AGH University of Science and Technology
Faculty of Physics and Applied Computer Science

Master's thesis

Andrzej Dworak

major: **applied computer science**

specialisation: **computer science in science and technology**

Testing and improving the middleware infrastructure for the CERN accelerator controls

Supervisor: **dr hab. inż. Mariusz Przybycień**

Cracow, June 2009

Aware of criminal liability for making untrue statements I decleare that the following thesis was written personally by myself and that I did not use any sources but the ones mentioned in the dissertation itself.

Cracow, 16 June 2009

The subject of the master's thesis and the internship by Andrzej Dworak, student of 5th year major in applied computer science, specialisation in computer science in science and technology

The subject of the master's thesis: **Testing and improving the middleware infrastructure for the CERN accelerator controls**

Supervisor:	dr hab. inż. Mariusz Przybycień
Reviewer:	dr inż. Piotr Gronek
A place of the internship:	CERN, Geneva

Programme of the master's thesis and the internship

1. First discussion with the supervisor on realization of the thesis.
2. Collecting and studying the references relevant to the thesis topic.
3. The internship:
   - getting to know the idea,
   - participation in the section's technical meetings,
   - testing schemas investigation and development planning,
   - project development.
4. Continuation of the software development concerning the thesis.
5. Testing, documenting and maintenance of the created software, theirs discussion and approval by the supervisor.
6. Typesetting the thesis.

Dean's office delivery deadline: 16 June 2009

....

(head of department signature) (supervisor signature)

Evaluation of the thesis by the supervisor:

The final grade of the thesis as given by the supervisor: .....

Date: ..... Signature: .....

Evaluation of the thesis by the reviewer:

The final grade of the thesis as given by the reviewer: .....

Date: ..... Signature: .....

Grading: (6.0 – excellent; used only in exceptional cases when the work surpasses distinctly the regular level), 5.0 – very good, 4.5 – more than good, 4.0 – good, 3.5 – more than satisfactory, 3.0 – satisfactory, 2.0 – fail

Abstract

This document presents concepts and implementation of the middleware infrastructure for the CERN accelerator controls redeveloped and improved for the needs of the Large Hadron Collider operation. The first part depicts the CERN organization. The second one takes a look at the fundamentals of control systems as they are implemented in nowadays CERN facilities. Next chapters describe work of the author with CERN Control Middleware system (CMW), which at the time of author's stay at CERN, underwent final phase of its development. Almost all main functionality had already existed. What was to be done was to implement a testing platform that would guarantee its reliability, stability and infallibility during the operation of the LHC. This included general testing of the libraries and development of scalability tests that would show the limits of the software, also with comparison to other, already existing solutions. These topics are described in chapter three. Chapter four shows the details on CMW-RecorderPlayer, an application developed by the author for more complex testing scenarios and diagnosis of running software. The last part focuses on the details of implementation and functionality of the CMW-Proxy, another application developed by the author of the thesis. CMW-Proxy is mainly being used to take off load from overloaded front-end servers and to decrease data transfer bandwidth between the servers and their clients. The proxy can also be used as a gateway from general network into the realm of the Technical Network by any authorized client seeking a transparent way to communicate with front-end computers.

Acknowledgements

Since the very beginning of my young-professional life it has been my good fortune to encounter well-disposed people. Their friendship, companionship, professional and personal help as well as the time they have spent to broaden my knowledge and competencies, is of the greatest value to me. Without them this work would have never been possible. But what is more, and for what I want to thank most – is their passion and in-depth knowledge of the subjects they work on professionally, and likewise number of past-time activities they pursue. This keeps on reminding me how important it is to draw happiness from one's work and life in general.

In the perspective of this work I would like to thank in particular my parents Alicja Dworak and Dominik Dworak who though did not subscribe me to singing lessons when I was young, always took care about my education and had time to help me with their own knowledge; prof. dr hab. Danuta Kisielewska, dr hab. inż. Mariusz Przybycień and dr inż. Krzysztof Malarz, science teachers from my home university for introducing me to the world of particle physics and CERN, and they constant help with all the official formalities during my absence; Pierre Charrue and Maciej Sobczak for supervising me during my stay at CERN as CERN Technical Student; Maciej Sobczak and Joel Lauener for their wide technical help and knowledge that supported me on every day basis; Czesław Fluder for introduction to the accelerator control systems; one more time to dr hab. inż. Mariusz Przybycień for supervising this thesis, dr inż. Piotr Gronek for reviewing it, Maciej Sobczak for consultancy, William Spearman and Maciej Woś for correcting my English style.

Thank You!

Contents

1 Introduction **9**
- 1.1 CERN, The European Organization for Nuclear Research . . . 9
- 1.2 The accelerator complex . . . 9
- 1.3 Experiments . . . 10

2 Control System **11**
- 2.1 Introduction . . . 11
- 2.2 Architecture . . . 11
- 2.3 Equipment access . . . 13
  - 2.3.1 The PLCs . . . 13
  - 2.3.2 The Fieldbuses . . . 13
- 2.4 Servers and operator consoles . . . 14
- 2.5 Machine timing and UTC . . . 14
  - 2.5.1 Central Beam and Cycle Management . . . 14
  - 2.5.2 Timing generation, transmission and reception . . . 15
  - 2.5.3 UTC generation, transmission and reception . . . 15
- 2.6 Data management . . . 16
- 2.7 Controls Middleware – the communication software framework . . . 16
  - 2.7.1 Device access model – RDA . . . 17
  - 2.7.2 Messaging model – JMS . . . 18
- 2.8 FEC Software Framework . . . 19
- 2.9 Java software – J2EE, Spring, JAPC and LSA . . . 20
- 2.10 UNICOS . . . 22

3 Remote Device Access **23**
- 3.1 Introduction . . . 23
- 3.2 Design choices . . . 23
  - 3.2.1 C++ generic data container . . . 24
  - 3.2.2 Java generic data container . . . 25
- 3.3 Architecture . . . 27
- 3.4 Example use . . . 28
  - 3.4.1 Java client side . . . 29
  - 3.4.2 C++ server side . . . 30
- 3.5 Testing and improving . . . 34
  - 3.5.1 Correctness testing and a unit-testing framework of choice . . . 34
  - 3.5.2 Tests performed and their results . . . 38
  - 3.5.3 Reliability tests . . . 40
  - 3.5.4 Performance testing environment . . . 40
  - 3.5.5 Response time . . . 41
  - 3.5.6 Scalability and performance . . . 43
  - 3.5.7 Summary . . . 46

4 CMW-RecorderPlayer **47**
- 4.1 Introduction . . . 47
- 4.2 Recorder . . . 47
- 4.3 Player . . . 48
- 4.4 Testing . . . 51

5 CMW-Proxy **53**
5.1 Introduction 53
5.2 Architecture 53
5.3 Optimization and execution issues 56
5.4 Utilities 58
5.5 Security policy 60
5.6 Testing 62
5.7 Usage 63

6 Conclusions **64**

1 Introduction

1.1 CERN, The European Organization for Nuclear Research

The European Organization for Nuclear Research, known as CERN, is the world's largest particle physics laboratory. Established in 1954 in the northwest suburbs of Geneva, spans its territory over Franco-Swiss border between Jura Mountains and Lake Geneva. The organization consists of twenty European member states. An additional eight international organizations or countries have "observer status". Apart from that, there are other forms of collaboration between CERN and other countries resulting in approximately 12,000 people from 80 different nationalities working together. This makes CERN a real multi-cultural melting pot adequate to the nearby international city of Geneva [1].

Soon after its establishment, CERN's main research objectives went beyond the study of the atomic nucleus. CERN, studying interactions between particles, have become European laboratory for particle physics, achieving many spectacular successes in the field. Amongst them the discovery of W and Z bosons (1983), the first creation of antihydrogen atoms (1995) and the discovery of the direct CP-violation[1] (1999) [2].

Apart from its achievements in particle physics CERN has greatly donated to the other fields of research: nanotechnology, vacuum technology, cryogenics and electronics, not to forget about computer science and networking with the World Wide Web, the GRID and speed records in long-distance data transfer [3].

At the moment, the main activity at CERN concerns the Large Hadron Collider (LHC) [4] and six accompanying experiments - ATLAS, CMS, Alice, LHCb, LHCf and TOTEM [5].

1.2 The accelerator complex

The main CERN's accelerator complex consists of the Antiproton Decelerator and 7 particle accelerators that can reach increasingly higher energies and is shown in Figure 1. The biggest of them is the LHC measuring 27 km in circumference. Its tunnel located 100 metres underground was previously occupied by the Large Electron-Positron Collider (LEP) [6], which was dismantled in November 2000. For the upcoming experiments both protons and lead ions are going to be accelerated in LHC. Protons obtained by removing electrons from hydrogen atoms will start their life cycle in the linear accelerator (LINAC2) from which they will be injected into the PS Booster, then to the Proton Synchrotron (PS), followed by the Super Proton Synchrotron (SPS), before finally reaching the Large Hadron Collider (LHC). In the LHC two beams circulating around the ring in opposite directions will collide in the so called interaction points at maximum energy of 14 TeV in the centre of mass.

Lead ions will start from a source of vaporized lead and enter LINAC3 before being collected and accelerated in the Low Energy Ion Ring (LEIR). Then they will follow exactly the same way as protons, reaching a final collision energy of 1150 TeV.

The whole complex of 8 accelerators, the cryogenic distribution system and other technical infrastructure is controlled from the recently established CERN Control Centre (CCC). The LHC has already been turned on, and first beams went around the whole ring on 10 September 2008. The beam calibration process went smoothly; both hardware and software were working properly. Unfortunately, due to a faulty connection between superconducting magnets the

[1] In particle physics, CP-violation is a violation of the CP symmetry. CP symmetry states that the laws of physics should be the same if a particle was interchanged with its antiparticle (C stands for charge), and left and right were swapped (P stands for parity) as in a mirror reflection. Various experiments have proved that this symmetry is violated during certain types of weak decay.

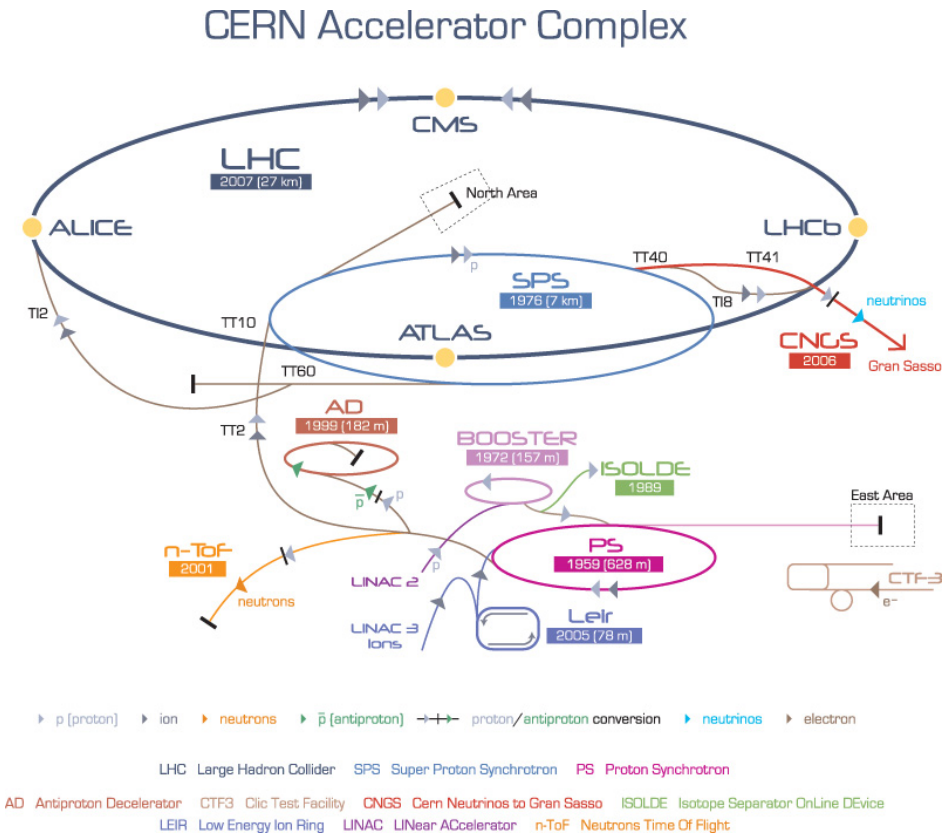


Figure 1: CERN Accelerator Complex

system was taken down for repairs on 19 September 2008. The restart of LHC is expected in October of 2009.

1.3 Experiments

All six LHC experiments have been developed and will be operated by international collaborations. Each experiment is distinct, characterized by its unique particle detector.

The two largest and most known are ATLAS (A Toroidal LHC Apparatus) [7] and CMS (Compact Muon Solenoid) [8]. They were designed to investigate a large range of physics topics - confirmation of the Standard Model (SM), investigation on the Higgs boson, CP violation (asymmetry between the behaviour of matter and antimatter), properties of top quarks and new physics beyond the SM, such as supersymmetry[2] and extra dimensions. Independent development and different design of the detectors is vital for cross-confirmation of any new discoveries.

The smaller experiment, ALICE (A Large Ion Collider Experiment) [9], will study heavy ion collisions (Pb-Pb), which are expected to generate a new state of matter called quark-gluon plasma, where quarks and gluons are deconfined. LHCb (Large Hadron Collider beauty[3]) [10] will focus on CP violation in the interactions of heavy particles containing a bottom quark.

The smallest, TOTEM [11] and LHCf [12], which are housed by the caverns of CMS and ATLAS respectively, are designed to focus on "forward particles", i.e. those that just brush past each other as the beams collide rather than meeting head-on.

[2]Supersymmetry (SuSy) is a symmetry that relates elementary particles of one spin to another particle that differs by half a unit of spin and are known as superpartners. In other words, for every type of boson there exists a corresponding type of fermion, and vice-versa. The theory has not been yet validated by any experiment.

[3]Where "beauty" refers to the bottom quark

2 Control System

The CERN accelerator control system is described in detail in [13]. Only the main features are presented here. This serves later on as an explanation of the software concepts and models used to implement the main controls communication library, namely Controls Middleware (CMW) [14]. Thus, emphasis has been put on presenting the impact of the control environment on the CMW and the communication model it delivers.

2.1 Introduction

The control systems at CERN have greatly evolved over time. Until 1993, industrial equipment was connected to the control network via CERN specific front ends. Since then, the experiments and corresponding complexity of apparatus have grown immensely, and so did the demands towards the control systems. To meet anticipations and to guarantee safe operation of the machine under expected high energies, the whole system was redesigned and reimplemented. Moreover, a new CERN control room – CERN Control Centre (CCC) – was established to bring together control and operation of the site services, cryogenics, injector chain and the new collider [15].

2.2 Architecture

The LHC control system architecture is mainly based on standard components employed worldwide for the control of accelerators as well as components used for the PS, SPS and LEP. Comparing current control systems with the ones used at CERN earlier, one can see two major differences:

- use of object oriented technologies in software systems,
- use of industrial controls solutions for the supervision of complete subsystems of the LHC.

As shown in Figure 2, the LHC control system consists of three hierarchical layers of equipment communicating through the CERN Technical Network, a flat Gigabit Ethernet network using the TCP/IP protocol. Starting from the bottom of the figure:

- At the equipment level, the various actuators, sensors, and measuring devices are interfaced to the control system through three different types of front-end computers:
  - VME computers running LynxOS[4], dealing with high performance acquisitions and real-time processing,
  - PC based gateways interfacing systems where a large quantity of identical equipment is controlled through fieldbuses,
  - Programmable Logic Controllers (PLCs)[5] driving various industrial actuators and sensors.

[4]The LynxOS [16] is a Unix-like real-time operating system. LynxOS features full POSIX (Portable Operating System Interface) [17] conformance and Linux compatibility. LynxOS is mostly used in real-time embedded systems, in applications for avionics, aerospace, the military, industrial process control and telecommunications.

[5]A programmable logic controller (PLC) is a digital computer used for automation of electromechanical processes, such as control of machinery on factory assembly lines. Difference between a PLC and a general-purpose computer is that PLC is designed for multiple inputs and output arrangements, immunity to electrical noise, extended temperature ranges, and resistance to vibration and impact. These let them operate in the vicinity of the LHC machine.

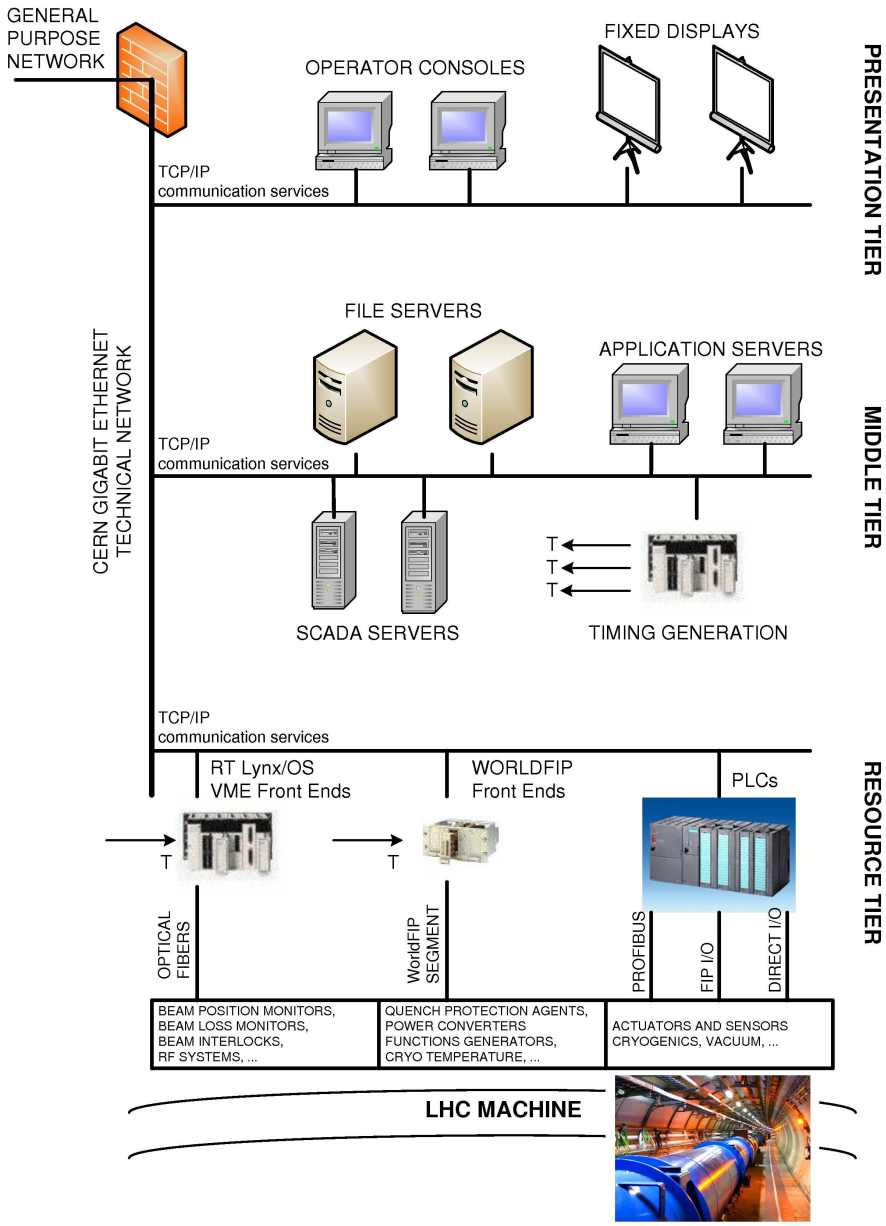


Figure 2: Control System Architecture

As TCP/IP protocol does not meet Quality of Service (QoS) expectations on the level required by communication between Front End and PLC computers, and measurement devices and other tunnel equipment, it is therefore not used for this part of communication. What is used instead is Fieldbus – a family of industrial computer network protocols used for real-time distributed control. Amongst them, the two major ones are Profibus [18] and WorldFIP [19]. They allow longer distance transmission (without repeaters) and are more robust than Ethernet protocols. In addition, they are the only means to connect equipment located in the LHC tunnel or other radioactive areas [20].

- In the middle tier, powerful UNIX servers host the operational files and run the LHC applications. Hardware reliability and availability is ensured by selecting multi-CPU architectures with redundant and hot swappable power supplies, discs and fans. Mirroring and RAID[6] techniques are used to ensure data integrity and error recovery. Main servers

[6]RAID, Redundant Array of Independent Disks is a technology that employs the simultaneous use of two or more hard disk drives to achieve greater levels of performance, reliability, and/or larger data volume sizes [21].

of the middle tier are:

  - **application servers** hosting the software required to operate the LHC beams and running the Supervisory Control and Data Acquisition (SCADA)[7] systems,
  - **data servers** containing the LHC layout and the control configuration as well as all the machine settings needed to operate the machine or to diagnose its behaviour,
  - **central timing** which provides the cycling information of the whole complex of machines involved in the production of the LHC beam and the timestamp reference.

- At the control room level, consoles running the Graphical User Interfaces (GUI) allow machine operators to control and optimise the LHC beams and to supervise the state of the whole system. Dedicated fixed displays also provide real-time summaries of key machine parameters.

2.3 Equipment access

2.3.1 The PLCs

Non-industrial hardware in the PS, SPS and LHC sites is connected to the control system via VME or PC based Front End Computers (FEC). These systems mainly run a LynxOS real-time operating system. Several hundred FECs are distributed in the surface buildings of the LHC and some in the underground areas. Where possible they are diskless in order to increase reliability and connected to the remote reboot and terminal server systems, thereby allowing remote management.

FECs run hardware access software, which is part of the resource tier (see Fig. 2) and provides access to equipment data. The data may be accessed by any authorized software agent in the control system via a subscription or a single demand/response call. A dedicated FEC software framework has been developed in order to simplify and standardize the software running in the FECs (see Sec. 2.8).

PLCs are chosen when the process is not synchronized to accelerator timing and when the sampling period required is not smaller than 100 msec. They are cost efficient, highly reliable, adaptable to industrial environment and offer ease of programming via high-level languages. Additionally, they benefit from generic facilities such as remote reset, monitoring and optionally clock synchronization. As market standards, only PLCs from Schneider and Siemens are used for LHC.

2.3.2 The Fieldbuses

The two fieldbuses used in the LHC accelerator, namely WorldFIP and Profibus, are part of the three fieldbuses recommended at CERN. They are both supported by an internal CERN service. These fieldbuses allow longer distance and more robust protocols than Ethernet. In addition they are the only means to connect equipment located in the LHC tunnel or other radioactive areas.

The WorldFIP Fieldbus [19] is used when the following features are required:

- Determinism (up to 10μs) required for:
  - real-time control and synchronization of LHC equipment,

[7] SCADA is an industrial control system. Usually it is a computer system which is monitoring and controlling various processes [22].

  - high precision time distribution,
  - management of periodic data.

- Robustness in severe environments and in particular:
  - resistance to electromagnetic noise,
  - good resistance to high radiation levels.

- Data rates:
  - WorldFIP is used to control largely distributed LHC systems at high (in the given environment) data rates of 1 MBits/s and 2.5 MBits/s,
  - high load factor possible (70 to 80% of the network bandwidth).

The Profibus Fieldbus [18] has been selected for several applications in the LHC, such as cooling and ventilation, vacuum, cryogenics, fast kicker magnets, and magnet and power interlock controllers. The main reasons to select the Profibus are:

- robustness of the protocol and simplicity of configuration,
- large variety of remote I/O systems available with Profibus,
- availability of radiation-tolerant remote I/O,
- ease of integration with Siemens PLCs,
- capacity to be used as an instrumentation bus with a wide range of instrumentation products offering Profibus as standard means of connection.

2.4 Servers and operator consoles

The presentation tier of the LHC control system (see Fig. 2) is deployed on operation consoles and fixed displays. The LHC middle tier software runs on the servers and hosts operational programs and data files, furthermore it offers specific services (web services for operation, fixed displays, SCADA servers, Database servers, etc.). The servers run the Linux operating system. Emphasis has been put on the hardware reliability and availability issues by selecting multi-CPU architectures with redundant and hot swappable power supplies, discs and fans. To ensure data integrity and error recovery mirroring, RAID techniques are used. The fixed displays and operator consoles are both based on desktop PCs with a large amount of memory. They run GUI applications from their local disks and use one to three screens to display the data. More than 20 fixed displays and around 10 consoles are needed in the CCC to operate the LHC machine. Around 50 servers located nearby and in the LHC surface buildings are used to deploy and run the operational LHC software.

2.5 Machine timing and UTC

2.5.1 Central Beam and Cycle Management

The production of the LHC beam involves a long chain of injectors which need to be tightly synchronized. Moreover, the various types of beams to be produced for the LHC are only a subset of all beams that the injectors must be capable of producing, e.g. beams for fixed

target physics, for ISOLDE (On-Line Isotope Mass Separator) [23] and AD (Antiproton Decelerator) [24]. These beams are produced in sequences of typically a few tens of seconds. The composition of these sequences changes many times a day and the transition between different sequences must be seamless.

To manage the settings of the machines and to synchronize precisely the beam transfer from one injector to another until the beam is injected into the LHC, a Central Beam and Cycle Manager (CBCM) was developed. The CBCM:

- delivers General Machine Timing (GMT),
- provides Beam Synchronous Timings (BST),
- elaborates the "telegrams" specific to each machine and broadcasts over the GMT networks. A "telegram" describes the type of beam that has to be produced and provides detailed information for sequencing real-time tasks running in the FECs and for setting up equipment.

In total the CBCM drives seven separate GMT networks dedicated to the different CERN accelerators. To achieve absolute reliability, a CBCM is running in parallel on two different machines. A selector module capable of seamlessly switching between them guarantees continuous operation of the system.

2.5.2 Timing generation, transmission and reception

Several hardware timing modules have been built either to produce reference clocks and timing events or to extract the timing pulses or the interrupts needed to drive the equipment from the distributed timing data.

- The CTSYNC module provides the "master" clocks from a 10MHz oscillator which has a stability greater than 10^{-10}s.
- The CTGU module encodes events to be transmitted over the GMT network: the millisecond events, machine timing events, telegrams, and the Universal Coordinated Time (UTC) events.
- The CTGB is the driver module for the BST network; Also, the CTGB sends the 1 Pulse per Second (1PPS) UTC time and some telegram events over these networks.
- The CTRP is a GMT multi-channel reception module. It can generate timing pulses with very low jitter. The CTRP recognizes the various types of timing events: UTC time, the telegrams, the millisecond and other machine events; furthermore it can use them to trigger pre-programmed actions.

2.5.3 UTC generation, transmission and reception

A Global Positioning System (GPS) time receiver connected to the CTGU serves as the source of date and time for all CERN accelerators. The GPS provides UTC referenced date and time plus a very accurate 1 PPS tick, which is used by the CTGUs to synchronize with the UTC time. At initialization the CTGUs receive the date and time information from the GPS receiver and then run independently, using the clock delivered by the CTSYNC until a manual resynchronization is made. CTGUs provide the time every second in UNIX standard format. This format allows easy conversion to year, month, day, hour, minute and second. UTC time

Table 1: LHC time stamping accuracy

Beam dump Beam instrumentation Radio frequency Injection (kickers) General machine timing	< 0.05 ms
Machine interlocks Quench protection Power converters	< 1 ms
Feedbacks (orbit, tune ...) Cryogenics Vacuum All other systems	1 – 10 ms

can be provided by the CTRP modules to timestamp the data to be logged with a 25 ns resolution. Using an optional CERN-developed chip the resolution can be brought down to 1 ns. The required time-stamping granularity for the different LHC systems is shown in Table 1. In principal, the 1 ns performance is required only in the injector chain.

2.6 Data management

The LHC machine is of unequaled complexity in terms of number of components and possibilities of their operational parameters settings. To name a few, the cryogenics, machine protection and beam monitoring systems alone, give the control room access to more than 100,000 signals. Similarly, the powering layout for the LHC, including 1232 main dipole magnets, about 450 quadrupole magnets and a few thousand corrector magnets, powered in more than 1600 electrical circuits is of enormous complexity. As operation of the machine relies on the precise setting of all those parameters, it is crucial to investigate, understand and control them, so as to make the LHC work optimally. To be able to do that, all the settings are stored in a database. Apart from that, the database system contains information about the whole control system, which is composed of a large number of application programs, a middleware layer, VME- and PC-based FECs with control software, industrial PLCs, fieldbuses, and hardware interface modules. Shared characteristics of these components enable all control room software to be data-driven and more generic, greatly reducing the software communication layer. Moreover, this also allows reuse of the database system to hold the control configuration of the injection accelerator chain of the PS and SPS complexes. The database system also provides the possibility of automatic configuration and bootstraping of all FECs: storing their configuration data and their physical addresses. The whole database system at CERN runs on Oracle systems.

2.7 Controls Middleware – the communication software framework

The Controls Middleware (CMW) [14] is a software framework that provides a common software communication infrastructure for the CERN accelerator controls. CMW provides its users with a consistent and easy to use Application Programming Interfaces (API) that allows seamless communication between the software entities of the control system. This API has full implementation in both C++ and Java programming languages and is called RDA (Remote Device Access) (see Fig. 4).

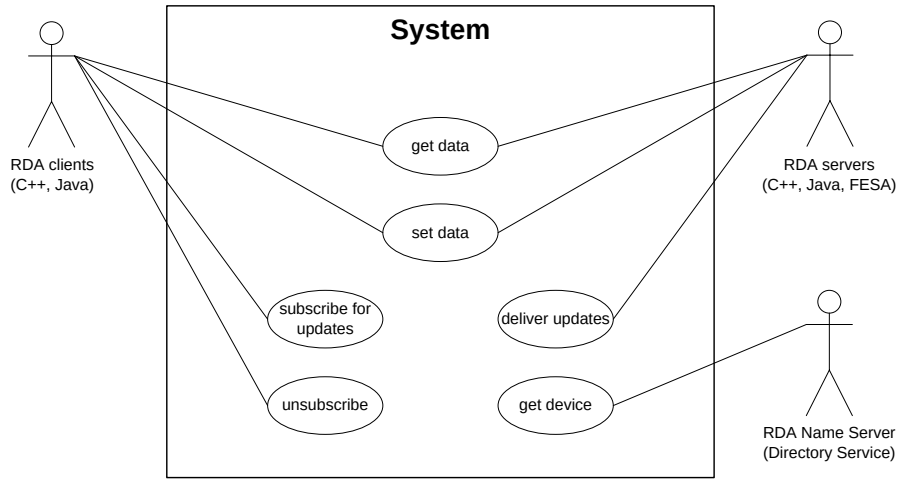


Figure 3: CMW use case.

In CMW two conceptual models are supported: the device access model and the messaging model [25]. The device access model is mainly used in the communication between the resource and middle tiers while the messaging model is mainly used within the business tier or between the business tier and applications running in the presentation tier.

2.7.1 Device access model – RDA

Within the device access model, all accelerator equipment can be accessed as a device. A device is a named entity of the control system, which usually corresponds to a physical device or equipment (e.g. beam position monitor, power converter) or to a virtual controls entity (e.g. transfer line). Each device has a number of properties that describe its state and through which it can be controlled. By getting the value of a property, the state of the device can be read. Accordingly, a device can be controlled by setting one of its properties with the required value. `Get` and `set` operations can be either synchronous or asynchronous. In addition to the `get` and `set` operations on device properties, it is possible to monitor changes of the property via listeners. Listeners are implemented as callbacks and set with `monitorOn` and `monitorOff` functions. Apart from sending commands to a device and receiving its state it is often necessary to specify under which conditions the operations had to take place, for instance to which cycle of the machine the operation applies. These conditions are commonly referred to as filters and as implementation code they are called `context`. The basic system use case is shown in Figure 3.

As communication abstraction, the device access model is usually used between CMW equipment servers running in FECs and Java applications running in the middle tier (see Fig. 2). These processes communicate together through the RDA library. Using accompanying services and CORBA[8] as its backbone, RDA implements this model in a distributed environment with devices hosted by servers that can run anywhere in the controls network [27]. From client's point of view, RDA provides reliable and location-independent access to the devices. This functionality is provided by an RDA name server called Directory Service.

CMW's device access model is shown in details in Figure 4. Beside the RDA's software layers

[8]CORBA (Common Object Request Broker Architecture), is Object Management Group's (OMG) open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, a CORBA-based program can interoperate with any other CORBA-based program, even if they were written in different languages and run on machines with different architectures and operating systems [26].

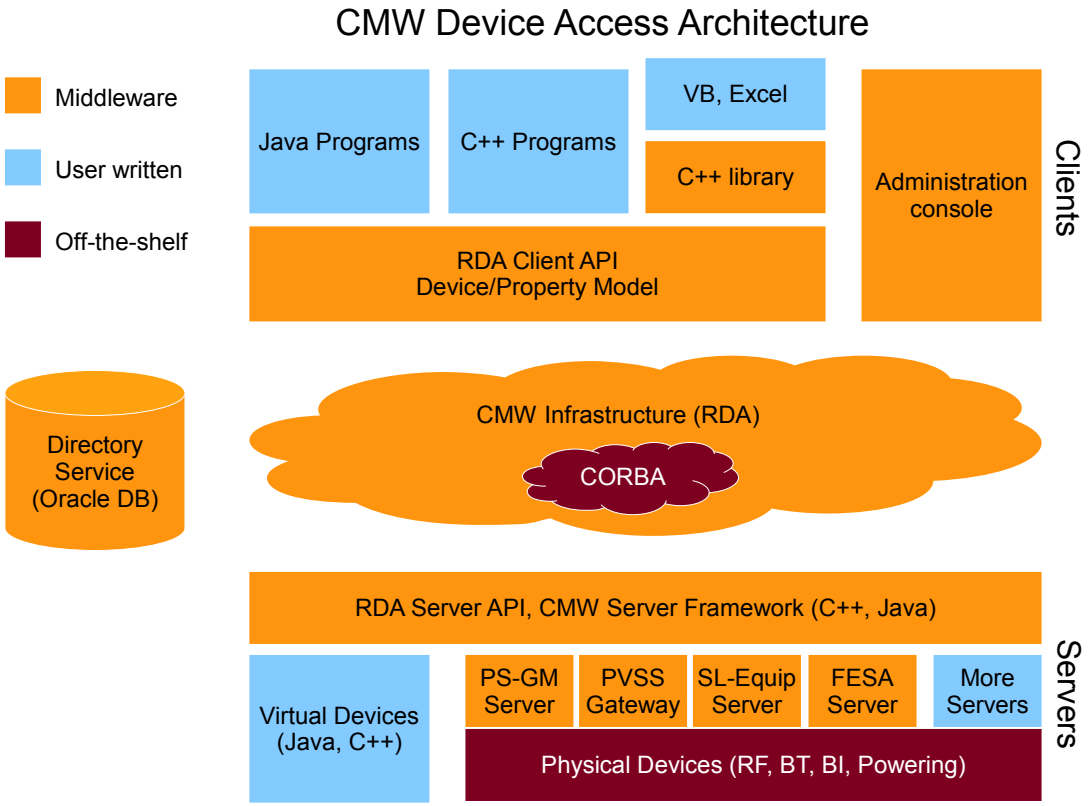


Figure 4: CMW Device Access Architecture

used on client's and server's side, one can also see more parts of the middleware infrastructure. One of them is the administration console, which officially is called CMW Admin. CMW Admin is a client side application that monitors states and conditions of all available CMW servers. Next, a few servers, the PS-GM and SL-Equip servers, give access to old but still used software operating on the equipment running in the PS and SPS tunnels [15]; PVSS gateway gives access to PVSS (Prozessvisualisierungs- und Steuerungs-System, an object-oriented process visualization and control system)), which is a SCADA system; FESA servers are all new servers developed with FESA application (see Sec. 2.8). Moreover there is the Directory Service, which maps logical names (device and server names) to their real locations (IP address and port number) and later to physical devices and pieces of hardware used in controlling and monitoring the accelerators.

2.7.2 Messaging model – JMS

Complementary to the device access model, the messaging model allows to send and receive asynchronous messages. This approach to communication, though simpler in concept, is also very important in the accelerator control system. It is a natural solution when it comes to exchange of timing and logging events, notifications and alarms. These events, characterized by asynchronous generation from a number of producers that potentially have multiple consumers.

To meet these demands Java Message Service (JMS)-compliant messaging system has been incorporated into the CMW system. JMS provides a common API, which with the addition of a provider framework, enables development of portable, message based applications in the Java environment [28]. Additionally, JMS is an integral part of the Java 2, Enterprise Edition (J2EE)[9] platform, which allows easy usage of all its components.

[9]Java 2, Enterprise Edition (J2EE), is build on the foundation of Java Standard Edition. Being the industry

2.8 FEC Software Framework

The Front-End Software Architecture (FESA) is the new CERN accelerator real-time software architecture, a complete environment providing access to the accelerators' physical elements. The project started in 2003 with its primary objective being to develop an infrastructure that would speed-up, simplify and standardize development of front-end software.

In order to simplify development of a server for a new FEC, needs and characteristics of each type of equipment were studied. As a result, a clear separation between the generic and the equipment specific parts was found and incorporated into FESA, letting it to become rather a whole structure-and-flow control framework than just a library implementing common functionalities. Thus, the framework organizes the server activity, and when needed calls the functions provided by the application developer to provide application-specific behaviour. Such approach not only cuts down to minimum development time but also enhances maintainability of the code.

To build the control software, the programmer is provided with a set of configuration tools:

- A graphical tool: developing a FESA class requires three XML[10] documents: Design, Deployment and Instantiation. As part of FESA framework there exists a graphical application, which is basically a generic XML editor, configured by dedicated W3C XML Schemas[11] To put it more clearly, the application itself is a general XML file editor, but its present look (graphical elements such as buttons, text input fields, etc.) and behaviour depends on an XML Schema that is loaded at the time of use. Following schemas have been defined for FESA:
  - FESA Design Schema drives the design tool and encodes the model of all possible models of equipment software. It allows equipment specialists to think of an equipment design as of:
    - a public Interface, a collection of get/set/subscribe properties. Equipment is controlled through remote invocation of these properties using the CMW;
    - a set of fields (data-holders) composing software abstraction of an underlying hardware device;
    - a set of server actions, implementing the properties get/set services;
    - a set of real-time actions transferring data between the hardware device and its software representation;
    - a set of events which trigger the equipment's action.
  - FESA Deployment Schema: conducts the deployment tool used to deploy the FESA class on a particular FEC.
  - FESA Instantiation Schema: drives the instantiation tool that configures and creates device instances of deployed FESA class on a FEC where it was previously deployed.

standard it is widely used for development of service-oriented architecture (SOA) applications, as well as next-generation web applications [29].

[10]Extensible Markup Language (XML) is a simple flexible text format. Originally designed to meet the challenges of large-scale electronic publishing, XML is getting more and more popular as a medium to store data or to exchange it on the Web and elsewhere [30].

[11]The W3C definition says: "XML Schemas express shared vocabularies and allow machines to carry out rules made by people. They provide a means for defining the structure, content and semantics of XML documents" [31].

- A code generation tool which using XML file with the previously specified equipment design allows automatic generation of makefiles and source code with C++ implementation of the equipment. This is done using another XML-based technology, namely XSL[12].

- Test environment: using XSL, FESA can automatically create a Java GUI application that gives access to each property of any defined device instance [33].

Despite the huge diversity of devices, such as Beam-Loss monitors, Cryogenic systems, Kickers and many others, FESA has successfully fulfilled all the aims, becoming the new development standard for the LHC and the whole injector chain. The framework allowing equipment specialists to design, develop, test and deploy real-time control software for FECs in a fast and easy way, speeds up the whole process of software development and its maintenance. On top of these all, using the high level language to describe the equipment software makes it easy to understand its structure and behaviour, just by examining the design document. Being a generic, well tested tool, FESA has found many clients outside CERN. Amongst the others, it is going to be used at the Fermi National Accelerator Laboratory (FERMILAB) and the Gesellschaft fr Schwerionenforschung research centre (GSI, Association for Heavy Ion Research), despite small hardware and equipment discrepancies [34].

2.9 Java software – J2EE, Spring, JAPC and LSA

Apart from CMW, depending on the need and ease of utilization, there are other communication systems or extra software layers used. The second concept – ease of utilization and extra software layers – mainly concerns Java applications and goes perfectly with its "yet-another-software-layer" language philosophy. Top-level applications controlling LHC, written mainly in Java must handle a huge variety of tasks such as equipment and database access, data visualization and quite often, significant computation. Considering the complexity of those undertakings Java philosophy does not seem to be superfluous anymore.

To perform the tasks, applications need some common services: security, transactions (atomicity of complex operations), remote access and remote resource management. In this expansive and confusing environment, it is very important to keep the application's code short and clear, and to reuse good existing solutions as often as possible. At CERN, at the beginning of the development process of new control applications for the LHC machine, majority of them chose J2EE as a common platform providing all the needed functionalities. As J2EE's container, Oracle's OC4J implementation was used [35]. But, because of J2EE's complexity and cumbersomeness it was mostly replaced with the Spring[13] framework [37].

Being an established industrial standard, based on the Java programming language and using all its functionality, Spring not only satisfies the previously mentioned requirements but also provides some extra advantages. Firstly, it provides implementation of needed services in the form of independent, strictly defined modules. Secondly, it offers a clear separation between the graphical-presentation layer and application logic layer. For applications developed at CERN such separation, between the implementation of graphic user interface (GUI), the control part of an application (the core business of the application) and the abstract model of the accelerator control on which it operates, leads to a simple and very popular three-tier

[12]XSL (The Extensible Stylesheet Language) is a family of languages defining XML document transformation and presentation, thus letting to transform one XML file into another XML document [32]. This in the end makes it possible to generate for instance C++ source file from an XML file.

[13]Spring is a leading platform to build and run enterprise applications. Features of its framework can be used by any Java application, but its extensions can also be used for building web applications on top of the J2EE platform [36].

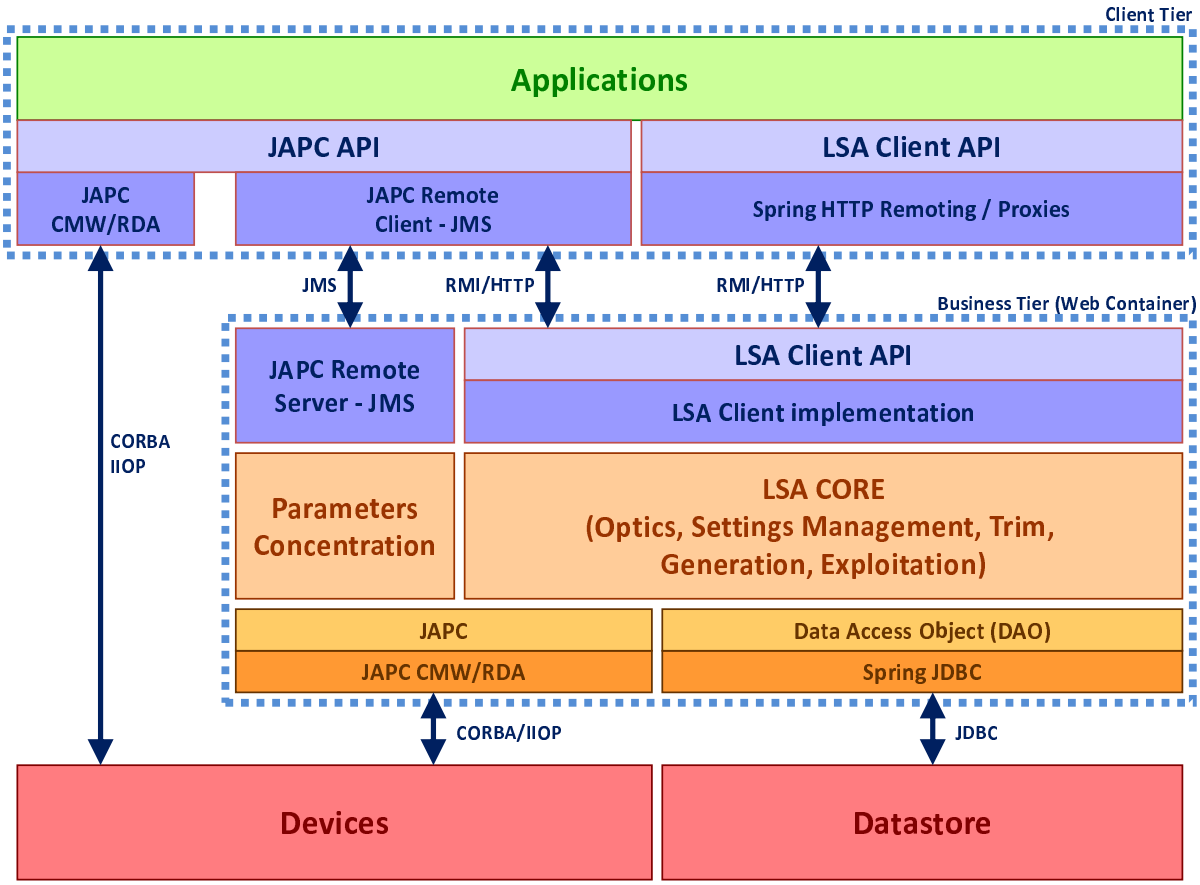


Figure 5: Typical usage of JAPC and LSA libraries.

architecture, with independent, clearly separated, well defined interfaces and tasks for each of the layers.

Coming back to what was said in the first paragraph about additional communication frameworks and layers, one of them is called Java API for Parameter Control (JAPC). JAPC is a framework to build Java applications that control accelerator devices. Its main concept is a parameter which represents a controlled entity, such as physical equipment, databases, timing events and others. Client programs can modify and check status of JAPC parameters with `set` and `get` functions, and they can subscribe for an automatic notification of its value changes. The concept is quite similar to the CMW device access model (see Sec. 2.7.1). In fact, JAPC not only serves as a wrapper for Java applications, providing a straightforward and uniform interface to equipment communicating using CMW but also to other entities using different means of communication. JAPC saves developers from the tediousness of writing specific applications for all the different kinds of devices, accessed through a large number of different protocols [38]. As a communication wrapper, JAPC usage can be seen in Figure 5.

Another important project is the LHC Software Architecture (LSA). The system covers all of the most important aspects of accelerator controls. Its main tasks are to store and give access to layout of machine optics and other control equipment and make it easier to set and read machine parameters. LSA organizes the parameters in tree structures. Their roots are usually physics-oriented, high-level abstraction parameters (e.g. momentum of the particles), while leaves correspond to hardware settings (such as value of currents creating the magnetic field). A change of the root element instantly propagates down to the lower levels in the hierarchy between parameters. The changes are specified by appropriate algorithms or equations. Operators usually control the machine by modifying the root parameters and letting the LSA system recalculate values of the dependent ones.

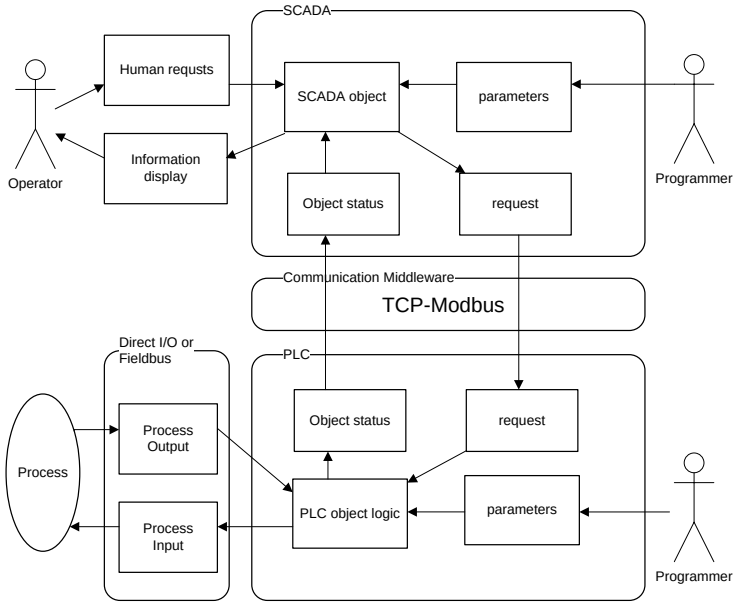


Figure 6: The UNICOS object model.

Since 2005, providing a clean and clear API, LSA has been a powerful tool used by all the operational applications in the LHC, SPS and LEIR. The system is entirely written in Java and uses the Spring framework; it provides services like transactions or remote access (JAPC or RMI[14]). Through a JAPC interface, it also gives an independent access to its database containing layout and parameters settings. LSA architecture and usage can be seen at Figure 5.

2.10 UNICOS

UNICOS (UNified Industrial COntrol System) is a hardware and software framework which provides components, methodology and tools to design, build and program industrial control systems for the LHC [40]. At CERN it is used mainly with the cryogenic equipment and experimental magnets in the LHC complex. In the supervision layer UNICOS uses popular industrial controls architecture, which is the PVSS SCADA[15]. Schneider PLCs are used for process control at the control layer. Communication is based on Ethernet and FIPIO (Factory Instrumentation Protocol Input Output).

From programmer's point of view, development philosophy inherits partly from object-oriented approach. That is, each process device is represented as an object, which integrates both its behaviour and the GUI. Though in one object, the two parts are separated. The GUI part is programmed in the SCADA. It includes interaction with the operator through graphical components which shows the current status of the object, and dedicated panels which allow sending orders. The PLC contains the process behaviour of the object, and orchestrates it either by direct I/O connection or by a fieldbus. The SCADA and PLC object parts are connected together by a communication layer based on the TCP-Modbus protocol. For simplified architecture of the UNICOS object model see Figure 6.

[14]The Java Remote Method Invocation (RMI) is one of the standards of remote communication for Java language. Using it an object running in one Java virtual machine can easily invoke methods of an object running in another Java virtual machine [39].

[15]SCADA (Supervisory Control And Data Acquisition) is a system supervising course of a technological or manufacturing process. Main responsibilities of a SCADA system are collecting data, on-line measurements visualization, controlling the ongoing process and alarming in case of a problem.

3 Remote Device Access

3.1 Introduction

The LHC machine consists of a number of important parts. Not less important is the steering equipment conducting the way the machine works and monitoring its states. As explained in the previous chapter the equipment consists of physical parts such as cable infrastructure, actuators, sensors, various measurement devices and computer hardware. Besides, there is also software operating on them:

- Programs specifying the actions of the underlying hardware.
- Protocols and communication layers responsible for transferring commands over the network.
- Programs that take care of on-line results filtering, collecting and presenting, as well as transferring and storing of the results into various database systems.
- Finally, applications used for off-line analysis of data, recorded during a run, about the machine behaviour. Such analysis helps to calibrate the machine's settings and improve its behaviour in the following runs.

The complexity and variety of necessary services led to an inclusion of many new software concepts and technologies. Some of them were already well established industrial standards taken as they were or slightly modified according to the needs. Others were developed from scratch. Amongst them was CMW.

In 1999 a proposition was raised to create a common software communication infrastructure for the CERN accelerator controls. The proposed new infrastructure would replace the old heterogeneous system used during the operation of LEP and would provide a new uniform facilities for the LHC era. In particular, that would include support for the standard Accelerator Device Model, publish and subscribe paradigms and inter-operability with industrial control systems solutions. Based on these requests, a more precise study on requirements and available middleware technology was started. Thus, the CMW project was launched, and as one of its main parts, the RDA package.

3.2 Design choices

Apart from already mentioned general requirements, the design of RDA was significantly influenced by its purpose of providing access to the accelerator devices from application programs. The parts of the system should inter-operate transparently and efficiently, independently of the programming language or platform used. Officially, the library supports applications developed in C++ and Java programming languages, running on any platform used at CERN accelerator controls (Linux, LynxOS, Microsoft Windows).

At that time the best solution satisfying all the demands was CORBA. Without hesitation it was chosen as the communication technology. Though, it was decided that RDA interface would hide all the CORBA-oriented details. This standard approach in object oriented programming would make eventual changes of underlying implementation easy without requiring any changes in the code developed by the library clients. Such an API also simplifies and speeds up the development process for people using the library as they do not need to know anything about CORBA or any other software used beneath the RDA.

To avoid compile-time dependencies because of some specific device data types, generic container objects were implemented and are used by RDA to pass the values between applications and devices. A container holds the values and their run-time type description allowing proper retrieval and interpretation of the data.

3.2.1 C++ generic data container

The first official versions of the C++ Standard Template Library (STL)[16] were implemented and released as early as 1994. However it still was not very popular in 1999 when work on CMW started. Furthermore, controls at CERN still use platforms for which there is no STL implementation or which do not fully support recent specifications of the C++ language. These yielded a code which does its work properly, but is hard to maintain and unfortunately does not fully exploit the robustness and power of the language.

The C++ implementation of the generic data container consists of three main classes: `rdaSet`, `rdaDataEntry` and `rdaData`, though only the last one is usually seen and used by a user. Together they allow the easy combination of data of various simple types (Booleans, integers, floating point numbers, strings, ...) and their arrays into objects of almost unlimited size and complexity (at the moment it is impossible to nest a structure into another one, yet such extension is planned for a future release). The class design, though not making everything out of the newer C++ specifications, makes it easy and intuitive to work with the classes. At the same time, the implementation details guarantee type safety of the created objects and their size optimization. The object size minimization is achieved mainly through the use of the C++ unions, and is a crucial element to improving the efficiency of any communication library.

In computer science, a union is a data structure that stores one of several types of data at a single location. There are two types of unions: a tagged union, where apart from the data stored there is also a tag describing what kind of data has been stored and an untagged union, which needs less space because it stores only the data without the describing name. On the other hand the lack of a describing field makes untagged unions type unsafe as it is the user of such a union who must remember what kind (type) of data has been placed there recently. The C++ language offers untagged unions with its `union` keyword. The `rdaDataEntry` class adds to the C++ union a tag and data type validation creating in principal a tagged union.

To create an object ready to send through a network; the user creates an object of `rdaData` type and feeds it with data using methods `insert` or `put`. As parameters such a call takes a basic data type (or an array of them) and a name by which it will be called and retrieved later on. Internally, this name is the union tag of which was mentioned in the previous paragraph. Figure 7 presents in a conceptual way the dependencies between the classes and main parts of their interfaces and structures. Corresponding Listing 1 shows an example use of the real classes.

A few things may need further explanation. When RDA was written for the first time the new features of the C++ language were not yet implemented for compilers on the LynxOS platform, which was the main system for controls at that time. This explains why instead of using a namespace, the C++ RDA classes have a prefix `rda`. There was also no standard STL implementation, and because of that a basic set like data structure was implemented and named `rdaSet`. Concerning the lack of C++ templates, all the needed insertion methods in the `rdaData` class were overloaded by hand. The methods support addition, extraction and

[16]The Standard Template Library, or STL, is a C++ library of container classes, algorithms and iterators. It provides many of computer science data structures (vectors, lists, maps,) and basic algorithms working with them. The STL is a generic library. It extensively uses C++ templates making all its components heavily parameterized and thus allows working in a type-safe manner with any built-in type or user defined class type [41].

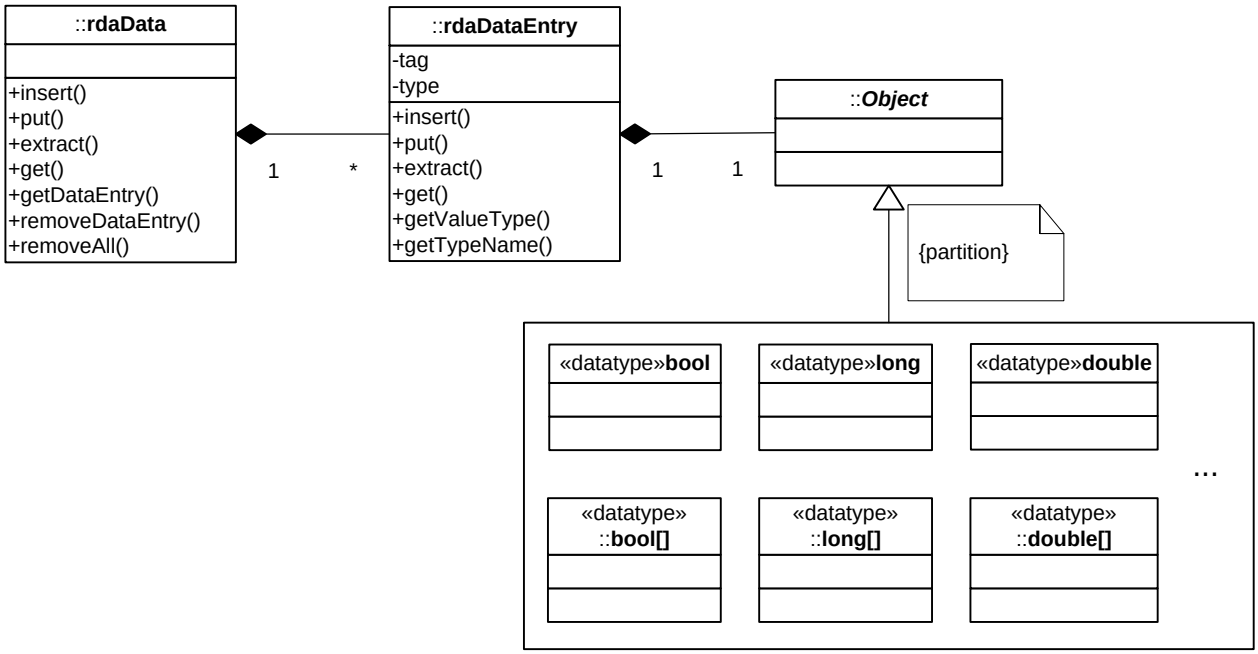


Figure 7: Concept view of the C++ data holding classes.

removal of an entry of any type specified by the union in the `rdaDataEntry`. There are a few ways to access the internally stored values of the `rdaData`. For basic types one can use `insert` and `extract` methods. These functions exist also for arrays of basic types. When used they perform a deep copy of the passed values. `Put` and `get` methods exist only for array types and instead of making a copy of the arrays they just set a pointer to them. As long as the user takes care of the lifetime of the objects, this can greatly speed up the creation of an `rdaData` object.

Listing 1: The rdaData use.

```
1  rdaData manipulateData( const rdaData & data )
2  {
3      unsigned long size;
4      const bool * arrayOfBools = data.getBooleanArray( "theBooleanArray", size );
5      rdaData newData;
6      newData.insert( "theFirstValue", arrayOfBools[ 0 ] );
7      newData.insert( "theLastValue", arrayOfBools[ size - 1 ] );
8
9      double dbl = data.extractDouble( "theDoubleValue" );
10     newData.insert( "doubledDoubleValue", 2 * dbl );
11
12     return newData;
13 }
```

3.2.2 Java generic data container

Since the days of its creation, shortly before the first release of the CMW, the Java language underwent even greater changes than C++ and its standard libraries. From a toy used mainly for writing Java Web applets[17] Java evolved into a powerful tool. New language structures

[17] A Java applet is an applet (a software component that runs in the context of another program) delivered to users in a form of Java bytecode. Java applets can run in a Web browser using a Java Virtual Machine (JVM), or in Sun's AppletViewer, a stand-alone tool for executing applets.

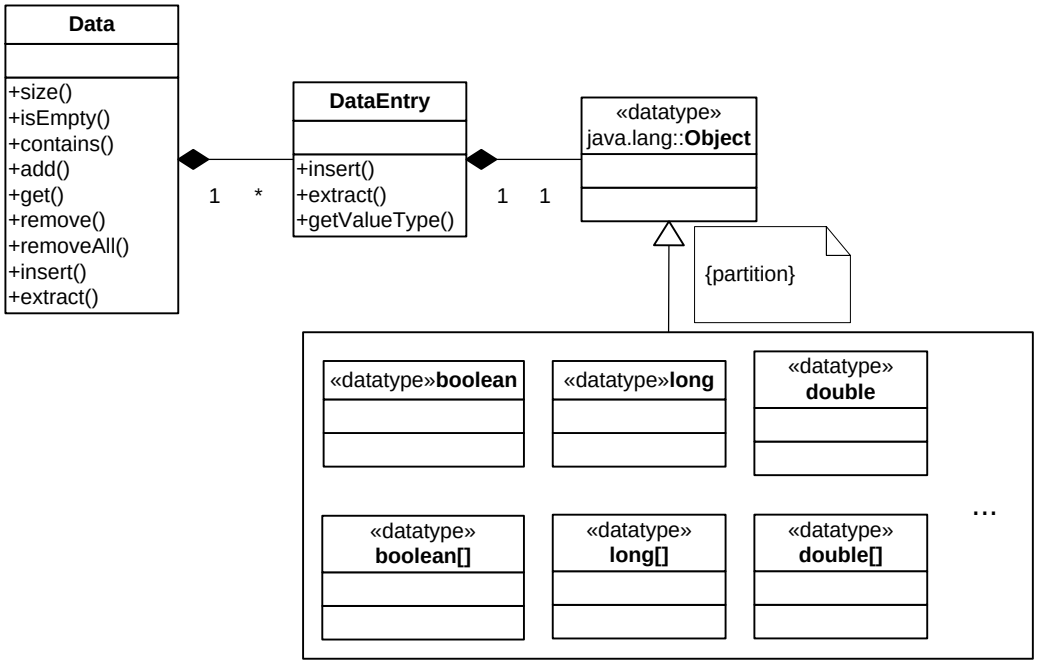


Figure 8: Concept view of the Java holding classes.

appeared, one of them are generics, which to some extent offer a functionality that is similar to that of C++ templates. Libraries grew large, supporting almost any type of functionality one could ever think of.

Thanks to existence of standard containers, the Java version of RDA data holding structures is less complicated than the C++ one. It consists of only two main classes kept in a `cern.cmw` package. These are `DataEntry` and `Data`. As in the C++ implementation, the `Data` class is responsible for storing and giving access to type safe tagged unions (`DataEntry` objects). Figure 8 presents in a conceptual way the dependencies between the classes and main parts of their interfaces and structures. Corresponding Listing 2 shows an example use of the real classes.

Listing 2: The cern.cmw.Data use.

```
1  private static Data manipulateData( final Data data ) throws BadParameter,
       TypeMismatch {
2      boolean[] arrayOfBooleans = data.extractBooleanArray( "theBooleanArray" );
3      Data newData = new Data();
4      newData.insert( "theFirstValue", arrayOfBooleans[ 0 ] );
5      newData.insert( "theLastValue", arrayOfBooleans[ arrayOfBooleans.length - 1
           ] );
6
7      double dbl = data.extractDouble( "theDoubleValue" );
8      newData.insert( "doubledDoubleValue", 2 * dbl );
9
10     return newData;
11 }
```

For both C++ and Java programs using the `rdaData/Data` structures, the type safety is checked at runtime. In case of a type mismatch or wrong name of an entry (not existing entry) an appropriate exception is thrown.

3.3 Architecture

RDA is based on a client-server model. Accelerator devices are implemented in device servers, and are accessed by applications using classes and interfaces of the RDA client part. The `RDAService` class manages communications with remote devices. It is also a factory for `DeviceHandle` objects. A client application uses a `DeviceHandle` to remotely invoke access methods on a device. The `DeviceHandle` delegates the calls over the transport layer to the `DeviceServerBase` object. `DeviceServerBase` is an abstract class that on its own implements and hides from users the way the calls are received. Being derived by the users leaves them to implement at least four abstract methods. They are automatically called whenever a corresponding method is invoked on an associated device handle.

The `get` and `set` methods come in two modes: synchronous (blocking) and asynchronous (nonblocking). The asynchronous calls require a reference to the object implementing the `ReplyHandler` interface as a parameter; when a reply to the request arrives at the client, the RDA passes the operation results to the specified object using methods defined in this interface.

Aparat from simple `get` and `set` calls, the `DeviceHandle` interface also has a `monitorOn` method dedicated for subscriptions. Similar to the asynchronous `get` and `set`, it requires an object that implements the `ReplyHandler` interface. This object receives subscription reports from the associated `ValueChangeListener` object on the server side. A new object of the `ValueChangeListener` class is created on the RDA server-side for each incoming subscription request. The server implementation uses this object to forward updates of a property value or errors to the client. There they are received with the above mentioned `ReplyHandler` object. A client can cancel a subscription with a call to the `monitorOff` method.

All the remote calls make use of the connection classes which are implemented as CORBA objects. This allows the object representing one end of the connection to invoke remotely methods of the associated object on the other end. On the client side the connection classes are represented by the `ServerConnection` and on the server side by the `ClientConnection`. These two classes provide all functionality needed to send operation requests and receive replies and also control the network connection they represent. To create a new connection, firstly, using CORBA objects, a server creates an entry point for the connection establishment. Then it notifies a Naming Service of its existence and location. When a client wants to send a request to that server it asks `RDAService` for the server location. Then using the CORBA object on its side it communicates with the server.

Under normal conditions the connection is closed on a client demand. Abnormal disconnections (e.g., due to a client or a server crash) are detected by the RDA connection monitoring mechanism which is based on a ping mechanism from a client to a server. When the client dies the server no longer receives the ping and releases all local resources related to it. On the other hand, if the client cannot successfully perform the ping call or an ordinary call to the server (due to some communication errors or a server crash), the client-side of the `ServerConnection` assumes that the server is inaccessible and invokes the disconnected method on all reply handlers waiting for subscription reports. Afterwards it starts to monitor the server reference on the Namig Service. When the server is up again it reregisters with the Naming Service. This triggers the reconnection procedure: the client tries to reconnect to the server and resend all pending subscription requests. The whole procedure is performed internally by the RDA library and is invisible to the users.

RDA architecture and basic flow of requests and data are depicted in Figure 9.

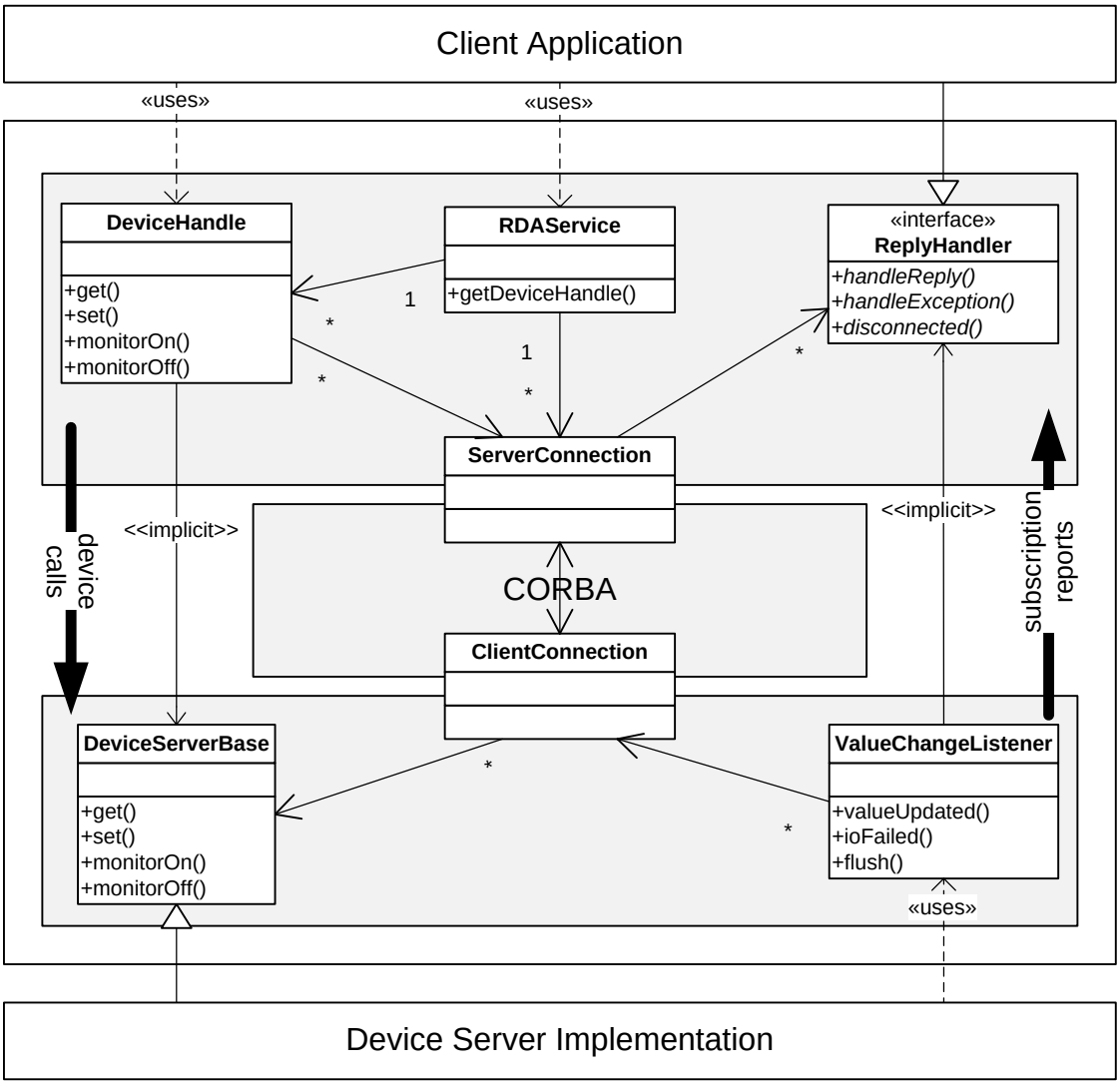


Figure 9: RDA architecture.

3.4 Example use

Below basic examples of client and server code using RDA are presented. The examples show how easy it is to send and receive data using the framework. Besides, looking at a basic implementation makes it easier to understand the architecture of the framework and the way the whole system works. Finally, these basics are cornerstones found in the implementation of all real client and server applications. Similarly, the client and server based programs described in the following parts of this work are also based on those snippets.

At CERN, but also outside of it, it is quite common to implement the client side with the Java language. Java makes GUI application development fast and easy. Unfortunately, due to many programming simplifications, Java also has many limitations. Java is expensive in terms of memory consumption and to run, it needs the Java Virtual Machine (JVM). Therefore C++, even though harder but at the same time much faster and giving to the users full control over the program behaviour for writing servers and similar services. Moreover, in the CERN environment almost all RDA servers run on FECs where it is impossible to run JVM. Following this thought, client code snippets are presented in Java, while the server ones in C++.

3.4.1 Java client side

Listing 3: Simple Java client using RDA library.

```
1  /// @brief Simple client example. Get a value from a server,
2  /// make a subscription and unsubscribe after a few seconds.
3  public class Client
4  {
5     public static void main(String[] args) {
6        try {
7           RDAService rda = RDAService.init();
8           DeviceHandle device = rda.getDeviceHandle("DEV_01");
9           Data data = device.get("PRP_08");
10          System.out.println("value = "+ data.extractDouble());
11
12          SubscriptionHandler handler = new SubscriptionHandler();
13          SubscriptionRequest rq = device.monitorOn("PRP_08", "CYCLE03_LHC",
               false, handler);
14          Thread.sleep(4000);
15          device.monitorOff(rq);
16       }
17       catch(Throwable ex) {
18          System.out.println("error: " + ex);
19       }
20       System.exit(0);
21    }
22 }
```

As described previously, for communication purposes, client side uses `DeviceHandle` class. In Listing 3, the user starts by initializing RDA service. The call to `RDAService.init` initializes RDA: loading configuration, setting up logging, initializing ORB and connecting to the Directory Service server name. Later on in the program, `RDAService` is important because it serves as a factory of `DeviceHandle` objects. It is used as in the example, by calling method `getDeviceHandle` with name of a device. This call queries the Directory Service for the physical location of the device and returns a handle to it. Later, the handle can be used to perform one of the `get`, `set` or `monitorOn` calls. In the example, a query for `DEV_01` device is made, then user asks for the property named `PRP_08` and in the end displays its value.

The next four lines of code present how to make a subscription. A user creates a `SubscriptionHandler` object shown in Listing 4 and passes it to `DeviceHandle` `monitorOn` method. This makes a remote subscription to the server and starts a new thread that waits for notifications from the server, and when they arrive, executes proper methods in the `SubscriptionHandler` object. That is why the main thread executes sleep method, putting itself to sleep and waiting for some data instead of just quitting the program. After a predefined four seconds, the main thread wakes up and executes `monitorOff` to cancel the subscription. This, again, makes a remote call to the server asking for the subscription cancelation. The program ends.

Listing 4: Simple Java subscription handler using RDA library.

```
1  ///
2  /// @brief Subscription handler. Print out received data and/or errors.
3  ///
4  class SubscriptionHandler extends ReplyAdapter {
5
6     @Override
7     public void handleReply(Request rq, Data value) {
8        System.out.println("VALUE RECEIVED: " + value.toString());
9     }
10
```

```
11      @Override
12      public void handleError(Request rq, CmwException ex) {
13          System.out.println("ERROR RECEIVED: " + ex.toString());
14      }
15
16      @Override
17      public void disconnected(SubscriptionRequest rq) {
18          System.out.println("DISCONNECTED FROM " + rq.toString());
19      }
20
21      @Override
22      public void reconnected(SubscriptionRequest rq) {
23          System.out.println("RECONNECTED TO " + rq.toString());
24      }
25
26      @Override
27      public void cancelled(SubscriptionRequest rq) {
28          System.out.println("CANCELLED " + rq.toString());
29      }
30  };
```

Listing 4 presents the implementation of a basic subscription handler class. The class extends `ReplyAdapter`, which implements the `ReplyHandler` interface. Inheriting from `ReplyAdapter` saves us from writing all of the `ReplyHandler` methods on our own, making the Client code shorter. The most important method is `handleReply` which is remotely called by the server. It delivers data from the server to the client side. Here, in the example, we only print it to the screen, but in a typical client application the information would be logged or displayed on a GUI panel. The second important method is `handleError` which notifies the client about various errors. This includes connection problems with the server, but also any error sent to the client by the server.

3.4.2 C++ server side

Though in CMW world, servers are usually created with the FESA framework; a fully hand-written code is presented here. Although writing everything by hand is slower than using a tool to generate nearly complete server code, doing so gives a full control over the program, and allows for greater flexibility when it comes to the designing and coding such a server. Moreover, the majority of the application examples described later on, are not quite typical servers.

Listing 5: Simple C++ Server declaration using RDA library.

```
1  ///
2  /// Basic server to test RDA communication methods.
3  ///
4  class Server : public rdaDeviceServerBase
5  {
6  public:
7
8      /// Ctor sets the server name and sleep time [ms] between the next
           subscription updates.
9      Server( const char * serverName, int sleep );
10
11     /// Get
12     virtual rdaData * get( const rdaIOPoint & iop, const rdaData & ctx );
13
14     /// Set
15     virtual void set( const rdaIOPoint & iop, const rdaData & ctx, const rdaData
           & value );
```

```
16
17      /// Add listener to the list.
18      virtual void monitorOn( const rdaIOPoint & iop, const rdaData & ctx,
            rdaValueChangeListener * listener );
19
20      /// Delete listener from the list.
21      virtual void monitorOff( const rdaIOPoint & iop, rdaValueChangeListener *
            listener );
22
23  private:
24
25      /// A thread generating data.
26      DataGeneratorThread * dataGenerator_;
27  };
```

As shown in Figure 9, the main server class inherits from the `DeviceServerBase` class. The interface of a stripped-down server is presented in Listing 5. Developer's work is to implement at least the four methods: `get`, `set`, `monitorOn` and `monitorOff`, which are automatically called by RDA when a respective call is recieved from the client side. Another important thing is to provide the server name to the `DeviceServerBase` constructor. That is why the Server constructor takes the `serverName` parameter and in the implementation (see Listing 6) passes it to its base class. In CMW, a server name advertises itself to the Directory Service and also allows clients to connect directly to the server (without device-parameter resolution by the Directory Service). Coming back to the constructor, it also creates and starts a thread that will provide data to its subscribers.

Listing 6: Simple C++ Server implementation using RDA library.

```
1   Server::Server : rdaDeviceServerBase(serverName)
2   {
3       dataGenerator_ = new DataGeneratorThread( 1000 * sleep );
4       dataGenerator_->start();
5   }
6
7   rdaData * Server::get( const rdaIOPoint & iop, const rdaData & ctx )
8   {
9       rdaData * data = new rdaData();
10      fillInData( *data );
11      return data;
12  }
13
14  void Server::set( const rdaIOPoint & iop, const rdaData & ctx, const rdaData &
        value )
15  {
16      setOptions( value );
17  }
18
19  void Server::monitorOn( const rdaIOPoint & iop, const rdaData & ctx,
        rdaValueChangeListener * listener )
20  {
21      dataGenerator_->add( listener );
22  }
23
24  void Server::monitorOff( const rdaIOPoint & iop, rdaValueChangeListener *
        listener )
25  {
26      dataGenerator_->remove( listener );
27  }
```

In case of `get` and `set` implementation, a call to `get` simply returns to the client some data provided by an external function `fillInData` (e.g. a read-out data from a device). A call to `set` passes the sent-by-client data to a function `setOptions`, e.g. which may alter some options of a physical device. A call to `monitorOn` adds a listener to the list of listeners while calling `monitorOff` removes the listener from the list.

Listing 7: Simple C++ data generating class definition.

```
1  /// \brief Generate some data every second.
2  class DataGeneratorThread : public IceUtil::Thread
3  {
4  public:
5
6      /// Set the sleep time.
7      DataGeneratorThread( int sleep );
8
9      /// Send new data to all the listeners.
10     virtual void run();
11
12     /// Add a listener in a concurent safe way.
13     void add( rdaValueChangeListener * listener );
14
15     /// Remove a listener in a concurent safe way.
16     void remove( rdaValueChangeListener * listener );
17
18 private:
19
20     /// Type definition of list of listeners.
21     typedef std::vector< rdaValueChangeListener * > ListOfListeners;
22
23     /// List of listeners.
24     ListOfListeners listeners_;
25
26     /// Sleep in microseconds between next updates.
27     const int sleep_;
28
29     /// Mutex object for synchronization of the class collection.
30     IceUtil::Mutex mutex_;
31
32     /// Copy of the data generated previously.
33     rdaData oldData;
34 };
```

The class `DataGeneratorThread` (Listing 7) is responsible for storing listeners in the form of pointers to `ValueChangeListener` objects. For this a standard collection is used and a mutex[18] that synchronizes access to the collection are used. But keeping client handlers is not its main responsibility. The main one is to send updates to them. This task is performed by the `run` method. As one can see in Listing 8 it is done by calling `valueUpdated` on the handlers, and passing to it `rdaData` objects with previous and present values, and a Boolean flag set to true if the previous and present data differ between each other.

Listing 8: Simple C++ data generating class implementation.

```
1  DataGeneratorThread::DataGeneratorThread( int sleep ) : sleep_(sleep)
2  {
3  }
4
```

[18]Mutex is an abbreviation for mutual exclusion, an algorithm used in concurrent programming to avoid the simultaneous use of a common resource.

```
5  void DataGeneratorThread::run()
6  {
7      while ( true )
8      {
9          usleep( sleep_ );
10
11         rdaData * data = new rdaData();
12         fillInData( *data );
13
14         IceUtil::Mutex::Lock lock(mutex_);
15         for ( unsigned int i = 0; i < listeners_.size(); ++i )
16         {
17             listeners_[ i ]->valueUpdated( oldData, *data, true );
18         }
19         oldData = *data;
20     }
21 }
22
23 void DataGeneratorThread::add( rdaValueChangeListener * listener )
24 {
25     IceUtil::Mutex::Lock lock( mutex_ );
26     listeners_.push_back( listener );
27 }
28
29 void DataGeneratorThread::remove( rdaValueChangeListener * listener )
30 {
31     IceUtil::Mutex::Lock lock( mutex_ );
32     ListOfListeners::iterator it = std::find( listeners_.begin(), listeners_.end
       (), listener );
33     listeners_.erase( it );
34 }
```

Now, the final thing is to instantiate the `Server` class and let it run. Listing 9 shows the initialization of the RDA, then the creation of the `Server` object and a call to the `runServer` method inherited from `DeviceServerBase`. This method should be called when the server is ready to accept client requests. By registering the server in the naming service and entering an event loop, it makes the server available to the clients. The event loop finishes, and if the `shutDown` method is called on the server, the `runServer` method returns. But as this basic server is supposed to run forever it is not needed to implement such functionality here.

Listing 9: Simple C++ server main function.

```
1  int main( int argc, char * argv[] )
2  {
3      rdaDeviceServerBase::init( argc, argv );
4
5      const char * serverName = "myServer";
6      int sleep = 1000;
7      Server server( serverName, sleep );
8      server.runServer();
9
10     return EXIT_SUCCESS;
11 }
```

The presented code implements a basic RDA server. The server is stripped of any exception and error handling, yet fully working. There is only one catch: this server always returns the same values regarless of the supplied device name and the property the client is supposed to make calls to. On the server side, data indicating the client's wish for the device and property names, with which he wants to communicate, is stored in the `IOPoint` class. Such

knowledge is enough to allow implementation of different handling methods for the different device-property pairs. For instance, it is enough to pass this data to the `fillInData` and `setOptions` functions, and store them along with the listeners in the data generating class. For simplicity, this functionality was not implemented in the example.

3.5 Testing and improving

Software Testing is the process of executing a program or system with the intent of finding errors [42] or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its specifications. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible [43].

Testing is more than just eliminating bugs and guarunteeing the correctness of the software. It includes quality assurance, reliability estimation and is performed to provide a generic metric on software efficiency (system resources used, performance, latency, etc.). Testing for correctness is performed to prove that functional requirements are fulfilled and that the system does what it is supposed to do. The other tests are performed to show that the system meets its non-functional requirements or to estimate the non-functional characteristics of the system.

3.5.1 Correctness testing and a unit-testing framework of choice

Correctness is the minimum requirement for software, the essential purpose of testing. Correctness tests are performed to prove that software meets its functional requirements. There are two basic approaches to test for correctness: black-box and white-box testing. In the first case, a tester does not care about the internal behaviour of the software. Usually these tests are the only choice when no access to the source code is given. The tester applies some input data and expects a specific answer according to the documentation or other specifications. In the second case, the structure and flow of the software under test are visible to the tester. He knows the inside details of the software, its control flow and data flow. Thus he is able to execute each line of code at least once (statement coverage), traverse each branch (branch coverage), or cover all the possible combinations of true and false conditional statements (multiple condition coverage) [44].

Unfortunately, even with white-box testing it is hard or almost impossible to cover all the possible flow branches. This is because of the exception mechanism and exception control flow. In C++ or Java, an exception can be thrown almost from any statement. In a typical case it at least doubles the number of execution paths that tests should follow. The exception can be almost of any type which only makes things worse.

In Java this problem is solved with method exception specification. Each method specifies what types of exceptions it can throw. The Java compiler checks this during compilation and fails accordingly in case of any inconsistency. More than that, the objects that can be thrown are limited only to those that are derived from the `Throwable` class. This limits the number of possibilities and guarantees a common interface with two key methods: `getMessage` and `printStackTrace`; Usually, standard exceptions are used or developers inherit their own exceptions from the `Exception` class (see Fig. 10 and Listing 10). Exceptions derived from the `Error` class are thrown in case of a hard failure in the Java virtual machine; the exceptions derived from `RuntimeException` are reserved to indicate incorrect use of the API. Java applications do not usually throw or catch exceptions of those two types, which are not mentioned in the method exception specification.

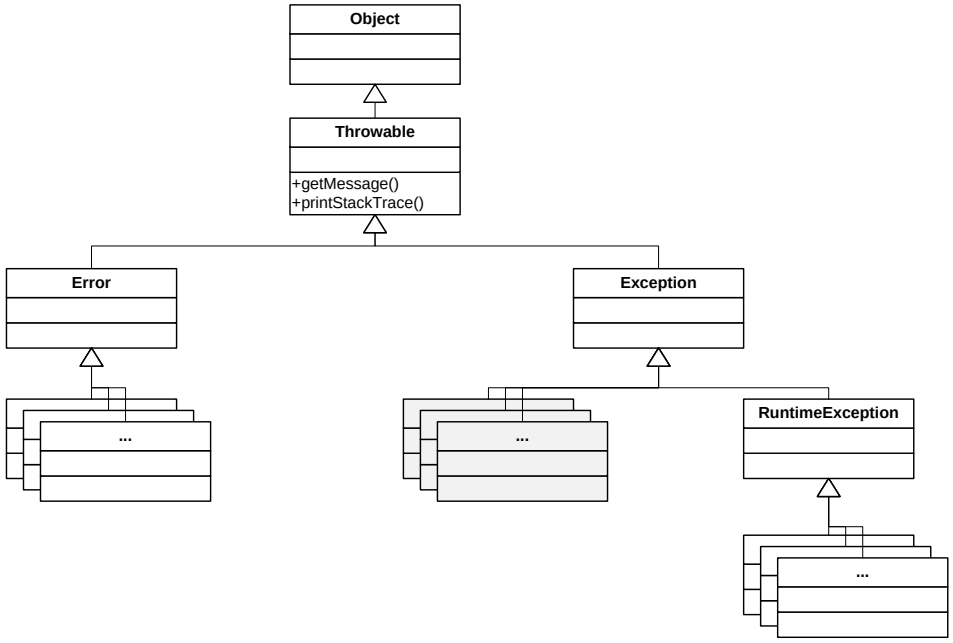


Figure 10: Java exception hierarchy. In grey user defined exceptions.

Listing 10: Java supports exception specification at compilation time.

```
1  class MyException extends Exception {}
2
3  class SomeOtherException extends Exception {}
4
5  class ExceptionTests {
6      // promise that only MyException can be thrown from this method
7      public void fun( int ex ) throws MyException {
8          switch (ex) {
9              case 0 : throw new MyException();
10             // this would not compile:
11             // case 1 : throw new SomeOtherException();
12         }
13     }
14 }
```

In C++, exception specification does not really work as expected. First, C++ exception specifications are optional (backward compatibility with old C++ standards before exception introduction), and second, no compile-time check is performed. A developer can specify that a function does not throw any exceptions, while in fact it does. A standard C++ compiler will compile such code without even a warning, and the only time a developer will be notified (that despite the specification an exception has been thrown) is only after executing the function. Moreover, such a call will do nothing more but unexpectedly finish the program execution without even indicating which line of the code was responsible for the termination (see Listing 11).

Listing 11: C++ lacks compile time support for exception specification.

```
1  void f() throw()    // promise that nothing will be thrown from this function
2  {
3      throw int();    // this compiles, but during runtime it immediately breaks
           the program
4  }
```

Listing 12 presents all the errors and misunderstandings to which the lack of compile time support for exception specification can lead. These include:

- The function specifies that it throws an exception which in fact it doesn't;
- The function can throw an exception against the exception specification;
- Throwing an exception against the specification terminates the program (calls `unexpected` handler which terminates the program by default); even having a proper catch block does not resolve this;
- Catch statements defined against the specification (catch blocks that should never be executed);

The problem is serious and difficult to detect. GCC[19] for instance, compiles the source with neither error nor warning, even if run with flags $-pedantic - Wall - Wextra$. The flag, $-pedantic$, issues all the warnings demanded by strict ISO C++, and rejects all programs that use forbidden extensions and some other programs that do not follow ISO C++. The $-Wall$ and $-Wextra$ flags together turn on all other GCC-supported warnings. [45]

Listing 12: C++ lacks compile time support for exception specification.

```
1  #include <cstdlib>
2
3  void f() throw( double )   // promise that only double can be thrown
4  {
5      throw int();   // this compiles!
6  }
7
8  int main()
9  {
10     try
11     {
12         f();
13     }
14     catch( int )   // this does not catch int, the program breaks all the same!
15     {
16     }
17     catch( char )   // this is against the specification, yet compiler allows it
           !
18     {
19     }
20
21     return EXIT_SUCCESS;
22 }
```

Similar to the Java `Exception`, the C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called `exception` and is defined in the <exception> header file under the namespace `std` [46]. This class has a virtual function, `what`, that returns a null-terminated character sequence (`char *`) and that can be overwritten in derived classes to contain some sort of description of the exception. Yet, there is no obligation to use that class, and a developer can throw a variable of any type.

[19]GCC, the GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages. It is the official compiler of the GNU system and has been adopted as the standard compiler by most other modern Unix-like computer operating systems, including GNU/Linux, the BSD family and Mac OS X.

On top of that, the C++ exception specification generates additional code which enlarges and slows down the executable; furthermore, it is no longer being promoted in modern coding guidelines.

Fortunately a tester can choose from a number of tools intended to help him with the tests. To facilitate correctness testing in Java there exists a standard, commonly-used JUnit library. For C++ no standard unit-testing library exists. For this reason, people writing tests in C++ either create their own testing tools or use one of the existing unit-testing frameworks. With respect to RDA, to find the unit-testing framework of choice a short list of expectations was created before actually writing the tests. In particular, tests should run on all CMW-supported operating systems (Windows, Linux and LynxOS). This implies that they should have no dependencies on non-standard libraries, nor should they rely on the newest C++ features (templates, STL, RTTI, etc). Writing test code should be fast and easy. Testing library should be small and easy to install, or it should be possible to copy it along with the test files without any installation. It should provide a good assert functionality, meaning that a failing assert statements should print the content of the variables that were compared and of course the place where it happened. Lastly, good documentation is necessary in order for any new person taking over the tests to understand their purpose and to maintain them in the future.

Bearing in mind the list of expectations a few libraries were examined: CppUnit, Unit++, Boost.Test and CppUnitLite. Here, a short outcome of the examination is presented:

- CppUnit [47] — fails with the most important issue because it requires STL and RTTI to run. This would be a problem when running the tests on LynxOS with the old gcc compiler. Besides, CppUnit requires quite a bit of work by the tester, even for the simplest possible test.

- Unit++ [48] — uses STL and `iostreams` which would not let it run on LynxOS. Besides, to run a test one has to register it manually and then add it to a suite, it is all a bit cumbersome. In addition, assertion support is very poor.

- Boost.Test [49] — as any C++ libraries with the Boost logo, it is very well documented and strictly follows the language definition; as a result there are no problems with running it on Windows and Linux systems in conjunction with new compilers. Unfortunately, it extensively uses new language features and includes quite a few supporting headers from other Boost libraries. Therefore it is not small and self-contained rendering it useless on LynxOS.

- CppUnitLite [50] — does not make use of any new C++ features like STL, templates, RTTI or even exceptions. To guarantee compatability with the old systems and compilers it defines its own string type instead of using `std::string`. As the name indicates, it is a light version of CppUnit, which consists of only a few files that could be distributed with the test files. Unfortunately, this is where the positives end. First CppUnitLite does not even have an official website. There is neither license to dictate the terms of usage nor documentation. Using assertions is uncomfortable because they are macros that do not use streams to print out the contents of their variables; as a result, one needs separate custom macros for each object type one wants to use. For anything else more than doubles, longs, or strings, one has to add custom macros by hand. It has only equality checks.

Of these libraries, CppUnitLite best fits the requirements, yet it does not provide much more functionality than a simple assertion. In conclusion, for the purpose of testing, standard

assertions and macros are sufficient. Such an approach yields a platform independent, simple, fast and easy to understand solution.

In this vein tests for Java version of the library use only a simple `assert` function. Even the standard JUnit library was abandoned to maintain consistency between the language-specific versions of the tests.

3.5.2 Tests performed and their results

With assertions used as a main weapon against bugs, RDA was thoroughly tested for correctness. Both black-box and white-box approaches were used when testing the library. The majority of the testing was focused on ensuring proper handling of the data structures. This is especially valid for the C++ implementation, for which the whole data structure system was developed from scratch (see Figure 7 in page 25). Java implementation needed less testing as it reused many standard Java containers wrapping around them and adding extra functionality.

The test suites[20] consist of a large number of calls to the data structures under various states. They cover all statements, branches and multiple conditions of the implementation of the data objects. A significant effort was also made to cover the execution paths caused by the exceptions. As previously described, it is hard, or rather impossible, to guarantee that such coverage is complete. A test case[21] is evaluated as passed when a data structure depending on its state returns an expected value or behaves in an expected way e.g. changes properly its internal state, throws an appropriate exception or sets a proper flag.

While performing the tests a few small bugs have been found. For example, wrong exceptions were thrown in the event of certain errors; sometimes not all the exceptions caught were handled properly, which led to memory corruptions and further errors. After mending the bugs, the C++ unit tests were rerun under the Valgrind [51] environment[22].

Valgrind was of great help, and found a few memory bugs. In some places memory was never released, and in others it was not released even when the exception was thrown. But the most common error found by Valgrind was due to discrepancy between the way memory chunks were allocated and deallocated afterwards. The problem appeared due to the fact that in C++ there are two independent ways of allocating and deallocating memory on the heap. They should not be mixed: if memory was allocated using one method it should also be released using the same method. The first one comes from C language in the <cstdlib> library in the form of two functions `malloc` and `free`. The other is the C++ `new` operator and corresponding `delete` operator. The `new` operator first allocates memory with a call to `operator new` function and afterwards executes object constructor. Similarly the `delete` operator first calls object destructor and then releases the memory with a call to the `operator delete` function. In RDA, implementation clashes appeared between the two allocation methods especially on the border between RDA and CORBA. It is common for CORBA to return heap allocated variables which users should release on their own or should use provided smart pointers that will release the memory automatically. Usually the CORBA variables were released in an appropriate way, but sometimes a copy of them was needed. The copy was created as memory chunks allocated with the old `malloc` function; these chunks should be later released with a call to the `free` function belonging to the same library. Yet, these variables were quite often incorrectly released with `delete` operator. Such code generally yields undefined behaviour. On some platforms, where the `operator new` and `operator delete` are implemented in terms of `malloc` and `free`,

[20] A test suite is a collection of test cases that are gathered together to test a piece of software and to show that it performs properly the tasks it promises to perform.

[21] A test case is a set of conditions or variables under which a tester will determine whether a piece of software meets specifications.

[22] Valgrind is a program which helps to find memory management and threading bugs.

this problem is illusive and many people do not consider it to be a bug as everything seems to be working correctly. In fact, it is a very serious bug, which may surface when moving the code to another platform or when the destructor of the allocated object fails to perform some important task because it was never called. All these errors have been fixed.

Listing 13: Typical C++ memory allocation inconsistency errors.

```
1  class C
2  {
3  public:
4      C() // ctor of the class may perform some important tasks
5      {
6          std::cout << "C() some important task" << std::endl;
7      }
8
9      ~C() // dtor of the class may perform some important tasks
10     {
11         std::cout << "~C() some important task" << std::endl;
12     }
13 };
14
15 void f()
16 {
17     { // fine, ctor and dtor called
18         C * c = new C;
19         delete c;
20     }
21     { // wrong!! unspecified behaviour; destructor not called
22         C * c = new C;
23         free(c);
24     }
25     { // fine, but note that neither constructor nor destructor are called
26         C * c = static_cast< C * >( malloc( sizeof(C) ) );
27         free(c);
28     }
29     { // wrong!! unspecified behaviour; ctor not called
30         C * c = static_cast< C * >( malloc( sizeof(C) ) );
31         delete c;
32     }
33
34     { // fine; 3 times ctor called and 3 times dtor called
35         C * c = new C[3];
36         delete [] c;
37     }
38     { // wrong!! unspecified behaviour; 3 times ctor called and dtor only once
39         C * c = new C[3];
40         delete c;
41     }
42 }
```

In addition to `new` operator and `delete` operator there are also `new[]` operator and `delete[]` operator in C++. They exist to allocate and release memory for arrays of elements. `new[]` operator first allocates memory for an array of objects with a call to `operator new[]` function and then calls constructor for each object. `delete[]` operator first calls destructors for each element in its array, and then calls function `operator delete[]` to release the storage. Again the `new[]` and `delete[]` operators should not be mixed with `malloc` and `free`, or with `new` and `delete` operators. Valgrind found in the RDA also a few bugs of this type. All of them have been fixed. Listing 13 presents in a simplified way most commonly found errors of the type.

The communication part of the library was tested by writing a server in C++ and clients both in C++ and Java. The clients performed various calls to the server sending and receiving data (via value and context parameters). Upon receiving the data, the object was checked for consistency. In particular, for `set` calls, the data was set on the client side and send to the server where it was checked. For `get` and `monitorOn` calls, the data was set on the server side and checked by the receiving clients. The consistency check relied on an extra parameter sent in the data object, which specified the other data of which the data object should consist.

3.5.3 Reliability tests

It is a hard task to measure software reliability[23], especially when the product is still undergoing improvements. Throughout its development cyle RDA has been used by thousands of real applications whose users reported all the bugs directly to the developers. Recently, the RDA library has stabilized. No new functionality has been introduced since then. A number of tests run on previous versions to prove their reliability were then rerun on the newest version and no flaws were found. No crash occurred nor did process memory consumption increase, which would indicate a memory leak. Apart from that many clients have already updated to the newest version of the library and are using it with success.

3.5.4 Performance testing environment

RDA provides a very user friendly API. This of course has an impact on its performance. To verify that the friendly API does not cost too much in terms of efficiency, the same performance tests were performed for two other communication standards, namely CORBA (omniORB ver. 4.1.2) and ZeroC's Ice[24].

RDA library runs on all platforms supported by CERN and is easily accessible on all CERN computers, and so is CORBA. Unfortunately, Ice is not widely used at CERN, and with strict software policies the library could not be installed on a typical CERN machine. Because of that, a special machine running Linux was devoted to testing, where all the needed libraries were installed. To keep as many similarities to RDA as possible, the libraries were compiled on the local machine without optimization options, and with the debug flag turned on.

The test machine was 64 bit and ran an unsupported version of SLC5[25]. Unfortunately, that was the only machine on which software not supported by CERN could be installed, and therefore the only place where tests of all three frameworks could be performed. The machine in the rest of the document is referred to as SLC5 machine.

Tests were repeated on official machines running SLC4[26] but only with RDA and CORBA libraries. The particular settings of the machines are not important as it is the ratio of the execution times that has more meaning than the absolute results.

[23]IEEE defines reliability as the ability of a system or component to perform its required functions under stated conditions for a specified period of time [52].

[24]The Internet Communications Engine (Ice) is a modern object-oriented middleware with support for C++, .NET, Java, Python, Ruby, and PHP. Supporting and making use of the newest standards of so many languages Ice is used in many mission-critical projects by companies all over the world. Ice is easy to learn, yet provides a powerful network infrastructure and vast array of features for demanding technical applications. It shines where technologies such as SOAP or CORBA are too slow and complex [53].

[25]SLC5, Scientific Linux CERN 5, is a Linux distribution build within the framework of Scientific Linux which in turn is rebuilt from the freely available Red Hat Enterprise Linux 5 (Server). Scientific Linux CERN is built to integrate into the CERN computing environment but it is not a site-specific product: all CERN site customizations are optional and can be deactivated for external users. Formal certification has not yet started and at the time of writing this thesis SLC5 should be used for tests and SLC4 for production use [54].

[26]SLC4, Scientific Linux CERN 4 [55], at the time of writing this paper it is the officially supported Linux system at CERN. For more information see the note on SLC5.

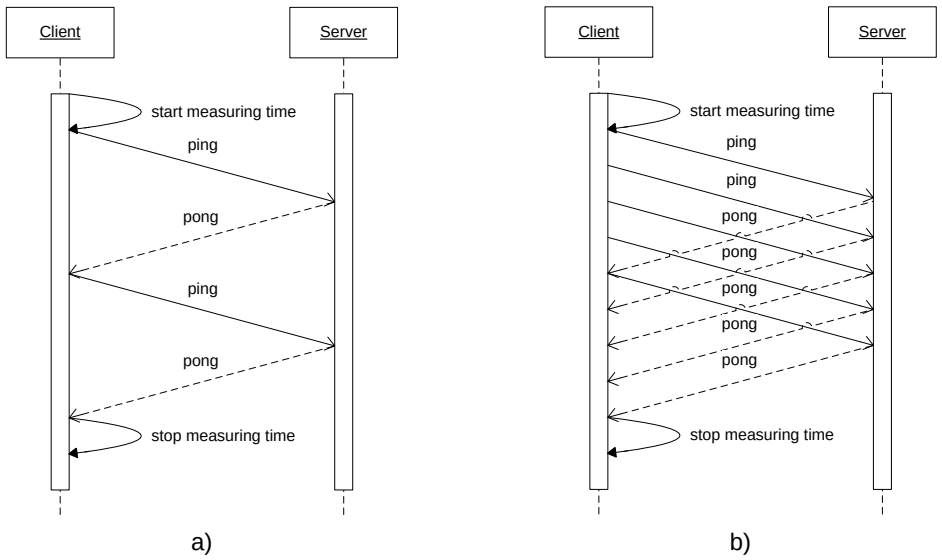


Figure 11: a) Proper performance test (proper to estimate number of messages per second). b) Scalability or throughput test (sometimes wrongly referred to as performance test).

3.5.5 Response time

The system response time is the interval between the moment at which an inquiry message has been fully sent and the beginning of the transmission of a response message to the station originating the inquiry. In terms of RDA, the response time characteristic might be interesting for those that exchange small packets of data (e.g. sending notifications that something has been done) or are interested in how much time it will take before the other side starts executing an action which they demanded. It would show the scale of delay that a client (a person sitting in CCC or a program running on an outside machine and managing FECs) may experience.

In this section only the time needed to exchange small packets of data is presented. The time needed to exchange big packets of data between a client and a server will be presented in the next section. This is because from the client's point of view, receiving a response from a server is an atomic operation, in the sense that a client receives either a whole data object or nothing. So it is impossible to measure response time in accordance with its definition. Thus, in fact, what we measure here as the response time is the time for the whole communication. For small objects the result could be treated as the response time approximation, while for the big ones, it is more natural to call it performance.

For the tests performed, the response time is defined as time span between sending an integer value from a client to a server and receiving a response back on the client side. Server's work for this test is receiving the client's message (the integer value) and sending it back to the client.

The results of the tests were also used to estimate the number of messages per second that RDA can handle. The client performs a synchronous call sending to the server a message (an integer value) after that it awaits for an answer (receiving back the same integer value). Only one message is sent at a time, the client side waits for the answer, and then sends another one. A message has to finish its full round trip before another one can start. Even if this sounds obvious, many developers do not understand the concept. They boast about their libraries being able to handle millions of messages per second, just because they send at a time, asynchronously, many of them, and then collect the responses. Of course, such a test is a scalability test, not a performance one. The idea is presented in Figure 11.

The response time was measured by writing appropriate client and server application and performing the communication 10^5 times to average the results. For comparison the same tests were written and performed with plain CORBA and Ice. The results are presented in the Table 2.

Table 2: Response time

SLC4	RDA	$55\mu s$	18182 messages/s
SLC4	CORBA	$25\mu s$	40000 messages/s
SLC5	RDA	$120\mu s$	8333 messages/s
SLC5	CORBA	$55\mu s$	18182 messages/s
SLC5	Ice	$110\mu s$	9090 messages/s

Table 3: Time needed to feed RDA Data object with an integer value and then to retrieve it.

SLC4	$2.5\mu s$
SLC5	$5.0\mu s$

At first, the results are a bit shocking. CORBA is approximately two times faster than RDA. RDA is slower because of all the work RDA does in the background and because of the RDA Data object transformation into a proper CORBA object that is afterwards sent by an underlying CORBA layer. By studying the time needed to fill in an RDA Data object with integer value, one can learn that in this case the container design on its own is not the main reason of extra time needed by RDA, see Table 3. There are a few factors which together sum up and make the big difference:

- Transformation of the data between RDA and CORBA.
- The fact that CORBA object to which RDA Data is transformed is not a plain CORBA integer type, but a dynamic structure corresponding to the RDA Data object but defined in the CORBA IDL. Similarly to the RDA Data, the CORBA version is a collection (sequence) of tagged unions. In case of an integer value, the wrapping object, which is sent automatically, is bigger than the plain integer we have started with. In the tests performed, the respective CORBA object is six times bigger than plain integer would be.
- It is not only the CORBA version of the RDA Data that is sent when communicating using RDA. The underlying CORBA layer sends additional data, e.g. strings describing the provenance of the Data object: device and property names. The more things to send, the more time it takes.
- All the objects created by CORBA are created dynamically.

There is no such excuse for Ice. It performed poorly when it was tested by sending the plain integer value back and forth between the client and server. The results are disappointing. Ice performs poorly with basic types.

3.5.6 Scalability and performance

Both scalability[27] and performance have been tested together. With a typical system, as with RDA, increasing the load decreases the performance. To perform the tests a new client and server programs were written. The client performed `get` calls on the server, sending data in the context field and retrieving its copy from the server through the return value. Thus only one connection was made for bidirectional communication. The data sent to and from the server had a typical structure of the data sent between FECs and their clients. It consisted of an array of fixed length, thirty-character long strings (which could be associated with the names of the measured values) and an array of doubles (the measurement values themselves). On the SLC5 machine, for the array of size one, the size of data was 34 bytes.

As expected, the results show a linear time consumption growth in the length of the array size[28]. Tests up to the number of two million elements (around 70 MB in size) were performed. Up to this threshold, no problems with the data transmission occurred. After exceeding the threshold value, with the precise limit hard to estimate, the data transfer failed occasionally. As the need to transfer such amounts of data at one go hardly ever arises, this does not pose a problem. Yet, if there was a need for such capacity, one could divide the data into a few smaller chunks and send them separately. Graphs below show results obtained within range $[10^3; 10^5]$ of elements, which corresponds approximately to [34; 3400] KB.

An important factor is that the data structures used for the mesurements both on the client and server side were not deep-copied. Instead the pointers were set to previously created structures. Because of that, the measurements include only the time spent internally by the library on preparing the data to be sent and the very act of communication. For each measured data quantity, the measurements were repeated ten times and an average value was taken; this cuts off random local distortions.

As it has been metnioned before, RDA provides a very user friendly API. This of course has an impact on its performance. To verify the impact of thefriendly API on efficiency, the same test cases were developed and executed for the other two communication standards, namely CORBA (omniORB ver. 4.1.2) and ZeroC's Ice. To read more about the testing environment see Section 3.5.4 on page 40.

Figure 12 presents the results for all described middleware systems obtained by running the client and server on the same 64 bit test machine running SLC5; the same tests were repeated on an SLC4 machine (without Ice). At first, the results are a bit shocking. RDA is approximately two times slower than its counterparts. This is the price that has to be paid for its internal data structures management mechanism. Dynamic allocation/deallocation of the memory is an expensive operation.

There are few things that explain why RDA underperforms. First, the performance cost comes not without a gain: there is no need to write an IDL or Ice interface specification for either the client or server; no need to generate stubs and skeletons and keep them all up to date and coherent with the IDLs; RDA interface is very flexible and lets users create type safe data structures that are of unlimited complexity. Another thing is that middleware systems like the ones here are meant not only for inter-process communication which in fact was just tested. They are usually used to transfer data over network. At CERN in particular between servers running on FECs and client programs running on machines widely spread over the whole CERN site. In such a case, difference in performance of this size might be acceptable as it can be only

[27] In the context of a computer application, scalability is the ability to continue to function well when the problem with which the system is dealing is changed in size. Typically, the rescaling is to a larger size. Often scalability concerns also system performance improvement proportional to the capacity of added hardware. Here, we consider only the first part of the definition.

[28] It gets slower as the array gets bigger

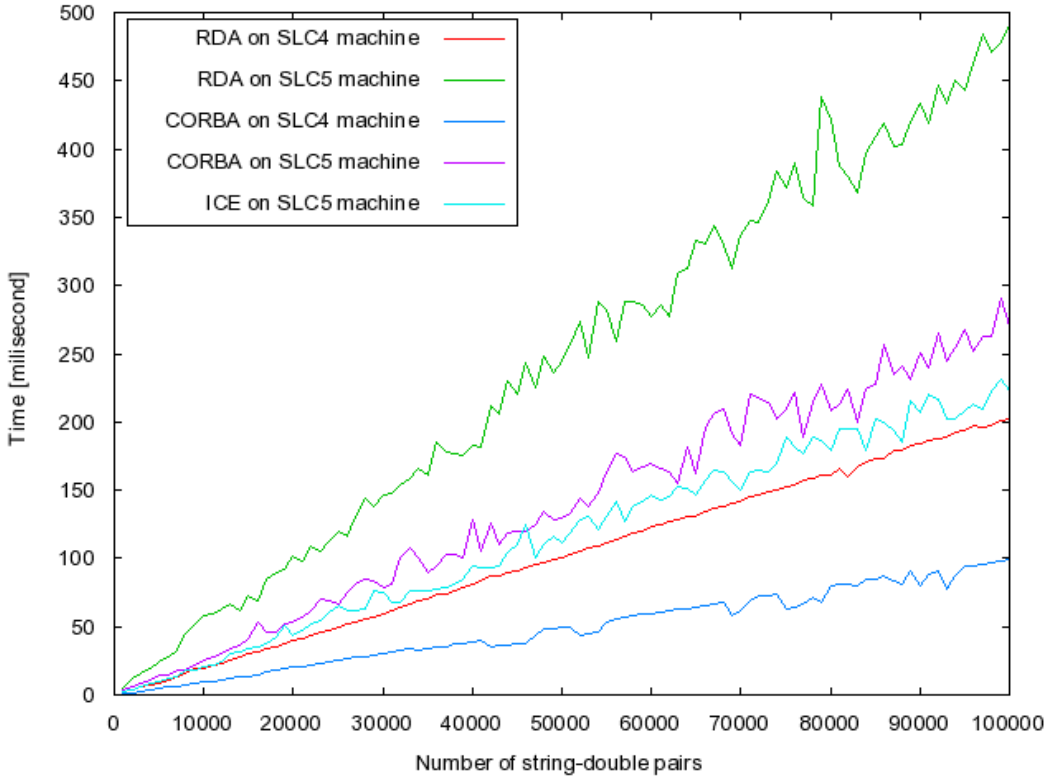


Figure 12: Performance tests (communication time).

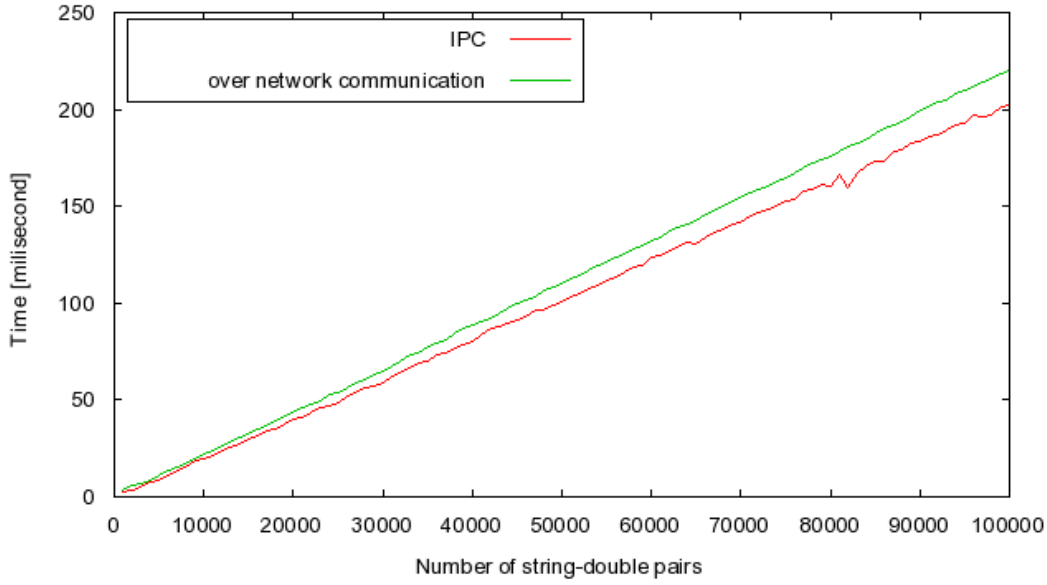


Figure 13: Performance tests. Comparison between IPC and over network communication.

a fraction of time spent on actual data transfer over network (loading and reading network card buffers, transmitting data). This strongly depends on the hardware used, network parameters and is hard to estimate or measure, and as shown on Figure 13, is not always the case.

Figure 13 presents comparison in communication time between client and server running on the same machine, and client and server running on different machines. Both computers were running SLC4 and were in the CERN LAN. In the second case, the test was run on one machine which was on the CERN Prevesin site, and the other one on Meyrin site. CERN

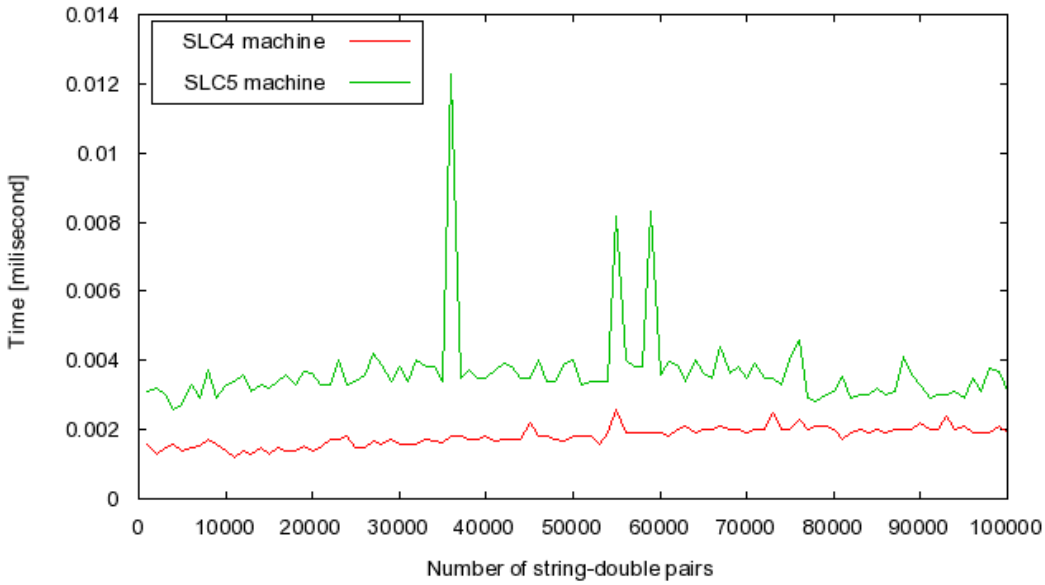


Figure 14: RDA Data put-get shallow copy performance test. Peaks are due to momentary server load increases.

Gigabit Ethernet network infrastructure works fast and on such small distances it does not introduce any measurable delays. The time spent on sending the data through the network is just a fraction of the whole communication procedure.

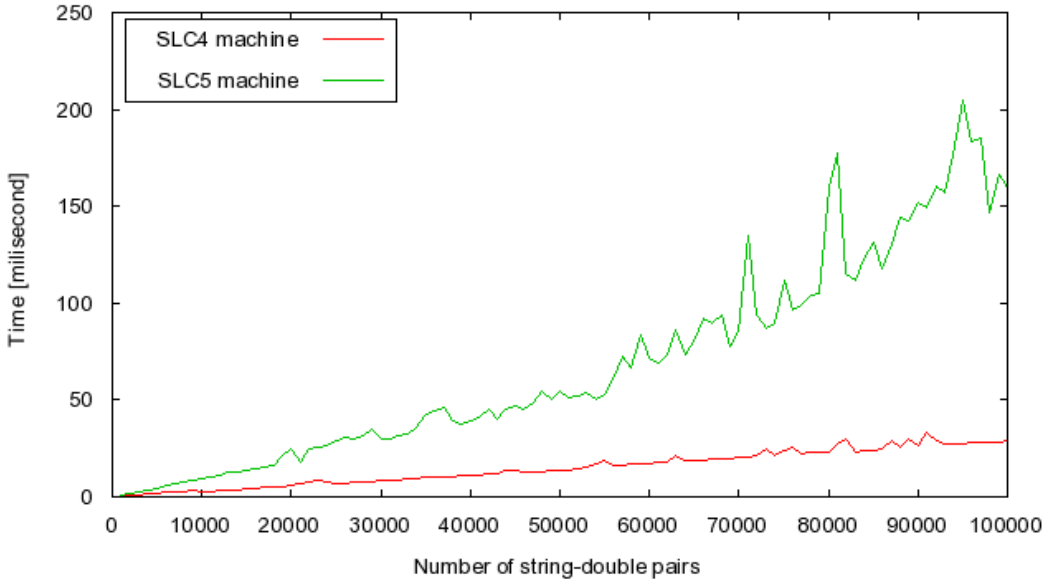


Figure 15: RDA Data insert-extract deep copy performance test.

Finally, a test was performed to see how much time on its own it takes to fill in an RDA Data object with arrays of strings and double values. Such tests could shed some light on the RDA efficiency shortcomings. Two methods of filling in an object were tested. First, filling by pointer (`put-get` calls; see Figure 14), which of course does not depend on the data size. Second, filling by a deep copy of the data (`insert-extract` calls; see Figure 15). The "by pointer" methods (`put-get`) were used in performance tests shown in Figure 12 when the client side was setting the RDA Data, and then when it was reading out the response from the server. Therefore it

is only the communication time that was measured, without the time needed for RDA Data object preparation. Yet, inside the RDA library the deep-copy functions (`insert-extract`) are used. In particular, when receiving the data both on the client and server side an RDA object is created by deep copying the data from an incoming CORBA object. As results show for the big data packets sent, the time spent on creating the RDA objects is more than half of the extra time that RDA uses.

3.5.7 Summary

RDA library was thoroughly tested. The tests cover the data storage classes, user interface classes along with exception specifications, and the whole process of communication (establishing a connection, exchanging data, closing the connection). They helped to eliminate a few bugs and provided more reliable code.

Recently an idea emerged to incorporate a continuous integration server[29] in the CERN BE-CO group for all its projects. The idea was widely approved and already a product has been chosen that will serve as the server. The server name is Bamboo and apart from other functionality it will also provide a testbed[30] for automatic test run (amongst them RDA tests). Integration and usage of the server will increase productivity of the whole group, improve code quality and facilitate information flow between the group sections.

Apart from RDA functionality tests also non functional software tests were performed. Reliability tests and efficiency tests (response time, scalability and performance tests) proved library dependability and gave an insight on its efficiency and limits.

[29] Continuous integration describes a set of software engineering practices that speed up the delivery of software by decreasing integration times. The practices include maintaining a code repository, automating build process and making the build process self-testing. Programmers should commit their code on every day basis and each of the head commits should end with a build, which should be kept as fast as possible. Finally everyone should see the results of the latest build, latest deliverables should be easily accessible and the deployment should be automated.

[30] Testbed is a platform for testing large development projects. With established tests, testbed allows rigorous, transparent and replicable testing.

4 CMW-RecorderPlayer

4.1 Introduction

Software products before their final release are usually thoroughly tested. It is the same case with libraries and utilities coming out with CMW logo. Unfortunately, testing does not guarantee error-free, 100% working and reliable products. On the contrary, as our own experience and experience of many other software development teams show – software is never bug free and there are always things to mend or improve. Bearing that in mind a suggestion came to develop another testing-oriented product. It came out under a name of CWM-RecorderPlayer.

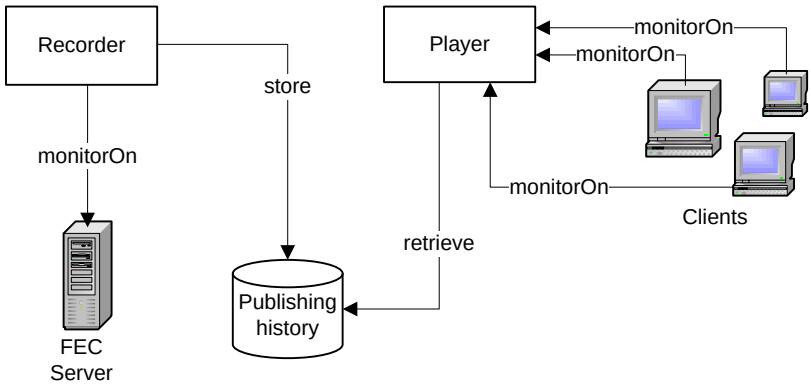


Figure 16: CMW-RecorderPlayer.

CMW-RecorderPlayer is a general testing product written to support eventual bug-hunt in case when an error was revealed during the run period of machines in the LHC complex. In particular, it is a tool that during the run will register the real data published by FECs. Afterwards, it will allow off-line diagnosis of the previously recorder data by republishing it as it was originally published by the FECs and recorded by the CMW-RecorderPlayer. By such functionality whole publishing scenarios can be played and analyzed without incorporation of the real software or hardware components. This can be used to track bugs spotted during the run, and being able to reproduce a bug is a vital part of finding and mending it. Product usage can be extended and further benefits may be obtained by analyzing the data send by the RDA and the way it is used. More profound understanding on such matters could lead to a stronger library optimization on its own, as well as server and client applications using the library.

CMW-RecorderPlayer consists of two independent yet cooperating parts. The first one, the Recorder part, is responsible for subscribing to servers while they are publishing, and afterwards storing the received data into files. The second one, the Player part, started on demand, republishes the data saved previously by the Recorder. Figure 16 presents the whole system.

4.2 Recorder

Recorder may serve as an example of a typical CMW client application. Figure 17 presents its static structure. Recorder inherits from `rdaReplyHandler` and implements its functions. Recorder main functionality is to subscribe to a server while it is publishing data, and then store the received data in files. On data arrival the `handleReply` function is called, which in

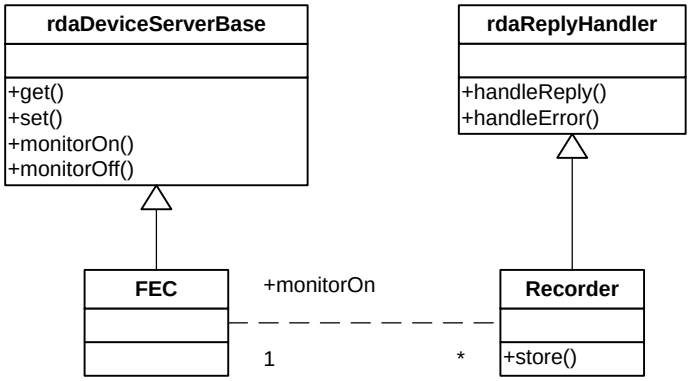


Figure 17: Recorder static structure.

turn calls store method, which actually writes down the received information into a file. When all desired data is stored `monitorOff` function is called and Recorder finishes its execution.

Figure 18 contains a sequence diagram with a typical run. There is an RDA server publishing data to which Recorder subscribes. Just after the subscription is established, Recorder starts receiving data. The first packet received is called first-update and it contains data that was most recently published by the server (functionality of the first-update is implemented in almost all servers using the RDA library). All the data packets received are immediately stamped by Recorder with present local machine time. The time stamp is made just in case if the data packet did not contain an acqStamp field and it is just an approximation of the original publishing time (which is synchronized among all the FECs). In the next step, Recorder checks if in the arriving data packet there is the acqStamp field. If it is there, it is used instead of the approximating local time stamp. Next, the received data is written to a file. File name contains the time stamp, device, property and eventually cycle names to which the received data belonged. This information is all what Player will need to republish the recorded data properly. The data itself is written to the file in a form of hex-coded RDA Data object.

4.3 Player

Player is a server application. Figure 19 presents its static structure. Player main functionality is performed by two classes: `Player` class which inherits `rdaDeviceServerBase` and implements its virtual functions and `Worker` class inheriting from `IceUtil::Thread`. `Player` class manages client subscriptions, while `Worker` class sends the previously recorded data to all listening clients.

A typical run of the application is shown on Figure 20. The main `Player` thread creates a `Worker` thread which starts publishing the data. If a client wants to receive the updates he subscribes for them by calling `monitorOn`. The call is performed directly on the server (without the device-property naming resolution) and the subscription covers everything that is being published, regardless device, property or cycle selector names. This is an intended functionality but can be easily changed to one supporting differentiation between the collected data objects corresponding to other device, parameter and cycle selector triple. After receiving `monitorOn` call, the `Player` thread adds the client to a list of subscribers and waits for eventual new ones. In the mean time `Worker` thread notifies the subscribers about the published data.

In fact, what `Worker` does is a little bit more complicated and the state diagram in Figure 21 sheds light on the details. After forking-off from the main thread `Worker` collects all the file names containing encoded data. As said before, they consist of timestamp (time when the

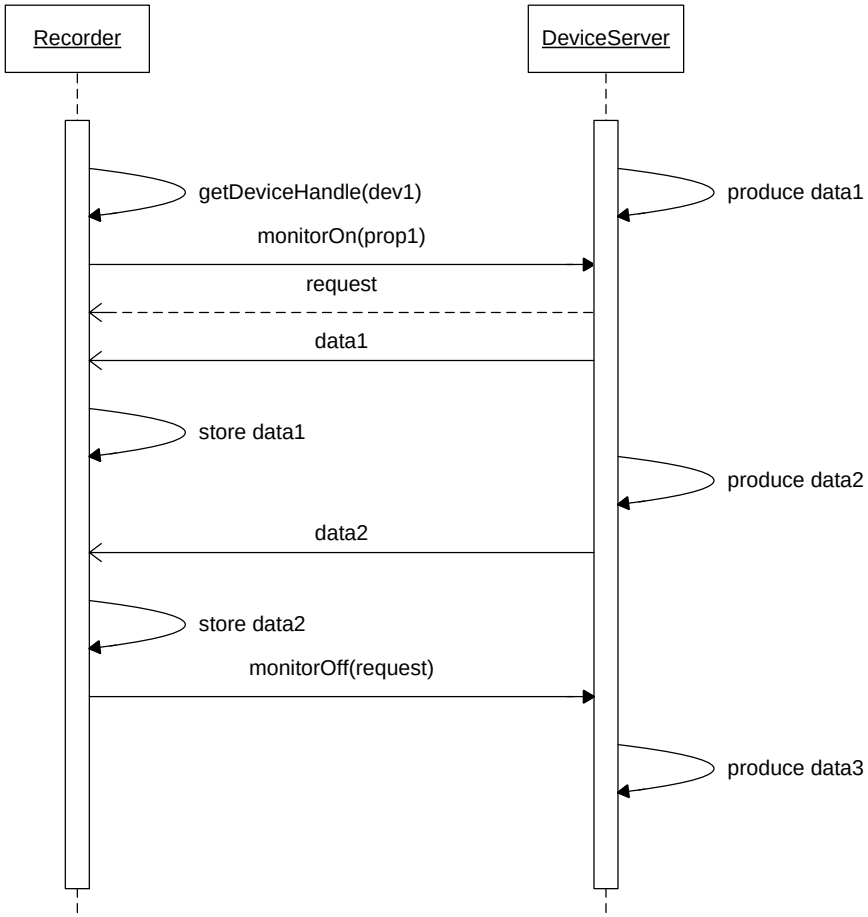


Figure 18: Recorder sequence diagram.

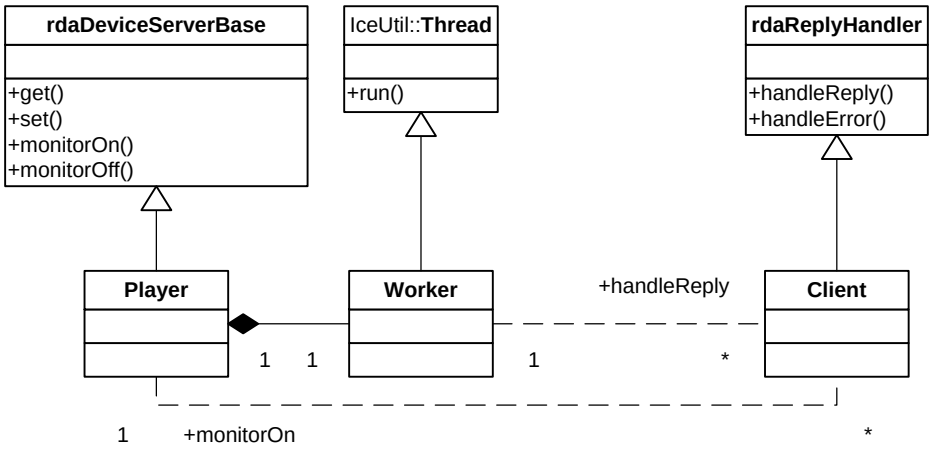


Figure 19: Player static structure.

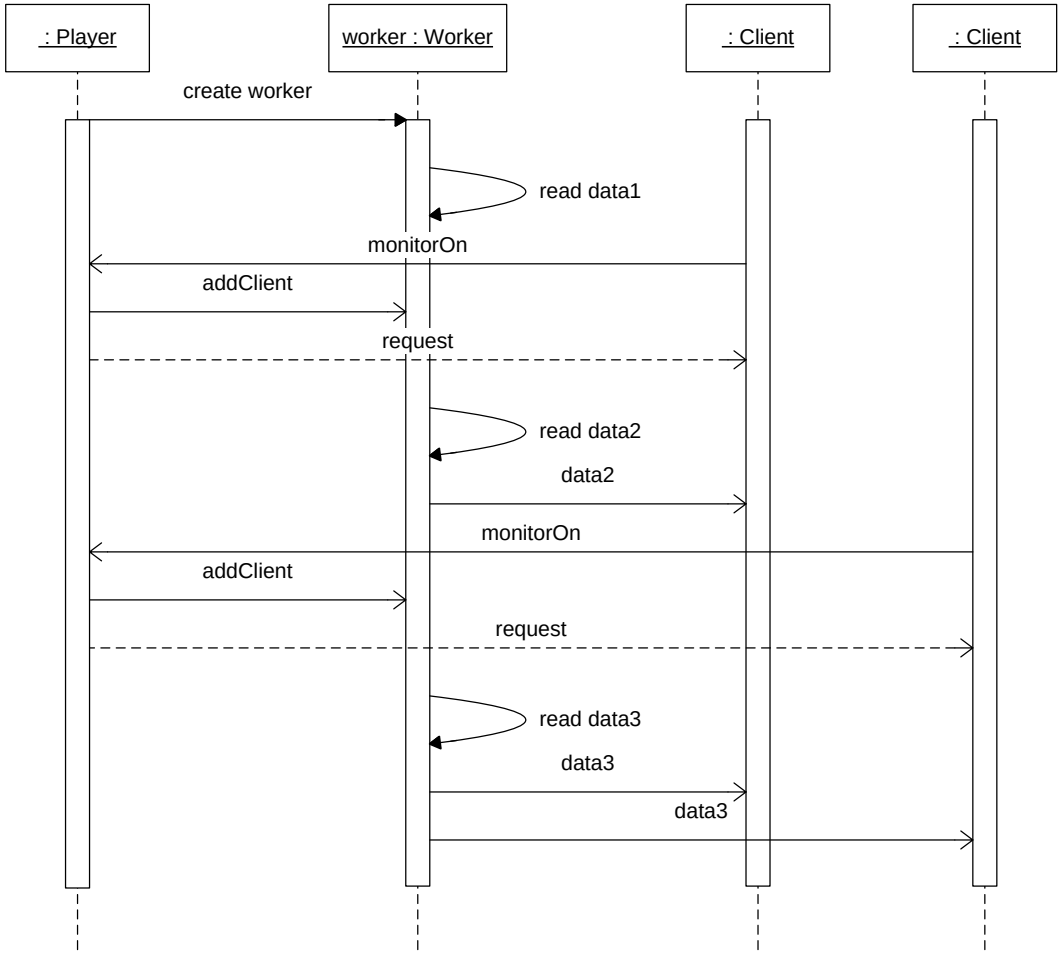


Figure 20: Player monitorOn and publishing sequence diagram.

data was recorded), device, property and eventual cycle selector names. The timestamp allows the sorting of all the data packets by the time at which they were recorded and the time at which they will be republished. The later three names might be crucial if the functionality of differentiation between device, property and cycle selector triple was needed on the level of CMW-RecorderPlayer (one can always guarantee that functionality by running more than one Player with segregated data files).

The files are sorted by the recording time. First the oldest one is read and decoded into RDA Data object. If the original server which published the object inserted into it an `acqStamp` timestamp field with publication time, the field is being updated with present time, so the data object seems to be present and not from the past. Next, the whole object is being published. `Worker` checks if the list of objects to sent is exhausted and if not it reads and decodes the next file on the list repeating the whole procedure. When the list is exhausted the `Worker` announces the fact to the main thread, awaits till all the data is delivered to the clients and finishes its execution. After receiving the message the `Player` thread also finishes its execution. The server stops.

In case when a client wants to stop receiving updates before they are finished, he calls `monitorOff` on the `Player`. As shown in Figure 22 he is being removed from the clients list, and does not receive any more updates. Obviously, this does not stop the `Worker` from publishing or other clients from receiving the updates.

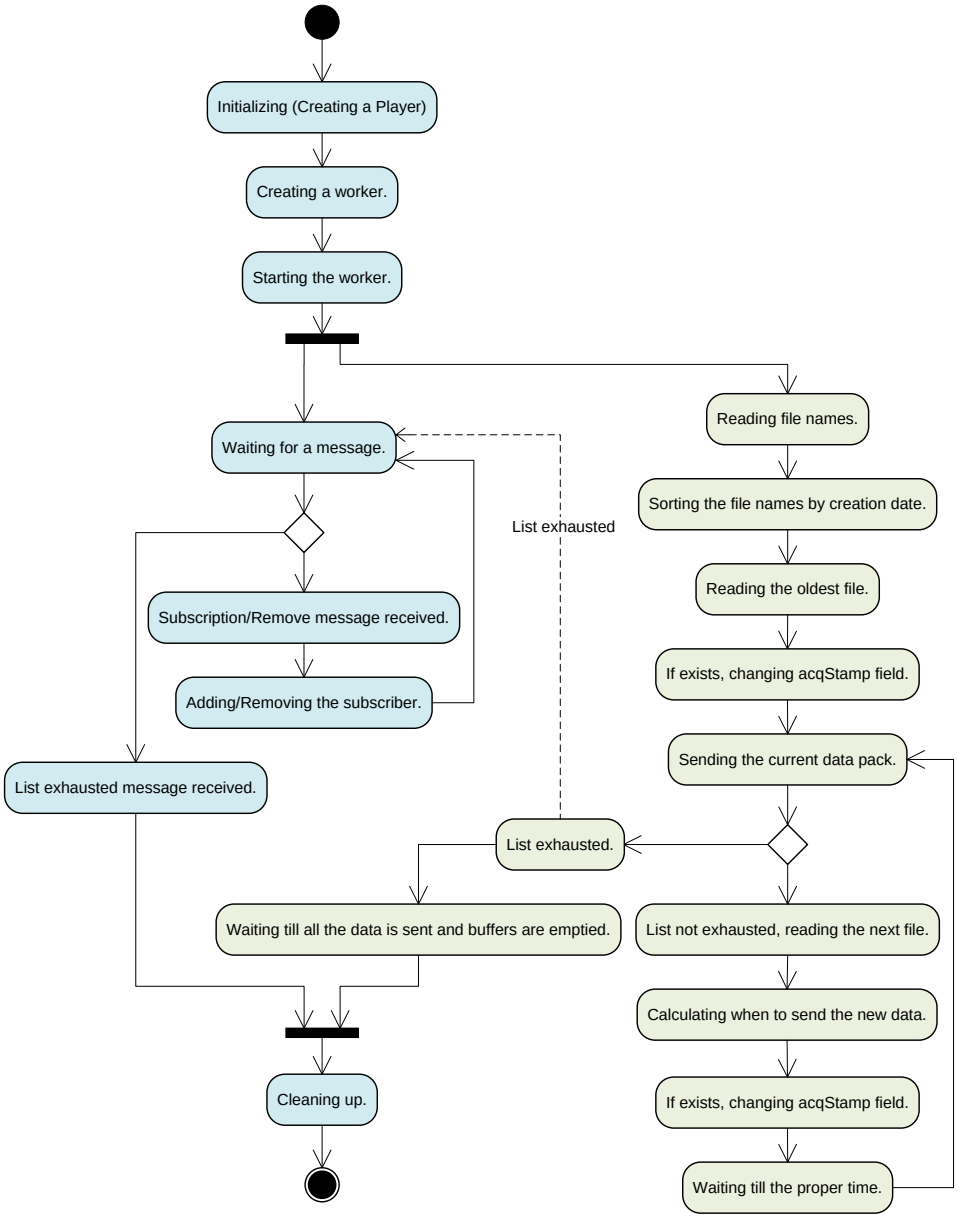


Figure 21: Player state diagram. The main thread is painted in blue and the Worker thread in green.

4.4 Testing

One of the most important part of the CMW-RecorderPlayer is its management of timing information. `Recorder` receives data and checks if the packet contains the `acqStamp` field, precise time of the data publication provided by FECs. If data contains the field it is being written as publication time. If data does not contain it, `Recorder` measures time on the local machine it is running on. This roughly estimates the value that should be provided by the `acqStamp` field. Afterwards, when the `Player` part is generating notifications it is using the time written down by the `Recorder` (either from `acqStamp` or the generated one) increased by the time that passed since the first publication was received till the first publication performed by the `Player`.

Time calculations are performed with accuracy to 1 μs. Even though it seems to be very accurate, it is much less than accuracy of the `acqStamp` field, which is 1 ns.

Tests were performed with a server publishing data at a rate of one publication per second. Recorder was receiving the updates and so was a client program which wrote down the time

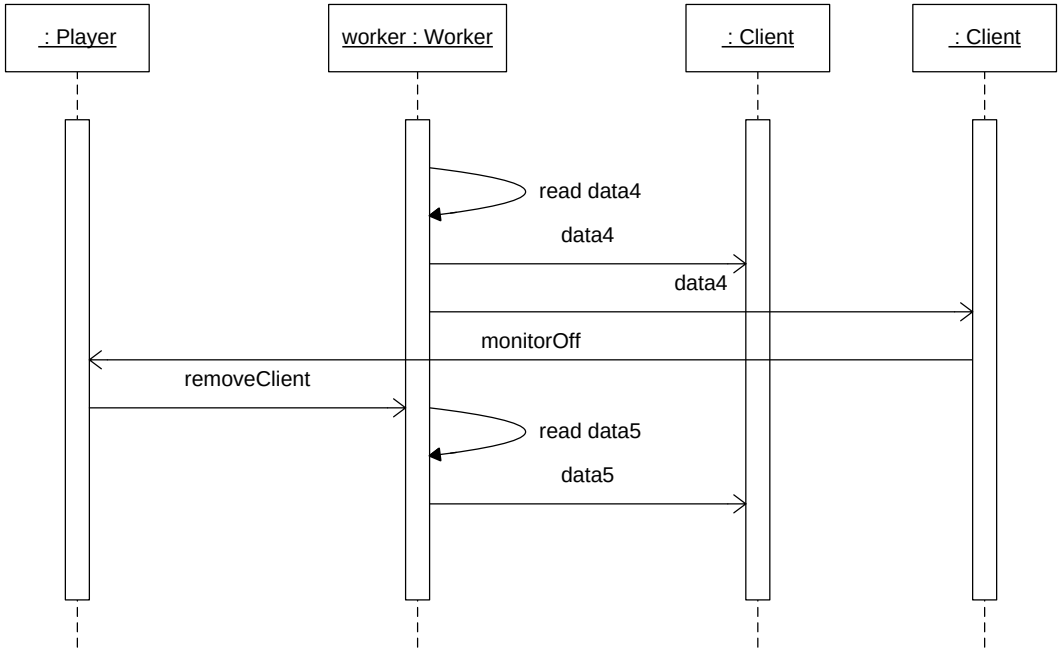


Figure 22: Player monitorOff sequence diagram.

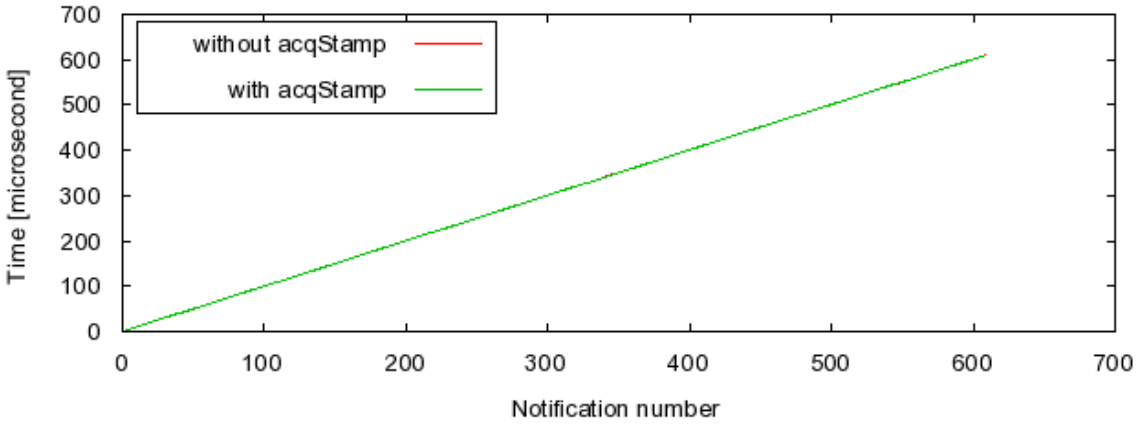


Figure 23: CMW-RecorderPlayer delay in relation to direct publication.

of notification arrival. 600 notifications were sent, received and recorded like that, which is a little over 10 minutes of the server run. After that, `Player` was publishing the data, and again a client subscribed to it was writing down time at which it received the updates. Later, the times measured by both clients (the one receiving from the server and the other one receiving from `Player`) were subtracted from each other. Tests were performed for both cases: a server sending the `acqStamp` field, and a one which did not send it. As can be seen in Figure 23, the result did not depend on the `acqStamp` field presence. CMW-RecorderPlayer performed equally in both cases.

CMW-RecorderPlayer prolonged each publication by 1 μs. This is not much, and is exactly on the border of time scales that CMW-RecorderPlayer functions operate at. After a day and night of server run (24h), that would result in 86,4 ms of difference between server and `Player`, and 24 hour long test would be quite a long one. Either way, in the future version of CMW-RecorderPlayer an update to support nanosecond scale might be considered.

5 CMW-Proxy

5.1 Introduction

Before the first LHC startup on Wednesday, 10 September 2008, the whole accelerator complex and the software that manages it were exhaustively tested. The tests indicated that under full load the FECs running CMW servers might be overloaded if there are too many client subscriptions. Such conditions may render the servers to be unstable, which is unacceptable during the machine operation.

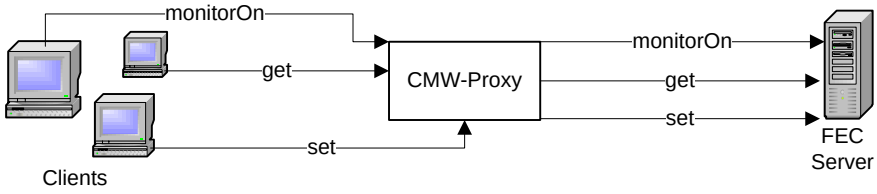


Figure 24: Client – Proxy – Server three-tier architecture.

As it was impossible to exchange the FEC hardware with any newer and stronger computers, other methods were looked for. As a solution a proxy application was suggested. It would run between client applications and FEC servers creating a nice three-tier architecture (Figure 24), and if possible it would decrease the number of calls to the servers. Overloading the proxy application would not be a problem, first, because it can run on a modern, strong machine running normal (non real-time) operating system. Second, because there could be many instances of the proxy running.

Apart from taking the load from the servers, an ideal proxy should be transparent to clients. They should not change anything in their application code; in fact they should not know that they are using proxy at all. It should allow clients to perform all the standard RDA calls (`get`s, `set`s, `monitorOn`s and `monitorOff`s). Being a server application it should support the same safety-guarantee procedures that real end-servers do, e.g. controlling what a typical client can and cannot do on the server side.

The idea led to creation of the CMW-Proxy application that fulfills all the needs stated above: it is transparent, and the only thing a client application has to do extra is to donate an argument with the proxy name it wants to use; it supports all main RDA ways of communication and standard safety procedures. Moreover, the application usage does not end with lessening the load from FECs; it may serve as a general purpose proxy forwarding messages between any clients and servers. As such it might be used as a gateway for clients that are running outside the Technical Network, yet they want to contact CMW servers running inside of it, e.g. on FECs.

5.2 Architecture

CMW-Proxy architecture is quite straightforward. There are only two basic concepts one has to know to understand how it performs its work and why it was implemented in the particular way.

First, CMW-Proxy is an application which actually implements both server and client sides. It is a server for all client applications that direct their requests through it. At the same time it is a client of any RDA server to which it redirects the clients calls. This concept is illustrated on Figure 25.

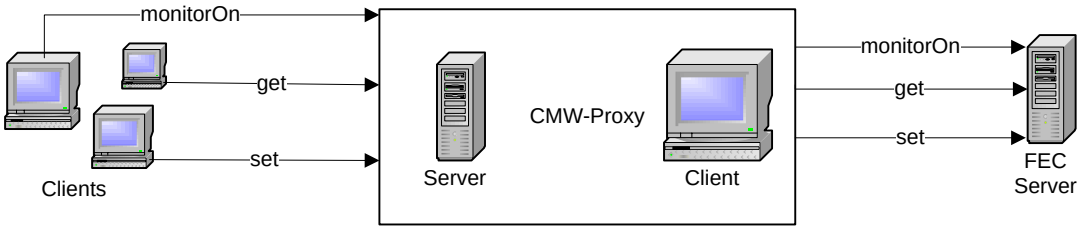


Figure 25: CMW-Proxy combines client and server functionality.

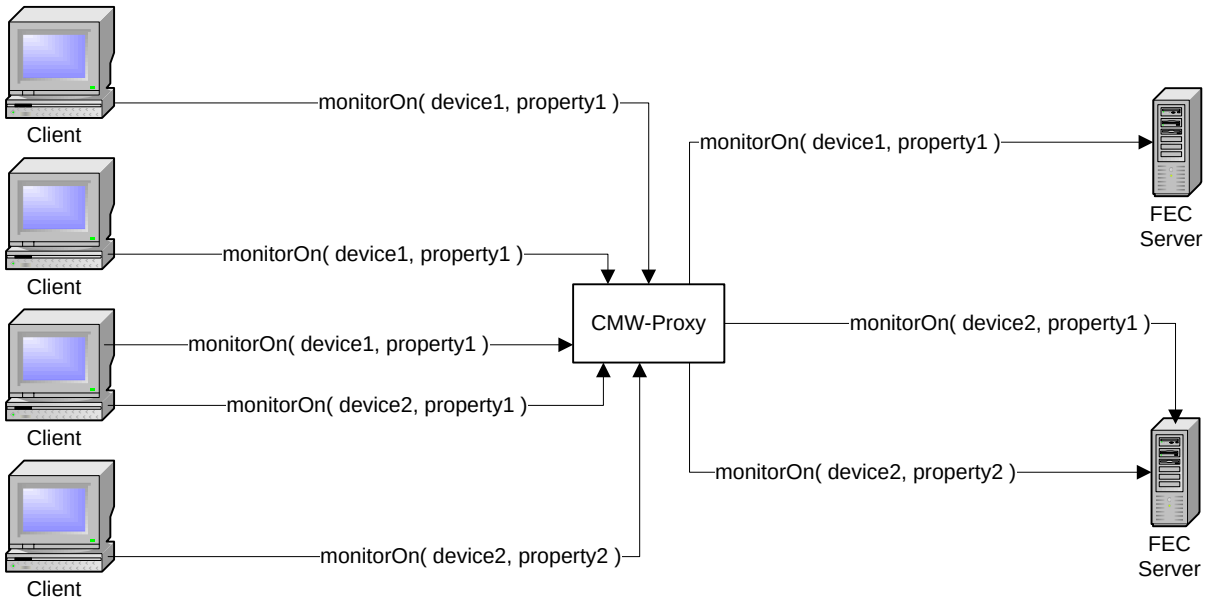


Figure 26: CMW-Proxy takes off load from FECs by combining subscriptions to the same device and property.

Second, the only time when CMW-Proxy can decrease load that is put on a FEC server is when there is more than one client subscribing to the same data published by the FEC. This means, that there are at least two clients subscribing to the same device-property pair (or device-property-cycle selector triple) without specifying the call context, see Figure 26. For any other calls: simple `get`s, `set`s or `monitorOn`s with context, CMW-Proxy just forwards the calls as they come without any effective optimization.

The mentioned reasons led to the structure presented on Figure 27. The main class called Server inherits from `rdaDeviceServerBase`. This makes it a CMW server, and methods that it implements from its base class: `get`, `set`, `monitorOn` and `monitorOff` are going to be executed by CMW system when a connected client calls them on his side. While `get` and `set` calls are forwarded directly to a destination server (Figure 28), `monitorOn` and `monitorOff` calls are passed to helper objects: `ManagerWithContext` if a call has a context or `ManagerWithoutContext` if it does not have it. Both managers keep a list of handler objects which extend `rdaReplyHandler` class. The difference between the two managers lies in the way they manage subscriptions between their clients and the destination servers.

In case of the manager with context for any `monitorOn` call a new handler object is created and added to the manager list of handlers. Afterwards, a new connection is established and a new handler object is set as the one responsible for receiving updates from the destination server and pushing them to an appropriate client. Because of such behaviour, the handler keeps

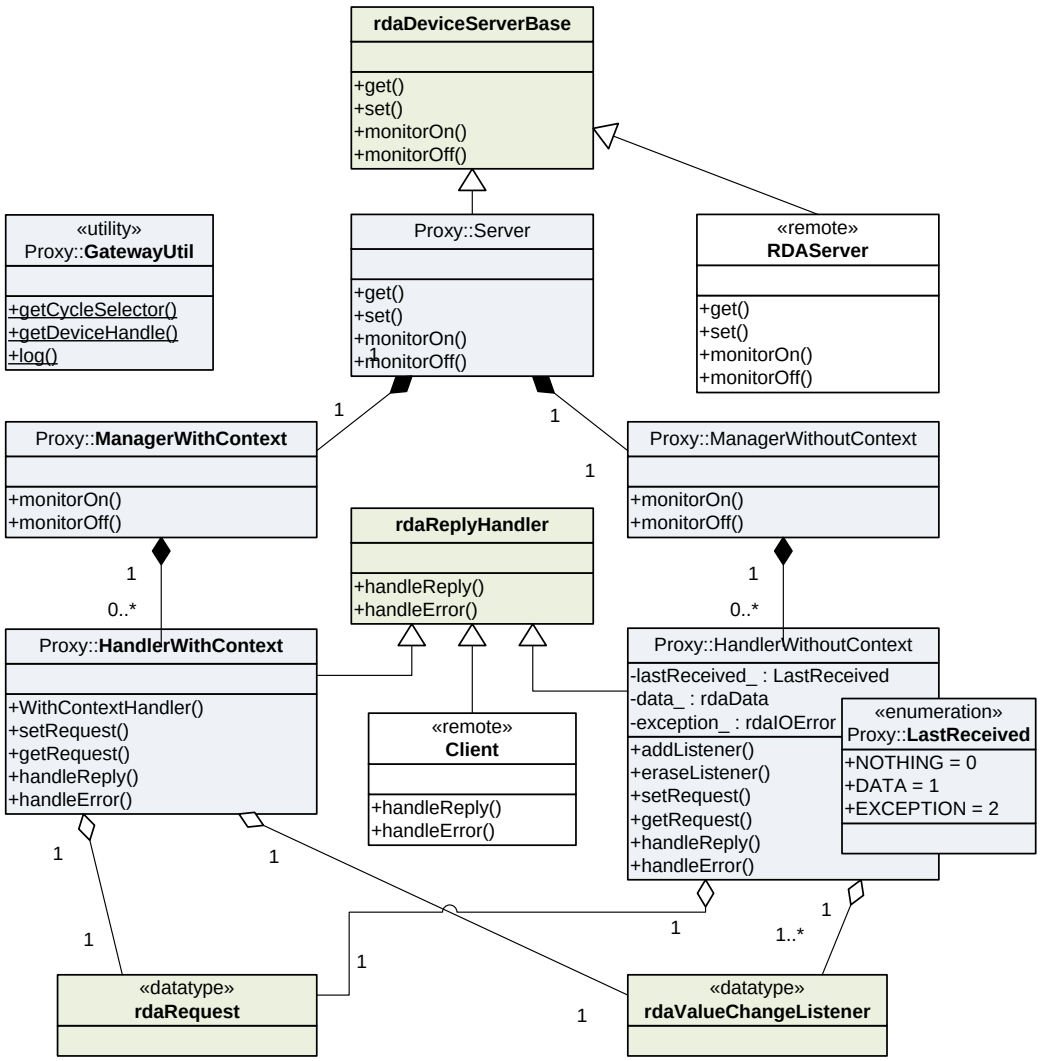


Figure 27: CMW-Proxy static diagram. CMW-Proxy classes are painted in blue while RDA library classes are in green. The two white classes represent an outside CMW-Proxy project RDA client and server, which knowing nothing about the proxy may still profit from its existence.

only one object of the type `rdaRequest`[31] and one of the `rdaValueChangeListener`[32].

In case of the manager without context, if it is a `monitorOn` call to a not yet subscribed device-property pair, everything happens similarly as for the manager with context. Yet, if a client tries to subscribe to a device and a property to which there already exists a connection the procedure is a bit different. No new handler object is created or new connection established; only to the existing appropriate handler object the client listener is added. Because of that the handler instead of keeping only one listener keeps a list of them.

Class `HandlerWithoutContext` compared to `HandlerWithContext` contains some extra class fields. They keep data which is used to support the first-update functionality. The `data_` field stores last received notification while the `exception_` field stores a last received exception. Field `lastReceived_` is an enumeration indicating what type of notification (normal data or

[31] An rdaRequest object identifies connection with a server on the client side. It is needed to finish a subscription with monitorOff call.

[32] An rdaValueChangeListener object is a server side object identifying a client that subscribed to the server with monitorOn call, and expects being notified about the server publications.

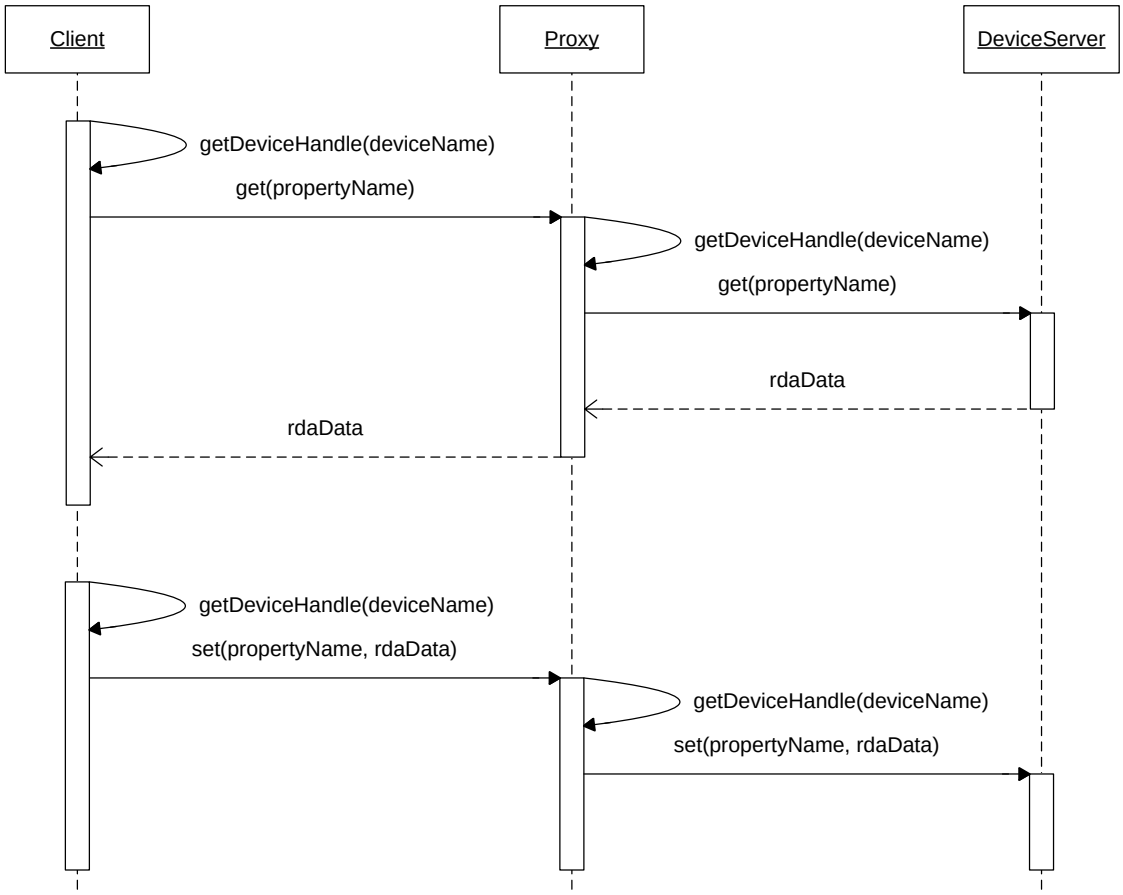


Figure 28: Get and set calls are forwarded by the main Server class to a destination server without any grouping or optimization.

an exception) the CMW-Proxy received most recently. It is set to `NOTHING` if the proxy has not yet been notified. `HandlerWithContext` does not need those fields because when it receives first update from an end server it resends it immediately to its only listener. It does not have to lay aside the data for future use because it will not have another subscriber.

Apart from the main implementation classes, the project contains also a set of utilities which are described in Section 5.4.

5.3 Optimization and execution issues

Even if the overall functionality of CMW-Proxy and the way it achieves it are not complicated, there are a few details worth looking at. As they all concern `ManagerWithoutContext` class let us concentrate on `monitorOn` calls carrying no context and respective `monitorOff` calls.

Figure 29 shows a sequence diagram depicting an optimization done by the proxy when there is more than one subscription to the same device-property pair. In the example the DeviceServer (FEC) instead of dealing with two subscriptions from clients deals only with one coming from CMW-Proxy. CMW-Proxy manages connection between itself and the server as well as with the clients. When CMW-Proxy receives request from a client to subscribe to a device-property pair to which the proxy was not subscribed yet the connection is established and forwards to the client received from the server first-update. From this time on the proxy regularly forwards received updates to the client.

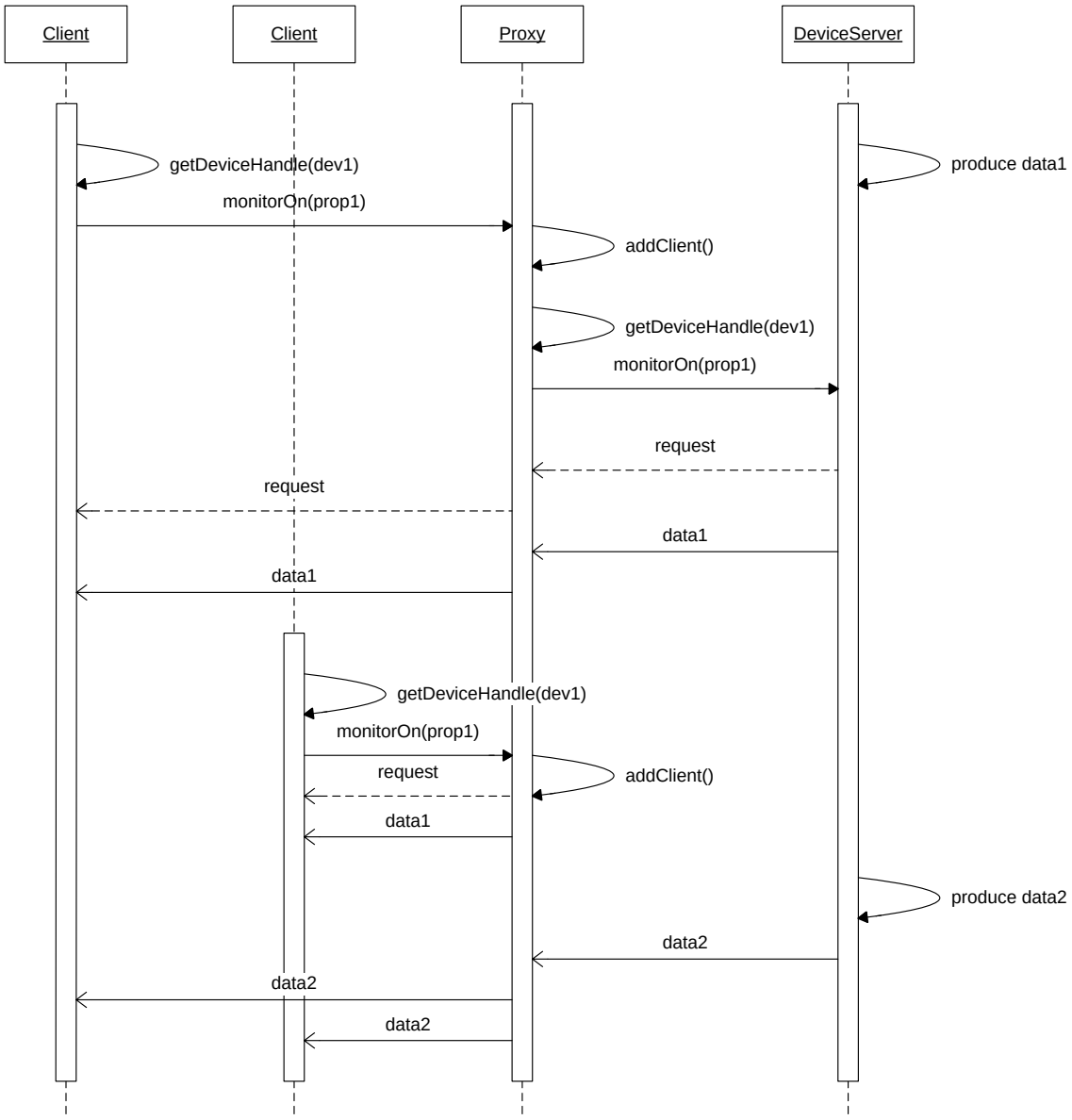


Figure 29: ManagerWithoutContext optimizes subscriptions to the same device-property pairs.

If there is a new client subscribing to the proxy to a pair that is already served by `ManagerWithoutContext`, the client handler is simply added to the list of listeners in an appropriate `HandlerWithoutContext` object. Next, the client is notified with a first-update and all consecutive ones.

The `monitorOn` call is presented in details on Figure 30. If the `ManagerWithoutContext` already has a connection established with an RDA server it just adds the new client listener to the list in an appropriate `HandlerWithoutContext` object. If the connection does not exist yet, the manager asks `getDeviceHandle` (which can perform extra optimization, see Section 5.4) for the address of the server responsible for the device. The address is obtained from Directory Service, and in the end used by the manager to establish the connection. As a result of creating the new connection, an `rdaRequest` object is received and saved for future use.

As long as a client who cancels a subscription is not the last one subscribed to a device-property pair, CMW-Proxy only removes its listener from an appropriate list of clients. In order to still be able to send updates to other clients it keeps connection with the FEC, and does not cancel it with `monitorOff` call. But if the client was the last one subscribed, apart

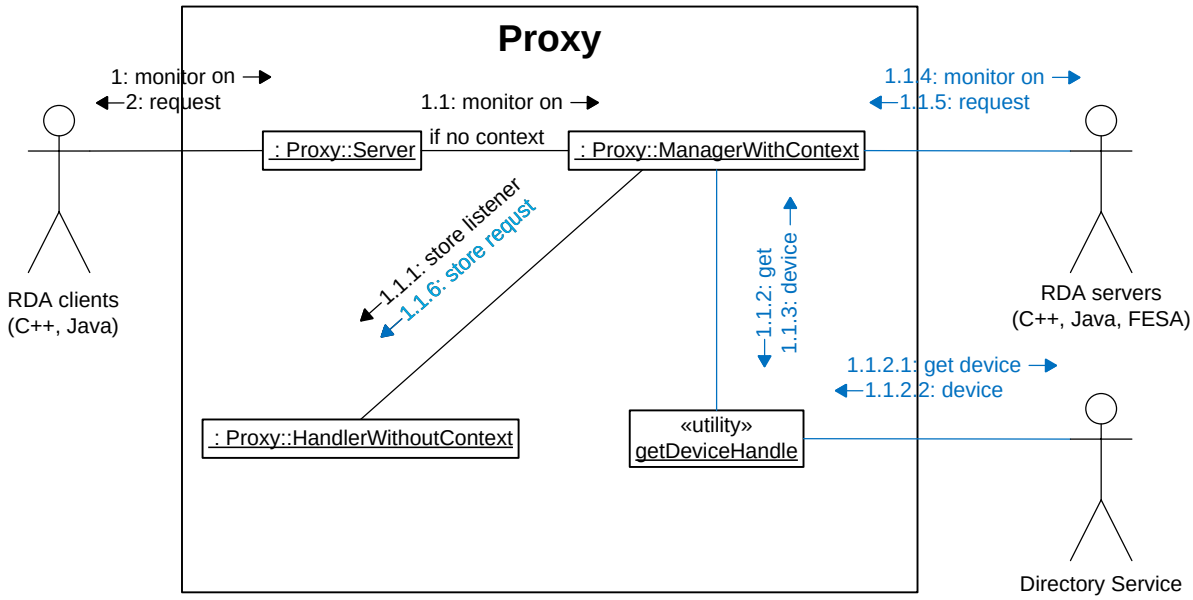


Figure 30: Details on monitorOn to the same device-property pairs. In blue the calls that are performed only for the first client subscription.

from deleting its handler the CMW-Proxy also cancels the connection with the FEC with a call to `monitorOff`. This behaviour is shown on Figure 31.

Figure 32 presents what specifically happens during a `monitorOff` call. If there are still other clients subscribed to device-property pair only client listener is erased. If the client is the last one, the `ManagerWithoutContext` calls also `monitorOff` on the destination server. To do that it obtains a device handle (in the same manner it did it in `monitorOn` call) and invokes `monitorOff` on that handle using `rdaRequest` object obtained when the `monitorOn` was performed.

5.4 Utilities

Apart from the main implementation classes, the CMW-Proxy project contains also a set of utilities. One of them is its own logging infrastructure.

A small `Log` class was implemented to support thread-safe yet almost non-blocking logging functionality. This property has been achieved thanks to the fact that for each logging a temporary object of the class is created. To its local buffer the log message is written. Because of that there is no need for synchronization with other log objects. When the log message is created it is written down by the class destructor which is called automatically and manages synchronization on its own. So, the user does not even have to flush the message. Everything is done automatically. An ease of use is provided by template overloaded bitwise left shift operator (`operator<<`) which makes using the logging object similar to using the stream classes from the standard C++ library.

Each log message starts with date and time of logging. Moreover, logging facility contains also a macro definition which creates a local `Log` object and adds to each message name of the file and line on which the logging was performed.

Short listings present the `Log` interface (see Listing 14), its basic usage (see Listing 15) and an exemplary output (see Listing 16).

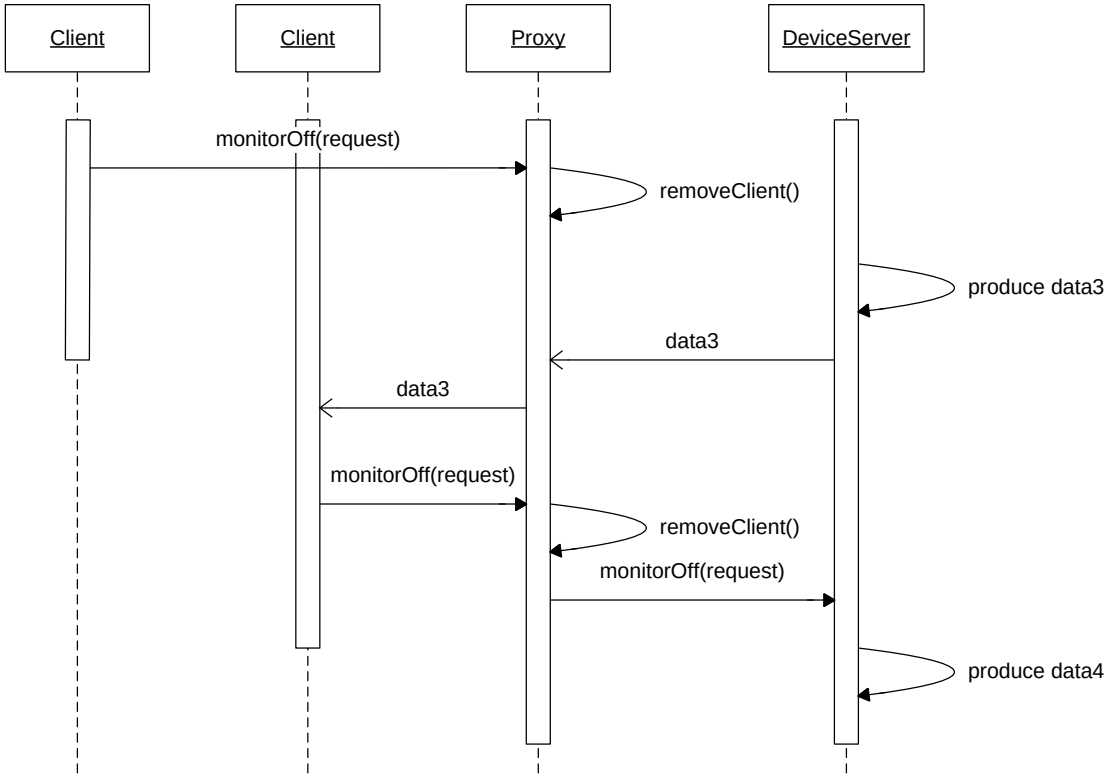


Figure 31: ManagerWithoutContext optimizes subscriptions to the same device-property pairs.

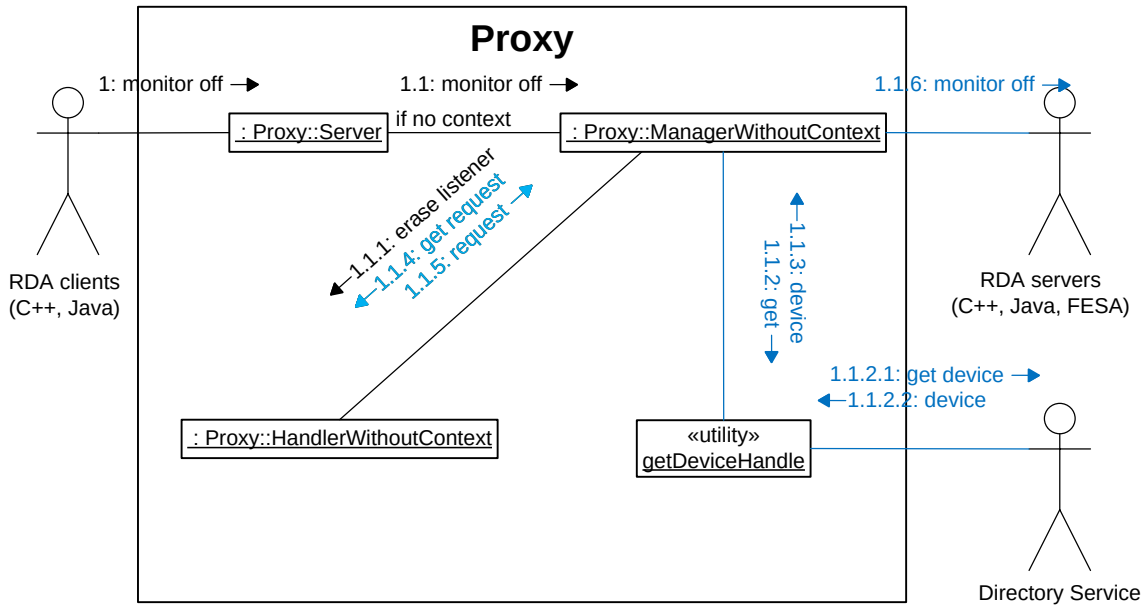


Figure 32: Details on monitorOff for clients subscribed to the same device-property pairs. In blue the calls that are performed only for the unsubscription of the last client.

Listing 14: Simplified Log interface.

```
/// @brief Logging class. Thread safe.
class Log
{
public:

    /// Log the occurrence date.
    Log( std::ostream & ostr );

    /// Flush the buffer with logs.
    ~Log();

    /// @brief Log the message/value. Do not flush it,
    /// it is going to be flushed by the class itself.
    template <typename T>
    Log & operator<<(T const & value);

private:

    /// Output stream.
    std::ostream & ostr_;

    /// Logging buffer.
    std::ostringstream buffer_;
};

#define logErr RecPlay::Log(std::cerr) << __FILE__ << ":" << __LINE__ << " "
```

Listing 15: Log ussage.

```
catch ( const rdaBadParameter & ex )
{
    logErr << " failed: set(" << iop.getDeviceName()
        << ", " << iop.getPropertyName() << ") " << ex;
    throw;
}
```

Listing 16: Log output.

```
2009/04/27 13:09:06 [1240830546:417582] Server.cpp:98  failed: set( someDevice,
    someProperty ) rdaIOError: [GW 0] Device 'someDevice' is not known by naming
    server
```

Another utility is `getDeviceHandle` function. Its role is to cache addresses of computers on which specific devices are running. If there is a request to the proxy to make a connection to a device first it looks up for it on the cache list. If it finds it there it simply returns the already possessed address. If it does not, it asks for it the Directory Service, stores it in the cache and returns the address. As a CMW-Proxy instance serves usually only a narrow group of devices such functionality has high hit rate. It shortens the proxy time response and reduces network traffic and number of calls received by Directory Service.

5.5 Security policy

In the virtual word of software, security is as important as it is in the real life. Without any security policy CMW middleware framework could not be safely used. Any client could perform any operation on any FEC. That would definitely lead to disastrous consequences. If

not because of some outside CERN hostile hacker attack, then by chance, by absent-mindedness of one of the casual users.

Fortunately the framework provides as its part the RBAC[33] library. Every operation (get, set or subscription demand) is subject to the authorization process and its execution can be denied in case of issuer having insufficient privileges.

Decision whether a particular operation is valid or not is dependent on a set of access rules. They are specified by an equipment specialist for every device, stored and managed centrally in the Controls database. The access rules are mirrored also in a tab-separated text file located in the Controls Network File System. Every server can read access rules (referred to as access map) for devices it is providing access to.

Access rules are parameterized using all factors related to the authorization process. More specifically, an equipment specialist has to specify the following fields to define an access rule:

- device class name,
- property name,
- device name,
- role name,
- application name,
- location name,
- accelerator mode,
- operation type (set, get, or subscribe).

The interactions between a client, the server and the RBAC server during an execution of a server access can be described in a few steps. First, upon start-up, the server reads its access map from a file. Consequently, it is ready to receive calls from client applications. To allow access to the server, a client must authenticate himself and obtain an RBAC token from the RBAC server. The token is transferred to the server in order to provide information needed for authorization. Now client access request can be sent to the server. To properly authorize a request, the server obtains current accelerator mode from the timing source (different rules may be in force at different stages of the run). Finally, authorization decision is taken and in case of access denial an RBAC exception is handled back to the client [56].

There are two modes into which CMW operations can be authorized: token per connection and token per operation. In the first case a token is transmitted to the server upon the connection establishment and is stored there. It is subsequently used to authorize every CMW operation. In the token per operation mode the RBAC token is sent encapsulated in the context of every CMW operation. This approach is more expensive to use than token per connection but allows a flexible switching between credentials used to authorize subsequent operations.

When connecting to a FEC, CMW-Proxy has to use RBAC as any other client. Following the security policies in this case is very important. To decrease the server load as much as possible, most commonly a token per connection method is used, and the proxy is authenticated and authorized only once at its startup. Besides, the RBAC role for the proxy usually authorizes it to perform all the calls on the server side, occasionally forbidding only the calls it definitely should not perform.

[33] RBAC (role-based access control) is an approach to restricting access to a system only to authorized users. Each user has assigned roles which specify what he is allowed to do in the system.

Of course if the things were left like that the system would be unsafe and vulnerable to attacks. To keep it safe the CMW-Proxy side takes on it part of the authorization process. Simply it uses whole access map defined previously for a FEC to which it gives access. Such solution brings no changes on the client side, and hardly any changes on the FEC side. It also generates no effort for anyone to develop new access rules to the proxy because they are copied from FEC ones. This is a simple and safe solution which additionally can take off some load from any destination server by taking over the authorization functionality.

5.6 Testing

CMW-Proxy has been carefully tested. It proved to perform properly all expected tasks. Tests consisted of various communication layouts: single client with a single server, many clients with a single server and many clients with many servers. All tests finished with positive results.

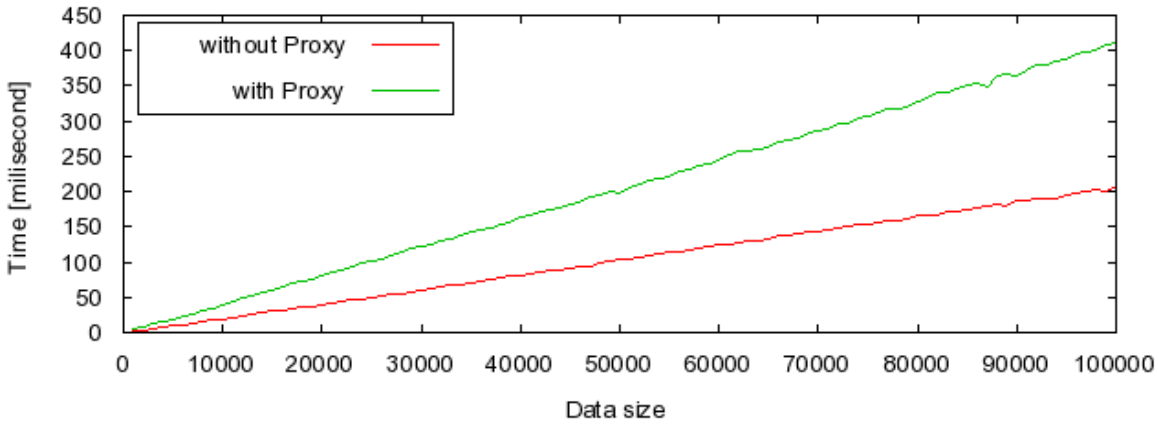


Figure 33: CMW-Proxy delay in relation to direct publication.

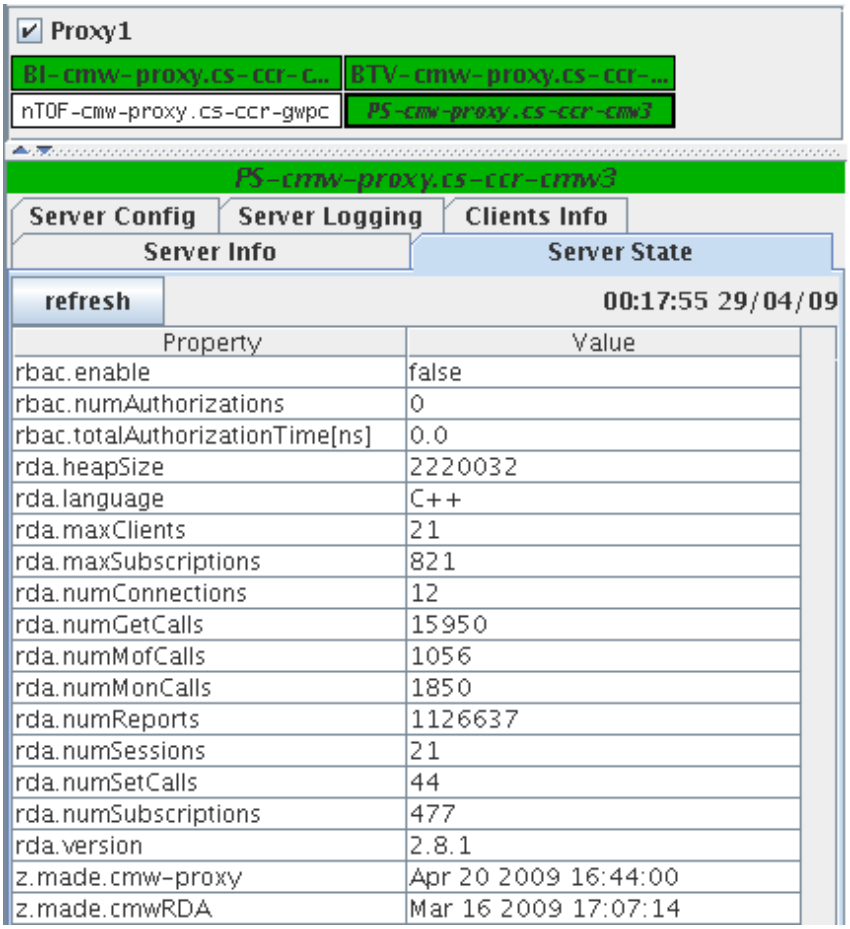
However, reliability tests revealed, that because of specific system and underlying libraries limits (CORBA), CMW-Proxy can break under heavy load. This is because of number of thread limits per process, which was estimated to be not much more than 300 on a typical nowadays Linux machine. With standard omniORB settings this limits the number of subscriptions to around 2000 (not a bad result achieved thanks to resources pooling).

A situation when CMW-Proxy resources are exhausted can happen, when many connections are made for a short period of time and after cancelation new ones are made on their place, while some resources are not yet released (they are cached for performance reasons). As tests showed, this can occasionally lead to program crash (probably memory corruption inside omniORB). Fortunately it does not happen too often. Also, because RDA implements auto-reconnection between clients and servers, it is not an important issue that a CMW-Proxy might lose its state, because it would regain it automatically on restart.

As tests show (see Fig. 33) in a simple "one client through CMW-Proxy to one server" communication, CMW-Proxy slows down the act of communication between a client and a server approximately two times. This is an expected result, and there is no place to optimize CMW-Proxy further, though in specific situations super linear effects may occur. They can be spotted most often when a server application running on a weak machine fills up network card buffers sending the same data to many clients. When CMW-Proxy is used to unload such an application, the data from it to the CMW-Proxy it sent only once, and thus the probability of filling up the buffers decreases. Then CMW-Proxy, running on stronger, bigger-buffered machine redistributes the data without buffer overflows on its site.

5.7 Usage

CMW-Proxy has been around, accompanying other CMW products for several weeks. So far, there were a few official instances running and serving real clients and unloading real FECs. Figure 34 shows a part of cmwAdmin application, written to monitor running servers. There are visible four proxies, of which three in green are running at the moment. For one of them, state of the server is presented. Among the others the number of clients and calls they made can be read, as well as number of subscriptions served by the CMW-Proxy at the very moment of watching (taking the screenshot).



Property	Value
rbac.enable	false
rbac.numAuthorizations	0
rbac.totalAuthorizationTime[ns]	0.0
rda.heapSize	2220032
rda.language	C++
rda.maxClients	21
rda.maxSubscriptions	821
rda.numConnections	12
rda.numGetCalls	15950
rda.numMofCalls	1056
rda.numMonCalls	1850
rda.numReports	1126637
rda.numSessions	21
rda.numSetCalls	44
rda.numSubscriptions	477
rda.version	2.8.1
z.made.cmw-proxy	Apr 20 2009 16:44:00
z.made.cmwRDA	Mar 16 2009 17:07:14

Figure 34: CMW-Proxy delay in relation to direct publication.

Proxies have proved to work properly and as recently reported by FEC specialists (people responsible for their implementation and maintenance) they had significantly helped with reducing load from FECs and had saved them from dangerous crashes encountered in the past. Because of that, the number of FEC developers interested in using the proxy started to grow. The fact that Proxy performs well, is accepted and used by more and more FEC specialists is very promising and gives an optimistic outlook for the future.

6 Conclusions

The thesis presents concepts and implementation of the middleware infrastructure for the CERN accelerator controls redeveloped and improved for the needs of the Large Hadron Collider operation.

A lot of effort was put to test and prove Control Middleware system functionality. Many tests were written, and thanks to them many bugs were found and eliminated. Yet, even though tests were prepared with uttermost care and scrutiny they did not cover all the possible execution paths and missed a few things. Only after long hours of debugging the errors were found, and when understood the missing tests were added. This only shows how a tough work it is to test a big library and guarantee its functionality.

Apart from functionality, the library was tested for its reliability and stability. To estimate efficiency of the library two other middleware systems (CORBA and Ice) were also tested. The results helped to draw a line between what can be expected from the RDA library in terms of performance, and what has to stay on the wish-list.

To support more complicated testing scenarios, especially during the operation of the machine, RecorderPlayer was written. Main purpose of the application is to allow post-mortem analysis and debugging, yet it can be used also in many other cases.

Finally, Proxy application was developed, tested and integrated with the system. Its main task is to take some work from the overloaded FEC servers and occasionally play a role of a gateway into the CERN Technical Network. The proxy proved to do its work well, therefore improving the Controls infrastructure and making its users happier.

The work performed by the author is presented in chapters 3, 4 and 5. It was performed during his 14 months stay at CERN as CERN Technical Student. In particular it includes the following: development and execution of the tests for the RDA library, further maintenance of the library as well as extending its functionality. The full extent of this work reaches outside the scope of the thesis and only relevant parts are presented here. In particular, the full development cycle (design, development, implementation and maintenance) of the CMW-RecorderPlayer and CMW-Proxy applications are described.

References

[1] CERN personnel statistics 2007. https://webh12.cern.ch/hr-info/stats/persstats/CERNPersonnelStatistics2007.pdf.

[2] LHC brochure (CERN faq, LHC the guide). http://cdsmedia.cern.ch/img/CERN-Brochure-2008-001-Eng.pdf.

[3] CERN Courier. Gigabits, the grid and the guinness book of records. http://cerncourier.com/cws/article/cern/29031.

[4] LHC website. http://public.web.cern.ch/public/en/LHC/LHC-en.html.

[5] The LHC experiments website. http://public.web.cern.ch/public/en/LHC/LHCExperiments-en.html.

[6] LEP website. http://public.web.cern.ch/public/en/Research/LEP-en.html.

[7] ATLAS experiment website. http://atlas.ch/.

[8] CMS experiment website. http://cms.cern.ch/.

[9] ALICE experiment website. http://aliceinfo.cern.ch/.

[10] LHCb experiment website. http://lhcb.web.cern.ch/.

[11] TOTEM experiment website. http://totem.web.cern.ch/.

[12] LHCf experiment website. http://public.web.cern.ch/public/en/LHC/LHCf-en.html.

[13] LHC design report volume I, chapter 14 - control system. https://edms.cern.ch/file/445863/5/Vol_1_Chapter_14.pdf.

[14] CMW website. http://cmw.web.cern.ch/CMW/.

[15] CERN P. Ninin et al. Industrial control systems for the technical infrastructure at CERN. http://www.aps.anl.gov/News/Conferences/1997/icalepcs/paper97/p133.pdf.

[16] LynxOS website. http://www.lynuxworks.com/.

[17] IEEE POSIX Certification Authority website. http://standards.ieee.org/regauth/posix/.

[18] Profibus and Profinet International (PI) website. http://www.profibus.com/pb/.

[19] WorldFIP website. http://www.worldfip.org/.

[20] Cz. Fluder et al. Experience in configuration, implementation and commissioning of a large scale control system (based on LHC cryogenic distribution control system). International Carpathian Control Conference ICCC 2008.

[21] Tech-FAQ. RAID. http://www.tech-faq.com/raid.shtml.

[22] Tech-FAQ. SCADA. http://www.tech-faq.com/scada.shtml.

[23] ISOLDE website. http://isolde.web.cern.ch/.

[24] AD website. http://public.web.cern.ch/public/en/research/AD-en.html.

[25] CERN B. Frammery. ICALEPS 2005, the LHC control system. http://icalepcs2005.web.cern.ch/.

[26] CORBA standard website. http://www.corba.org/.

[27] CERN N. Trofimov, V. Baggiolini et al. Remote device access in the new CERN accelerator controls middleware. http://arxiv.org/PS_cache/physics/pdf/0111/0111166v1.pdf.

[28] JMS website. http://java.sun.com/products/jms/.

[29] J2EE website. http://java.sun.com/javaee/.

[30] W3C XML website. http://www.w3.org/XML/.

[31] W3C XML Schema website. http://www.w3.org/XML/Schema.

[32] W3C XSL website. http://www.w3.org/Style/XSL/.

[33] CERN Michel Arruat et al. Front-end software architecture. http://ics-web4.sns.ornl.gov/icalepcs07/WOPA04/WOPA04.PDF.

[34] GSI T. Hoffmann. FESA – the front-end software architecture at FAIR. http://accelconf.web.cern.ch/AccelConf/pc08/papers/wep007.pdf.

[35] CERN S. Deghaye, L. Bojtar et al. OASIS: status report. http://project-oasis.web.cern.ch/project-oasis/pdfdocs/ICALEPCS%2005%20-%20Oasis%20Status%20report.pdf.

[36] Spring framework website. http://www.springsource.org/.

[37] CERN G. Kruk, S. Deghaye et al. LHC software architecture [LSA] evolution toward LHC beam commissioning. http://ics-web4.sns.ornl.gov/icalepcs07/WOPA03/WOPA03.PDF.

[38] JAPC website. http://wikis.cern.ch/display/JAPC/Home.

[39] RMI guide. http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html.

[40] Home page of UNICOS project. http://ab-project-unicos.web.cern.ch/ab-project-unicos/.

[41] SGI Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/.

[42] G. J. Myers. *The Art of Software Testing.* Wiley, 1979.

[43] C. Kaner, J. Bach. *Lessons Learned in Software Testing.* Wiley, 2002.

[44] M. Roper, N. Parrington. *Understanding Software Testing.* Wiley, 1989.

[45] Free Software Foundation. Gcc website. http://gcc.gnu.org/.

[46] The C++ Resources Network. cplusplus.com. http://www.cplusplus.com/.

[47] M. Feathers. Cpp unit website. http://sourceforge.net/projects/cppunit/.

[48] Unit++ website. http://unitpp.sourceforge.net/.

[49] G. Rozental. Boost test library website. http://www.boost.org/doc/libs/1_38_0/libs/test/doc/html/index.html.

[50] J. Kulaviak, A. Glew. Cpp unit lite unofficial website. http://c2.com/cgi/wiki?CppUnitLite.

[51] Valgrind website. http://valgrind.org/.

[52] Institute of Electrical and Electronics Engineers website. http://www.ieee.org/.

[53] ZeroC Ice website. http://www.zeroc.com/.

[54] SLC5 website. http://linux.web.cern.ch/linux/scientific5/.

[55] SLC4 website. http://linux.web.cern.ch/linux/scientific4/.

[56] S. Gysin (Fermilab) K. Kostro (CERN), W. Gajewski (CERN). Role-based authorization in equipment access at CERN. http://accelconf.web.cern.ch/AccelConf/ica07/PAPERS/WPPB08.PDF.