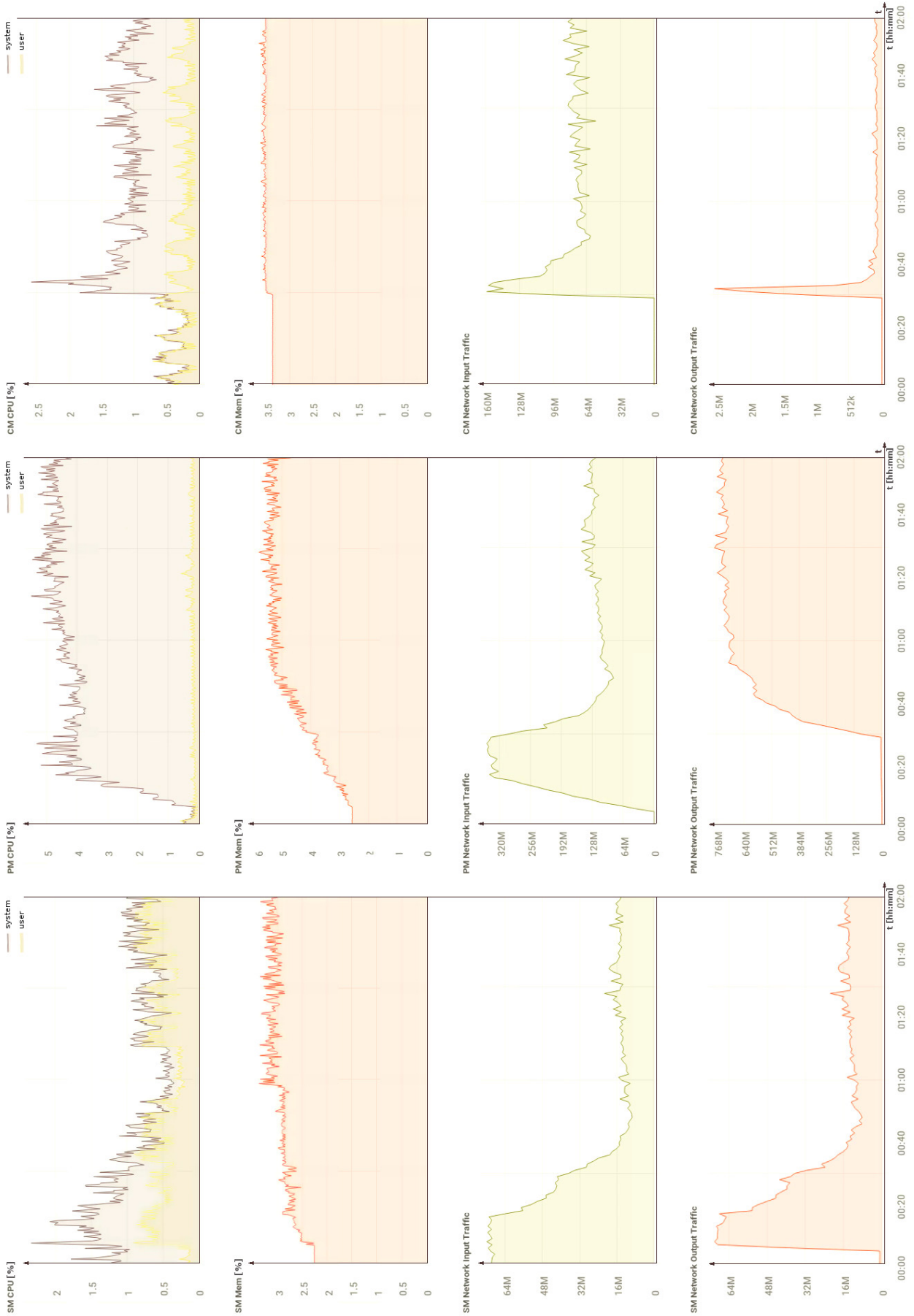


Figure 4.11: Results of test 4 (CPs using unique keys).

single stream and single CPU core results for LAN (NIC 100 Gbit/s) for the WLCG as slightly above the 22% for normal NIC conditions and less than 26% for NIC with tuning. Their results will be showed on HEPiX Autumn 2019 Workshop in Amsterdam [78]. Considering their results the behavior of the proxy-based monitoring library benchmarks reached similar values. The single incoming steam of PP reached its maximum value at around 350 MB/s which corresponds to about 28% of NIC declared speed. The tuning is made by the library so this result seems quite adequate. The PP incoming value was dropping also while CPs were connecting. The maximum outgoing value that PP reached was around 700 MB/s. If we add the incoming value of around 130 MB/s, we get the result of up to 900 MB/s which is the value closer to the declared NIC speed. Individual speeds relative to the entire experiment were most likely also related to the network load balancing.

Final test – adjusted amount of machines

The final test was done exactly as previous test, but with fewer machines. The amount of machines was determined after all tests were done. It was performed to show working architecture bypassing external limits from previous tests. This test was performed faster – in a way closer to the real conditions of the ALICE experiment simulation. It started as always from the ALICE experiment simulation working alone, then monitoring by proxy project consisting of 3 SPs and 1 PP was started simultaneously. After a few minutes CPs were connecting – one every 5 minutes. This condition was maintained until the end of the test. Results are shown in figure 4.12.

In the final test one can see that the monitoring by proxy project is doing well. The data rate from the ALICE experiment simulation did not drop. There is only small difference between incoming and outgoing stream of PP. This case was tested separately and the result showed that it was also caused by network factors. CP is able to receive faster as well as PP is able to send faster, but the network connection is the bottleneck. The PP acted correctly and dropped some events, so CP received slightly less data.

There were no more tests performed as the network limitations is not the subject of this thesis. To examine the performance limits of created programs, these tests must be repeated in a test environment consisting of machines equipped with faster NICs.

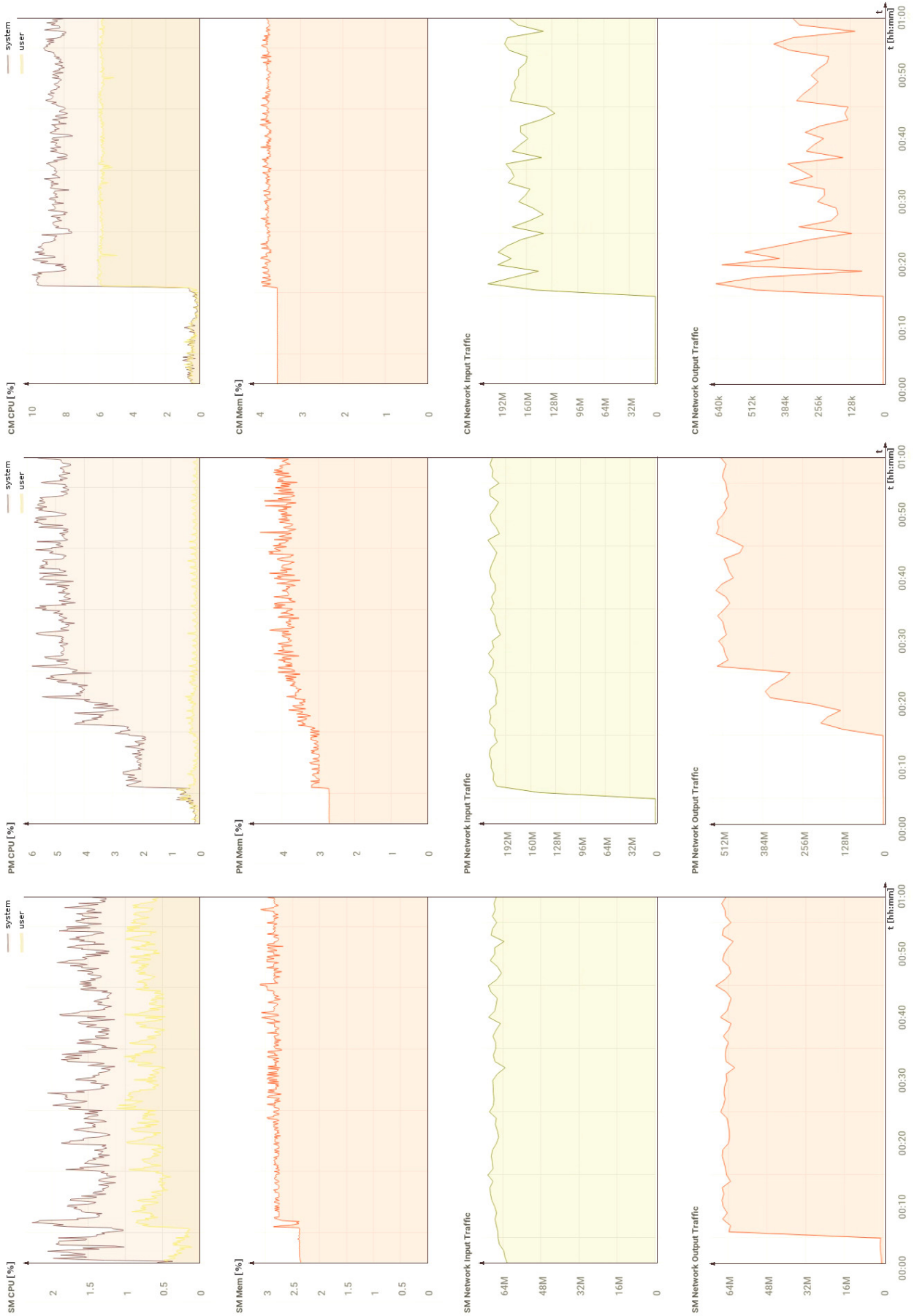


Figure 4.12: Results of final test 4.

4.4 Tests of programs source code correctness

The code correctness test was performed using the Valgrind framework [79]. Valgrind is a programming tool for detecting memory issues as well as thread bugs and other code defects. It checks both stack and heap memory, and also has the function of predicting program behavior in various branches. The tests were made for every piece of code. The `-leak-check=full` parameter was used. An example is presented in listing 4.4.1.

Listing 4.4.1: Valgrind parameters used during test

```
valgrind --leak-check=full --track-origin=yes ./SP
```

Test results with its interpretation for each of the programs are presented below.

SP test

The SP test results are presented in listing 4.4.2.

Listing 4.4.2: Test of SP source code

```
HEAP SUMMARY:
  in use at exit: 79,409 bytes in 64 blocks
  total heap usage: 51,757 allocs, 51,693 frees, 545,248,590 bytes
    allocated

LEAK SUMMARY:
  definitely lost: 0 bytes in 0 blocks
  indirectly lost: 0 bytes in 0 blocks
  possibly lost:   0 bytes in 0 blocks
  still reachable: 80,855 bytes in 106 block
  suppressed:     0 bytes in 0 blocks

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 10 from 6)
```

The test result shows that there are no memory leaks or any serious code errors that could affect the correct operation of the program. There are a few bugs visible in the “still reachable” category, but they are caused by functions from “dateDb” and EML libraries that are used by this program. The number of lost bytes is constant regardless of how long and under what conditions the program has been running. Therefore, they are considered harmless. The only way to get rid of them is to correct these libraries source code.

PP test

The PP test results are presented in listing 4.4.3.

Listing 4.4.3: Test of PP source code.

HEAP SUMMARY:

```
in use at exit: 78,040 bytes in 22 blocks
total heap usage: 5,032 allocs, 5,010 frees, 2,814,887 bytes allocated
```

LEAK SUMMARY:

```
definitely lost: 0 bytes in 0 blocks
indirectly lost: 0 bytes in 0 blocks
possibly lost:   0 bytes in 0 blocks
still reachable: 78,040 bytes in 22 blocks
suppressed:     0 bytes in 0 blocks
```

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 10 from 6)

As in the case of the SP code, the PP source code test showed no major errors. As in the previous test, several errors from the “still reachable” category are visible, but as before, they are caused by the use of the “dateDb” library, which is used by this program. The number of lost bytes is constant regardless of how long and under what conditions the program has been running. Therefore, they are considered harmless. The solution is exactly the same as in the previous test.

Proxy-based monitoring library test

The correctness of the proxy-based monitoring library code is checked indirectly by checking the SP, PP and CP programs, as all of them use this library for normal operation. The “still reachable” category errors, which come from the “dateDb” library (shown in tests of other programs) just come through this proxy-based monitoring library, as the functions which cause these errors are placed exactly here.

CP test

The CP test was performed on the example code which is presented in section 3.5.4. The results are presented in listing 4.4.4.

Listing 4.4.4: Test of CP source code.

HEAP SUMMARY:

in use at exit: 78,086 bytes in 24 blocks
total heap usage: 1,584 allocs, 1,560 frees, 12,084,104 bytes allocated

LEAK SUMMARY:

definitely lost: 0 bytes in 0 blocks
indirectly lost: 0 bytes in 0 blocks
possibly lost: 0 bytes in 0 blocks
still reachable: 78,040 bytes in 22 blocks
suppressed: 0 bytes in 0 blocks

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 10 from 6)

This result is exactly the same as in the case of PP source code. The reason and solution are also the same as in that case. The number of lost bytes is constant regardless of how long and under what conditions the program has been running. Therefore, they are considered harmless.

Chapter 5

Summary and conclusions

The LHC with its experiments is a great place to create modern projects in the field of computer science. Huge computer architecture as well as variety of IT issues allow to study wide range of the IT topics, from programming to networking.

ALICE as one of the four main experiments located on the LHC ring is full of IT solutions. The data, collected by this project, is qualified as being the so-called “big data”. However, the time spent on hardware construction and equipment configuration is sometimes equal or even greater than the time in which the expected quality data is collected. So that supporting tools such as an example a live data monitoring system are required for efficient data quality checking and equipment configuration.

In this work the proposal for an upgrade of a part of the ALICE experiment data quality software – the monitoring tool, is presented. The existing library works directly on the experiment machines and has some limitations which are required to be improved. The new solution is based on the addition of proxy servers between the experiment machines and the clients. The ZMQ library was used for data transmission along their entire route.

As a result of the proposed solution, a set of computer programs was created. The proposed design met the initial assumptions. It has many advantages over previous solution. First of all as a consequence of the proxy architecture end users are not connecting directly to the experiment machines. This reduces the risk of unexpected actions and makes the monitoring system interact with the experiment machines in a more structured way. Additionally, one client can use the prepared library in multi-part programs and not receive duplicated data. This was achieved by using keys identifying separate fragments of the one monitoring program. Furthermore, there is no limit on the size of the event, so now end users can obtain data of even the largest collision. New solution can easily replace the existing monitoring library due to the fact that most of new functions have

their old format counterpart.

This solution, in addition to fulfilling its original role, which it has been performed for, can be successfully used in other similar physical experiments as well as in other areas of economy, wherever real-time data transfer between different platforms is required, also using various programming languages.

For the purpose of this thesis the test environment of the ALICE experiment for Run 2 conditions was created on machines located at Warsaw University of Technology. Due to lack of a hardware trigger the COLE which creates ALICE-like events was used. The prepared prototype has been tested in different use cases. Tests confirmed the correct operation of the project. Unfortunately, full performance tests, which would simulate an environment close to the conditions of a fractional part of the real ALICE experiment, were impossible due to the weak network interface controllers. To carry out more appropriate tests better machine hardware is needed. To meet the limitations of slower network interface controllers, this solution can be adapted later to the specifics of the network (for example, by duplicating data streams) or to the specifications of machines on which the solution will ultimately run. The target environment for the created prototype will be the Run 3 – the next running period of the LHC, which will start in the first quarter of the year 2021.

The author considers as her own achievements:

- installation and configuration, on test machines, the whole Run 2 ALICE experiment architecture, using a COLE emulator which served as a trigger system as well as data production,
- development of a new concept of the monitoring system, based on proxy servers and using the ZMQ library for data transfer, which retains all existing monitoring capabilities and extends them with additional functionalities,
- creation of a set of working function prototypes that fit into this idea, which can easily replace the existing monitoring library and can be used in other similar physical experiments as well as in other areas of economy.

Bibliography

- [1] J. Niedziela and B. von Haller, *Event visualisation in ALICE - current status and strategy for Run 3*, Journal of Physics: Conference Series, vol. 898, pp. 1–8, 2017.
- [2] CERN, *Upgrade of the Online-Offline computing system Technical Design Report*, June 2015. ALICE Run 3 Project, CERN-LHCC-2015-006, ALICE-TDR-019.
- [3] *About CERN*, <https://home.cern/about>. (visited on 12/10/2017).
- [4] W. Yao *et al.*, (Particle Data Group), *Review of particle physics*, Journal of Physics G: Nuclear and Particle Physics, vol. 33, 7 2006.
- [5] H. P. Nilles, *Supersymmetry, Supergravity and Particle Physics*, Phys. Rept., vol. 110, pp. 1–162, 1984.
- [6] G. Jungman, M. Kamionkowski, and K. Griest, *Supersymmetric dark matter*, Physics Reports, vol. 267, no. 5, pp. 195 – 373, 1996.
- [7] M. S. Edmund J. Copeland and S. Tsujikawa, *Dynamics of Dark Energy*, International Journal of Modern Physics D, vol. 15, no. 11, pp. 1753 – 1935, 2006.
- [8] P. Foka and M. A. Janik, *An overview of experimental results from ultra-relativistic heavy-ion collisions at the CERN LHC: Bulk properties and dynamical evolution*, Reviews in Physics, vol. 1, pp. 154 – 171, 2016.
- [9] P. Foka and M. A. Janik, *An overview of experimental results from ultra-relativistic heavy-ion collisions at the CERN LHC: Hard probes*, Reviews in Physics, vol. 1, pp. 172 – 194, 2016.
- [10] E. Shuryak, *Quark-gluon plasma and hadronic production of leptons, photons and psions*, Physics Letters B, vol. 78, no. 1, pp. 150 – 153, 1978.
- [11] P. Braun-Munzinger, V. Koch, T. Schäfer, and J. Stachel, *Properties of hot and dense matter from relativistic heavy ion collisions*, Physics Reports, vol. 621, pp. 76 – 126, 2016. Memorial Volume in Honor of Gerald E. Brown.

- [12] *CERN faq: LHC the guide*, 2017.
- [13] *CERN member states*, <https://home.cern/about/member-states>. (visited on 20/11/2018).
- [14] H. R. Department, *CERN Personnel Statistics 2016*, March 2017.
- [15] T. G. Pickavance, *Synchrocyclotrons, and the CERN 600 MeV machine*, Nuovo Cim., vol. 2, no. 1suppl, pp. 403–412, 1955.
- [16] C. Fabjan, T. Taylor, D. Treille, and H. Wenninger, *Technology Meets Research*. WORLD SCIENTIFIC, 2017.
- [17] L. Evans and P. Bryant, *LHC Machine*, Journal of Instrumentation, vol. 3, pp. S08001–S08001, aug 2008.
- [18] *CERN Badge Logo*, <https://design-guidelines.web.cern.ch/guidelines/logo>. (visited on 12/10/2017).
- [19] *CERN Logo*, <https://design-guidelines.web.cern.ch/guidelines/outline-logo>. (visited on 12/10/2017).
- [20] *The birth of the web*, <https://home.cern/topics/birth-web>. (visited on 18/06/2018).
- [21] *Forty years since the first PET image at CERN*, <https://home.cern/news/news/knowledge-sharing/forty-years-first-pet-image-cern>. (visited on 18/06/2018).
- [22] *WLCG Worldwide LHC Computing Grid*, <http://wlcg-public.web.cern.ch/about>. (visited on 20/11/2018).
- [23] *The Nobel Prize*, <https://www.nobelprize.org/>. (visited on 18/06/2018).
- [24] *The CERN accelerator complex*, <http://cds.cern.ch/record/2197559>. (visited on 20/11/2018).
- [25] G. Aad *et al.*, (ATLAS), *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST, vol. 3, p. S08003, 2008.
- [26] S. Chatrchyan *et al.*, (CMS), *The CMS Experiment at the CERN LHC*, JINST, vol. 3, p. S08004, 2008.

- [27] K. Aamodt *et al.*, (ALICE), *The ALICE experiment at the CERN LHC*, JINST, vol. 3, p. S08002, 2008.
- [28] A. A. Alves, Jr. *et al.*, (LHCb), *The LHCb Detector at the LHC*, JINST, vol. 3, p. S08005, 2008.
- [29] *CERN experiments*, <https://home.cern/about/experiments>. (visited on 20/11/2018).
- [30] *ALICE*, <http://aliceinfo.cern.ch/Public/Welcome.html>. (visited on 20/11/2018).
- [31] *ALICE*, <https://home.cern/science/computing/processing-what-record>. (visited on 20/11/2018).
- [32] CERN, *Upgrade of the ALICE Experiment Letter of Intent*, September 2012. ALICE Run 3 Project, ALICE Letter of Intent, CERN-LHCC-2012-012, ALICE-UG-001.
- [33] P. L. Rocca and F. Riggi, *The upgrade programme of the major experiments at the Large Hadron Collider*, Journal of Physics: Conference Series, vol. 515, p. 012012, may 2014.
- [34] B. Schmidt, *The High-Luminosity upgrade of the LHC: Physics and Technology Challenges for the Accelerator and the Experiments*, Journal of Physics: Conference Series, vol. 706, p. 022002, apr 2016.
- [35] *Longer term LHC schedule*, <https://lhc-commissioning.web.cern.ch/lhc-commissioning/schedule/LHC-long-term.htm>. (visited on 20/11/2018).
- [36] F. Carena, W. Carena, S. Chapeland, V. C. Barroso, F. Costa, E. Dénes, R. Divià, U. Fuchs, G. Simonetti, C. Soós, A. Telesca, P. V. Vyvre, and B. V. Haller, *ALICE DAQ and ECS Manual*. CERN, December 2010. ALICE DAQ Project, ALICE ECS Project, ALICE Internal Note/DAQ, ALICE-INT-2010-001.
- [37] R. Brun, P. Buncic, F. Carminati, A. Morsch, F. Rademakers, and K. Safarik, *Computing in ALICE*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, vol. 502, no. 2-3, pp. 339–346, 2003.

- [38] F. Carena, W. C. S. Chapeland, V. C. Barroso, F. Costa, E. Dénes, R. Divià, U. Fuchs, A. Grigore, T. Kiss, G. Simonetti, C. Soós, A. Telesca, P. V. Vyvre, and B. von Haller, (ALICE), *The ALICE data acquisition system*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, vol. 741, pp. 130–162, 21 March 2014.
- [39] *ALICE Off-line Project*, <http://alice-offline.web.cern.ch/>. (visited on 20/11/2018).
- [40] P. Saiz, L. Aphecetche, P. Bunčić, R. Piskač, J.-E. Revsbech, V. Šego, A. Collaboration, *et al.*, *AliEn: ALICE environment on the GRID*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, vol. 502, no. 2-3, pp. 437–440, 2003.
- [41] *ROOT Data Analysis framework*, <https://root.cern.ch/>. (visited on 20/11/2018).
- [42] R. Brun, F. Bruyant, M. Maire, A. C. McPherson, and P. Zanmarini, *GEANT 3: user's guide Geant 3.10, Geant 3.11; rev. version*. Geneva: CERN, 1987.
- [43] *Geant4 - a simulation toolkit*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, vol. 506, no. 3, pp. 250 – 303, 2003.
- [44] T. Sjostrand, *High-energy physics event generation with PYTHIA 5.7 and JETSET 7.4*, Comput. Phys. Commun., vol. 82, pp. 74–90, 1994.
- [45] *ALICE Technical Design Report on Computing*, Journal of Instrumentation. CERN-LHCC-2005-018.
- [46] *The Worldwide LHC Computing Grid*, <https://home.cern/science/computing/worldwide-lhc-computing-grid>. (visited on 20/11/2018).
- [47] G. Bird, *LHC Computing (WLCG): Past, present, and future*, 2016.
- [48] A. K. A. Klimentov, M. Grigorieva and A. Zarochentsevb, (ALICE), *BigData and computing challenges in high energy and nuclear physics*, Journal of Instrumentation, vol. 12, pp. 1–8, 29 June 2017.

- [49] T. Anticic, V. Barroso, F. Carena, W. Carena, S. Chapeland, O. Cobanoglu, E. Dénes, R. Divià, U. Fuchs, T. Kiss, I. Makhlyueva, F. Ozok, F. Roukoutakis, K. Schossmaier, C. Soós, P. Vande Vyvre, and S. Vergara, *Commissioning of the ALICE data acquisition system*, Journal of Physics Conference Series, vol. 119, pp. 2006–, 07 2008.
- [50] *Technical Design Report for the Upgrade of the ALICE Inner Tracking System*, Tech. Rep. CERN-LHCC-2013-024. ALICE-TDR-017, CERN, Nov 2013.
- [51] *Upgrade of the ALICE Time Projection Chamber*, Tech. Rep. CERN-LHCC-2013-020. ALICE-TDR-016, Oct 2013.
- [52] *Addendum to the Technical Design Report for the Upgrade of the ALICE Time Projection Chamber*, Tech. Rep. CERN-LHCC-2015-002. ALICE-TDR-016-ADD-1, Feb 2015.
- [53] *Technical Design Report for the Muon Forward Tracker*, Tech. Rep. CERN-LHCC-2015-001. ALICE-TDR-018, CERN, Jan 2015.
- [54] *Addendum of the Letter of Intent for the upgrade of the ALICE experiment : The Muon Forward Tracker*, Tech. Rep. CERN-LHCC-2013-014. LHCC-I-022-ADD-1, CERN, Geneva, Aug 2013. Final submission of the presetn LoI addendum is scheduled for September 7th.
- [55] P. Antonioli, A. Kluge, and W. Riegler, *Upgrade of the ALICE Readout and Trigger System*, Tech. Rep. CERN-LHCC-2013-019. ALICE-TDR-015, Sep 2013.
- [56] P. Cortese, F. Carminati, C. W. Fabjan, L. Riccati, and H. de Groot, *ALICE computing: Technical Design Report*. Technical Design Report ALICE, Geneva: CERN, 2005. Submitted on 15 Jun 2005.
- [57] *ALICE O2 Project*, <https://alice-o2-project.web.cern.ch/>. (visited on 20/11/2018).
- [58] J. Carlson *et al.*, *Nuclear Physics Exascale Requirements Review: An Office of Science review sponsored jointly by Advanced Scientific Computing Research and Nuclear Physics, June 15 - 17, 2016, Gaithersburg, Maryland*,

- [59] B. Marian, F. Ivan, H. Nicholas, L. Hector, L. Daniel, S. Miroslav, and W. Denis, *LEMON - LHC Era Monitoring for Large-Scale Infrastructures*, Journal of Physics: Conference Series, vol. 331, 12 2011.
- [60] S. Chapeland, F. Carena, W. Carena, V. Chibante Barroso, F. Costa, E. Denes, R. Divia, U. Fuchs, A. Grigore, C. Ionita, C. Delort, G. Simonetti, C. Soos, A. Telesca, P. Vande Vyvre, and B. Von Haller, (ALICE), *The ALICE DAQ infoLogger*, J. Phys.: Conf. Ser., vol. 513, p. 012005. 7 p, 2014.
- [61] S. Chapeland, F. Carena, W. Carena, V. Chibante Barroso, F. Costa, E. Denes, R. Divia, U. Fuchs, A. Grigore, G. Simonetti, C. Soos, A. Telesca, P. Vande Vyvre, and B. von Haller, (ALICE Collaboration), *Orthos, an alarm system for the ALICE DAQ operations*, J. Phys.: Conf. Ser., vol. 396, p. 012013. 9 p, 2012.
- [62] V. Altini, F. Carena, W. Carena, S. Chapeland, V. Chibante Barroso, F. Costa, R. Divia, U. Fuchs, I. Makhlyueva, F. Roukoutakis, K. Schossmaier, C. Soos, P. Vande Vyvre, and B. Von Haller, (Alice Collaboration), *The ALICE Electronic Logbook*, J. Phys.: Conf. Ser., vol. 219, p. 022027, 2010.
- [63] F. Roukoutakis, S. Chapeland, and O. Cobanoglu, *The ALICE-LHC Online data quality monitoring framework: Present and future*, Nuclear Science, IEEE Transactions on, vol. 55, pp. 379 – 385, 03 2008.
- [64] B. von Haller, F. Roukoutakis, S. Chapeland, V. Altini, F. Carena, W. Carena, V. C. Barroso, F. Costa, R. Divià, U. Fuchs, I. Makhlyueva, K. Schossmaier, C. Soós, P. V. Vyvre, and the Alice collaboration, *The ALICE data quality monitoring*, Journal of Physics: Conference Series, vol. 219, p. 022023, apr 2010.
- [65] B. von Haller, F. Roukoutakis, S. Chapeland, V. Altini, F. Carena, W. Carena, V. Chibante Barroso, F. Costa, R. Divia, U. Fuchs, I. Makhlyueva, K. Schossmaier, C. Soos, and P. Vande Vyvre, (Alice Collaboration), *The ALICE data quality monitoring*, J. Phys.: Conf. Ser., vol. 219, p. 022023, 2010.
- [66] F. Carena, W. Carena, S. Chapeland, R. Divià, I. Mkhlyueva, J. C. Marin, K. Schossmaier, C. Soos, P. Van de Vyvre, and A. Vascotto, (ALICE Collaboration), *The ALICE Data-Acquisition Software Framework DATE V5*, 2006.

- [67] S. Chapeland, F. Carena, W. Carena, V. Chibante Barroso, F. Costa, E. Denes, R. Divia, U. Fuchs, A. Grigore, G. Simonetti, C. Soos, A. Telesca, P. Vande Vyvre, and B. von Haller, (ALICE Collaboration), *Experiences and evolutions of the ALICE DAQ detector algorithms framework*, J. Phys.: Conf. Ser., vol. 396, p. 012012. 10 p, 2012.
- [68] S. Chapeland, V. Altini, F. Carena, W. Carena, V. Chibante Barroso, F. Costa, R. Divia, U. Fuchs, I. Makhlyueva, F. Roukoutakis, K. Schossmaier, C. Soos, P. Vande Vyvre, and B. von Haller, (Alice Collaboration), *Online processing in the ALICE DAQ The detector algorithms*, J. Phys.: Conf. Ser., vol. 219, p. 022004, 2010.
- [69] C. E. Vandoni, *Visualization of Scientific Data for high Energy physics: basic Architecture and a Case Study*, in *Visualization in Scientific Computing* (W. H. M. Grave, Y. le Louis, ed.), ch. 5, pp. 43–52, Springer-Verlab, 1994.
- [70] *ZeroMQ official Website*, <https://zeromq.org/>. (visited on 03/02/2016).
- [71] P. Hintjens, *0MQ - The Guide*, <http://zguide.zeromq.org/page:all>. (visited on 03/02/2016).
- [72] *Scientific Linux Cern 6*, <http://linux.web.cern.ch/linux/scientific6/>. (visited on 10/12/2018).
- [73] *ALICE Data Acquisition Website*, <https://alice-daq.web.cern.ch/>. (visited on 20/11/2018).
- [74] P. Hintjens, *ZeroMQ*. O'Reilly Media Inc., 2013.
- [75] *ZeroMQ 4.1 API*, http://api.zeromq.org/4-1:_start. (visited on 21/11/2016).
- [76] *Definition: Benchmark*, <http://searchcio.techtarget.com/definition/benchmark>.
- [77] *Influxdata Website*, <https://www.influxdata.com>. (visited on 08/03/2019).
- [78] M. Babik and S. McKee, *WLCG/OSG Network Activities, Status and Plans*, HEPiX Autumn 2019 Workshop, 2019.
- [79] *Valgrind official Website*, <http://valgrind.org/>. (visited on 08/03/2019).

Appendices

Appendix A

Source Program source code

SP.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include "monitor.h"
#include "monitoringbyproxy.h"

typedef struct socket_info{
    void* socket;
    char* socketAddr;
    void* socketHb;
    char* socketHbAddr;
    pthread_t threadHB;
    int tnum;
    int alive;
}socket_info;

int HB_TIMEOUT;
int HB_SND_TIMEOUT;
int REC_SLEEP;
int HB_DELAY;
void* CONTEXT;
int ALIVE;
socket_info *SOCKETS;

int findFirstAlive(int pstart, int pamount)
{
    int control = 0, pret = pstart;
    if(pstart == -1) pret = 0;
    do{
        pret++; if(pret>=pamount) pret = 0;
        control++;
    }while(SOCKETS[pret].alive!=1 && control<=pamount);
    if(SOCKETS[pret].alive!=1) pret = -1;
    return pret;
}
```

```

void *heartbeatThread(void *args)
{
    int snum = *((int *) args);
    SOCKETS[snum].alive = 0;
    SOCKETS[snum].socketHb = zmq_socket(CONTEXT, ZMQ_REQ);
    if(SOCKETS[snum].socketHb!=NULL)
    {
        zmq_setsockopt(SOCKETS[snum].socketHb, ZMQ_RCVTIMEO, &HB_TIMEOUT,
            sizeof(HB_TIMEOUT));
        zmq_setsockopt(SOCKETS[snum].socketHb, ZMQ_SNDTIMEO,&HB_SND_TIMEOUT,
            sizeof(HB_SND_TIMEOUT));
        int rc = zmq_connect(SOCKETS[snum].socketHb, SOCKETS[snum].socketHbAddr);
        if(rc==0)
        {
            printf("SP: hbt: Connected to heartbeat port for proxy %s\n",
                SOCKETS[snum].socketHbAddr);
            zmq_msg_t hbmsg;
            zmq_msg_t rmsg;
            int drc = 0, crc = 0;

            while(ALIVE==1)
            {
                if(rc== -1) //reconnection
                {
                    int linger = 0;
                    zmq_setsockopt (SOCKET_S[snum].socketHb, ZMQ_LINGER, &linger, sizeof
                        (linger));
                    rc = zmq_disconnect(SOCKET_S[snum].socketHb,
                        SOCKETS[snum].socketHbAddr);
                    if(rc!= -1) drc = 1;
                    rc = zmq_close(SOCKET_S[snum].socketHb); //to prevent deadlock
                    if(rc!= -1) crc = 1;
                    sleep(REC_SLEEP);
                    if(rc!= -1)
                    {
                        SOCKETS[snum].socketHb = zmq_socket(CONTEXT, ZMQ_REQ);
                        if (SOCKET_S[snum].socketHb!=NULL)
                        {
                            zmq_setsockopt(SOCKET_S[snum].socketHb, ZMQ_RCVTIMEO, &HB_TIMEOUT,
                                sizeof(HB_TIMEOUT));
                            zmq_setsockopt(SOCKET_S[snum].socketHb,
                                ZMQ_SNDTIMEO,&HB_SND_TIMEOUT, sizeof(HB_SND_TIMEOUT));
                            rc = zmq_connect(SOCKET_S[snum].socketHb,
                                SOCKETS[snum].socketHbAddr);
                            if(rc==0)
                            {
                                drc = 0; crc = 0;
                                fprintf(stderr,"SP: Reconnected to proxy %s
                                    socket\n",SOCKET_S[snum].socketHbAddr);
                            }
                        }
                    }
                }
            }
        }
        else
        {
            zmq_msg_init_size(&hbmsg,strlen(SOCKET_S[snum].socketHbAddr)+1);

```

```

memcpy(zmq_msg_data(&hbmsg), SOCKETS[snum].socketHbAddr,
       zmq_msg_size(&hbmsg));
rc = zmq_msg_send(&hbmsg, SOCKETS[snum].socketHb, 0);
zmq_msg_close(&hbmsg);

zmq_msg_init(&rmsg);
rc = zmq_msg_recv(&rmsg, SOCKETS[snum].socketHb, 0);
if(rc!=-1 &&
    strcmp(zmq_msg_data(&rmsg),SOCKETS[snum].socketHbAddr)==0)
{
    if(SOCKETS[snum].alive!=1)
        SOCKETS[snum].alive = 1;
}
else
{
    if(SOCKETS[snum].alive==1)
    {
        SOCKETS[snum].alive = 0;
        fprintf(stderr,"SP: Heartbeat from proxy %s
                    lost\n",SOCKETS[snum].socketHbAddr);
    }
}
zmq_msg_close(&rmsg);
}
sleep(HB_DELAY);
}

int linger = 0;
zmq_setsockopt (SOCKETS[snum].socketHb, ZMQ_LINGER, &linger, sizeof
(linger));
if(drc==0)
    zmq_disconnect(SOCKETS[snum].socketHb, SOCKETS[snum].socketHbAddr);
if(crc==0)
    zmq_close(SOCKETS[snum].socketHb);
SOCKETS[snum].socketHb = NULL;
free(SOCKETS[snum].socketHbAddr);
SOCKETS[snum].socketHbAddr = NULL;
zmq_setsockopt (SOCKETS[snum].socket, ZMQ_LINGER, &linger, sizeof
(linger));
zmq_disconnect(SOCKETS[snum].socket, SOCKETS[snum].socketAddr);
zmq_close(SOCKETS[snum].socket);
SOCKETS[snum].socket = NULL;
free(SOCKETS[snum].socketAddr);
SOCKETS[snum].socketAddr = NULL;
SOCKETS[snum].alive = 0;
}
else
{
    fprintf(stderr,"SP: hbt: Error while connecting to proxy %s heartbeat
                port\n",SOCKETS[snum].socketHbAddr);
    int linger = 0;
    zmq_setsockopt (SOCKETS[snum].socketHb,ZMQ_LINGER, &linger, sizeof
(linger));
    zmq_close(SOCKETS[snum].socketHb);
    SOCKETS[snum].socketHb = NULL;
    free(SOCKETS[snum].socketHbAddr);
}

```

```

        SOCKETS[snum].socketHbAddr = NULL;
        zmq_disconnect(SOCKETS[snum].socket, SOCKETS[snum].socketAddr );
        zmq_setsockopt (SOCKET_S[snum].socket,ZMQ_LINGER, &linger, sizeof
            (linger));
        zmq_close(SOCKETS[snum].socket);
        printf("SP: hbt: Data socket %d closed\n",snum);
        SOCKETS[snum].socket = NULL;
        free(SOCKETS[snum].socketAddr);
        SOCKETS[snum].socketAddr = NULL;
        SOCKETS[snum].alive = 0;
    }
}
}

/*===== MAIN =====*/
int main(int argc, char **argv)
{
    int status;
    char* role = ":";
    ALIVE = 1;

    if(argc>1)
        role = argv[1];
    else
        printf("SP: Program started without specified role, default role is a
            random local online role\n");

    CONTEXT = zmq_ctx_new();
    //get configuration from database
    char* cport = getPort("SPPP_SourceDataPort");
    char* hport = getPort("SPPP_SourceHeartbeatPort");
    int MAX_EV_SIZE = getLimit("MaxEventSize");
    int HWM = getLimit("SP_SendHWM");
    int STIM = getLimit("SP_SendTimeout");
    int buffsize = MAX_EV_SIZE * HWM;
    char** proxies = NULL;
    const int pamount = getIPs(&proxies);
    if(pamount==--1 || pamount==0 || cport==NULL || hport==NULL || STIM==--1 ||
        HWM==--1)
    {
        if(pamount==--1 || pamount==0)
            fprintf(stderr,"SP: Zero proxies in configuration file. Program
                exits.\n");
        else
            fprintf(stderr,"SP: Error while getting parameters from file. Program
                exits.\n");
        zmq_ctx_destroy(CONTEXT);
        exit(1);
    }
    SOCKETS = malloc(pamount * sizeof(socket_info));
    HB_TIMEOUT = getLimit("SP_HeartbeatTimeout");
    HB_SND_TIMEOUT = HB_TIMEOUT/4;
    REC_SLEEP = getLimit("SP_ReconnectSleep");
    HB_DELAY = getLimit("SP_HeartbeatDelay");

    int rc;

```

```
//connecting to all proxies
for(int i=0; i<pamount; i++)
{
    SOCKETS[i].threadHB = 0;
    SOCKETS[i].socketAddr = malloc(strlen(proxies[i])+strlen(cport)+1);
    strcpy(SOCKETS[i].socketAddr ,proxies[i]);
    strcat(SOCKETS[i].socketAddr ,cport);
    SOCKETS[i].socket = zmq_socket(CONTEXT, ZMQ_PUSH);
    zmq_setsockopt(SOCKETS[i].socket, ZMQ_SNDHWM, &HWM, sizeof(HWM));
    zmq_setsockopt(SOCKETS[i].socket, ZMQ_SNDTIMEO, &STIM, sizeof(STIM));
    zmq_setsockopt(SOCKETS[i].socket, ZMQ_SNDBUF, &buffsize, sizeof(buffsize));

    rc = zmq_connect(SOCKETS[i].socket, SOCKETS[i].socketAddr );
    if(rc==0)
    {
        SOCKETS[i].socketHbAddr = malloc(strlen(proxies[i])+strlen(hport)+1);
        strcpy(SOCKETS[i].socketHbAddr ,proxies[i]);
        strcat(SOCKETS[i].socketHbAddr ,hport);
        SOCKETS[i].tnum = i;
        rc = pthread_create(&SOCKETS[i].threadHB, NULL,
            heartbeatThread, (void*)&SOCKETS[i].tnum);
        if(rc!=0)
        {
            fprintf(stderr, "SP: Error while creating heartbeat thread for proxy
                %s\n",SOCKETS[i].socketHbAddr);
            zmq_disconnect(SOCKETS[i].socket, SOCKETS[i].socketAddr );
            int linger = 0;
            zmq_setsockopt (SOCKETS[i].socket,ZMQ_LINGER, &linger, sizeof
                (linger));
            zmq_close(SOCKETS[i].socket);
            SOCKETS[i].socket = NULL;
            free(SOCKETS[i].socketAddr);
            SOCKETS[i].socketAddr = NULL;
            SOCKETS[i].socketHb = NULL;
            free(SOCKETS[i].socketHbAddr);
            SOCKETS[i].socketHbAddr = NULL;
            SOCKETS[i].alive = 0;
        }
    }
}
else
{
    fprintf(stderr,"SP: Error while connecting to proxy %s data
        port\n",SOCKETS[i].socketAddr);
    int linger = 0;
    zmq_setsockopt (SOCKETS[i].socket,ZMQ_LINGER, &linger, sizeof (linger));
    zmq_close(SOCKETS[i].socket);
    SOCKETS[i].socket = NULL;
    SOCKETS[i].socketHb = NULL;
    free(SOCKETS[i].socketAddr);
    SOCKETS[i].socketAddr = NULL;
    SOCKETS[i].alive = 0;
}
}
free(cport);
free(hport);
```

```

int x = 0;
struct eventStruct *event;
zmq_msg_t msg;
bool nc = false;
bool disp = false;

status = monitorSetDataSource(role);
if(status !=0)
{
    fprintf(stderr, "Error in monitorSetDataSource: %s\n",
        monitorDecodeError(status));
    exit(1);
}

status = monitorDeclareMp("monitoringbyproxy");
if(status!=0)
{
    fprintf(stderr, "Error in monitorDeclareMp: %s\n",
        monitorDecodeError(status));
    exit(1);
}

char *table[] = {
    "All events", "all",
    NULL
};

status = monitorDeclareTable(table);
if(status!=0)
{
    fprintf(stderr, "Error in monitorDeclareTable: %s\n",
        monitorDecodeError(status));
    exit(1);
}

monitorSetWait();
void* ptr = NULL;
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(STDIN_FILENO, &readfds);
fd_set savefds = readfds;
struct timeval usertimeout;
usertimeout.tv_sec = 0;
usertimeout.tv_usec = 0;
char input[100];
clock_t start, end, mid1, mid2;
double cpu_time_used;

printf("SP: Reading data...\n");
while(ALIVE==1)
{
    if(select(1, &readfds, NULL, NULL, &usertimeout))
    {
        fscanf(stdin, "%s", input);
        if(strcmp(input, "quit")==0)
        {

```

```

        printf("SP: Quitting program\n");
        ALIVE = 0;
    }
}
readfds = savefds;

status = monitorGetEventDynamic(&ptr);
if(status!=0)
    fprintf(stderr,"Error in monitorGetEventDynamic: %s\n",
        monitorDecodeError(status));

if(ptr!=NULL && status==0)
{
    event = (struct eventStruct*)ptr;
    try_again:
    x = findFirstAlive(x, pamount);
    if(x!=-1)
    {
        disp = false;
        zmq_msg_init_size(&msg,event->eventHeader.eventSize);
        memcpy(zmq_msg_data(&msg), event, event->eventHeader.eventSize);
        rc = zmq_msg_send (&msg, SOCKETS[x].socket, 0);
        zmq_msg_close(&msg);

        if(rc===-1)
        {
            fprintf(stderr,"SP: Error: Problem with sending message to %s
                (%s)\n", SOCKETS[x].socketAddr, zmq_strerror(errno));
            goto try_again;
        }
    }
    else
    {
        if(disp==false)
        {
            fprintf(stderr,"SP: No proxy alive - starting to dispose events\n");
            disp = true;
        }
    }
}
free(ptr);
ptr = NULL;
}

int s;
void **RETVAL2 = NULL;

for(int i=0; i<pamount; i++)
{
    if(SOCKETS[i].threadHB!=0)
        pthread_join(SOCKETS[i].threadHB,RETVAL2);
    if(proxies[i]!=NULL)
        free(proxies[i]);
}
free(proxies);
free(SOCKETS);

```

```
    zmq_ctx_destroy(CONTEXT);  
    return 0;  
}
```

Makefile Source code:

MakefileSP

```
DATE_CPPFLAGS := $(shell date-config --cflags)  
DATE_LIBS     := $(shell date-config --libs --monitorlibs=dyn)  
  
all: SP.c  
    gcc $(DATE_CPPFLAGS) $(DATE_LIBS) -g -std=gnu99 -L/opt/date/mbp/Linux  
        -lmbp -lzmq -o SP SP.c  
  
clean:  
    rm SP
```

Appendix B

Proxy Program source code

PP.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <cstdlib>
#include <iostream>
#include <pthread.h>
#include <zmq.hpp>
#include <ctime>
#include <mutex>
#include <unistd.h>
#include <cstdlib>
#include "event.h"
#include "monitoringbyproxy.h"
using namespace std;

void *CONTEXT;
bool *ALIVE = new bool();
char* SP_PORT = NULL;
char* SP_HB_PORT = NULL;
char* CLIENT_COMM_PORT = NULL;
int CLIENT_FIRST_PORT;
int LINGER;
int INNER_THREADS_TIMEOUT;
int MAX_EV_SIZE;
int SP_HB_TIMEOUT;
int SP_HB_SND_TIMEOUT;
int SP_RCV_HWM;
int SP_RCV_TIMEOUT;
int CLIENT_FH_TIMEOUT;
int CLIENT_HWM;
int MAX_CLIENTS;
int MAX_PREFERENCES;
int TID_SHIFT = 1000;

void *SourceHbThread(void *args);
void *SourceDataThread(void *args);
void *KeyThread(void *args);
void *ClientHbThread(void *args);
void *ClientDataThread(void *args);
```

```

//==== Thread Data =====//
typedef struct thread_data{
    int thread_id;
    bool* thread_condition;
    bool* var;
}thread_data;

void fillThreadData(thread_data *td, int id, bool* condition, bool* var)
{
    td->thread_id = id;
    td->thread_condition = condition;
    td->var = var;
};

int sendVal(const char* val, void* socket)
{
    zmq_msg_t nport;
    zmq_msg_init_size(&nport,strlen(val));
    memcpy(zmq_msg_data(&nport), val, strlen(val));
    int rc = zmq_msg_send(&nport, socket, 0);
    zmq_msg_close(&nport);
    return rc;
}

//==== PROXY DB =====//
typedef struct client_data{
    std::mutex mtx;
    bool occupied;
    bool connected;
    int thread_id;
    pthread_t thread;
    char* comm_port;
    char* data_addr;
    char* inproc_addr;
    char* client_key;
    struct mbpPreferenceConverted *preferences;
    int pref_nb;
}client_data;

class proxy_db{

    int TA;
    client_data *db;

public:

    int get_TA(){
        return TA;
    };

    void change_TA(bool t){
        if(t==true)
            TA++;
        else
            TA--;
    };
};

```

```
int get_thread_id(int n){
    lock_guard<mutex> guard(db[n].mtx);
    return db[n].thread_id;
};

pthread_t get_thread(int n){
    lock_guard<mutex> guard(db[n].mtx);
    return db[n].thread;
};

char* get_client_key(int n){
    lock_guard<mutex> guard(db[n].mtx);
    return db[n].client_key;
};

void set_client_key(int n, char* s){
    lock_guard<mutex> guard(db[n].mtx);
    strcpy(db[n].client_key,s);
};

bool get_occupied(int n){
    lock_guard<mutex> guard(db[n].mtx);
    return db[n].occupied;
};

void set_occupied(int n, bool s){
    lock_guard<mutex> guard(db[n].mtx);
    db[n].occupied=s;
};

void set_connected(int n, bool s){
    lock_guard<mutex> guard(db[n].mtx);
    db[n].connected=s;
};

char* get_comm_port(int n){
    lock_guard<mutex> guard(db[n].mtx);
    return db[n].comm_port;
};

char* get_inproc_addr(int n){
    lock_guard<mutex> guard(db[n].mtx);
    return db[n].inproc_addr;
};

char* get_data_addr(int n){
    lock_guard<mutex> guard(db[n].mtx);
    return db[n].data_addr;
};

int get_pref_nb(int n){
    lock_guard<mutex> guard(db[n].mtx);
    return db[n].pref_nb;
};
```

```

void set_pref_nb(int n, int s){
    lock_guard<mutex> guard(db[n].mtx);
    db[n].pref_nb = s;
};

int createHBThread(int n){
    return pthread_create(&db[n].thread, NULL,
        ClientHbThread, (void*)&db[n].thread_id);
};

int getPreference(int n, int m, mbpPreferenceConverted &pref){
    lock_guard<mutex> guard(db[n].mtx);
    if(db[n].connected==true)
    {
        memcpy(&pref,&(db[n].preferences[m]),sizeof(pref));
        return 0;
    }
    else
        return -1;
};

bool checkBlock(int n){
    lock_guard<mutex> guard(db[n].mtx);
    for(int i=0;i<db[n].pref_nb;i++)
        if(db[n].preferences[i].percent== -1)
            return true;
    return false;
};

void fillDb()
{
    TA = 0;
    db = new client_data[MAX_CLIENTS];
    for(int i=0;i<MAX_CLIENTS;i++)
    {
        lock_guard<mutex> guard(db[i].mtx);
        db[i].occupied = false;
        db[i].connected = false;
        db[i].thread_id = TID_SHIFT+i;
        db[i].comm_port = (char*)malloc(11);
        sprintf(db[i].comm_port,"%d",CLIENT_FIRST_PORT+i);
        db[i].inproc_addr = (char*)malloc(strlen("inproc://#") +
            strlen(db[i].comm_port) + 1);
        sprintf(db[i].inproc_addr,"inproc://#%s",db[i].comm_port);
        db[i].client_key = (char*)malloc(101);
        strcpy(db[i].client_key,"");
        db[i].data_addr = (char*)malloc(101);
        strcpy(db[i].data_addr,"");
        db[i].preferences = NULL;
        db[i].pref_nb = 0;
    }
    cout << "PP: Proxy DB filled up" << endl;
};

```

```
void updateDb(int nr, struct mbpConnection *client)
{
    lock_guard<mutex> guard(db[nr].mtx);
    strcpy(db[nr].client_key, client->key);
    strcpy(db[nr].data_addr, client->dataaddr);
    db[nr].pref_nb = client->PrefNb;

    int cdbnr = db[nr].thread_id-TID_SHIFT;
    int l = 0;
    while(l<client->PrefNb && l<MAX_PREFERENCES)
    {
        db[cdbnr].preferences = (client->preferences)+l;
        db[cdbnr].preferences++;
        l++;
    }
    db[cdbnr].preferences-=client->PrefNb;
};

void freeDb()
{
    for(int i=0; i<MAX_CLIENTS; i++)
    {
        lock_guard<mutex> guard(db[i].mtx);
        db[i].thread_id=0;
        free(db[i].comm_port);
        free(db[i].client_key);
        free(db[i].inproc_addr);
        free(db[i].data_addr);
    }
    delete[] db;
};

int findFirstFreeSlot()
{
    for(int i=0; i<MAX_CLIENTS; i++)
    {
        db[i].mtx.lock();
        if(db[i].occupied==false)
        {
            db[i].mtx.unlock();
            return i;
        }
        else
            db[i].mtx.unlock();
    }
    return -1;
};

};
proxy_db PROXY_DB;
```

```

//===== KEY_DB =====//

typedef struct key_data{
    int changed;
    bool occupied;
    bool ready;
    char* key;
    bool block;
    char* rcv_addr;
    char** inproc_addrs;
    int* inproc_clients;
    int thread_id;
    pthread_t thread;
    int client_nb;
}key_data;

class key_db{
    key_data *db;
    int keythread_nb;
    mutex mtx;

private:
    int findFreeKeySlot()
    {
        for(int i=0;i<MAX_CLIENTS;i++)
            if(db[i].occupied==false)
            {
                db[i].occupied = true;
                return i;
            }
        return -1;
    };

public:

    int get_keythread_nb(){
        lock_guard<mutex> guard(mtx);
        return keythread_nb;
    };

    void set_keythread_nb(int n){
        lock_guard<mutex> guard(mtx);
        keythread_nb += n;
    };

    void set_rcv_addr(int n, char* addr)
    {
        lock_guard<mutex> guard(mtx);
        if(addr==NULL)
        {
            if(db[n].rcv_addr!=NULL)
            {
                free(db[n].rcv_addr);
                db[n].rcv_addr = NULL;
            }
        }
    }
}

```

```
else
{
    if(db[n].rcv_addr==NULL)
    {
        db[n].rcv_addr = (char*)malloc(sizeof(char)*strlen(addr)+2);
        memcpy(db[n].rcv_addr,addr,strlen(addr)+1);
    }
}
};

bool get_block(int n){
    lock_guard<mutex> guard(mtx);
    return db[n].block;
};

int get_client(char** &iaddr, int* &icli, int n)
{
    lock_guard<mutex> guard(mtx);
    if(db[n].client_nb>0)
    {
        iaddr = db[n].inproc_addr;
        icli = db[n].inproc_clients;
    }
    return db[n].client_nb;
};

int get_client_nb(int n){
    lock_guard<mutex> guard(mtx);
    return db[n].client_nb;
};

int get_changed(int n){
    lock_guard<mutex> guard(mtx);
    return db[n].changed;
};

void reset_changed(int n){
    lock_guard<mutex> guard(mtx);
    db[n].changed = 0;
};

bool get_occupied(int n){
    lock_guard<mutex> guard(mtx);
    return db[n].occupied;
};

pthread_t get_thread(int n){
    lock_guard<mutex> guard(mtx);
    return db[n].thread;
};

void reset_occupied(int n){
    lock_guard<mutex> guard(mtx);
    db[n].occupied = false;
};
```

```

void set_ready(int n, bool st){
    lock_guard<mutex> guard(mtx);
    db[n].ready=st;
};

void fillDb()
{
    lock_guard<mutex> guard(mtx);
    keythread_nb = 0;
    db = new key_data[MAX_CLIENTS];
    for(int i=0;i<MAX_CLIENTS;i++)
    {
        db[i].changed = 0;
        db[i].occupied = false;
        db[i].ready = false;
        db[i].key = NULL;
        db[i].rcv_addr = NULL;
        db[i].thread = NULL;
        db[i].client_nb = 0;
        db[i].block = false;
        db[i].inproc_addrs = NULL;
        db[i].inproc_clients = NULL;
    }
};

void freeDb(){
    lock_guard<mutex> guard(mtx);
    delete[] db;
};

int copyRcvAddr(int n, char* &str)
{
    lock_guard<mutex> guard(mtx);
    if(db[n].key!=NULL && db[n].ready==true && db[n].rcv_addr!=NULL)
    {
        int len = strlen(db[n].rcv_addr);
        if(str==NULL)
        {
            str = (char*) malloc(len+2);
            memcpy(str,db[n].rcv_addr,len+1);
        }
        return 0;
    }
    else
        return -1;
};

int findKeyThread(char* k)
{
    lock_guard<mutex> guard(mtx);
    for(int i=0;i<MAX_CLIENTS;i++)
    if(strcmp(db[i].key,k)==0)
        return i;
    return -1;
};

```

```
void addKeyThreadClient(char* k, int cl)
{
    lock_guard<mutex> guard(mtx);
    int ft = -1;
    for(int i=0; i<MAX_CLIENTS; i++)
        if(db[i].key!=NULL)
            if(strcmp(db[i].key, k)==0)
                ft = i;
    if(ft==-1) //create new thread
    {
        ft = findFreeKeySlot();
        if(ft!=-1)
        {
            db[ft].key=(char*)malloc(strlen(k)+1);
            memcpy(db[ft].key, k, strlen(k)+1);
            db[ft].client_nb = 1;
            db[ft].inproc_clients = (int*)malloc(1*(sizeof(int))+1);
            db[ft].inproc_clients[0] = cl;
            db[ft].inproc_addrs = (char**)malloc(1*(sizeof(char*))+1);
            db[ft].inproc_addrs[0] = PROXY_DB.get_inproc_addr(cl);
            db[ft].changed = 1;
            db[ft].thread_id = TID_SHIFT+ft+2*MAX_CLIENTS;
            int rc = pthread_create(&db[ft].thread, NULL,
                                   KeyThread, (void*)&db[ft].thread_id);
            if(rc!=0)
            {
                free(db[ft].inproc_addrs);
                db[ft].inproc_addrs = NULL;
                db[ft].inproc_clients[0] = -1;
                free(db[ft].inproc_clients);
                db[ft].inproc_clients = NULL;
                free(db[ft].key);
                db[ft].key = NULL;
                db[ft].client_nb = 0;
                db[ft].occupied = false;
                db[ft].changed = 0;
            }
        }
        else
            db[ft].occupied = false;
    }
    else //add client to existing thread
    {
        db[ft].client_nb++;
        db[ft].inproc_clients = (int*)realloc(db[ft].inproc_clients,
                                                db[ft].client_nb*(sizeof(int))+1);
        db[ft].inproc_clients[db[ft].client_nb-1] = cl;
        db[ft].inproc_addrs = (char**)realloc(db[ft].inproc_addrs,
                                                db[ft].client_nb*(sizeof(char*))+1);
        db[ft].inproc_addrs[db[ft].client_nb-1] = PROXY_DB.get_inproc_addr(cl);
        db[ft].changed = db[ft].client_nb;
    }
    if(db[ft].block==false)
        if(PROXY_DB.checkBlock(cl))
            db[ft].block = true;
};
```

```

void removeKeyThreadClient(char* k, int cl)
{
    lock_guard<mutex> guard(mtx);
    int ft=-1;
    for(int i=0;i<MAX_CLIENTS;i++)
        if(db[i].key!=NULL)
            if(strcmp(db[i].key,k)==0)
                ft = i;

    if(ft!=-1)
    {
        if(db[ft].client_nb>1)
        {
            int ctr = 0;
            while(db[ft].inproc_clients[ctr]!=cl || ctr>=db[ft].client_nb)
                ctr++;

            if(ctr<db[ft].client_nb)
            {
                db[ft].inproc_addrs[ctr] = NULL;
                db[ft].inproc_clients[ctr] = -1;
                db[ft].client_nb--;
                if(ctr<db[ft].client_nb)
                {
                    for(int i=ctr;i<db[ft].client_nb;i++)
                    {
                        db[ft].inproc_addrs[i] = db[ft].inproc_addrs[i+1];
                        db[ft].inproc_clients[i] = db[ft].inproc_clients[i+1];
                    }
                    db[ft].inproc_addrs[db[ft].client_nb] = NULL;
                    db[ft].inproc_clients[db[ft].client_nb] = -1;
                }
                db[ft].inproc_clients = (int*)realloc(db[ft].inproc_clients,
                    db[ft].client_nb*(sizeof(int))+1);
                db[ft].inproc_addrs = (char**)realloc(db[ft].inproc_addrs,
                    db[ft].client_nb*(sizeof(char*))+1);

                if(ctr==0)
                    db[ft].changed = -1;
                else
                    db[ft].changed = (-1*ctr)-1;

                if(db[ft].block==true)
                {
                    db[ft].block = false;
                    for(int i=0;i<db[ft].client_nb;i++)
                        if(PROXY_DB.checkBlock(i))
                            db[ft].block = true;
                }
            }
        }
    }
    else if(db[ft].client_nb==1) //remove whole key thread
    {
        db[ft].client_nb = 0;
        free(db[ft].key);
        db[ft].key = NULL;
    }
}

```

```
        free(db[ft].inproc_clients);
        db[ft].inproc_clients = NULL;
        free(db[ft].inproc_addrs);
        db[ft].inproc_addrs = NULL;
        db[ft].block = false;
        db[ft].changed = -1;
    }
    else
        cerr << "PP: rmktc: Error while removing client from key db" << endl;
    }
};
};
key_db KEY_DB;

//===== checking preferences =====//

//return -1 if random > percent, 0 if mustall, 1 if random < percent and not
//mustall
int checkPercent(int percent)
{
    if(percent!=-1) //if not 100% or must all 100%
    {
        int r = rand()%100;
        if(r<percent)
            return 1;
        else
            return -1;
    }
    else
        return 0;
}

//returns 0 if equal, 1 if not
int checkEventType(eventTypeType et, int type)
{
    if(type == 0 || type == et)
        return 0;
    else
        return 1;
}

//returns 0 if equal 1 if not, -1 if error
int checkSize(int es, int sign, int isize)
{
    switch(sign)
    {
        case 0:
            return 0;
            break;

        case 1:
            if(es>isize) return 0; else return 1;
            break;
    }
}
```

```

    case 2:
        if(es>=isize) return 0; else return 1;
        break;

    case 3:
        if(es<isize) return 0; else return 1;
        break;

    case 4:
        if(es<=isize) return 0; else return 1;
        break;

    case 5:
        if(es==isize) return 0; else return 1;
        break;

    default:
        return -1;
        break;
}

return -1;
}

//returns 0 if equal 1 if not, -1 if error
int checkTriggerPattern(eventTriggerPatternType etp, char* str)
{
    eventTriggerPatternType ctp;
    int nTrig=0;
    if (etp==NULL || str==NULL)
        return -1;

    // "*" = disable selection criteria
    if(str[0]=='*')
        return 0;

    // Must contain numbers or '&'s or '|'s
    int numAnds = 0;
    int numOrs = 0;
    for (int i = 0; i<strlen(str); i++ ) {
        if ( str[i] != '&' && str[i] != '|' && !isdigit( (int)str[i] ) )
            return -1;
        if ( str[i] == '&' ) numAnds++;
        if ( str[i] == '|' ) numOrs++;
    }

    // Must contain only '&'s or '|'s (no mixtures allowed)
    if ( numAnds != 0 && numOrs != 0 ) return -1;

    // The numbers must be within the allowed range
    ZERO_TRIGGER_PATTERN(ctp); //sets zeros
    for ( int i = 0; str[i] != 0; )
    {
        int num;

```

```
if ( str[i] == '&' || str[i] == '|' )
    i++;
else
{
    if ( sscanf( &str[i], "%d", &num ) != 1 )
        return -1;
    if ( num < EVENT_TRIGGER_ID_MIN || num > EVENT_TRIGGER_ID_MAX )
        return -1;

    nTrig++;
    SET_TRIGGER_IN_PATTERN( ctp, num ); //set 1 in num position
    while ( str[i] != '&' && str[i] != '|' && str[i] != 0 )
        i++;
}
}

if ( nTrig != 0 )
    VALIDATE_TRIGGER_PATTERN(ctp); //sets last element
else
    INVALIDATE_TRIGGER_PATTERN(ctp); //sets last element

if ( nTrig != 0 && numAnds != 0 )
{
    int s=0;
    for(int i=0;i<EVENT_TRIGGER_PATTERN_WORDS-1;i++)
        if(etp[i]&ctp[i]==ctp[i])s++;

    if(s==EVENT_TRIGGER_PATTERN_WORDS-1) return 0;
    else return 1;
}
else if( nTrig != 0 && numOrs != 0 )
{
    int s=0;
    for(int i=0;i<EVENT_TRIGGER_PATTERN_WORDS-1;i++)
        if(etp[i]&ctp[i]!=0)s++;
    if(s>0) return 0;
    else return 1;
}
else
    return -1;
}

int checkDetectorPattern(eventDetectorPatternType edp, char* str)
{
    eventDetectorPatternType cdp;
    int nDet=0;
    if (edp==NULL || str==NULL)
        return -1;

    // "*" = disable selection criteria
    if(str[0]=='*')
        return 0;

    // Must contain numbers or '&'s or '|'s
    int numAnds = 0;
    int numOrs = 0;
```

```

for (int i = 0; str[i] != 0; i++ ) {
    if ( str[i] != '&' && str[i] != '|' && !isdigit( (int)str[i] ) )
        return -1;
    if ( str[i] == '&' ) numAnds++;
    if ( str[i] == '|' ) numOrs++;
}

// Must contain only '&'s or '|'s (no mixtures allowed)
if ( numAnds != 0 && numOrs != 0 ) return -1;
// The numbers must be within the allowed range
ZERO_DETECTOR_PATTERN(cdp); //sets zeros
for ( int i = 0; str[i] != 0; )
{
    int num;
    if ( str[i] == '&' || str[i] == '|' )
        i++;
    else
    {
        if ( sscanf( &str[i], "%d", &num ) != 1 )
            return -1;
        if ( num < EVENT_DETECTOR_ID_MIN || num > EVENT_DETECTOR_ID_MAX )
            return -1;

        nDet++;
        SET_DETECTOR_IN_PATTERN( cdp, num ); //set 1 in num position
        while ( str[i] != '&' && str[i] != '|' && str[i] != 0 )
            i++;
    }
}

if ( nDet != 0 )
    VALIDATE_DETECTOR_PATTERN(cdp); //sets last element
else
    INVALIDATE_DETECTOR_PATTERN(cdp); //sets last element

if ( nDet != 0 && numAnds != 0 )
{
    int s=0;
    for(int i=0;i<EVENT_DETECTOR_PATTERN_WORDS-1;i++)
        if(edp[i]&cdp[i]==cdp[i])s++;

    if(s==EVENT_DETECTOR_PATTERN_WORDS-1) return 0;
    else return 1;
}
else if( nDet != 0 && numOrs != 0 )
{
    int s=0;
    for(int i=0;i<EVENT_DETECTOR_PATTERN_WORDS-1;i++)
        if(edp[i]&cdp[i]!=0)s++;

    if(s>0) return 0;
    else return 1;
}
else
    return -1;
}

```

```
int checkTypeAttribute(eventTypeAttributeType eat, char* str)
{
    eventTypeAttributeType cat;
    int nAttr=0;
    if ( str == NULL || eat == NULL)
        return -1;

    nAttr = 0;
    // "*" = disable selection criteria
    if ( str[0] == '*')
        return 0;

    // Must contain numbers or '&'s or '|'s
    int numAnds = 0;
    int numOrs = 0;
    for ( int i = 0; str[i] != 0; i++ ) {
        if ( str[i] != '&' && str[i] != '|' && !isdigit( (int)str[i] ) )
            return -1;
        if ( str[i] == '&' ) numAnds++;
        if ( str[i] == '|' ) numOrs++;
    }

    // Must contain only '&'s or '|'s (no mixtures allowed)
    if ( numAnds != 0 && numOrs != 0 ) return -1;

    // The numbers must be within the allowed range
    RESET_ATTRIBUTES( cat );
    for ( int i = 0; str[i] != 0; ) {
        int num;
        if ( str[i] == '&' || str[i] == '|' )
            i++;
        else{
            if ( sscanf( &str[i], "%d", &num ) != 1 )
                return -1;
            if ( num < 0 || num >= ALL_ATTRIBUTE_BITS )
                return -1;
            nAttr++;
            SET_ANY_ATTRIBUTE( cat, num );
            while ( str[i] != '&' && str[i] != '|' && str[i] != 0 )
                i++;
        }
    }

    if ( nAttr != 0 && numAnds != 0 )
    {
        int s=0;
        for(int i=0;i<3;i++)
            if(eat[i]&cat[i]==cat[i])s++;

        if(s==3) return 0;
        else return 1;
    }
    else if( nAttr != 0 && numOrs != 0 )
    {
        int s=0;
        for(int i=0;i<3;i++)
```

```

        if(eat[i]&cat[i]!=0)s++;

        if(s>0) return 0;
        else return 1;
    }
    else
        return -1;
}

//return -1 if don't match, 0 if match and mustall, (number of preference +1)
//if match and not mustall
int checkPref(struct eventStruct event, int clientnr)
{
    mbpPreferenceConverted pref;
    int p;
    for(int i=0;i<PROXY_DB.get_pref_nb(clientnr);i++)
    {
        p = PROXY_DB.getPreference(clientnr,i, pref);
        if(p!=-1)
        {
            if(checkEventType(event.eventHeader.eventType,pref.eventType) == 0 &&
                checkSize(event.eventHeader.eventSize, pref.eventSizeSign,
                    pref.eventSize) == 0)
            {
                if(pref.eventGdcId==-1 ||
                    pref.eventGdcId==event.eventHeader.eventGdcId)
                {
                    if(checkTriggerPattern(event.eventHeader.eventTriggerPattern,
                        pref.eventTriggerPattern)==0 &&
                        checkDetectorPattern(event.eventHeader.eventDetectorPattern,
                            pref.eventDetectorPattern)==0 &&
                        checkTypeAttribute(event.eventHeader.eventTypeAttribute,
                            pref.eventTypeAttribute)==0)
                    {
                        int p = checkPercent(pref.percent);
                        if(p==1) //ok - return nb of pref
                            return i+1;
                        else if(p==0) //must all
                            return 0;
                        else
                            return -1;
                    }
                }
            }
        }
    }
    return -1;
}

```

```
//===== THREADS =====//
// ===== Source Threads =====//

//Heartbeat thread for data source
void *SourceHbThread(void *id)
{
    int tid = *((int*)id);
    void* RHBsocket = zmq_socket(CONTEXT, ZMQ_REP);

    if(RHBsocket!=NULL)
    {
        char addr4bind[strlen("tcp://*:") + strlen(SP_HB_PORT) + 1];
        strcpy(addr4bind, "tcp://*:"); strcat(addr4bind, SP_HB_PORT);
        zmq_setsockopt(RHBsocket, ZMQ_RCVTIMEO, &SP_HB_TIMEOUT,
            sizeof(SP_HB_TIMEOUT));
        zmq_setsockopt(RHBsocket, ZMQ_SNDTIMEO, &SP_HB_SND_TIMEOUT,
            sizeof(SP_HB_SND_TIMEOUT));

        int rc = zmq_bind(RHBsocket, addr4bind); assert(rc==0);
        if(rc==0)
        {
            cout << "PP: shbt: Heartbeat socket for SP bound: " << addr4bind << endl;
            zmq_msg_t hb;

            while(*ALIVE==true)
            {
                zmq_msg_init(&hb);
                rc = zmq_msg_recv(&hb, RHBsocket, 0);
                if(rc== -1)
                    cerr << "PP: shbt: Heartbeat receiving timeout rc: " << rc << endl;
                rc = zmq_msg_send(&hb, RHBsocket, 0);
                if(rc== -1)
                    cerr << "PP: shbt: Heartbeat sending timeout rc: " << rc << endl;
                zmq_msg_close(&hb);
            }
        }
        else{
            cerr << "PP: shbt: Cannot bind heartbeat zmq socket" << endl;
            exit(-1);
        }

        zmq_setsockopt(RHBsocket, ZMQ_LINGER, &LINGER, sizeof(LINGER));
        zmq_unbind(RHBsocket, addr4bind);
        zmq_close(RHBsocket);
    }
    else{
        cerr << "PP: shbt: Cannot create heartbeat zmq socket" << endl;
        exit(-1);
    }
    pthread_exit(NULL);
}
```

```

//Thread for receiving data and propagate it between clients
void *SourceDataThread(void *id)
{
    int tid = *((int*)id);
    void* RCVsocket = zmq_socket(CONTEXT, ZMQ_PULL);
    void* PreKEYsocket = zmq_socket(CONTEXT, ZMQ_PUB);

    int rcvbufferSize = MAX_EV_SIZE * SP_RCV_HWM;
    int rc, pubhwm = 10, nodrop = 1;

    if(RCVsocket != NULL && PreKEYsocket != NULL)
    {
        zmq_setsockopt(RCVsocket, ZMQ_RCVTIMEO, &SP_RCV_TIMEOUT,
            sizeof(SP_RCV_TIMEOUT));
        zmq_setsockopt(RCVsocket, ZMQ_RCVHWM, &SP_RCV_HWM, sizeof(SP_RCV_HWM));
        zmq_setsockopt(RCVsocket, ZMQ_RCVBUF, &rcvbufferSize,
            sizeof(rcvbufferSize));
        zmq_setsockopt(PreKEYsocket, ZMQ_SNDHWM, &pubhwm, sizeof(pubhwm));
        zmq_setsockopt(PreKEYsocket, ZMQ_XPUB_NODROP, &nodrop, sizeof(nodrop));

        char addr4bind2[strlen("inproc://")+strlen(SP_PORT)+1];
        strcpy(addr4bind2,"inproc://"); strcat(addr4bind2,SP_PORT);
        zmq_bind(PreKEYsocket,addr4bind2);
        cout << "PP: sdt: Data PUB socket for prekey threads bound: " <<
            addr4bind2 << endl;

        char addr4bind[strlen("tcp://*")+strlen(SP_PORT)+1];
        strcpy(addr4bind,"tcp://*"); strcat(addr4bind,SP_PORT);
        zmq_bind(RCVsocket,addr4bind);
        cout << "PP: sdt: Data socket for SP bound: " << addr4bind << endl;

        zmq_msg_t rmsg, smsg;

        while(*ALIVE==true)
        {
            zmq_msg_init(&rmsg);
            rc = zmq_msg_recv(&rmsg,RCVsocket,0);

            if(rc != -1) //any event received
            {
                try_again:
                zmq_msg_init(&smsg);
                zmq_msg_copy(&smsg,&rmsg);
                rc = zmq_msg_send(&smsg,PreKEYsocket,ZMQ_DONTWAIT);
                zmq_msg_close(&smsg);

                if(rc == -1)
                {
                    cout << "PP: sdt: Problem with sending event to prekey socket"<< endl;
                    if(KEY_DB.get_keythread_nb(>0)
                        goto try_again;
                    else
                    {
                        zmq_setsockopt(PreKEYsocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
                        zmq_unbind(PreKEYsocket,addr4bind2);
                        zmq_close(PreKEYsocket);
                    }
                }
            }
        }
    }
}

```

```
        zmq_bind(PreKEYsocket,addr4bind2);
        cerr << "PP: sdt: Publisher buffer flushed" << endl;
    }
}
else if(rc == 0)
    cerr << "PP: sdt: Empty msg to publish" << endl;
}
else
    cout << "PP: sdt: Did not receive any event"<< endl;
    zmq_msg_close(&rmsg);
}

zmq_setsockopt (RCVsocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
zmq_unbind(RCVsocket,addr4bind);
zmq_close(RCVsocket);
zmq_setsockopt(PreKEYsocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
zmq_close(PreKEYsocket);
}
else
{
    zmq_setsockopt (RCVsocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
    zmq_close(RCVsocket);
    zmq_setsockopt (PreKEYsocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
    zmq_close(PreKEYsocket);
    cerr << "PP: sdt: Error while creating zmq sockets. Errno: " <<
        zmq_strerror(errno) << endl;
}
pthread_exit(NULL);
}

//===== Key Threads =====//

//Inner pre key thread
void *PreKeyThread(void *args)
{
    thread_data *my_data = (thread_data *) args;
    long tid = (long)my_data->thread_id;
    bool* condition = (bool*)my_data->thread_condition;
    bool* block = (bool*)my_data->var;

    int rc, hwm = 5;
    void* SUBsocket = zmq_socket(CONTEXT,ZMQ_SUB);
    void* KEYsocket = zmq_socket(CONTEXT, ZMQ_PAIR);

    zmq_setsockopt(SUBsocket, ZMQ_RCVTIMEO, &INNER_THREADS_TIMEOUT,
        sizeof(INNER_THREADS_TIMEOUT));
    zmq_setsockopt(SUBsocket, ZMQ_RCVHWM, &hwm,sizeof(hwm));
    zmq_setsockopt(KEYsocket, ZMQ_SNDHWM, &hwm,sizeof(hwm));

    char buffer[11]; sprintf(buffer, "%d", tid-MAX_CLIENTS);
    char pushAddr[strlen("inproc://")+strlen(buffer)+1];
    strcpy(pushAddr,"inproc://"); strcat(pushAddr,buffer);
    zmq_connect(KEYsocket,pushAddr);

    char addr4bind[strlen("inproc://")+strlen(SP_PORT)+1];
```

```

strcpy(addr4bind,"inproc://#"); strcat(addr4bind,SP_PORT);
zmq_connect(SUBsocket,addr4bind);
zmq_setsockopt(SUBsocket, ZMQ_SUBSCRIBE, "", 0);

KEY_DB.set_keythread_nb(1);
zmq_msg_t rmsg, smsg;

while(*condition == true)
{
    zmq_msg_init(&rmsg);
    rc = zmq_msg_recv(&rmsg,SUBsocket,0);

    if(rc != -1) //any event received
    {
        try_again:
        zmq_msg_init(&smsg);
        zmq_msg_copy(&smsg,&rmsg);
        rc = zmq_msg_send(&smsg,KEYsocket,ZMQ_DONTWAIT);
        zmq_msg_close(&smsg);

        if(rc == -1){
            if(*block == true)
                goto try_again;
        }
    }
    zmq_msg_close(&rmsg);
}

KEY_DB.set_keythread_nb(-1);
zmq_setsockopt (SUBsocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
zmq_setsockopt(SUBsocket, ZMQ_UNSUBSCRIBE, "", 0);
zmq_disconnect(SUBsocket,addr4bind);
zmq_close(SUBsocket);
zmq_setsockopt (KEYsocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
zmq_disconnect(KEYsocket,pushAddr);
zmq_close(KEYsocket);
pthread_exit(NULL);
}

//Inner key thread
void *KeyThread(void *id)
{
    int tid = *((int*)id);
    int dbnr = tid-TID_SHIFT-2*MAX_CLIENTS;
    PROXY_DB.change_TA(true);
    pthread_detach(pthread_self());
    int rc, hwm = 5;
    void* ThreadPULLSocket = zmq_socket(CONTEXT,ZMQ_PAIR);
    void* ThreadPUSHSocket = zmq_socket(CONTEXT,ZMQ_PUSH);

    if(ThreadPULLSocket != NULL && ThreadPUSHSocket != NULL)
    {
        zmq_setsockopt(ThreadPULLSocket, ZMQ_RCVTIMEO, &INNER_THREADS_TIMEOUT,
            sizeof(INNER_THREADS_TIMEOUT));
        zmq_setsockopt(ThreadPULLSocket, ZMQ_RCVHWM, &hwm,sizeof(hwm));
    }
}

```

```
zmq_setsockopt(ThreadPUSHSocket, ZMQ_SNDHWM, &hwm, sizeof(hwm));

char buffer[11]; sprintf(buffer, "%d", tid);
char pullAddr[strlen("inproc://")+strlen(buffer)+1];
strcpy(pullAddr, "inproc://"); strcat(pullAddr, buffer);
KEY_DB.set_rcv_addr(dbnr, pullAddr);
zmq_bind(ThreadPULLSocket, pullAddr);

int prf, ws=0, trsnd=0, snd=0, clam=1;
int* cls = NULL;
char** addrs = NULL;
struct eventStruct rcvd_event;
zmq_msg_t rmsg, smsg;
int ch;

KEY_DB.set_ready(dbnr, true);

bool* trc = new bool(true);
bool* block = new bool(false);
thread_data td;
fillThreadData(&td, tid+MAX_CLIENTS, trc, block);
pthread_t threadPK;
rc = pthread_create(&threadPK, NULL, PreKeyThread, (void*)&td);

if(rc==0)
{
    while(clam>0 && *ALIVE==true)
    {
        if(KEY_DB.get_changed(dbnr)!=0)
            clam = KEY_DB.get_client_nb(dbnr);

        zmq_msg_init(&rmsg);
        rc = zmq_msg_recv(&rmsg, ThreadPULLSocket, 0);
        if(rc!=-1)
        {
            memcpy(&rcvd_event, zmq_msg_data(&rmsg), sizeof(rcvd_event));
            trsnd = 0; snd = 0;
            while(snd==0 && trsnd<clam && clam>0 && *ALIVE==true)
            {
                if(KEY_DB.get_changed(dbnr)!=0)
                {
                    clam = KEY_DB.get_client(addrs, cls, dbnr);
                    cout << "PP: kt: Number of clients in thread " << tid << " : " <<
                        clam << endl;
                    KEY_DB.reset_changed(dbnr);
                    ws = 0;
                }
                if(clam>0 && ws<clam)
                {
                    try_again:
                    if(cls[ws]!==-1)
                        prf = checkPref(rcvd_event, cls[ws]);
                    else
                        prf = -1;

                    if(prf!=-1)
```

```

        {
            zmq_connect(ThreadPUSHSocket, addr[ws]);
            zmq_msg_init(&smsg);
            zmq_msg_copy(&smsg, &rmsg);
            rc = zmq_msg_send(&smsg, ThreadPUSHSocket, ZMQ_DONTWAIT);
            zmq_msg_close(&smsg);
            zmq_disconnect(ThreadPUSHSocket, addr[ws]);

            if(rc == -1)
            {
                ch = KEY_DB.get_changed(dbnr);
                if(ch < 0 && ws < ((-1*ch)-1))
                    ws--;

                if(ch < 0 && ws < ((-1*ch)-1))
                    ws = 0;
                else if(prf == 0 && *ALIVE == true)
                    goto try_again;
            }
            else
                snd = 1;
        }
        ws++; if(ws >= clam) ws = 0;
        trsnd++;
    }
}

else
    cerr << "PP: kt: Key thread " << tid << " reached timeout while
        receiving message" << endl;
    zmq_msg_close(&rmsg);
}

*trc = false;
pthread_join(threadPK, RETVAL2);
}

delete trc;
delete block;
KEY_DB.set_ready(dbnr, false);
zmq_setsockopt (ThreadPULLSocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
zmq_unbind(ThreadPULLSocket, pullAddr);
zmq_close(ThreadPULLSocket);
zmq_setsockopt (ThreadPUSHSocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
zmq_close(ThreadPUSHSocket);
KEY_DB.set_rcv_addr(dbnr, NULL);
}
else
    cerr << "PP: kt: Error while creating zmq sockets. Errno: " <<
        zmq_strerror(errno) << endl;

KEY_DB.reset_occupied(dbnr);
PROXY_DB.change_TA(false);
pthread_exit(NULL);
}

```

```
//===== Clients Threads =====//

//Data Thread for clients
void *ClientDataThread(void *args)
{
    thread_data *my_data = (thread_data *)args;
    long tid = (long)my_data->thread_id;
    int cdbnr = tid-MAX_CLIENTS-TID_SHIFT;
    bool* condition = (bool*)my_data->thread_condition;

    int rc, hwm = 1;
    int clientbuffersize = MAX_EV_SIZE * CLIENT_HWM;
    void* ThreadPULLSocket = zmq_socket(CONTEXT,ZMQ_PULL);
    void* ClDataSocket = zmq_socket(CONTEXT,ZMQ_PUSH);

    if(ThreadPULLSocket!=NULL && ClDataSocket!=NULL)
    {
        zmq_setsockopt(ThreadPULLSocket, ZMQ_RCVTIMEO, &INNER_THREADS_TIMEOUT,
            sizeof(INNER_THREADS_TIMEOUT));
        zmq_setsockopt(ThreadPULLSocket, ZMQ_RCVHWM, &hwm, sizeof(hwm));
        zmq_setsockopt(ClDataSocket, ZMQ_SNDHWM, &CLIENT_HWM, sizeof(CLIENT_HWM));
        zmq_setsockopt(ClDataSocket, ZMQ_SNDBUF, &clientbuffersize,
            sizeof(clientbuffersize));

        char fullClAddr[strlen(PROXY_DB.get_data_addr(cdbnr))+1];
        strcpy(fullClAddr,PROXY_DB.get_data_addr(cdbnr));
        rc = zmq_connect(ClDataSocket,fullClAddr);

        if(rc==0)
        {
            char* gcp = PROXY_DB.get_comm_port(cdbnr);
            char pullAddr[strlen("inproc://")+strlen(gcp)+1];
            strcpy(pullAddr,"inproc://"); strcat(pullAddr, gcp);
            zmq_bind(ThreadPULLSocket,pullAddr);
            PROXY_DB.set_connected(cdbnr,true);

            int p;
            struct eventStruct event;
            zmq_msg_t ev, smsg;

            while(*condition==true && *ALIVE==true)
            {
                rc = zmq_msg_init(&ev);
                rc = zmq_msg_recv(&ev, ThreadPULLSocket, 0);

                if(rc>0)
                {
                    memcpy(&event,zmq_msg_data(&ev),sizeof(event));

                    try_again:
                    zmq_msg_init(&smsg);
                    zmq_msg_copy(&smsg,&ev);
                    rc = zmq_msg_send(&smsg, ClDataSocket, ZMQ_DONTWAIT);
                    zmq_msg_close(&smsg);

                    if(rc== -1)
```

```

    {
        p = checkPref(event, cdbnr);
        if(p==0 && *condition==true && *ALIVE==true)
        {
            cout << "PP: cdt: Trying again" << endl;
            goto try_again;
        }
        else
            cout << "cdt: Client thread " << tid << " dropped an event due to
                an error: " << zmq_strerror(errno) << " (buffer full)" << endl;
    }
}
else
    cout << "PP: cdt: Client thread " << tid << " reached timeout while
        receiving message" << endl;
    zmq_msg_close(&ev);
}

    PROXY_DB.set_connected(cdbnr,false);
    zmq_setsockopt(ThreadPULLSocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
    zmq_unbind(ThreadPULLSocket,pullAddr);
    zmq_close(ThreadPULLSocket);
    cout << "PP: cdt: Thread PULL socket closed" << endl;
    zmq_setsockopt(ClDataSocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
    zmq_disconnect(ClDataSocket,fullClAddr);
}
    zmq_close(ClDataSocket);
}
else
    cerr << "PP: cdt: Error while creating zmq sockets. Errno: " <<
        zmq_strerror(errno) << endl;

pthread_exit(NULL);
}

//Heartbeat Thread for clients
void *ClientHbThread(void *id)
{
    int tid = *((int*)id);
    int cdbnr = tid-TID_SHIFT;
    PROXY_DB.change_TA(true);
    pthread_detach(pthread_self());

    bool* trc = new bool(false);
    char* port = PROXY_DB.get_comm_port(cdbnr);
    char hbAddr[strlen("tcp://*:") + strlen(port) + 1];
    strcpy(hbAddr,"tcp://*:"); strcat(hbAddr, port);
    void* ClCommSocket = zmq_socket(CONTEXT, ZMQ_REP);
    if(ClCommSocket!=NULL)
    {
        int rc;
        zmq_setsockopt(ClCommSocket, ZMQ_RCVTIMEO, &CLIENT_FH_TIMEOUT,
            sizeof(CLIENT_FH_TIMEOUT));
        zmq_setsockopt(ClCommSocket, ZMQ_SNDTIMEO, &CLIENT_FH_TIMEOUT,
            sizeof(CLIENT_FH_TIMEOUT));
    }
}

```

```
rc = zmq_bind(ClCommSocket,hbAddr);
zmq_msg_t hb;

wrong_client:
zmq_msg_init(&hb);
rc = zmq_msg_recv(&hb,ClCommSocket,0);

if(rc != -1)
{
    struct mbpConnection pmsg;
    memcpy(&pmsg, zmq_msg_data(&hb), sizeof(pmsg));
    struct mbpConnection pmsgo;

    if(strcmp(PROXY_DB.get_client_key(cdbnr),pmsg.key) == 0)
    {
        char* key = pmsg.key;
        KEY_DB.addKeyThreadClient(key,cdbnr);

        *trc = true;
        thread_data td;
        fillThreadData(&td, PROXY_DB.get_thread_id(cdbnr)+MAX_CLIENTS,trc,
            NULL);
        pthread_t threadCD;
        rc = pthread_create(&threadCD, NULL, ClientDataThread,(void*)&td);

        if(rc==0)
        {
            while(rc != -1 && *trc == true && *ALIVE==true)
            {
                rc = zmq_msg_send(&hb,ClCommSocket,0);
                zmq_msg_close(&hb);
                if(rc !=-1)
                {
                    zmq_msg_init(&hb);
                    rc = zmq_msg_recv(&hb,ClCommSocket,0);

                    if(rc != -1)
                    {
                        memcpy(&pmsgo, &pmsg, sizeof(pmsg));
                        memcpy(&pmsg, zmq_msg_data(&hb), sizeof(pmsg));

                        if(memcmp(&pmsgo,&pmsg,sizeof(pmsg))!=0)
                            PROXY_DB.updateDb(cdbnr, &pmsg);
                    }
                    else
                        cerr << "PP: chbt: Cannot receive a heartbeat from client: " <<
                            cdbnr << endl;
                }
                else
                    zmq_msg_init(&hb);
            }

            *trc = false;
            KEY_DB.removeKeyThreadClient(key,cdbnr);
            cout << "PP: chbt: Waiting for data thread to close" << endl;
            int s = pthread_join(threadCD,RETVAL2);
```

```

        PROXY_DB.set_connected(cdbnr, false);
        if(s!=0)
            cout << "PP: chbt: Error with data thread - did not exit" << endl;
    }
    else
        cout << "PP: chbt: Thread for data cannot be created" << endl;
    }
    else
    {
        memcpy(zmq_msg_data(&hb), "error", 6);
        rc = zmq_msg_send(&hb, ClCommSocket, ZMQ_DONTWAIT);
        cout << "hbt: Wrong client tried to connect" << endl;
        goto wrong_client;
    }
    }
    else
        cout << "PP: chbt: Did not receive correct first heartbeat" << endl;
}
else
    cout << "PP: chbt: Cannot create zmq socket" << endl;

if(*ALIVE==true)
    PROXY_DB.set_pref_nb(cdbnr, 0);

zmq_setsockopt(ClCommSocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
zmq_unbind(ClCommSocket, hbAddr);
zmq_close(ClCommSocket);
delete trc;

if(*ALIVE==true)
    PROXY_DB.set_client_key(cdbnr, (char*)"");

PROXY_DB.set_occupied(cdbnr, false);
PROXY_DB.change_TA(false);
pthread_exit(NULL);
}

//===== MAIN =====//

int main(int argc, char **argv)
{
    srand(time(NULL));
    int rc;
    *ALIVE = true;
    CONTEXT = zmq_ctx_new();

    SP_PORT = getPort("SPPP_SourceDataPort");
    SP_HB_PORT = getPort("SPPP_SourceHeartbeatPort");
    INNER_THREADS_TIMEOUT = 1000;
    LINGER = 0;

    MAX_EV_SIZE = getLimit("MaxEventSize");
    SP_HB_TIMEOUT = getLimit("PP_SourceHeartbeatTimeout");
    SP_HB_SND_TIMEOUT = SP_HB_TIMEOUT/4;
    SP_RCV_HWM = getLimit("PP_ReceiveHWM");

```

```

SP_RCV_TIMEOUT = getLimit("PP_ReceiveTimeout");
CLIENT_COMM_PORT = getPort("PP_ClientsCommPort");
MAX_CLIENTS = getLimit("PP_MaxClients");
MAX_PREFERENCES = getLimit("PP_MaxPreferences");
CLIENT_FH_TIMEOUT = getLimit("PP_ClientFirstHeartbeatTimeout");
CLIENT_HWM = getLimit("PP_ClientHWM");

char* cfport = NULL;
cfport = getPort("PP_ClientFirstPort");
CLIENT_FIRST_PORT = atoi(cfport);
free(cfport);

char* end;
if(argc>1)
    SP_RCV_HWM = (int)strtol(argv[1],&end,10);

PROXY_DB.fillDb();
KEY_DB.fillDb();

pthread_t buffer_thread;
int sdt_id = 1;
rc = pthread_create(&buffer_thread, NULL, SourceDataThread,(void*)&sdt_id);
if(rc!=0){
    cerr << "PP: Error: unable to create thread, error: " <<
        zmq_strerror(errno) << endl;
    exit(-1);
}

pthread_t buffer_hb_thread;
int shbt_id = 2;
rc = pthread_create(&buffer_hb_thread, NULL, SourceHbThread,(void*)&shbt_id);
if(rc!=0){
    cerr << "PP: Error: unable to create thread, error: " <<
        zmq_strerror(errno) << endl;
    exit(-1);
}

void* COMMsocket = zmq_socket(CONTEXT, ZMQ_REP);
if(COMMsocket==NULL){
    cerr << "PP: Error: unable to create zmq socket, error: " <<
        zmq_strerror(errno) << endl;
    exit(-1);
}
zmq_setsockopt(COMMsocket, ZMQ_RCVTIMEO, &INNER_THREADS_TIMEOUT,
    sizeof(INNER_THREADS_TIMEOUT));
char addr[strlen("tcp://*:") + strlen(CLIENT_COMM_PORT) + 1];
strcpy(addr, "tcp://*:"); strcat(addr, CLIENT_COMM_PORT);
rc = zmq_bind(COMMsocket, addr);

if(rc==0)
{
    zmq_msg_t clientmsg;
    int cslot;

    fd_set readfds;
    FD_ZERO(&readfds);

```

```

FD_SET(STDIN_FILENO, &readfds);
fd_set savefds = readfds;

struct timeval usertimeout;
usertimeout.tv_sec = 0;
usertimeout.tv_usec = 0;
string input;

//waiting for new client
while(*ALIVE==true)
{
    if(select(1, &readfds, NULL, NULL, &usertimeout))
    {
        cin >> input;
        if(input.compare("quit")==0)
        {
            cout << "PP: Quitting program" << endl;
            *ALIVE=false;
        }
    }
    readfds = savefds;

    zmq_msg_init(&clientmsg);
    rc = zmq_msg_recv(&clientmsg, COMMsocket, 0);
    if(rc!=-1)
    {
        cslot = PROXY_DB.findFirstFreeSlot();
        if(cslot!=-1)
        {
            struct mbpConnection client;
            memcpy(&client, zmq_msg_data(&clientmsg), zmq_msg_size(&clientmsg));

            PROXY_DB.set_occupied(cslot, true);
            PROXY_DB.updateDb(cslot, &client);

            rc = PROXY_DB.createHBThread(cslot);
            if(rc!=0)
            {
                PROXY_DB.set_occupied(cslot, false);
                sendVal("-1", COMMsocket);
            }
            else
                sendVal(PROXY_DB.get_comm_port(cslot), COMMsocket);
        }
        else
            sendVal("-1", COMMsocket);
    }
    zmq_msg_close(&clientmsg);
}
else
{
    cerr << "PP: Error: cannot bind socket for new clients, error: " <<
        zmq_strerror(errno) << endl;
    exit(-1);
}

```

```

rc = pthread_join(buffer_thread, RETVAL2);
buffer_thread = NULL;
rc = pthread_join(buffer_hb_thread, RETVAL2);
buffer_hb_thread = NULL;

while(PROXY_DB.get_TA()!=0){}

zmq_setsockopt(COMMsocket, ZMQ_LINGER, &LINGER, sizeof (LINGER));
zmq_unbind(COMMsocket, addr);
zmq_close(COMMsocket);

PROXY_DB.freeDb();
KEY_DB.freeDb();

delete ALIVE;
free(SP_PORT);
SP_PORT = NULL;
free(SP_HB_PORT);
SP_HB_PORT = NULL;
free(CLIENT_COMM_PORT);
CLIENT_COMM_PORT = NULL;

zmq_ctx_destroy(CONTEXT);
return 0;
}

```

Makefile Source code:

MakefilePP

```

DATE_CPPFLAGS := $(shell date-config --cflags)
DATE_LIBS      := $(shell date-config --libs --monitorlibs=dyn)

all: PP.cpp
    gcc $(DATE_CPPFLAGS) $(DATE_LIBS) -O2 -std=c++0x -pthread -g
        -L/opt/date/mbp/Linux -lzmq -lmbp -o PP PP.cpp

clean:
    rm PP

```

Appendix C

Libraries source code

monitoringbyproxy.h

```
#ifndef __monitoringbyproxy_h__
#define __monitoringbyproxy_h__

#include <stdbool.h>
#include <pthread.h>
#include "zmq.h"
#include "event.h"
#include "dateDbFile.h"
#define MAX_PROXADDR_LEN 1024
#define PROX_CONF_FILE "proxies.config"

#ifdef __CINT__
# ifdef __cplusplus
extern "C" {
# endif
#endif

int RETVAL;
void** RETVAL2 = NULL;

typedef struct mbpPreference{
    char percent[10]; //default: 100 (-1 => 100%+block)
    char eventType[50]; //default: * = ALL
    char eventSize[20]; //default: * = ALL (size of whole event in bytes)
    char eventTriggerPattern[100]; //default: * = ALL
    char eventDetectorPattern[100]; //default: * = ALL
    char eventTypeAttribute[100]; //default: * = ALL
    char eventGdcId[10]; //default: -1 = ALL
}mbpPreference;

typedef struct mbpPreferenceConverted{
    int percent; // -1 - 100 (-1 => 100%+block)
    int eventType; // 0 - 14 (0 = ALL)
    int eventSizeSign; // 0 = ALL size, 1=">", 2=">=", 3="<", 4="<=" 5="="
    int eventSize; // 0 - 65535 (size in bytes)
    char eventTriggerPattern[100]; //default: * = ALL
    char eventDetectorPattern[100]; //default: * = ALL
    char eventTypeAttribute[100]; //default: * = ALL
    long eventGdcId; // -1 - 65535 (-1 = ALL)
}mbpPreferenceConverted;
```

```
typedef struct mbpConnection{
    bool connected;
    char key[100];
    char dataaddr[100];
    int PrefNb;
    struct mbpPreferenceConverted preferences[100];
    void* datasocket;
    char** commaddrs;
    pthread_t hbthread[100];
    bool threadalive[100];
    bool wait;
}mbpConnection;

int  getIPs(char***);
char* getPort(const char*);
int  getLimit(const char*);

/* public */
int  mbpCreateContext();
struct mbpConnection* mbpCreateConnection(char*, char*);
int  mbpSetPreferences(mbpConnection*, mbpPreference*, int);
int  mbpConnect(mbpConnection*, int);
int  mbpGetData(mbpConnection*, struct eventStruct*);
int  mbpDisconnect(mbpConnection*);
void mbpDestroyConnection(mbpConnection*);
int  mbpDestroyContext();
void mbpControlWait(mbpConnection*, bool);
void mbpSetWait(mbpConnection*);
void mbpSetNoWait(mbpConnection*);
void mbpSetNoWaitTimeout(mbpConnection*, int);
int  mbpFlush(mbpConnection*);

/*===== The "old" deprecated entries from old monitoring library =====*/

/* The clients interface */
int  monitorSetDataSource( const char * ); /* 0: OK, else: error */
int  monitorDeclareTable( char * const * ); /* 0: OK, else: error */
int  monitorDeclareTableExtended( char * const * ); /* 0: OK, else: error */
int  monitorDeclareTableWithAttributes( char * const * ); /* 0: OK, else:
    error */
int  monitorDeclareMp( const char * ); /* 0: OK, else: error */
const char *monitorDecodeError( int error );/* NULL: error, else: OK */
int  monitorGetEvent( void *buffer, eventSizeType sizeOfBuffer );/* 0: OK,
    else: error */
int  monitorGetEventDynamic( void** );/* 0: OK, else: error */
int  monitorSetWait();/* 0: OK, else: error */
int  monitorSetNowait();/* 0: OK, else: error */
int  monitorControlWait( int );/* 0: OK, else: error */
int  monitorSetWaitTimeout( int );/* 0: OK, else: error */
int  monitorSetSwap( int, int ); /* 0: OK, else: error */
int  monitorFlushEvents(); /* 0: OK, else: error */
int  monitorLogout();/* 0: OK, else: error */
```

```

/* The "old" deprecated entries: */
int  SetDataSource( const char* );/* 0: OK, else: error */
void SetGetEventInteractive();/* 0: OK, else: error */
void SetGetEventBatch(); /* 0: OK, else: error */
void FlushGetEventBuffer();/* 0: OK, else: error*/
int  getevent( char *ptr, eventSizeType size ); /* >0:OK (byte count), else:
        error */

#ifdef __CINT__
# ifdef __cplusplus
}
# endif
#endif

#endif // __monitor_h__

```

monitoringbyproxy.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <unistd.h>
#include <limits.h>
#include <time.h>
#include <ifaddrs.h>
#include "monitoringbyproxy.h"

void *CONTEXT;
char **PROXIES;
int PA;

// macro used for "shortened" strings
#define MATCHES( t, s ) ( strncasecmp( t, s, strlen( s ) ) == 0 )

/* ===== monitoring by proxy new functions ===== */

//===== Reading config file functions =====//

int getIPs(char*** proxies)
{
    int error = dbOpen();
    if(error!=-1)
    {
        int amount;
        dbFile db = NULL;
        error = dbFile_open(PROX_CONF_FILE,"",&db);
        if(error==0 && db!=NULL)
        {
            error = dbFile_getNextSection(&db, "Proxies");
            if(error==0)
            {
                char** pr = NULL;
                error = dbFile_getVal_charArray(&db, "ProxyList", &pr, &amount);
            }
        }
    }
}

```

```
dbFile_close(&db);
db = NULL;
dbClose();

if(error==0)
{
    char addr[strlen("tcp://") + strlen("000.000.000.000") +
        strlen(":")+1];
    *proxies = (char**)malloc((sizeof(addr))*amount);
    for(int i=0;i<amount;i++)
    {
        (*proxies)[i] = (char*)malloc(sizeof(addr));
        strcpy(addr,"tcp://");
        strcat(addr,pr[i]);
        strcat(addr,":");
        memcpy((*proxies)[i],addr,sizeof(addr)-1);
    }
    for(int i=0;i<amount;i++)
        if(pr[i]!=NULL)
            free(pr[i]);
    if(pr!=NULL)
        free(pr);
    return amount;
}
else
{
    for(int i=0;i<amount;i++)
        if(pr[i]!=NULL)
            free(pr[i]);
    if(pr!=NULL)
        free(pr);
    fprintf(stderr, "mbp: No ProxyList variable inside the configuration
        file\n");
    return -1;
}
}
else
{
    fprintf(stderr, "mbp: No Proxies section inside the configuration
        file\n");
    dbFile_close(&db);
    db = NULL;
    dbClose();
    return -1;
}
}
else
{
    printf("mbp: Proxies configuration file missing\n");
    dbClose();
    return -1;
}
}
fprintf(stderr, "mbp: Cannot open database\n");
return -1;
}
```

```

char* getPort(const char* type)
{
    int error = dbOpen();
    if(error!=-1)
    {
        char* port = NULL;
        dbFile db = NULL;
        error = dbFile_open(PROX_CONF_FILE,"",&db);
        if (error==0 && db!=NULL)
        {
            error = dbFile_getNextSection(&db, "Ports");
            if(error==0)
            {
                error = dbFile_getVal(&db, type, &port);
                dbFile_close(&db);
                db = NULL;
                dbClose();
                if(error==0)
                {
                    printf("mbp: Port %s found in config file and set to: %s\n", type,
                        port);
                    return port;
                }
            }
            else
            {
                printf("mbp: No Ports section inside the configuration file\n");
                dbFile_close(&db);
                db = NULL; dbClose();
            }
        }
        else
        {
            printf("mbp: Proxies configuration file missing\n");
            dbFile_close(&db);
            db = NULL; dbClose();
        }
    }

    if(strcmp(type,"SPPP_SourceHeartbeatPort")==0) {port=strdup("49998");}
    else if(strcmp(type,"SPPP_SourceDataPort")==0) {port=strdup("49999");}
    else if(strcmp(type,"PP_ClientsCommPort")==0) {port=strdup("50000");}
    else if(strcmp(type,"PP_ClientFirstPort")==0) {port=strdup("51000");}
    else{port=strdup("unknown");}
    printf("mbp: No port %s found in config file. Default port number is:
        %s\n", type, port);
    if((strcmp (port,"unknown")==0))
    {
        free(port);
        return NULL;
    }
    else
        return port;
}
fprintf(stderr,"mbp: Cannot open database\n");
return NULL;
}

```

```
int getLimit(const char* type)
{
    char* lmt = NULL;
    int error = dbOpen();
    if(error!=-1)
    {
        dbFile db = NULL;
        error = dbFile_open(PROX_CONF_FILE,"",&db);

        if (db!=NULL && error==0)
        {
            error = dbFile_getNextSection(&db, "Limits");
            if(error==0)
            {
                error = dbFile_getVal(&db, type, &lmt);
                dbFile_close(&db);
                db = NULL;
                dbClose();
                if(error==0)
                {
                    int k = atoi(lmt);
                    free(lmt);
                    printf("mbp: Limit %s found in config file and set to: %d\n", type,
                        k);
                    return k;
                }
            }
            else
            {
                printf("mbp: No Limits section inside the configuration file\n");
                dbFile_close(&db);
                db = NULL;
                dbClose();
            }
        }
        else
        {
            printf("mbp: Proxies configuration file missing\n");
            dbFile_close(&db);
            db = NULL;
            dbClose();
        }
    }

    if(strcmp(type,"MaxEventSize")==0) {lmt=strdup("25000000");}
    else if(strcmp(type,"SP_SendHWM")==0) {lmt=strdup("1");}
    else if(strcmp(type,"SP_SendTimeout")==0) {lmt=strdup("1000");}
    else if(strcmp(type,"SP_HeartbeatTimeout")==0) {lmt=strdup("100");}
    else if(strcmp(type,"SP_HeartbeatDelay")==0) {lmt=strdup("1");}
    else if(strcmp(type,"SP_ReconnectSleep")==0) {lmt=strdup("10");}
    else if(strcmp(type,"PP_ReceiveHWM")==0) {lmt=strdup("20");}
    else if(strcmp(type,"PP_ReceiveTimeout")==0) {lmt=strdup("1000");}
    else if(strcmp(type,"PP_ClientHWM")==0) {lmt=strdup("1");}
    else if(strcmp(type,"PP_SourceHeartbeatTimeout")==0) {lmt=strdup("1000");}
    else if(strcmp(type,"PP_MaxClients")==0) {lmt=strdup("1000");}
    else if(strcmp(type,"PP_MaxPreferences")==0) {lmt=strdup("10");}
    else if(strcmp(type,"PP_ClientFirstHeartbeatTimeout")==0)
```

```

        {lmt=strdup("5000");}
    else if(strcmp(type,"PP_SendHWM")==0) {lmt=strdup("1");}
    else if(strcmp(type,"CP_NewPortTimeout")==0) {lmt=strdup("2000");}
    else if(strcmp(type,"CP_HeartbeatTimeout")==0) {lmt=strdup("5000");}
    else{lmt=strdup("unknown");}

    printf("mbp: No limit %s found in config file. Default value is: %s\n",
           type,lmt);
    if((strcmp (lmt,"unknown")==0))
    {
        free(lmt);
        return -1;
    }
    else
    {
        int k = atoi(lmt);
        free(lmt);
        return k;
    }
}
fprintf(stderr,"mbp: Cannot open database\n");
return -1;
}

//===== Threads =====//
struct thread_data{
    int proxynr;
    struct mbpConnection *connection;
};

struct thread_data* FillThreadData(struct mbpConnection *connection, int
proxynr)
{
    struct thread_data* td = malloc(sizeof(struct thread_data));
    td->connection = connection;
    td->proxynr = proxynr;
    return td;
}

//===== Client functions =====//

int mbpCreateContext()
{
    srand(time(NULL));

    if(CONTEXT==NULL)
        CONTEXT = zmq_ctx_new();
    else
    {
        printf("mbp: Warning: Context has been already created!\n");
        return -1;
    }
}

```

```
PA = getIPs(&PROXIES);
if(PA<1)
{
    printf("mbp: No proxies found!\n");
    return -1;
}
return PA;
}

mbpConnection* mbpCreateConnection(char* key, char* addr)
{
    mbpConnection* connection = malloc(sizeof(struct mbpConnection));
    if (connection)
    {
        memset(connection,0,sizeof(struct mbpConnection));
        connection->connected = false;

        snprintf(connection->key,99,"%s",key);

        //allocate memory for proxies (with address and port)
        connection->commaddrs = malloc((sizeof(char*)*PA)+1);
        strcpy(connection->dataaddr,"tcp://");
        strncat(connection->dataaddr,addr,93);

        for(int i=0;i<PA;i++)
            connection->threadalive[i] = false;

        //set default values
        connection->datasocket = NULL;
        connection->PrefNb = 1;
        connection->wait = true;
        connection->preferences->percent = 100;
        connection->preferences->eventType = 0;
        connection->preferences->eventSize = 0;
        sprintf(connection->preferences->eventTriggerPattern,"%s","*");
        sprintf(connection->preferences->eventDetectorPattern,"%s","*");
        sprintf(connection->preferences->eventTypeAttribute,"%s","*");
        connection->preferences->eventGdcId = -1;
        printf("mbp: Connection created with key: %s, and default preferences:
            [%d, %d, %d, %s, %s, %s, %d]\n",connection->key,
            connection->preferences->percent, connection->preferences->eventType,
            connection->preferences->eventSize,
            connection->preferences->eventTriggerPattern,
            connection->preferences->eventDetectorPattern,
            connection->preferences->eventTypeAttribute,
            connection->preferences->eventGdcId);

        return connection;
    }
    else
        return NULL;
}
```

```

int mbpSetPreferences(mbpConnection *connection, mbpPreference
    *newPreferences, int PrefNb)
{
    int pref = 0;
    mbpPreference preference;
    if(connection->key!=NULL) //if connection exists
    {
        for (int l=0; l<PrefNb; l++) //set new preferences
        {
            if(l<100)
            {
                preference = *newPreferences;

                if(preference.percent[0]=='\0')
                    (connection->preferences[l]).percent=100;
                else
                {
                    char* ptr;
                    long int di = strtol(preference.percent,&ptr,10);
                    if(di==0)
                    {
                        char* str = preference.percent;
                        int percent;
                        char* ptr;
                        if (isdigit((int)str[0]))
                        {
                            percent = strtol(str,&ptr,10);
                        } else if ( MATCHES( "monitor", str ) ||
                            MATCHES( "yes", str )){
                            percent = 50;
                        } else if ( MATCHES( "ignore", str ) ||
                            MATCHES( "no", str ) ||
                            MATCHES( "dont", str ) ) {
                            percent = 0;
                        } else if ( MATCHES( "all", str ) ||
                            MATCHES( "must", str ) ||
                            MATCHES( "mustall", str ) ||
                            MATCHES( "must_all", str ) ||
                            MATCHES( "must all", str ) ) {
                            percent = -1;
                        } else if ( MATCHES( "most", str ) ) {
                            percent = 100;
                        } else if ( MATCHES( "few", str ) ) {
                            percent = 25;
                        } else {
                            fprintf(stderr, "mbp: Error while setting preferences. Percent
                                value in %d preference is invalid\n",l);
                            return -1;
                        }
                    }
                    (connection->preferences[l]).percent = percent;
                }
            }
        }

        if(preference.eventType[0]=='\0')
            (connection->preferences[l]).eventType = 0;
        else

```

```
{
    char* str = preference.eventType;
    int type = -1;

    if(str[0]=='*')
        return 0;

    if ( MATCHES( "all", str ) || MATCHES( "all events", str ) ||
        MATCHES( "*", str ) || MATCHES( "-1", str ) ) {
        type = 0;
    } else if ( ( MATCHES( "start of run files", str ) && !MATCHES(
        "start of run", str ) ) ||
        ( MATCHES( "start_of_run_files", str ) && !MATCHES(
        "start_of_run", str ) ) ||
        ( MATCHES( "sorf", str ) && !MATCHES( "sor", str ) ) || MATCHES(
        "3", str ) ) {
        type = 3;
    } else if ( ( MATCHES( "end of run files", str ) && !MATCHES( "end of
        run", str ) ) ||
        ( MATCHES( "end_of_run_files", str ) && !MATCHES( "end_of_run",
        str ) ) ||
        ( MATCHES( "eorf", str ) && !MATCHES( "eor", str ) ) || MATCHES(
        "4", str ) ) {
        type = 4;
    } else if ( MATCHES( "start of burst", str ) ||
        MATCHES( "start_of_burst", str ) ||
        MATCHES( "sob", str ) || MATCHES( "5", str ) ) {
        type = 5;
    } else if ( MATCHES( "end of burst", str ) ||
        MATCHES( "end_of_burst", str ) ||
        MATCHES( "eob", str ) || MATCHES( "6", str ) ) {
        type = 6;
    } else if ( MATCHES( "physics event", str ) ||
        MATCHES( "physics_event", str ) || MATCHES( "7", str ) ) {
        type = 7;
    } else if ( MATCHES( "calibration event", str ) ||
        MATCHES( "calibration_event", str ) || MATCHES( "8", str ) ) {
        type = 8;
    } else if ( MATCHES( "start of run", str ) ||
        MATCHES( "start_of_run", str ) ||
        MATCHES( "sor", str ) || MATCHES( "1", str ) ) {
        type = 1;
    } else if ( MATCHES( "end of run", str ) ||
        MATCHES( "end_of_run", str ) ||
        MATCHES( "eor", str ) || MATCHES( "2", str ) ) {
        type = 2;
    } else if ( MATCHES( "event format error", str ) ||
        MATCHES( "event_format_error", str ) ||
        MATCHES( "ferr", str ) || MATCHES( "9", str ) ) {
        type = 9;
    } else if ( MATCHES( "start of data", str ) ||
        MATCHES( "start_of_data", str ) ||
        MATCHES( "sod", str ) || MATCHES( "10", str ) ) {
        type = 10;
    } else if ( MATCHES( "end of data", str ) ||
        MATCHES( "end_of_data", str ) ||
```

```

        MATCHES( "eod", str ) || MATCHES( "11", str )) {
    type = 11;
} else if ( MATCHES( "system software trigger event", str ) ||
    MATCHES( "system software trigger event", str ) ||
    MATCHES( "sst", str ) || MATCHES( "12", str ) ) {
    type = 12;
} else if ( MATCHES( "detector software trigger event", str ) ||
    MATCHES( "detector software trigger event", str ) ||
    MATCHES( "dst", str ) || MATCHES( "13", str ) ) {
    type = 13;
} else if ( MATCHES( "sync_event", str ) ||
    MATCHES( "sync event", str ) || MATCHES( "14", str ) ) {
    type = 14;
} else {
    fprintf(stderr, "mbp: Error while setting preferences. EventType
        value in %d preference is invalid\n",1);
    return -1;
}
}

if(preference.eventSize[0]=='\0')
    (connection->preferences[1]).eventSizeSign = 0;
else
{
    char* str = preference.eventSize;
    if(str==NULL)
    {
        fprintf(stderr, "mbp: Error while setting preferences. EventSize
            value in %d preference is invalid\n",1);
        return -1;
    }

    // "*" = disable selection criteria
    if(str[0]=='*' || (str[0]=='a' && str[1]=='l' &&
        str[2]=='l') || (str[0]=='A' && ((str[1]=='L' &&
        str[2]=='L') || (str[1]=='l' && str[2]=='l'))))
        (connection->preferences[1]).eventSize = 0; //return 0;

    char sign[2];
    char size[19];
    int sn=0, se=0;

    for (int i = 0; i<strlen(str); i++ ) {
        if ( str[i] == '<' || str[i] == '=' || str[i] == '>')
        {
            if(se!=0 || sn>1)
            {
                fprintf(stderr, "mbp: Error while setting preferences.
                    EventSize value in %d preference is invalid\n",1);
                return -1;
            }

            if ( sscanf( &str[i], "%c", &sign[sn] ) != 1 )
            {
                fprintf(stderr, "mbp: Error while setting preferences.
                    EventSize value in %d preference is invalid\n",1);

```

```

        return -1;
    }

    sn++;
}
else if (isdigit((int)str[i]))
{
    if(sn==0)
    {
        fprintf(stderr, "mbp: Error while setting preferences.
            EventSize value in %d preference is invalid\n",1);
        return -1;
    }

    if ( sscanf( &str[i], "%d", &size[se] ) != 1 )
    {
        fprintf(stderr, "mbp: Error while setting preferences.
            EventSize value in %d preference is invalid\n",1);
        return -1;
    }

    se++;
}
else
{
    fprintf(stderr, "mbp: Error while setting preferences. EventSize
        value in %d preference is invalid\n",1);
    return -1;
}
}

if(sign[0] == '>')
    if(sign[1] == '=')
        (connection->preferences[1]).eventSizeSign = 2;
    else
        (connection->preferences[1]).eventSizeSign = 1;
else if(sign[0]=='<')
    if(sign[1] == '=')
        (connection->preferences[1]).eventSizeSign = 4;
    else
        (connection->preferences[1]).eventSizeSign = 3;
else if(sign[0] == '=')
    (connection->preferences[1]).eventSizeSign = 5;
else
    return -1;

(connection->preferences[1]).eventSize = atoi(size);
}

if(preference.eventTriggerPattern[0]=='\0')
    sprintf((connection->preferences[1]).eventTriggerPattern, "%s", "*");
if(preference.eventDetectorPattern[0]=='\0')
    sprintf((connection->preferences[1]).eventDetectorPattern, "%s", "*");
if(preference.eventTypeAttribute[0]=='\0')
    sprintf((connection->preferences[1]).eventTypeAttribute, "%s", "*");

```

```

        if(preference.eventGdcId[0]=='\0')
            (connection->preferences[1]).eventGdcId = -1;

        newPreferences+=1;
        pref++;
    }
}
connection->PrefNb=PrefNb;
newPreferences-=PrefNb;
}
else
    return -1;

return pref;
}

void *mbpHeartbeatThread(void *args)
{
    struct thread_data *my_data = (struct thread_data *)args;
    int proxynr = (int)my_data->proxynr;
    struct mbpConnection* connection = (struct
        mbpConnection*)my_data->connection;
    connection->threadalive[proxynr]=true;

    void* UCommSocket = zmq_socket(CONTEXT, ZMQ_REQ);
    int timeout = getLimit("CP_HeartbeatTimeout");
    if(UCommSocket!=NULL)
    {
        int rc = zmq_setsockopt(UCommSocket, ZMQ_RCVTIMEO, &timeout,
            sizeof(timeout));
        rc = zmq_connect(UCommSocket, connection->commaddrs[proxynr]);
        if(rc==0)
        {
            while(connection->connected==true && rc!=-1)
            {
                if(rc != -1)
                {
                    zmq_msg_t hbmsg;
                    rc = zmq_msg_init_size(&hbmsg, sizeof(*connection));
                    memcpy(zmq_msg_data(&hbmsg), connection, zmq_msg_size(&hbmsg));

                    rc = zmq_msg_send(&hbmsg, UCommSocket, 0);
                    zmq_msg_close(&hbmsg);
                }

                if(rc != -1)
                {
                    zmq_msg_t rmsg;
                    zmq_msg_init(&rmsg);
                    rc = zmq_msg_recv(&rmsg, UCommSocket, 0);
                    if(rc==-1 || strcmp(zmq_msg_data(&rmsg), (const char*)connection)!=0)
                        printf("mbp: Error in received HB\n");
                    zmq_msg_close(&rmsg);
                }
            }
        }
    }
}

```

```
    }
    zmq_disconnect(UCommSocket, connection->commaddrs[proxynr]);
}
else
    fprintf(stderr,"mbp: Error: Cannot connect to heartbeat port for proxy:
               %s, zmq error:
               %s\n",connection->commaddrs[proxynr],zmq_strerror(errno));

    zmq_close(UCommSocket);
}
else
    fprintf(stderr,"mbp: Error: Cannot create socket for heartbeat for proxy:
               %s, zmq error:
               %s\n",connection->commaddrs[proxynr],zmq_strerror(errno));

printf("mbp: Disconnected from proxy: %s\n",connection->commaddrs[proxynr]);
free(connection->commaddrs[proxynr]);
connection->threadalive[proxynr] = false;
free(args);
}
```

```
int mbpConnect(struct mbpConnection *connection, int buffhwm)
{
    int ptc = PA;
    int rc = 0;
    int timeout = getLimit("CP_NewPortTimeout");

    if(connection!=NULL)
    {
        connection->datasocket = zmq_socket(CONTEXT, ZMQ_PULL);
        if(connection->datasocket!=NULL)
        {
            int maxevsize = getLimit("MaxEventSize");
            int buffsize = maxevsize * buffhwm;
            zmq_setsockopt(connection->datasocket, ZMQ_RCVHWM, &buffhwm,
                           sizeof(buffhwm));
            zmq_setsockopt(connection->datasocket, ZMQ_RCVBUF, &buffsize,
                           sizeof(buffsize));
            rc = zmq_bind(connection->datasocket, connection->dataaddr);
            if(rc!=0)
            {
                connection->connected = false;
                fprintf(stderr,"mbp: Error: Cannot bind socket for data pull for
                               conection %s due to zmq error: %s\n", connection->dataaddr,
                               zmq_strerror(errno));
                zmq_close(connection->datasocket);
                connection->datasocket = NULL;
            }
            else
            {
                connection->connected = true;

                char* ccport = getPort("PP_ClientsCommPort");
                int portlen = 6;
            }
        }
    }
}
```

```

for(int i=0;i<PA;i++)
{
    void* CommSocket = zmq_socket(CONTEXT, ZMQ_REQ);
    if(CommSocket!=NULL)
    {
        char newClientConn[strlen(PROXIES[i])+strlen(ccport)+1];
        strcpy(newClientConn,PROXIES[i]);
        strcat(newClientConn,ccport);
        zmq_setsockopt(CommSocket, ZMQ_RCVTIMEO, &timeout, sizeof(timeout));
        zmq_setsockopt(CommSocket, ZMQ_SNDTIMEO, &timeout, sizeof(timeout));
        rc = zmq_connect(CommSocket, newClientConn);

        if(rc!=0)
        {
            zmq_close(CommSocket);
            fprintf(stderr, "mbp: Error: Cannot connect to proxy COMM Port
                due to ZMQ error: %s\n", zmq_strerror(errno));
        }
        else
        {
            zmq_msg_t msg;
            rc = zmq_msg_init_size(&msg,sizeof(*connection));
            memcpy(zmq_msg_data(&msg),connection, sizeof(*connection));
            rc = zmq_msg_send(&msg, CommSocket, 0);
            zmq_msg_close(&msg);

            zmq_msg_t port;
            zmq_msg_init(&port);
            rc = zmq_msg_recv(&port, CommSocket, 0);
            if(rc== -1)
            {
                fprintf(stderr,"mbp: Warning: Connection with proxy nr %d
                    cannot be done by error: %s\n",i+1,zmq_strerror(errno));
                zmq_msg_close(&port);
                zmq_disconnect(CommSocket, newClientConn);
                zmq_close(CommSocket);
            }
            else if(rc>=portlen || rc==0)
            {
                fprintf(stderr,"mbp: Received port is too short or too
                    long\n");
                zmq_msg_close(&port);
                rc = zmq_disconnect(CommSocket, newClientConn); assert(rc==0);
                zmq_close(CommSocket);
            }
            else //if everything is ok
            {
                char* UCommPort = (char*)malloc(sizeof(char)*portlen);
                memcpy(UCommPort, zmq_msg_data(&port),rc);
                UCommPort[rc]='\0';
                printf ("mbp: Received new port: %s\n", UCommPort);

                zmq_msg_close(&port);
                zmq_disconnect(CommSocket, newClientConn);
                zmq_close(CommSocket);
            }
        }
    }
}

```

```
if(atoi(UCommPort)==0)
    fprintf(stderr, "mbp: Received port is not a number!\n");
else if(atoi(UCommPort)==-1)
    fprintf(stderr, "mbp: No free ports in proxy %s!\n",
            PROXIES[i]);
else
{
    char ucommsockaddr[strlen(PROXIES[i])+strlen(UCommPort)+1];
    strcpy(ucommsockaddr,PROXIES[i]);
    strcat(ucommsockaddr,UCommPort);
    connection->commaddrs[i] =
        (char*)malloc(strlen(ucommsockaddr)+1);
    memcpy(connection->commaddrs[i], &ucommsockaddr[0],
            sizeof(ucommsockaddr));

    struct thread_data* td = FillThreadData(connection, i);
    rc = pthread_create(&(connection->hbthread[i]), NULL,
        mbpHeartbeatThread,(void*)td);

    if(rc==0)
        ptc-=1;
    else
    {
        fprintf(stderr,"mbp: Error: Cannot create heartbeat thread
            for connection %s!\n",PROXIES[i]);
        free(connection->commaddrs[i]);
        connection->commaddrs[i] = NULL;
        free(td);
        td = NULL;
    }
}
free(UCommPort);
}
}
}
}
free(ccport);
}
}

if(ptc==PA)
{
    fprintf(stderr,"mbp: Error: None proxy connected!\n");
    return -1;
}
else
{
    fprintf(stderr,"mbp: Error: Connection does not exist!\n");
    return ptc;
}

return ptc;
}
```

```

void mbpControlWait(struct mbpConnection *connection, bool flag)
{
    if(flag == true)
        connection->wait = 0;
    else
        connection->wait = ZMQ_DONTWAIT;
}

void mbpSetWait(struct mbpConnection *connection)
{
    connection->wait = 0;
}

void mbpSetNoWait(struct mbpConnection *connection)
{
    connection->wait = ZMQ_DONTWAIT;
}

void mbpSetWaitTimeout(struct mbpConnection* connection, int timeout)
{
    if(connection->datasocket!=NULL)
    {
        zmq_setsockopt(connection->datasocket, ZMQ_RCVTIMEO, &timeout,
            sizeof(timeout));
    }
    else
        fprintf(stderr, "mbp: No datasocket to set timeout\n");
}

int mbpGetData(struct mbpConnection* connection, struct eventStruct *event)
{
    if(connection->datasocket!=NULL)
    {
        zmq_msg_t dmsg;
        int rc = zmq_msg_init(&dmsg);
        rc = zmq_msg_recv(&dmsg, connection->datasocket, connection->wait);

        if(rc>0)
        {
            memcpy(event, zmq_msg_data(&dmsg), sizeof(*event));
            zmq_msg_close(&dmsg);
            return rc;
        }
        else if(rc== -1)
        {
            fprintf(stderr, "mbp: ZMQ Error: %s\n", zmq_strerror(errno));
            zmq_msg_close(&dmsg);
            return -1;
        }
        else if(rc==0)
        {
            fprintf(stderr, "mbp: Empty data\n");
            zmq_msg_close(&dmsg);
            return 0;
        }
    }
}

```

```
}
else
{
    fprintf(stderr, "mbp: No datasocket to get data\n");
    return -1;
}
}

int mbpFlush(struct mbpConnection* connection)
{
    int ling = 0;
    if(connection->datasocket != NULL)
    {
        zmq_unbind(connection->datasocket, connection->dataaddr);
        zmq_setsockopt(connection->datasocket, ZMQ_LINGER, &ling, sizeof(ling));
        zmq_close(connection->datasocket);
        connection->datasocket = NULL;
        connection->datasocket = zmq_socket(CONTEXT, ZMQ_PULL);
        int rc = zmq_bind(connection->datasocket, connection->dataaddr);
        if(rc!=0)
        {
            fprintf(stderr, "mbp: Error while flushing events\n");
            return -1;
        }
        return 0;
    }
    else
        return -1;
}

int mbpDisconnect(struct mbpConnection* connection)
{
    connection->connected = false;

    int s=0;
    for(int i=0; i<PA; i++)
    {
        if(connection->threadalive[i]==true)
        {
            s = pthread_join(connection->hbthread[i], RETVAL2);
            if(s!=0)
                fprintf(stderr, "mbp: Error with heartbeat thread %d - did not\n", i);
            printf("mbp: End of heartbeat thread %d\n", i);
        }
        else
            printf("mbp: Heartbeat thread %d does not exists\n", i);
    }

    if(connection!=NULL && connection->datasocket!=NULL)
        zmq_unbind(connection->datasocket, connection->dataaddr);

    printf("mbp: Disconnected\n");
    return 0;
}
```

```

void mbpDestroyConnection(struct mbpConnection* connection)
{
    if(connection!=NULL && connection->datasocket!=NULL)
    {
        int ling = 0;
        zmq_setsockopt(connection->datasocket,ZMQ_LINGER,&ling,sizeof(ling));
        zmq_close(connection->datasocket);
    }

    if(connection->commaddrs!=NULL)
    {
        free(connection->commaddrs);
        connection->commaddrs=NULL;
    }

    if(connection!=NULL)
    {
        free(connection);
        connection = NULL;
    }
}

int mbpDestroyContext()
{
    for(int i=0;i<PA;i++)
        free(PROXIES[i]);
    free(PROXIES);
    return zmq_ctx_destroy(CONTEXT);
}

/*===== The "old" deprecated entries from old monitoring library =====*/

char* oldmp_key;
struct mbpConnection* oldmp_cd;
struct mbpPreference oldmp_pd[] = {"100"};

void printError(char *where, char* error)
{
    fprintf(stderr,"mbp: Error in %s: %s\n", where, error);
    exit(1);
}

int monitorSetDataSource(const char * const dataSource )
{
    mbpCreateContext();
    return 0;
}

int monitorDeclareMp(const char * const mpName )
{
    struct ifaddrs *id;
    int val;
    val = getifaddrs(&id);

```

```
char* ip;
sprintf(ip,"%d",id->ifa_addr);
oldmp_cd = mbpCreateConnection((char*)mpName,ip);
if(oldmp_cd!=NULL)
{
    mbpSetPreferences(oldmp_cd,oldmp_pd,1);
    return 0;
}
else
    printError("monitorDeclareMp","Cannot create connection");
}

int monitorDeclareTable( char * const * const table )
{
    int x=0;
    while(table[x]!=NULL)
    {
        x++;
    }
    if(x==0 || x%2!=0)
        printError("monitorDeclareTable","Table empty or incorrect");
    else
    {
        struct mbpPreference pd[x];
        x=0;
        while(table[x]!=NULL)
        {
            strcpy(pd[x].eventType,table[x]);
            x++;
            strcpy(pd[x].percent,table[x]);
            x++;
        }
        mbpSetPreferences(oldmp_cd,pd,x);
    }
    return 0;
}

int monitorDeclareTableExtended( char * const * const table )
{
    int x=0;
    while(table[x]!=NULL)
    {
        x++;
    }
    if(x==0 || x%4!=0)
        printError("monitorDeclareTableExtended","Table empty or incorrect");
    else
    {
        struct mbpPreference pd[x];
        x=0;
        while(table[x]!=NULL)
        {
            strcpy(pd[x].eventType,table[x]);
            x++;
            strcpy(pd[x].percent,table[x]);
            x++;
        }
    }
}
```

```

        strcpy(pd[x].eventTypeAttribute,table[x]);
        x++;
        strcpy(pd[x].eventTriggerPattern,table[x]);
        x++;
    }
    mbpSetPreferences(oldmp_cd,pd,x);
}
return 0;
}

int monitorDeclareTableWithAttributes( char * const * const table )
{
    int x=0;
    while(table[x]!=NULL)
    {
        x++;
    }
    if(x==0 || x%4!=0)
        printError("monitorDeclareTableExtended","Table empty or incorrect");
    else
    {
        struct mbpPreference pd[x];
        x=0;
        while(table[x]!=NULL)
        {
            strcpy(pd[x].eventType,table[x]);
            x++;
            strcpy(pd[x].percent,table[x]);
            x++;
            strcpy(pd[x].eventTypeAttribute,table[x]);
            x++;
            strcpy(pd[x].eventTriggerPattern,table[x]);
            x++;
        }
        mbpSetPreferences(oldmp_cd,pd,x);
    }
    return 0;
}

const char *monitorDecodeError( const int errorCode )
{
    return "";
}

int monitorGetEvent(void * const buffer,const eventSizeType sizeOfBuffer )
{
    mbpGetData(oldmp_cd, buffer);
    return 0;
}

int monitorGetEventDynamic( void ** const buffer )
{
    mbpGetData(oldmp_cd, *buffer);
    return 0;
}

```

```

int monitorSetWait() {
    mbpSetWait(oldmp_cd);
    return 0;
}

int monitorSetNowait() {
    mbpSetNowait(oldmp_cd);
    return 0;
}

int monitorControlWait( const int waitFlag ) {
    if(waitFlag==0)
        mbpSetWait(oldmp_cd);
    else
        mbpSetNowait(oldmp_cd);
    return 0;
}

int monitorSetSwap( const int doByteSwap, const int doWordSwap ) {
    return 0;
}

int monitorFlushEvents() {
    if(mbpFlush(oldmp_cd)==0)
        return 0;
    else
        printError("monitorDeclareMp","Error while flushing events");
}

int monitorLogout() {
    return 0;
}

```

Makefile Source code:

MakefileStaticLibrary

```

DATE_CPPFLAGS := $(shell date-config --cflags)
DATE_LIBS      := $(shell date-config --libs --monitorlibs=dyn)

all: libmbp.a

libmbp.a: monitoringbyproxy.o monitoringbyproxy.h
    ar rcs libmbp.a monitoringbyproxy.o

monitoringbyproxy.o: monitoringbyproxy.c
    gcc $(DATE_CPPFLAGS) $(DATE_LIBS) -std=gnu99 -L/opt/date/mbp/linux/ -lzmq
    -c monitoringbyproxy.c -o monitoringbyproxy.o

clean:
    rm monitoringbyproxy.o libmbp.a

```

MakefileSharedLibrary

```
DATE_CPPFLAGS := $(shell date-config --cflags)
DATE_LIBS      := $(shell date-config --libs --monitorlibs=dyn)

all: libmbp.so

libmbp.so: monitoringbyproxy.o monitoringbyproxy.h
    gcc -shared -Wl,-export-dynamic -o libmbp.so monitoringbyproxy.o
    -L/opt/date/mbp/Linux/ -lzmq $(DATE_LIBS)

monitoringbyproxy.o: monitoringbyproxy.c
    gcc -g -std=gnu99 -fPIC $(DATE_CPPFLAGS) -o monitoringbyproxy.o -c
    monitoringbyproxy.c

clean:
    rm monitoringbyproxy.o libmbp.so
```

Appendix D

Client Program source code

CP.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <zmq.h>
#include "monitoringbyproxy.h"

int main()
{
    int c = mbpCreateContext();
    mbpConnection* cd = mbpCreateConnection("key1", "192.168.1.123");

    struct mbpPreference pd[] = {
        {"50", "PHY", "*", "*", "*"},
        {"must", "CAL", "*", "*", "*"}
    };

    int p = mbpSetPreferences(cd, pd, sizeof(pd)/sizeof(pd[0]));
    p = mbpConnect(cd, 20);

    int rc;
    struct eventStruct* pevent = calloc(sizeof(struct eventStruct), 1);

    if(p!=-1)
    {
        while(true)
        {
            rc = mbpGetData(cd, pevent);
            if(rc!=-1)
                printf("Event received: Run: #%d, Size: %lu, Type: %d\n",
                    pevent->eventHeader.eventRunNb, pevent->eventHeader.eventSize,
                    pevent->eventHeader.eventType);
        }
    }

    mbpDisconnect(cd);
    mbpDestroyConnection(cd);
    free(pevent);
    mbpDestroyContext();
    return 0;
}
```

Makefile Source code:

MakefileCP

```
DATE_CPPFLAGS := $(shell date-config --cflags)
DATE_LIBS      := $(shell date-config --libs --monitorlibs=dyn)

all: CP.c
    gcc $(DATE_CPPFLAGS) $(DATE_LIBS) -g -std=gnu99 -L/opt/date/mbp/Linux
        -lmbp -lzmq -o CP CP.c

clean:
    rm CP
```
