FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Upgrading and Enhancing the LHC Logging System

Daniel Dinis Teixeira

Report of project
Master in Informatics and Computing Engineering
Supervisors: Jaime Villate, Ronny Billen and Chris Roderick
July, 2008

Upgrading and Enhancing the LHC Logging System

Daniel Dinis Teixeira

Report of project
Master in Informatics and Computing Engineering

Approved in oral examination by the committee:
Chair: Pedro Alexandre Ferreira do Souto

____

External Examiner: António Amorim

____

Internal Examiner: Jaime Villate

____

Abstract

At CERN, Switzerland, the Large Hadron Collider (LHC), the world's most powerful particle accelerator ever built, is about to start operational service for the first time. This complex machine of 27km circumference, located at 100m underground is subjected to many physical properties such as temperatures, electrical currents, vacuum pressures, etc...

The Logging System was launched with the aim to track variations of these properties over time. This mission-critical system has been in operation since 2003: supporting the commissioning of the LHC sub-systems. It currently receives and stores more than 10´000 time series data per second coming from a wide range of heterogeneous clients. With the advancement of technology, the Data Loading element of the Logging System was faced with becoming obsolete.

The work presented in this document involved the analysis and implementation of solutions in order to ensure the maintenance of the security and manageability of the Logging System Data Loading – using the latest available software platform.

In addition a new aspect of the Logging System Data Loading was designed and fully implemented: to instrument and monitor Data Loading activity in an efficient, complete and powerful manner. With the instrumentation in place, effective analysis could be performed; resulting in correlations being identified between various aspects of the Data Loading activity/configurations and system performance. In turn, optimal configurations could be established to improve system performance.

Resumo

No CERN, na Suíça, o *Large Hadron Collider* (LHC), o mais poderoso acelerador de partículas jamais construído, está no ponto eminente de iniciar as suas actividades operacionais pela primeira vez. Este instrumento complexo, com 27km de circunferência, encontra-se a 100 m de profundidade e está sujeito a imensas propriedades físicas, tais como: temperaturas, correntes eléctricas, vácuo, etc...

O *Logging System*, é um projecto que foi criado com o objectivo de registar qualquer variações dessas propriedades no decorrer do tempo. Este projecto de missão crítica foi lançado em 2003. Neste momento, o sistema recebe cerca de 10 000 registos por segundo vindos de uma vasta quantidade de clientes heterogéneos dispersos à volta do LHC. Com os avanços tecnológicos alguns módulos do sistema deixaram de funcionar nas novas plataformas.

O trabalho desenvolvido tem por objectivo, analisar, propor e implementar soluções para manter a segurança e capacidade de gestão do *Logging System Data Loading* nas últimas plataformas disponíveis.

Para além disso, uma sofisticada ferramenta de instrumentação baseada em três níveis de abstracção foi desenvolvida especificamente para o sistema. Esta ferramenta permite monitorizar, correlacionar e analisar várias variáveis para determinar o desempenho do sistema e o comportamento dos clientes. Para finalizar, estatísticas produzidas pela instrumentação puderam ser analisadas e tornaram possível definir ajustamentos no sistema e no comportamento dos clientes para um melhor desempenho global.

Acknowledgements

The achievements and results obtained in this project were only possible thanks to the successive help of my colleagues at CERN, my home university in Porto, my family and friends.

I would like to express my gratitude to Ronny Billen, whose expertise, understanding and cheerfulness made my period of stay at CERN as much constructive as possible. Secondly, I very special thanks goes to Chris Roderick for his invaluable guidance throughout this work. I must also acknowledge all my other colleagues, Julien Mariethoz, Zornitsa Zaharieva, Manuel Marquez, Pascal Le Roux and Maciej Peryt for the coffee breaks.

A warm thanks goes to Jaime Villate and Eugénio Oliveira who believed in me and made my dream come true.

I would also like to thank my family for the support they gave me during all my life and particularly during this period.

Finally, I must acknowledge all my friends and especially Fernando Pereira, who cooked the most original cake that I have ever ate for my birthday.

Contents

1 Introduction **1**

1.1 CERN 1

1.2 LHC 3

1.2.1 Experiments 5

1.2.2 The aim of the LHC 6

1.3 The LHC Logging Project 6

1.3.1 Clients 7

1.3.2 Data loading 7

1.3.3 Data Extraction 8

1.4 Chapter overview 8

2 Logging System Analysis and Objectives **9**

2.1 An extensively used system 9

2.1.1 Roles 10

2.1.2 Environments 10

2.1.3 Challenge 11

2.2 Objectives 11

2.2.1 Security 11

2.2.2 Manageabilty 12

2.2.3 Diagnostics ability 12

2.2.3.1 System performance 12

2.2.3.2 Clients behaviour 12

2.3 Chapter overview 13

3 Security and Manageability evolution **15**

3.1 Core Technology 15

3.1.1 ORACLE 15

3.1.2 Java 15

3.1.3 Mixing both 16

3.1.4 Logging Project context 16

3.2 Security . . . 18
3.2.1 JAAS . . . 19
3.2.2 Logging Project security Mechanism . . . 19
3.3 Management . . . 20
3.3.1 JMX . . . 21
3.3.1.1 Components . . . 21
3.3.1.2 Types . . . 22
3.3.1.3 Architecture . . . 22
3.3.1.4 Support . . . 23
3.3.2 Managing the Logging System . . . 24
3.4 Chapter overview . . . 24

4 Security 25
4.1 LoginModule solution . . . 25
4.2 Declarative Security configuration . . . 26
4.3 Discussion . . . 28

5 Manageability 29
5.1 Logging Data Loading application management . . . 29
5.1.1 JMX and OEM . . . 30
5.1.2 Parameters classification . . . 30
5.2 Architecture . . . 31
5.3 Design and Implementation . . . 32
5.3.1 Standard MBeans . . . 32
5.3.2 Dynamic registration . . . 33
5.3.3 Validations . . . 34
5.4 OEM Console interaction . . . 35
5.5 Additional MBeans for configuration . . . 35
5.6 Summary . . . 36

6 Instrumentation 37
6.1 Specification . . . 37
6.2 Architecture . . . 39
6.3 Design and Implementation . . . 42
6.3.1 File . . . 42
6.3.2 Memory . . . 43
6.3.2.1 Data structure . . . 44
6.3.2.2 Properties and statistics measurements . . . 44
6.3.2.3 Update statistics . . . 45

6.3.2.4 Daily Summaries 47
6.3.2.5 Management 48
6.3.3 Logging Database 50
6.3.3.1 Considerations 50
6.3.3.2 Instrumentation table 51
6.3.3.3 Instrumentation view 53
6.3.3.4 Job 53
6.4 Testing 53
6.5 Interfaces and direct results 54
6.5.1 Specific requests 54
6.5.2 JMX browser in OEM 54
6.5.3 TIMBER and historical data 56
6.6 Chapter Overview 57

7 System analyses for performance tuning 59
7.1 Operational times 59
7.2 Data Distribution 60
7.2.1 Checking Time 60
7.2.2 Writing Time 61
7.2.3 Interpretation 62
7.3 Duplicate Data 63
7.4 Real case tuning 66
7.5 Summary 67

8 Conclusion and outlook 69

Bibliography 72

Appendix 77

A XML File 77

B Prototypes 79
B.1 Custom web application 79
B.2 JConsole 80

C Logging Database 81
C.1 Schema of the Logging database 81
C.2 Variables mapping 83
C.3 SQL scripts 85

C.3.1 View 85
C.3.2 Procedure 86

D MBeans Information **89**
D.1 Attributes 89
D.2 Methods 90
D.3 Middle class 90

List of Figures

1.1 20 member states 2
1.2 First Server 3
1.3 LHC 4
1.4 ATLAS Experiment 5
1.5 Logging Architecture 8

2.1 Roles 10

3.1 LoginContext 19
3.2 JMX Architecture 23

4.1 Users data model 26

5.1 JMX integration in the Logging System 32
5.2 MBeans Design 33
5.3 JMX browser interface in OEM 35

6.1 File process 38
6.2 LHC Cooling process 39
6.3 3 levels of instrumentation 40
6.4 Data Structure 44
6.5 Statistics, Property and Measurements 46
6.6 MBeans types 48
6.7 Table synonym 52
6.8 Database problem? 54
6.9 Interaction 55
6.10 Graphs in OEM 55
6.11 TIMBER interface 56

7.1 Checking time vs Number of variables 61
7.2 Writing time vs Number of records 62
7.3 Ratio Variables/Records vs Writing Time 63

7.4 **INSERT vs MERGE statement** 64
7.5 **File Size vs Total Time** 65
7.6 **Sensitivity analysis** 66

B.1 **Custom web application** 79
B.2 **JConsole plugin** 80

C.1 **Loggin data model** 81

List of Tables

5.1 Management requirements 30

7.1 Operational Times 59
7.2 Variables and Records 60

C.2 Counters - Logging variables suffix 1 83
C.4 Valiation and Parsing time - Logging variables suffix 83
C.6 Checking, Writing and Total time - Logging variables suffix . . . 84
C.7 File and Variables - Logging variables suffix 84
C.8 Records - Logging variables suffix 85

Chapter 1

Introduction

CERN [1] has always been an ambitious organization, pushing knowledge always further and technology beyond the current limits. Synchrocyclotron (SC), built in 1957, was the first accelerator dedicated to particle and nuclear physics experiments. Two years later, CERN built a more powerful accelerator, the Proton Synchrotron (PS) which focused mainly in particle physics experiments. Next came the Super Proton Synchrotron (SPS) a much larger one with almost 7km of circumference. Then, in the 80's, CERN constructed the biggest accelerator ever built, the Large Electron Positron with 27km of circumference. This one was operated as from 1989 and was finally dismantled in the year 2000 to made room to the current more powerful accelerator of the Earth, the Large Hadron Collider (LHC). The level of energy of the particles will be approximately one million times more important in the LHC than in the first accelerator. CERN has always been aware of the importance of the technology and tried as much as possible to anticipate it. However keeping up with the best techniques is not an easy task and can turn out to be a struggled operation. The same happens in software upgrades. Many times migrations imply costly operations and can turn modules already developed into an incompatible or deprecated state. This thesis discusses how an upgrade can affect a system and presents important enhancements that couldn't have been reached before.

1.1 CERN

CERN is the European Organization for Nuclear Research, it is situated near the city of Geneva, Switzerland. The acronym is derived from the French and stands for "Conseil Européen pour la Recherche Nucléaire" (European Council for Nuclear Research). It was found in 1952 as a council, in 1954 the council was abolished and the Organization took place, although the acronym CERN was kept. At that time the research made at CERN were mainly related to atoms, that's why the name "Nuclear". After the construction of the PS accelerator, CERN started mainly to study "particles". Because of this, nowadays

the laboratory operated by CERN is commonly referred to as the European Laboratory for Particle Physics. It was founded by 12 countries, but today it is composed by 20 member states. The current Member States are: Austria, Belgium, Bulgaria, the Czech Republic, Denmark, Finland, France, Germany, Greece, Hungary, Italy, the Netherlands, Norway, Poland, Portugal, the Slovak Republic, Spain, Sweden, Switzerland and the United Kingdom. In addition to member states there are other countries involved in many ways. In Figure 1.1 on page 2 the member states are showed.

Figure 1.1: 20 member states

At present, CERN's main area research is focused on high energy particle physics. They study the fundamental constituents of matter and the forces acting between them. Physics research involves many engineers and scientists in numerous areas, inevitably the research goes much deeper than simply physics. Thanks to the research in informatics, CERN gave birth to the World Wide Web. In fact, “info.cern.ch” was the address of the world's first-ever web site and web server, running on a computer at CERN. The first web page address was:

```
http://info.cern.ch/hypertext/WWW/TheProject.html
```

This page was centered on information regarding the WWW project. It is still possible to see the first server in the museum at CERN with an interesting advice saying: "This machine is a server. DO NOT POWER IT DOWN!!". The Figure 1.2 on page 3 shows the historical advice.

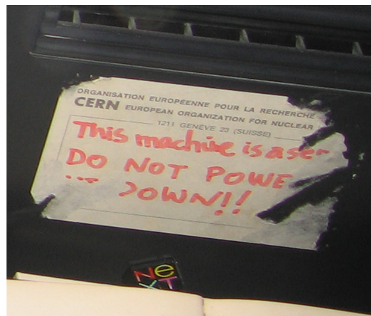

Figure 1.2: First Server

It is also worth mentioning that several scientists were distinguished with the Nobel Prize in physics. In 1984, Carlo Rubbia and Simon Van der Meer received the Nobel Prize in physics for "their decisive contributions to the large project which led to the discovery of the field particles W and Z, communicators of the weak interaction." Eight years later, Georges Charpak received the physics Nobel for "his invention and development of particle detectors, in particular the multiwire proportional chamber, a breakthrough in the technique for exploring the innermost parts of matter."

CERN is structured in departments, each one containing groups composed by sections. The work involved was done in the Accelerator Beams department, Controls group and Data Management Section (AB/CO/DM).

1.2 LHC

The Large Hadron Collider (LHC) [2] is the biggest scientific experience ever attempted [3]. More than 10 thousands physicists and engineers from 85 countries have combined their knowledge to build this impressive machine. It is a huge machine with 27km of circumference that crosses the border between Switzerland and France. It is placed in a tunnel at a mean depth of 100 m. Due to geological considerations, the depth varies from 50 m (near the lake) and 175 m (near the Jura mountain range). It is interesting to know that the circumference of the machine is not absolutely constant. Indeed, the tides have not only an effect on the oceans and sees, but also on the earth itself. They cause the level of water on the edge of the sea to rise and fall with a cycle of some 12 hours. The ground is also subject to the effect of lunar attraction because the rocks that make it up are flexible. At the new Moon and when the Moon is full, the Earth's crust rises by some 25 cm in the Geneva area under the effect of these 'ground tides'. This movement causes a variation of 1 mm in the circumference of the LHC and this produces changes in beam energy. Thus, physicists must take the Moon into account in their measurements and calculations of particle energy.

Two beams of subatomic particles called hadrons will travel in opposite directions

inside an ultrahigh vacuum tube, gaining energy with every lap. At the highest level of energy the two counter-rotating beams will be smashed together. The LHC should be able to recreate the conditions just after the Big Bang, this means 13.7 billion years ago. At the top speed, 0.999999991 times the speed of light, a beam of protons will make 11 245 loops around the LHC every second. If it circulates for 10 hours, it would travel more that 10 billion kilometers (enough to get to the planet Neptune and come back again). The LHC accelerator will be used by numerous physicists from all over the world to analyze the particles created in the collisions using special detectors in several dedicated experiments. In Figure 1.3 on page 4 a diagram of the LHC is exposed.

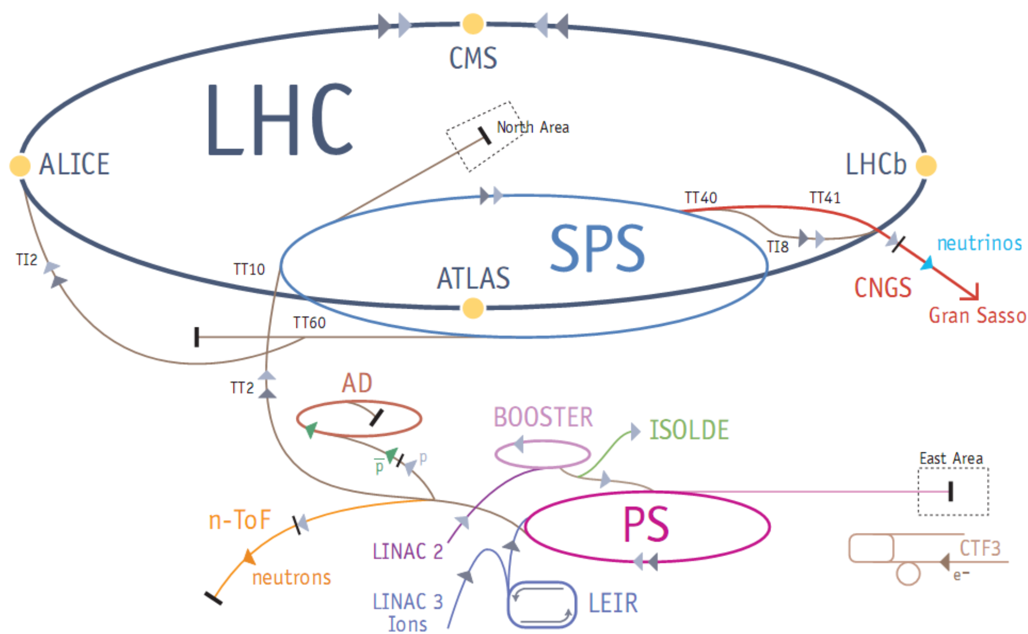

Figure 1.3: LHC

What makes the LHC so interesting compared to other particle accelerators is the fact that it uses a technique never experienced in other accelerators before: cryogeny. In order to create a super conductor circuit, the superconducting-magnets need to operate at a temperature of 271.3 ºC below 0 ºC (1.9 K). The LHC dipoles use niobium-titanium cables, which become superconducting below a temperature of 10 K (–263.2°C), that is, they conduct electricity without resistance. This permits to save a lot of energy for such a powerful system. If the magnets were normal and therefore introduced some resistance, the amount of energy required to accelerate the particles would be much higher. Anyhow, the LHC consumes more or less the same amount of electricity as the city of Geneva (120 MW). Assuming an average of 270 working days for the accelerator (the machine will not work in the winter period), the estimated yearly energy consumption costs of the LHC in 2009 is about 19 million Euros. To reach these extreme temperatures, the magnets will

use super fluid helium at 1.9 K. Nevertheless, the tunnel temperature remains at some 18ºC. In order to maintain the almost 300ºC temperature difference between the machine and its surroundings, vacuum is also used for isolation. To protect it against the boiling temperatures that human beings are used to (around 18º), another vacuum system was designed between the magnets and the exterior. In fact, the LHC is the largest cryogenic system in the world with more than 100 tons of liquid helium. That's why CERN people like to refer to their institute as "the coolest place in the Universe".

1.2.1 Experiments

For most of the ring, the beams travel in two separate vacuum pipes, but at four points they collide at the center of the main experiments [4]: ALICE (A Large Ion Collider Experiment) , ATLAS (A Toroidal LHC ApparatuS) , CMS (the Compact Muon Solenoid), and LHCb (the Large Hadron Collider beauty). The experiments' detectors will watch carefully as the energy of colliding protons transforms quickly into a vast number of exotic particles. Two other experiments, but with a smaller dimension are the LHCf (the Large Hadron Collider forward) and TOTEM (TOTal Elastic and diffractive cross section Measurement) experiments. ALICE. ALICE, ATLAS, CMS and LHCb are installed in four huge underground caverns built around the four collision points of the LHC beams. TOTEM will be installed close to the CMS interaction point and LHCf will be installed near ATLAS. Each one has different proposes or uses different techniques. For instance, CMS and ATLAS have the same goals, however they use different technical solutions and design. In Figure 1.4 on page 5 it is possible to compare the dimension of ATLAS with an human.

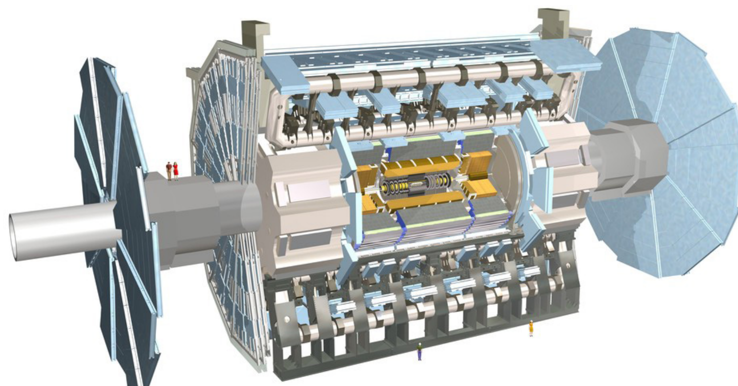

Figure 1.4: ATLAS Experiment

Each beam will consist of nearly 3000 bunches of particles and each bunch will contain as many as 100 billion particles. The particles are so tiny that the chance of any two colliding is very small. When the bunches cross, there will be only about 20 collisions between 200 billion particles. Bunches will cross on average about 30 million times per

second, so the LHC will generate up to 600 million particle collisions per second. The data recorded by the experiments will be enough to fill around 100 000 DVDs every year.

1.2.2 The aim of the LHC

The buzz began as long as the equation $E = mc^2$ exists. In 1905, Einstein defined how the energy was proportional to its mass, however one century later on physicists didn't still understand what gives the matter mass. The current theory about the elementary particles and the forces that act between them is called the "Standard Model". Experiments such as the LHC aim to demonstrate the predictions and thus confirm the theory (or the contrary). However, the Standard Model does not explain the origin of mass. The current aim is to demonstrate the existence of the "Higgs boson", the particle that is supposed to be responsible for the mass of all particles. If this particle actually exists, the LHC should detect it. Other questions such as: "What the invisible 96% of the Universe is made of?" "Why nature prefers matter to antimatter and how matter evolved from the first instants of the Universe's existence?" might be answered. The LHC is making the final preparations before embarking on a new era of discovery at the high energy frontier. Not only physicists are involved in these discoveries but also engineers, scientists and whoever sensitive to the couple of words "Big Bang".

1.3 The LHC Logging Project

The LHC is a huge machine with an unimaginable complexity that needs to be carefully monitored and controlled. Each of the subsystems has its own data acquisition system to track parameter values over time. Hundred-thousands of signals coming from equipment surveillance and beam observation need to be "logged" in a central database to carefully examine the behaviour of the machine. Additionally, end-users should be able, through this database, to correlate heterogeneous information, visualize and extract any data, compare over time, examine trends, find patterns and discover relationships between apparently unlinked parameters. In this way, the LHC Logging project [5] was launched, with the objective to log heterogeneous time series data [6], including: Cryogenics temperatures, magnetic field strengths, power dissipation, vacuum pressures, beam intensities, positions... The logging project started in 2001 and the first operational implementation was used in autumn 2003. The importance of such a logging system was already demonstrated on the previous big accelerator LEP. At that time, unexpected influences such as the tides due to the moon and the TGV passing in Geneva that influenced the behaviour of the LHC could be confirmed with the Logging System. The LHC Logging System was built and it is currently maintained by the data management section, which belongs to the control group within the accelerator and beams department.

1.3.1 Clients

For industrial services such as cooling, ventilation, vacuum, cryogenics, Supervisory Control And Data Acquisition (SCADA) systems built on top of PLCs are used to monitor part of the data coming from the LHC. The choice of the SCADA ended up in PVSS, which is developed by ETM [7] (an Austrian company). Clients have built application based on this system to send their data to the Logging Database. The most important part of the data (more than 90%) logged in the Logging Database comes from these clients. PVSS is able to filter some data and then only data of interest is sent to the Logging Database. A second category of clients are mostly beam-related instrumentation for which the data acquisition passes through custom home-made applications without local persistence. These clients, who cannot filter their data, have to send their data to another database, the Measurement Database for data persistence. Since the data is unfiltered it is sent at a higher rate with raw time series data measured at up to 2Hz. This database will filter interesting data and send it to the Logging Database, where it will be stored for the long term. Whereas the Measurement Database stores data up to 7 days, the Logging Database must do it for the LHC's life time, this means at least 20 years. The data coming from the Measurement represents currently almost 10 % of the Logging data. Additionally, the Measurement Database is also able to create some statistics about the performance of the accelerator. Finally the rest of the data (less than 1%), comes from the Technical Services department, which is responsible for systems such as electricity and water.

1.3.2 Data loading

The Measurement Service as well as the Technical Services, which in both cases store the data in an ORACLE database, write into the Logging Database using a PL/SQL API. As above mentioned, the most significant part of the data comes from PVSS, which loads the records in a different way. Basically, those systems send the data in an XML format to a servlet over HTTP. The XML file is then parsed and converted into Java objects to be written into the Logging Database via JDBC. Thousands of records are contained in these files, so it is imperative to have the best performance as possible. It is difficult to estimate the final throughput, but currently there are close to 1 billion records being sent per day via the XML path.The application is running in several servers. This is not only due to availability reasons (server crash), but also to ensure scalability and load balancing between different clients (some clients can be heavier than others). The clients of the LHC Logging System are typically related directly to the LHC machine like for example vacuum, cryogenics, general services, powering, beam systems... In addition, the very important cryogenics data of the experiments Atlas, LHCb, CMS experiments are also major clients. The Figure 1.5 on page 8 tries to shematize the Logging architecture.

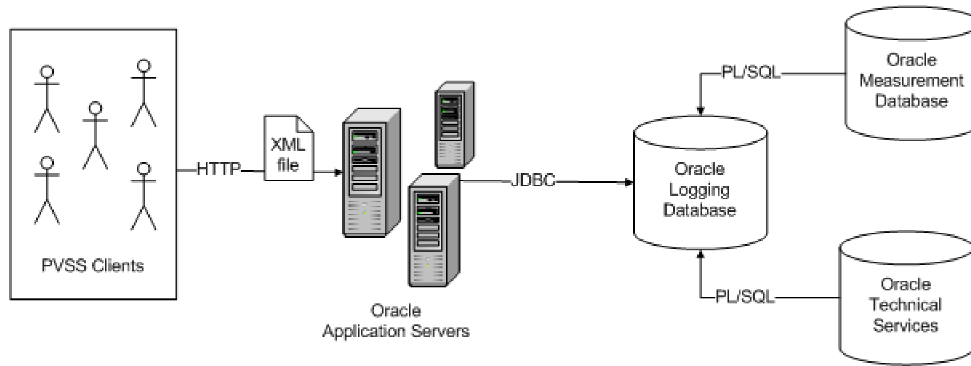

Figure 1.5: Logging Architecture

Clients are firstly submitted to a test environment, where their data loading methods need to be validated. Once a client's logging activities have been successfully validated, the client can start to log data in the production environment. Data logged in the Production environment is guaranteed to be available for at least the lifetime of the LHC machine. The validation tests will be explained in the next chapter.

1.3.3 Data Extraction

In consequence of the gigantic amount of data and the complexity of the database, a dedicated Java tool was built to extract the data in a user-friendly and efficient way. This GUI tool, baptized TIMBER, makes easy to select variables of interest and analyse them in many ways such as statistics, charts, etc... Additionally, for specialized systems and applications that need to directly access logged data, Java and PL/SQL data extraction APIs are provided.

1.4 Chapter overview

In this chapter, a top down approach has been presented, beginning with the presentation of the CERN Organization, passing through the Large Hadron Collider and finalizing with the Logging Project itself. Chapter 2 looks into the extensive use of the system and provides perspectives about the migration issues and benefits. Chapter 3 offers a look at what kind of technologies and techniques can be used in the migration process. Chapter 4 proposes the solution to maintain a secure system. Chapter 5 looks into the manageability and describes a solution. Chapter 5 focuses on the new aspect of the Logging System, i.e., the instrumentation. Chapter 6 exposes some analyses, results and enhancements thanks to the instrumentation aspect. Finally, chapter 7 is about the conclusion and outlook.

Chapter 2

Logging System Analysis and Objectives

2.1 An extensively used system

The Logging System was designed in a very flexible and customizable way, trying to satisfy a wide range of heterogeneous clients. The data sent by the clients is basically a measured value at a moment in time for a determined signal. The timestamp can be very precise for certain clients, reaching the nanoseconds, while for others it doesn't need to be so accurate. The time used is always written in Coordinated Universal Time (UTC). According to the requirements of the clients, the measured value could be generalized into four different data types. The following data types are supported:

- **NUMERIC** — numeric values: floats, integers, etc...
- **VECTORNUMERIC** — array of NUMERIC
- **NUMERICSTATUS** — NUMERIC with some status information
- **STRING** — textual alphanumeric character strings

The signals can be related to numerous systems like cryogenic systems, power consumption, vacuum... A prerequisite to log data in the system is to register a Logging variable giving it a name and provide its so-called meta-data. A naming convention was established to assure the quality of the Logging variable names. The client needs to provide information concerning the precision of the timestamps, the description and unit of the variable.... Once this step is accomplished, the client can, actually proceed to load time series data into the database, providing in each request the name of the variable with a measured value for a given timestamp. Clients should never send the same time series data (same timestamp) for a single variable, if this happens it means that duplicate data is being sent and that the client application is badly configured.

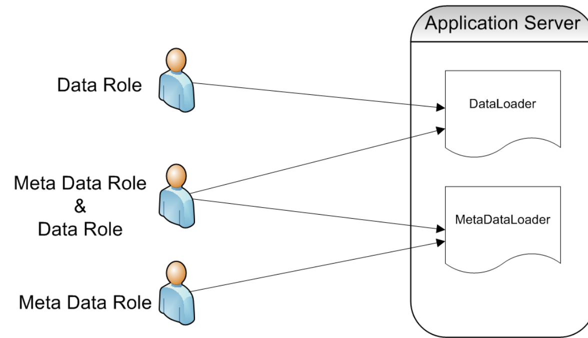

Figure 2.1: Roles

2.1.1 Roles

According to the two aforementioned steps , two different roles can be given to clients: the registration of the variable and the loading of the data. A client, who wants to register a new variable, needs to have the meta-writer role, while the one who wants to load the data needs to have the data-writer role. There is one servlet used to register variables, while the other is used to load the data. In order to ensure the security of the system, the execution of the servlet can only be done by authenticated and authorized clients. The Figure 2.1 on page 10 shows clients with different roles and the resources they can access.

2.1.2 Environments

The data loading is mission-critical and therefore the system must be as reliable and efficient as possible in the production environment. To ensure this reliability and high-performance, new clients who use the Logging system for the first time must start out using a test environment, with dedicated applications and database account. These clients need to pass some validations in the test environment before they can start using the production environment. Once the clients have successfully passed validation, they are allowed to use the production environment. Once using the production database, long-term persistence of the clients data is guaranteed. The validation tests consist of the following:

- Verification of the use of official naming conventions to give appropriate names to logging channels.
- Verification of the use of appropriate logging data filtering (application of deadbands, and rounding).
- Verification of the correctness of data.
- Verification of the sending of logging data to the LHC Logging System in appropriate sized batches.

- Verification of the sending of logging data to the LHC Logging System at an appropriate frequency.

2.1.3 Challenge

As explained in 1.3.2, PVSS systems send their data in an XML file via HTTP. This approach allows the Logging System to receive information from many different clients, which can be related to some properties of the LHC tunnel itself or experiments. Victim of its success, the Logging System is being used by an increasing number of clients. At the moment, the total amount of data logged is anticipated to be 10 TB per year, however this value may still increase due to new client logging requests. It is hard to predict but in 20 years the amount of data will be probably more than 200TB. A huge challenge is underway, and consists not only to store all that data, but also to perform efficient data extraction during the next 20 years. To face this challenge, the data management section is always following new technology developments, and trying as much as possible to keep up to date with technology evolution. Upgrading to the latest stable technology brings the benefits of new features and performance, and greatly eases system maintenance - all vital for the Logging System.

2.2 Objectives

The upgrade of the technology is crucial for high performance and for the ease of maintenance over the next decades. However, the migration of Oracle Application Server (OAS [8]) and Oracle Enterprise Manager (OEM [9]) from $9i$ to $10g$ introduces some obsolete modules that need to be replaced. The present work expects to ensure that the service can run on the latest software platforms in a secure manner, taking into account technology evolution and maintaining existing administrative functionalities.

On the other side the migration by itself doesn't ensure the best performance of the system. In fact, the Logging System is extensively used and its activity is expected to grow further over the coming years. The system reached a state, whereby it is extremely crucial to know what is going on. The additional aim of this work is to build an instrumentation tool able to provide significant information to analyse system performance and clients behaviour, and indentify potential for performance enhancements.

2.2.1 Security

The Data Loading application is based on two different servlets. One to register new Logging variables where meta-information about the Logging variables is sent, and the other used to load the time series data corresponding to these Logging variables. Each one of these servlets can be executed only by authenticated and authorized clients.

In OAS 10*g*, the security module in use is deprecated [10]. The security module needs to be replaced in order to run the application securely on the latest available platform.

2.2.2 Manageabilty

The application is partially managed using *context servlet parameters*. In 9*i* these parameters can be changed on the fly using the OEM interface. This functionality is no longer available in the OEM 10*g* interface, so there is no possibility for managing the system using *context servlet parameters* following the migration from OAS 9*i* to OAS 10*g*. The next objective consists of maintaining these administrative functionalities taking into consideration that the OEM interface is actually the only interface used for the management of the Data Loading application.

2.2.3 Diagnostics ability

The Logging System is extensively used and its activity is expected to grow further over the next years. The system reached a state that it is extremely hard to know what is going on. The additional aim of this work is to build a diagnostic tool to help analyse system performance and clients behaviour.

2.2.3.1 System performance

The only way to see the performance of the Data Loading system is by analysing the throughput (data volumes versus processing time), and by analysing the CPU and memory usage of the container/environment. Some debug information related to the time spent on a few operations is printed in a file. The tool used to print such information is powered by log4j [11], a project of the Apache Software Foundation [12]. In this debug file not only diagnostic information is printed but also any kind of error, such as connection failures, operations cancelled and other exceptions that can occur during the execution of the application. This file is not a good candidate to have a perception about what is going on. Firstly, too many concurrent threads are writing to the file at the same time, which means that the lines written in the file don't necessarily have a comprehensive structure. Secondly, the quantity of information is huge. In a few days, the log file can easily reach 1GB of plain text information. Finally, too much different information is mixed together in the file. All these reasons make the file very hard / impossible for a human to look into and understand how the overall system is performing.

2.2.3.2 Clients behaviour

Many clients running on different machines are using the Logging system. There is a need to know how they are behaving. It is extremely important to know many things about the

clients such as the activity of each client, the type of data they are sending, the frequency of requests of each clients, etc... Besides doing some queries in database, there is no way of having perceptions of the most active clients. Because the quantity of data is enormous and the database was not designed for diagnostics purposes, each audit query takes many hours to analyse records received during a 10 minutes sample period.

2.3 Chapter overview

This chapter looked into the Logging Project more deeply and presented the obvious need to migrate to the latest available software platforms to reach better performance and benefit from new features. Then it exposed the problems caused by the migration, with regards to security and manageability. In addition, it also showed the lack of a diagnostics tools in the Logging Data Loading System. The Logging System reached a state that a sophisticated instrumentation is crucial. It is more and more important because the system data rates are growing, as are the number of clients. It becomes extremely difficult to know how the system is performing and how the clients are behaving without a powerful diagnostic tool.

The next chapter speaks about the context of the Logging System as far as technology is concerned. Some different technologies are exposed to solve different issues and develop the instrumentation aspect.

Chapter 3

Security and Manageability evolution

This chapter starts to describe the core technologies and the main techniques present in the Logging System. After that, follows a section which speaks about security, focusing on the authentication and authorization of clients. Finally, the last section covers the manageability topic.

3.1 Core Technology

3.1.1 ORACLE

ORACLE [13] technology has been used at CERN for more than 25 years. It was one of the first companies to purchase the database license (Version 2) in 1983. The in-house experience gained over a quarter of century is invaluable. Not only mission-critical services related to accelerator's domains, but also administrative services have been developed using this technology. ORACLE has proven to be up to date, keeping up with the technology evolution, reliable and with an incomparable support. It is worth mentioning that the necessity for timestamping of data at nanosecond precision was a CERN requirement. ORACLE implemented a new data type called *timestamp* in its 9i database to satisfy this requirement. Considering an alternative Relational DataBase Management System in the development of the Logging System doesn't make sense after such accumulated investment and satisfaction.

3.1.2 Java

In the 80s and 90s, all the applications for the control of accelerators, were written in a procedural code in C. Just before getting into the new millennium several discussions were made in order to consider the use of an Object-Oriented Language (OOL). At that time,

C++ and Java [14] were emerging and the choice ended up with the Java Technology. At present, the Controls group only recommends writing new applications for accelerator control using the Java technology. The main factor for this choice was because of its facilitated usability and maintenance.

Traditionally people speculate that Java is not as good as C++, because the fact of running code in a JVM loses much performance. This is a fact when the Java code runs in a JVM and no optimization techniques are used to obtain better results. In turn, it has been demonstrated [15, 16] that the choice of virtual machine has a great impact on performance and the use of techniques like Native code or Just in Time (JIT) compilers make a also big difference. The compiled code can actually keep up closely with C++ performance. Also the understanding of some operations on strings, object reuse, have to be properly taken into account to obtain the best performance in Java based applications.

3.1.3 Mixing both

To sum up, ORACLE has been a leading and enthusiastic supporter of Java technology since its emergence in 1995. Oracle has been developing a reliable Application Server (OAS) to run J2EE applications in Oracle Java Containers (OC4J). In addition, Oracle has developed powerful JDBC drivers to improve the connectivity between Java applications and Oracle databases. The integration of other programming languages, such as C++, C#, PHP with the Oracle database is far away in comparison with Java.

3.1.4 Logging Project context

The Logging Project started some five years before its full operational use, therefore the evolution of the technology had to be taken into account. At that time the Oracle 9*i* techonology still wasn't released, but the development team carefully considered it's arrival in order to design the Logging System according to latest available features. Anticipating the technology turned out to be a clever choice, benefiting in the way of increased performance and easier maintenance. In fact, the evolution of the technology requires near constant maintenance, and at present the OAS 10*g* technology is the latest available application server platform. In order to be able to benefit from new features, enhancements and continued maintenance the Data Loading application must be upgraded to run on the OAS 10*g* platform. An overview of the application components used in the Logging System and techniques are now described.

Components

The Logging Project is made up of three main ORACLE components:

- **Database** — The relational database management system (RDBMS).

- **Application Server** — The Oracle Application Server (OAS) is a middleware product composed of an HTTP Server (based on Apache HTTP Server) and OracleAS Containers for J2EE (OC4J) where J2EE-based applications are deployed.

- **Enterprise Manager** — Oracle's Enterprise Manager (OEM) aims to manage software produced by Oracle. At the moment only the Application Server Control release is heavily used, which allows simple monitoring, diagnostics and repair of the deployed service. Tests have started with the OEM Grid Control application, which enables the combined management of the application servers and the database.

Efficient techniques

As mentioned in 3.1.2, the exploitation of adequate optimization techniques is key for the system's performance. In this section, some important techniques already in use in the servlet loaders (dataloader and metaloader) of the Logging System are presented:

- **Batch processing** — Client's Data Loading Files can contain more than 20000 records. If the insertion would be performed one by one it would take a considerable time, therefore the batch technique is used which allows to insert a high number of records together at the same time. The batch is done with an INSERT statement, however when the clients send duplicate data (data that was already inserted in the database), the INSERT fails and the MERGE statement must be used. This change of statement also implies a rollback of the records already inserted in the session, but not committed yet.

- **Statement caching** — Statement caching improves performance by caching executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. The cursors are ready to be used without the need to parse the statement again before execution. When the statement caching is enabled, a statement object is cached when the close method is called. Statement caching can be used with connection pooling.

- **Connection pooling** — The system needs to create for each client's request a connection to the database. It is known that database connections are expensive to create because of the overhead of establishing a network connection. In addition, connection session initialization often requires time consuming processing to perform user authentication, establish transactional contexts and establish other aspects of the session that are required for subsequent database usage [17]. Valuable database resources such as locks, memory, cursors, transaction logs, statement handles and temporary tables all tend to increase based on the number of concurrent connection

sessions. Therefore connection pooling should be used to minimize the creation of database connections and the number of concurrent sessions.

3.2 Security

Security is a vague term and can refer to various aspects, such as network security, application security [18]. In fact, the introduction of the networking and web applications introduced an increased necessity for reliable security mechanisms. As far as application security is concerned, five main functions can be defined to provide the necessary background on security [19]:

1. **Authentication** — The authentication determines who is currently executing the code. It is usually based on a username and password.
2. **Authorization** — The authorization ensures that the users have the access control rights (permissions) required to do the actions performed.
3. **Creation** — The creation is the process of registering a new user for the application.
4. **Maintenance** — The maintenance is the act of changing account information, such as contact information, usernames, or passwords.
5. **Deletion** — The deletion is the process of removing an account from the database, in many cases the account is not deleted but only inactivated.

Many applications only take care of the two first mentioned (Authentication and Authorization). Depending on the application the other actions can be performed only by administrators. There are two different types of authorization :

- **Environment** — The environment where the application is running determines if an user is authorized to see a resource before executing any code. An example of this is the execution of a servlet. The container firstly checks whether the current user has permissions to execute a servlet or not. This type of authorization is also known as declarative security [20] because it is declared in the configuration files of the web application. This is usually done via a resource URL constraint.
- **Application —** By contrast, the authorization based on the application determines if a user is authorized to view a resource when the application is running. Hence, the code is executed and the application decides in real time whether or not some components should be shown.

The Java 2 Security Model enables configuration of security at many levels of restrictions. It is possible to control over many aspects of enterprise applets, component, servlet and

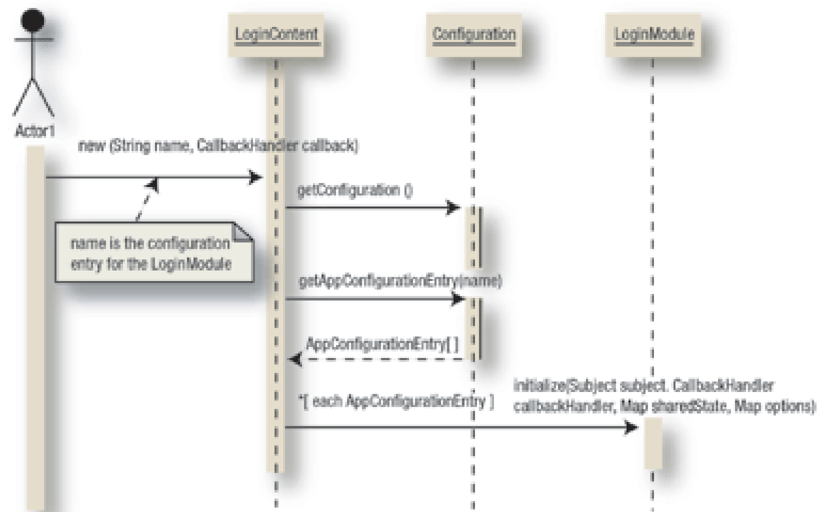

Figure 3.1: LoginContext

application security. The Java Authentication and Authorization Service (JAAS) is a Java package that was designed to complement the Java Security Model.

3.2.1 JAAS

The JavaTM Authentication and Authorization Service was first an optional package and afterwards was integrated into the J2SE. It can be used for the first two purposes aforementioned (Authentication and Authorization) . It implements a Java version of the standard Pluggable Authentication Mechanism (PAM) framework [21]. This allows applications to remain independent of the underlying authentication service. The authentication process is enabled by a LoginContext object instantiated by the application. The object references a configuration to determine the authentication technology or the LoginModule to be used in performing the authentication. Typically the LoginModules prompt for an username and password and verify them, but other LoginModules may read and verify a voice or fingerprint sample. Each LoginModule is initialized with a Subject and CallbackHandler. The Subject represents the user or service currently being authenticated and the CallbackHandler is used to communicate with the users. The 3.1 summarizes the actions.

Once the user or service executing the code has been authenticated, the JAAS authorization component works in conjunction with the core Java SE access control model to protect access to resources.

3.2.2 Logging Project security Mechanism

The Logging Project security mechanism only requires the implementation of the first two functions (Authentication and Authorization). The other actions are performed by the

database administrators according to the clients' requirements. In OAS 9*i*, the Logging Project security mechanism is supported by a class provided by Orion Server [22], however in OAS 10*g* this class is deprecated. The adoption of the JAAS technology is a good candidate to upgrade the security mechanism, since it is supported by the OAS 10*g* and it is used for both required purposes (Authentication and Authorization).

3.3 Management

The management of the applications can be very different depending on the purposes. Applications can vary widely in size, architecture, and criticality. An application typically has a lifecycle [23] that can be defined as follows:

- **Deployment** — Move the file to the system to be ready to start
- **Execution**
  - Monitor — Periodically check a resource to ensure that everything goes well.
  - Operate — Invoke an operation or function to execute.
  - Configure — Configure the application to behave differently.
- **Maintenance / Undeployment** — Apply new code to the application and deploy it again. Sometimes maintenance is not feasible and the cycle finishes with the removal of resource from the system.

A brief overview of the existing management technologies available for those purposes are described:

- **SNMP** — The Simple Network Management Protocol (SNMP) is used for device and network management. It is composed by a set of standards for network management, including an Application Layer protocol, a database schema, and a set of data objects. In spite of its important availability concerning network management, it has not been widely deployed for application management.
- **CMIP** — Common Management Information Protocol (CMIP) is a standard protocol for a larger standard management architecture used in the telecommunication domain and telecommunication devices. Many systems may not be capable of supporting the resource requirements of CMIP and difficulties may exist in the procurement of CMIP software because of limited availability. As far as Network Management is concerned, SNMP can be still an alternative.

- **CIM / WBEM** — Common Information Model (CIM) is a standard information model for describing management data about devices, networks, systems and applications of computer systems. Web-Based Enterprise Management (WBEM) defines CIM operations as an interface to the model. Although the device and system models are quite complete, the application model is also still a work in progress.

- **AIC** — Application Information and Control (AIC) is a C language and Java API for exposing application metrics and thresholds. It is relatively new standard and has not seen any widespread adoption so far.

- **ARM** — Application Response Measurement (ARM) includes a C and JAVA API and it is used to capturate the amount of time it takes to perform units of work inside applications.

- **JMX** — Java Management Extensions (JMX) is a Java technology that provides tools for managing and monitoring applications, system objects, devices and service oriented networks.

It is possible to notice that each technology has its pros and cons. Some are more tailored to a specific area while others can be adapted to a vaster field. The JMX technology has been broadly accepted by the Java community in numerous fields such as embedded systems, enterprise systems and telephony [24]. In fact it is well adapted to take care of the three functions defined in the execution phase (operate, monitor and configure).

3.3.1 JMX

Java Management Extensions (JMX) belongs to J2EE 1.4, since 2005. The JMX API defined a way to create manageable objects called MBeans and to store those objects in a repository managed by an agent - the MBeanServer. The access to the objects is done exclusively through this MBean Server.

3.3.1.1 Components

The MBeans are normally composed by:

- **Attributes** — Each attribute represents a value or a set of values. This attribute can be get or set remotely by the JMX client depending on the permissions defined for the attribute.

- **Operations** — The operations are methods that the JMX client can invoke on the MBean.

- **Notifications** — The notifications can generate broadcast errors or specific events. They can be used to alert about a value that reached a threshold for instance.

3.3.1.2 Types

The MBeans can be divided in several types. The division presented here, includes five different types:

- **Standard MBeans** — This type is the simplest model to design and implement. The operations and attributes of these MBeans are defined in an interface, which has the same name of the class but with the suffix MBean.
- **Dynamic MBeans** — This type implements a specific interface, and exposes its management interfaces on the fly for greatest flexibility. It can be useful to read a configuration file.
- **Open MBeans** — Dynamic MBeans that rely on basic data types for universal manageability, they are self-describing for user-friendliness.
- **Model MBeans** — Dynamic MBeans that are fully configurable and self described on the fly. They provide a generic MBean class with default behaviour for dynamic instrumentation of resources.
- **MXBeans** — An MXBean is the newest type of MBean introduced in J2SE 5.0. This MBean has the special quality to be usable by any client, including remote clients, without any requirement of a model-specific class representing the type of the MBean. In addition, MXBeans provide a convenient way to bundle related values together.

According to JMX best practices, as long as one can use standard MBeans to manage resources, the use of dynamic MBeans should be avoided (Open and Model included) [25].

3.3.1.3 Architecture

The JMX architecture is defined in three levels [26]:

- **Remote Management Level** — Protocol adaptors and standard connectors make a JMX agent accessible from remote management applications outside the agent's Java Virtual Machine (JVM).
- **Agent Level** — The main component of a JMX agent is the MBean server. This is a core managed object server in which MBeans are registered. A JMX agent also includes a set of services for handling MBeans. JMX agents directly control resources and make them available to remote management agents. The API provides agent services, which are also MBeans. Services allow MBean attributes (properties) to be monitored periodically; notifications to be sent at a scheduled time or times,

etc.... JSR 160 (JMX Remote API) defined a framework for remote access to an MBean Server, and standardized an access protocol based on RMI. These two JSRs are included in the J2SE platform. The Java Virtual Machine is instrumented using them, and they are also the default way for user applications to define their own instrumentation.

- **Probe** — Represents any resources, such as applications, devices, or services, are instrumented using Java objects called Managed Beans (MBeans). MBeans expose their management interfaces, composed of attributes and operations, through a JMX agent for remote management and monitoring. The communication is done through Connectors and Adaptors:
  - A connector provides full remote access to the MBeanServer API using various communication frameworks (RMI, IIOP, JMS, ...)
  - An adaptor adapts the API to other protocols such as SNMP or to Web-based GUI (HTML/HTTP, WML/HTTP, ...)

The figure 3.2 schematizes the three levels of the architecture.

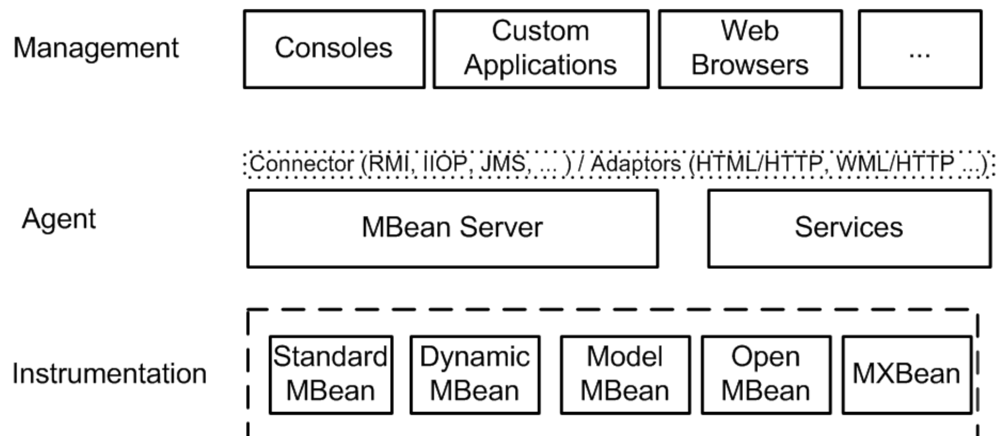

Figure 3.2: JMX Architecture

3.3.1.4 Support

JMX is supported by multiple Java application servers such as JBoss, JOnAs, WebSphere Application Server, WebLogic, Sun Java System Application Server and by the Oracle Application Server since 10g.

The Oracle Enterprise Manager 10g Control Console user interface is built on a JMX client that can be used to completely manage and monitor an OC4J instance, including the Java Virtual Machine (JVM). In addition to this console, numerous other consoles were also developed. The most well-known is the Java Monitoring and Management Console (JConsole) that is included in the JDK distribution. It is a graphical tool that allows

monitoring a local or remote JVM and manage applications through the exposed MBeans. Other consoles can be also referenced such as MC4J, XtremJ, JManage.

As one can notice, the lack of support of application servers and consoles regarding this technology is far away from being a problem.

3.3.2 Managing the Logging System

The management of the mission-critical Logging System is crucial. Applications belonging to mission-critical systems cannot be re-deployed each time a configuration must be modified. The application should be re-deployed as least often as possible, and so many administrative tasks must be operated on the fly. Since OAS 10*g*, Oracle Enterprise Manager is built on a JMX-compliant client that can be used to completely manage and monitor a running OC4J instance.

3.4 Chapter overview

This chapter showed the technology used by the Logging System. When it comes to upgrading the application server, the Java technology is decisive. Therefore, the security and manageability technologies referenced in this chapter are mostly focused on the Java technology.

Chapter 4

Security

4.1 LoginModule solution

As seen in 3.2.1, the JAAS security framework can be used for two main functions: the authentication function and the authorization function. The Logging Project requires precisely these two functions in order to execute both servlets (DataLoader and MetaLoader) securely. Since the Oracle Application Server 10*g* supports LoginModules, JAAS is a good candidate. The proposed solution uses a JAAS LoginModule. A LoginModule can be totally customized, however several application servers have developed generic LoginModules adapted to standard designs.

OAS 10*g* provides a Login Module (DBTableOraDataSourceLoginModule) since version 10.1.3.1. This module authenticates the client against database tables storing account information. After the authentication of the user, the Oracle Container checks if the user has privileges to access the web resource. If the request is authenticated and authorized, the web resource is returned to the client. If the client doesn't have permissions to access a resource, an error code is returned (403 Forbidden HTTP status code). In order to be able to use this Login Module, the database design must obey to several requirements:

1. The database must have a table where the users are defined, which must have a primary key on the username column.

2. The database must have a table of roles, where the username column is defined as a foreign key corresponding to the primary key of the users' table.

The Figure 4.1 on page 26 shows the users and roles tables of the Logging database design . One can notice that there is a table of users (META_USER) and a table of roles (META_USER_ROLE). The table of users has the primary key defined on the username (USER_NAME) column and the table of roles has the corresponding foreign key defined on its username (USER_NAME) column as well. Therefore after this short analysis, it is possible to conclude that the requirements are completely met.

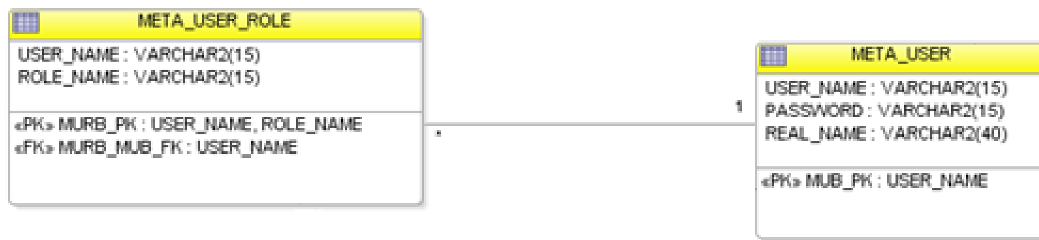

Figure 4.1: Users data model

In some cases the development of a module from scratch can be avoided without losing any significant feature. Usually, the use of a component that already exists brings many benefits compared with its development from scratch [27]. The dependability increases because it is a system that has already been extensively tested and the development process typically speeds up because both development time and validation process are reduced. On the other hand, it can be hard to adapt the component if the dessign changes. Anyway, the authentication and authorization is a quite direct process and the only case that could invalidate the use of this module is the change of the database design. Since the design is not likely to change, it is undoubtedly reasonable to use this module.

4.2 Declarative Security configuration

As mentioned in Section 3.2, declarative security is the process of determining the permissions of the user before the execution of any code. Therefore the configuration is defined in configuration files and the servlet is only executed if the user is correctly authenticated and authorized.

Currently, there are only two types of roles :

- **META_WRITER** — This role gives access to the LoggingMetadataLoader servlet to register new logging variables.
- **DATA_WRITER** — This role gives access to the LoggingDataLoader servlet to load values of logging variables already registered in the system.

The following XML code shows how to configure the application to use the Login Module. The first step is to tell the application, which module it should use for security. In this case, the DBTableOraDataSourceLoginModule is chosen. Then the Login Module must be configured properly. Since this module validates users directly against database tables, it needs to establish a connection with the database. The data_source_name represents a data structure that contains the information about the Logging database connection. Then, it is necessary to tell the login module, which tables in the database

and which fields should be used. The table META_USER is used to store the users, then the field USER_NAME is the field where the name of client is stored and the field PASSWORD contains the password of the client. The table which defines the roles is also given (META_USER_ROLE), specifying the field containing the role name, in this case the field is ROLE_NAME.

```
<jazn-loginconfig>
  <application>
    <name>xml-data-loaders</name>
    <login-modules>
      <login-module>
<class>oracle.security.jazn.login.module.db.DBTableOraDataSourceLoginModule</class>
        <control-flag>required</control-flag>
        <options>
          <option>
            <name>data_source_name</name>
            <value>jdbc/lhclogdb</value>
          </option>
          <option>
            <name>table</name>
            <value>META_USER</value>
          </option>
          <option>
            <name>usernameField</name>
            <value>USER_NAME</value>
          </option>
          <option>
            <name>passwordField</name>
            <value>PASSWORD</value>
          </option>
          <option>
            <name>groupMembershipTableName</name>
            <value>META_USER_ROLE</value>
          </option>
          <option>
            <name>groupMembershipGroupFieldName</name>
            <value>ROLE_NAME</value>
          </option>
        </options>
      </login-module>
    </login-modules>
  </application>
</jazn-loginconfig>
```

After the module configuration, it is necessary to indicate in a declarative way, what are the resources that can actually be accessed and by which clients. The following XML code defines the policy for each resource. The code shows that clients who want to access

the *DataLoader* servlet needs to have the *data_writer* role. The clients who want to access the *MetadataLoader* servlet need the *meta_writer* role. The authentication method and the existing security roles are also provided.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>MetadataLoader</web-resource-name>
        <url-pattern>/LoggingMetadataLoader</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>meta_writer</role-name>
    </auth-constraint>
</security-constraint>
<security-constraint>
    <web-resource-collection>
        <web-resource-name>DataLoader</web-resource-name>
        <url-pattern>/LoggingDataLoader</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>data_writer</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Logging Data Loader</realm-name>
</login-config>
<security-role>
    <role-name>meta_writer</role-name>
</security-role>
<security-role>
    <role-name>data_writer</role-name>
</security-role>
```

4.3 Discussion

In fact, since the database design is not likely to change the Login Module provided by ORACLE is tailored to the given problem. An added advantage compared to the previous security implementation is the fact that clients are made aware of authorization issues while in the previous version they were not: If a client introduces the correct credentials without the privileges to do an action, an error message stating the forbiddance of the action is received. In the previous version, the user couldn't differentiate between an authorization or authentication issue, since the received message only reported about wrong credentials. The Data Loading application configured to use the DBTableOraDataSourceLoginModule module can run securely on the latest available version of the OAS (10*g*).

Chapter 5

Manageability

This chapter describes a solution to maintain the previously available functionalities related to the management of the Data Loading application on the fly.

5.1 Logging Data Loading application management

The Logging Data Loading application was managed on the fly using *context servlet parameters* exposed via OEM 9*i* interface. These parameters are no longer exposed in the OEM 10*g* interface. The OEM is one of the main components of the Logging System, actually it is the sole interface used to manage the application servers and the Logging applications. Therefore it is absolutely necessary to adapt the application management so that it can continue to be done via OEM. The management must be performed on the fly, since the mission-critical system cannot be interrupted each time a configuration change is required. Every single parameter previously defined using *context servlet parameters* in the obsolete 9*i* based version must be made available in the new 10*g* version.

In addition, some functionality not performed by *context servlet parameters* is also requested. An inexperienced administrator who is not accustomed to the parameters, needs to have clear descriptions of the parameters he needs to change, without having to look at the documentation. In addition, since there are many parameters, they should be well-organized to facilitate their use. Finally, human error is inevitable - erroneous values inserted by an administrator could cause serious problems in the execution of the application. Therefore when a parameter value is entered or modified, it must be validated.

The requirements defined for this module are sumamrized in Table 5.1 on page 30. The requirements with the highest priority are related to the previously available management functionality. The other priorities are related to the newly requested functionality.

Description	Priority
Configuration on the fly	High
Configuration integrated with OEM	High
All parameters must be configurable	High
Validation of input values	Medium
Parameters must provide a description	Low
Parameters must be organized	Low

Table 5.1: Management requirements

5.1.1 JMX and OEM

One saw in 3.3.1 that OAS 10g supports the JMX technology for Java application and system management. In addition a JMX browser is completely integrated into the new version of the OEM interface. Therefore, in a Java context JMX seems the best candidate to manage Java applications on the fly. The highest priority requirements related to application configuration changes on the fly, and the integration with OEM interface are met by using this technology.

5.1.2 Parameters classification

After analysing some of the functionalities that could be performed on the fly with the *context servlet parameters*, one could organize them in into four different categories:

- **Auditing** — As explained in 2.1.2, new clients need to be validated. In addition, suspicious conducts of clients running in the production environment could also be audited. For these purposes, some actions such as capturing the data files sent by clients, or writing in the database information concerning the client activities were put in place. The configuration of theses actions could be done using the following parameters.
  - Status: A flag to define if auditing must be done or not.
  - Clients: The list of clients that must be audited (several information are sent to the database).
  - CaptureClients: A list of clients, whose files are captured on local disk.
- **Batch** — When writing to the Logging database, records are sent in batches. It is possible to modify the size of the batches and the batch writing mode:
  - Mode: The JDBC batch mode to be used (either ORACLE or NATIVE).
  - Size: Defines the threshold at which batch records should be flushed to the database, this only concerns ORACLE JDBC batch mode.

- **Datasource** — The data source name is defined in the OC4J environment. When the application wants to connect to a database, it finds the datasource using the specified JNDI name.
  - Name: The JNDI name of the datasource to be used
- **Files** — These parameters are related to the handling of XML data files sent by clients, and also where erroneous or captured files are stored.
  - DiskSize: The maximum file size to be stored on disk (in case of user capture)
  - MaxFiles: The maximum number of files to be stored on disk per client
  - MemorySize: The maximum file size that is accepted by the server
  - TempDirectory: The directory where captured and erroneous files are stored

JMX provides MBeans to manage resources. Since the parameters are divided in four categories Section 5.1, it makes sense to create an MBean for each category. Previously when using *context servlet parameters*, each parameter value was defined as a simple string. MBeans attributes support this object type and many more, which means that all parameters can be managed. The list of clients was done using delimited (;) strings, can now instead be done using an array of string objects; flags can be defined using the boolean object type. Part of the parameter value validation process can in fact be done just by using the appropriate object types.

5.2 Architecture

The Logging Data Loading infrastructure consists of several application servers, and subsequently two options can be envisaged to integrate JMX:

- JMX is integrated in only one application server and all application management is done from only one server
- JMX is integrate in all the application servers

The first approach can be good from the point of view of the administrator because he would only need to access one server. On the other side, the applications running in the other servers would have to access the management parameters remotely through the agent layer. This remote access would introduce a significant lag to the system. In addition if the application server used for management crashes, no more values would be available for the applications still running on the remaining servers. The second option is much more interesting because the application can access the management parameters locally. Therefore the second approach has been chosen. In Figure 5.1 on page 32 one can see the JMX integration in the system and the different MBeans deployed.

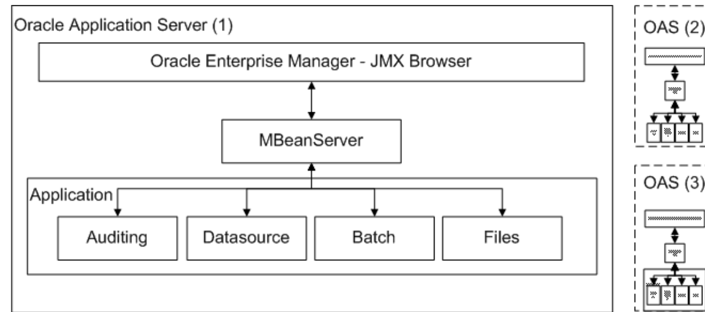

Figure 5.1: JMX integration in the Logging System

5.3 Design and Implementation

5.3.1 Standard MBeans

In spite of its basic functionalities, standard MBeans, seem to match all the requirements in order to substitute the *context servlet parameters*. Moreover, the *JMX best practices* [25] strongly recommends specifying MBeans using the standard model whenever possible. They are more understandable and can be documented using the familiar Javadoc tool. In addition clients can interact with them straightforwardly via proxies. All methods and attributes are defined in the interface of the manageable object, therefore it is easy to analyse the code or Javadoc to exactly know what are the attributes and operations provided by the MBean.

In the current Java version, the JMX specification still has some limitations. One limitation concerns the use of an interface ending with the suffix *MBean* for each manageable object. For instance, if an MBean is called *Test*, it has to implement an interface called *TestMBean*. Another limitation concerns the meta-information. Most of the existing JMX consoles are able to display the MBeans information, however the implementation of the MBeans requires some workaround. In addition to attributes, meta-information can also refer to methods and relative parameters, as well as notifications. The enhancement of these limitations and other interesting features are planned to be integrated in the next JMX API available in Java 7 [28].

To overcome some of these limitations, the classes are created based on the technique explained in [29]. This technique consists of directly subclassing the javax.management.StandardMBean class and overriding the *getMBeanInfo* method to provide meta-information about the MBeans and its attributes, operations and notifications. Actually in this context only the information about the MBean and its attributes are concerned, however the implementation is designed to support any kind of information that might be used in other contexts.

An easy way of editing the information of different MBeans in only one file and also implementing MBeans without taking care of the *getMBeanInfo* method can be achieved

with the introduction of an intermediate class between the MBean and the StandardMBean. This intermediate class has a reference to an enumeration class, where the respective meta-information is provided. The Figure 5.2 on page 33 shows this implementation. Part of the enumeration class can be found in Annex D.

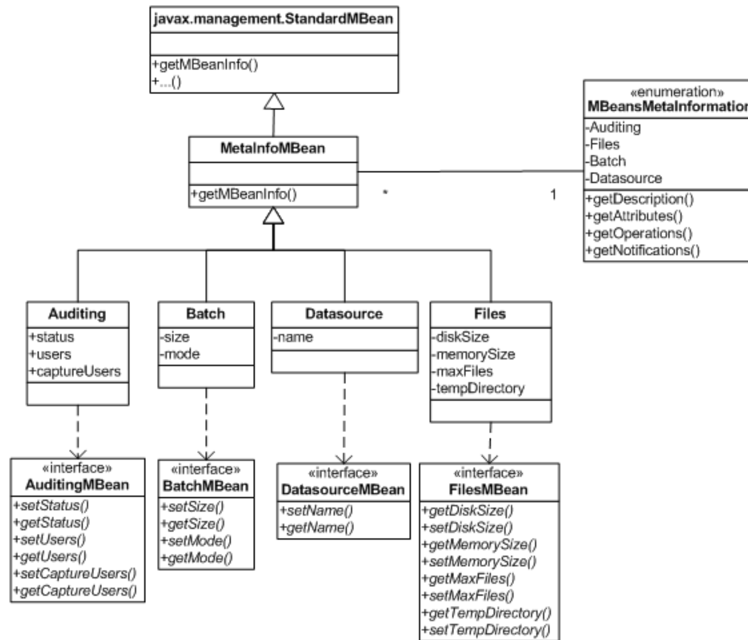

Figure 5.2: MBeans Design

5.3.2 Dynamic registration

The manageable beans can be registered either statically or dynamically:

- **Statically** — One way used to register MBeans is through configuration files. This approach has the main advantage that MBeans are registered as soon as the application is deployed. The main disadvantage of this approach is that the servlets cannot directly share resources with them, since there is no reference to the manageable objects from the servlet. The only way to access the MBean is through the MBeanServer, however this approach introduces some extra time and the implementation is much more complex. Additionally, the MBeans must be put in a JAR archive file and deployed together with the application. This approach is feasible when the applications do not need to directly access the MBeans.

- **Dynamically** — On the other hand, MBeans can be registered during the execution of the application when the code is first executed. When the MBean is instantiated, the application can keep a reference to the MBean object and access its methods and attributes directly. This approach is much more efficient.

The registration is done dynamicaly because the faster the attributes are read - the better the performance of the system; the implementation is also less error-prone. This registration is done only once, when one of the two servlets is executed. The following steps summarize the operations that need to be executed to register the Auditing MBean in the server.

1. It's necessary to get the MBeanServer running in the platform, this operation is normally done only once.

   ```
   MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
   ```

2. One must provide information concerning the name of the application (*xml-data-loaders*) and the name of the MBean (*Auditing*). Additionally, a 'type' is specified in order to display the MBean in an organized structure. Once the name of the MBean is created, the ObjectName can be instantiated.

   ```
   String applicationName = "xml-data-loaders";
   String type = "parameters";
   String mbean = "Auditing";
   String beanName = applicationName + ":type=" + type + ",name=" + mbean;
   ObjectName obj = new ObjectName(beanName);
   ```

3. Then, the MBean needs to be instantiated.

   ```
   Auditing auditing = new Auditing();
   ```

4. Finally, one must proceed to register the MBean in the container.

   ```
   mBeanServer.registerMBean(auditing,obj);
   ```

Since the OEM 10*g* provides a JMX browser and the application runs locally, it is possible to access those MBeans directly from the OEM interface without developing any additional code.

5.3.3 Validations

Part of the validation is done in the JMX layer: using the appropriate object type for each one of the managed parameters values one is ensuring part of the validation.

For other attributes defined as string objects, one must still use another type of validation based on a defined domain . If an entered value is out of domain, the value is not accepted and an error message shows up in the OEM interface. For instance the batch mode parameter can only be *NATIVE* or *ORACLE* . If an administrator inserts *XPTO*, the MBean detects that the value is not contained in the domain, and so the batch mode is not updated; instead the JMX browser throws an explicit exception.

5.4 OEM Console interaction

The results presented in Figure 5.3 on page 35 shows how the Oracle Enterprise Manager Console Interface displays the MBeans according to their types (in this case only the type parameter is displayed - left image). Clicking on the Auditing MBean (first arrow) one can see its details (attributes and description : *Controls Client Auditing*). For each of the MBeans attributes, the attribute description, value, and access type are shown. In this case all the attributes are RW (read and write).

A boolean value can be easily changed with a box (false/true) e.g. the *status* attribute. The attribute values of *users* and *captureUsers* corresponds to the list of clients being audited. They can also be edited in an easy way - clicking on the attribute name (second arrow). The image on the right shows the intuitive interface to add a new user or edit/remove it.

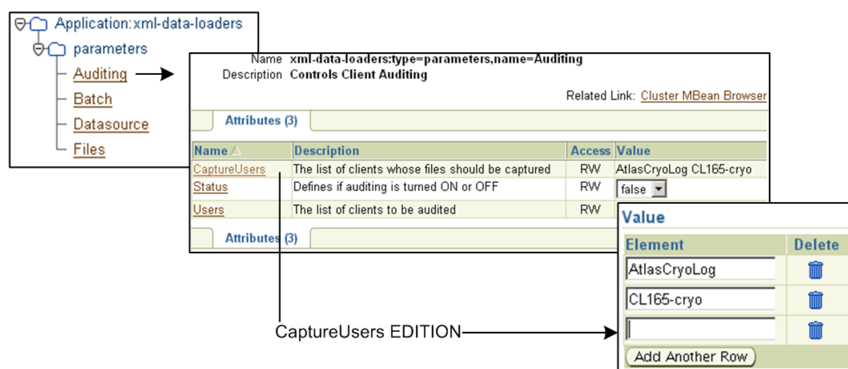

Figure 5.3: JMX browser interface in OEM

5.5 Additional MBeans for configuration

The JMX technology turned out to be perfectly adapted to making run-time configuration changes. The Logging Data Loading application was also suffering a lack of control regarding the logging files generated by Log4j. When the application is deployed for the first time, one could be interested in low level messages (debug or info mode). However, if everything goes fine, a higher log level (warn, error, fatal) will likely be required - avoiding the huge amount of information received in the logging file and improving the performance of the system (since the I/O operations are reduced). Therefore the implementation of the log4J loggers was modified in order that it could be managed in real time using a dedicated MBean.

5.6 Summary

This chapter focused on the potential of JMX for making changes to applicaion configuration in real time. This technology was used to replace the obsolete use of *context servlet parameters*, and additionally to manage the level of messages written by the log4J loggers. Due to the integration of a JMX browser in OEM 10*g*, all requirements could be met. In the next chapter one can see how JMX is also adapted to efficiently monitor an application. Other important features are presented about JMX such as notifications, methods and the use of MXBeans.

Chapter 6

Instrumentation

In this chapter an extension of the Logging Data Loading system is presented. This mission-critical system needs to be equipped with a powerful instrumentation tool to monitor system behaviour and identify potential problems. The instrumentation service is partially supported by JMX technology. This technology proved to be not only a solution for the configuration of an application in real time, but also as a powerful tool to monitor, analyse and notify of numerous situations on the fly.

6.1 Specification

An extensively used, mission-critical system, with an extremely high throughput must perform as well as possible. In order to achieve high level of performance, the use of top technology (both hardware and software) is not enough. Every action performed by the system must be highly optimized. In addition, clients must be well-behaved, i.e. they should not make unnecessarily frequent low volume data loading requests and they should avoid the redundant sending of duplicate data. It is extremely important to be aware of how each clients behaves, since a single badly behaving client can have a huge negative impact on the overall system performance. Receiving more than one billion records per day from different clients located in a widespread area all around the LHC can be hard to analyse without a powerful and sophisticated instrumentation.

PVSS clients store signal measurements in temporary local archives. These measurements are mapped to variables previously registered in the database (using the metadataloader servlet), and then periodically sent in files to the dataloader servlet. The file must conform to a specific XML schema definition. An example of file can be viewed in Annex A.

When the servlet receives the file it performs a sequence of actions:

1. **Validation** — The XML file is validated against the XML Schema Definition.

2. **Parsing** — The file is parsed in order to map the Logging variables and related time series data records to Java objects.

3. **Checking** — This step consists of checking the existence of the variables in the database and at the same time retrieving the database identifiers (primary key values) corresponding to the given variable names.

4. **Writing** — Finally, once the identifiers are selected, the records can be inserted into the database.

The Figure 6.1 on page 38 shows a schematic of the whole process. The time spent on each one of these actions needs to be instrumented by the service in order to identify potential bottlenecks in the system.

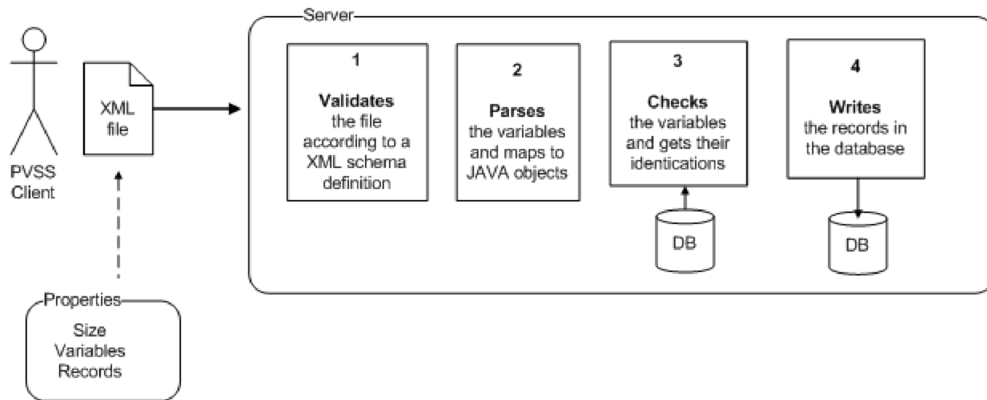

Figure 6.1: File process

The behaviour of a client can be hard to predict, because it depends on the LHC's state. A client is typically replicated on different host machines all around the LHC and can act differently depending on the sector he is located. The Figure 6.2 on page 39 shows a typical scenario of the cooling process. The cooling process in the LHC actually takes a few weeks because the sectors cannot be cooled all at the same time. The figure shows that sectors 12,23,34,45 and 56 are already cooled to the intended temperature (2 kelvin), therefore related measurements might be more or less constant. The sector 67 is still being cooled and so the measured values are probably changing more frequently. The rest of clients located in sectors at ambient temperature (300 kelvin) are also probably sending constant measured values.

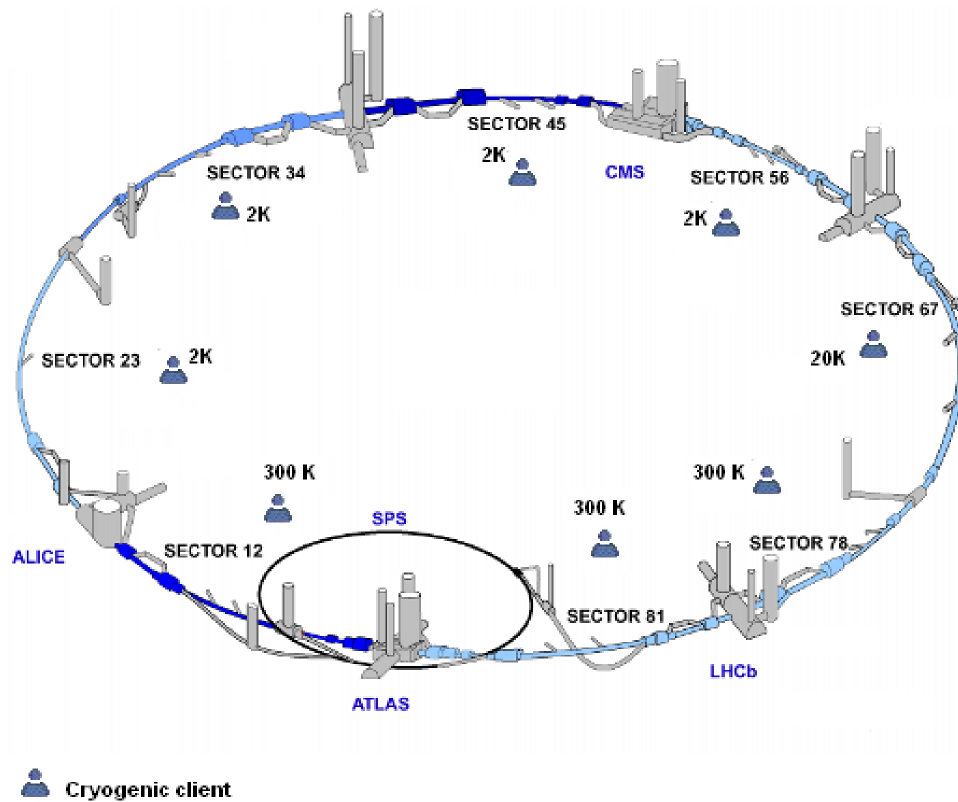

Figure 6.2: LHC Cooling process

The instrumentation must be able to give an insight into the behaviour of each client with the possibility to compare behaviour in different locations.

In addition, the instrumentation must not have a significant impact on the available system resources, such as CPU time, available memory and disk space.

One can notice that functional requirements and non-functional requirements are somewhat opposing: The functional requirements of the instrumentation are to capture all information about the properties of each client request. If one chose to do this in memory to assure minimal execution time, the instrumentation data would exceed the memory capacity in a few minutes. On the other hand, if one chooses to store the information on local disk, the execution time will drastically increase and subsequently the overall system performance will be reduced.

To solve this dillema the instrumentation is based on three levels of abstraction.

6.2 Architecture

The instrumentation can be done at file, memory and database levels. Each one of these approaches is explained in the next sections and schematized in Figure 6.3 on page 40.

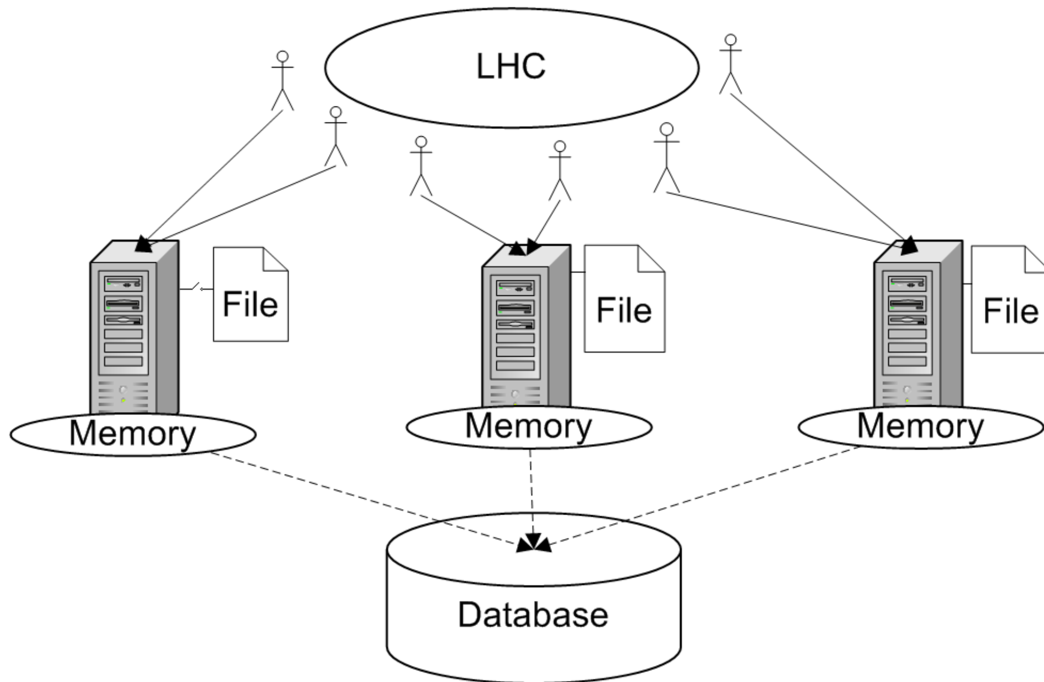

Figure 6.3: 3 levels of instrumentation

File

For each client request all the properties are written in a file. This approach has many drawbacks: Firstly, keeping all the request properties in a file is time consuming. Then, the more requests properties that are stored - the harder the analysis becomes. Finally, the storage increases linearly with the number of incoming client requests. Therefore, this type of approach should only be used to answer specific questions, such as:

- What happened during a specific short period of time (few minutes)? (e.g. Identify database issues)
- How client requests properties vary during a short period of time? (e.g. Analyse the distribution of requests during the 24 hours of a few days)

Due to the drawbacks mentioned, this approach will normally not be used during normal operation, therefore one must be able to turn it on and off in real-time.

Main memory

A solution to overcome the disadvantages of file based instrumentation, but without losing meaningful information is achieved with the creation of a structure in main memory in which new request properties are examined and corresponding statistical measurements immediately worked out. Only statistics measurements (e.g. average, maximum, minimum, last value) are kept, which means that the memory required is constant with re-

spect to the number of incoming client requests. For real time analysis the structure is integrated with JMX, which has proved to be not only useful to configure an application on the fly, but also to monitor it. In addition JMX technology provides operations and notifications. Notifications can be sent when a threshold limit value is reached (e.g. writing time too long). Operations can be useful to dynamically register a dedicated manageable resource for a specific client. This efficient approach is able to answer numerous important questions such as:

- Who are the clients using the application server?
- Are the clients respecting the limits imposed for the size of files?
- How many requests per minute are being received from a client?
- Who is the client with the most duplicate data?
- How long do the validation/parsing/checking/writing phases take on average?
- What is the minimum time for writing records to the database?
- Who are the most active clients?
- In how many host machines is a client replicated, and is he behaving differently depending on the machine?
- ...

The clients behaviours are directly connected to the LHC state (cooling process, running...), this means that their conduct is not always the same. Because of this, it is impossible to extract valuable information from statistics measurements derived over extended periods of time (more than some days). In addition, the performance of the system can also change if some tunings are done or a new server is added. In order to compare values from different days some statistics measurement are written to the database and statistics reset on the application server every day at midnight (UTC).

Database

The Logging Database was built to store heterogeneous time series data, and to ensure the persistence of the data during the lifetime of the LHC. Therefore it is possible to store the statistics measurements data in the database. In addition, the TIMBER1.3.3 tool built on top of it is quite powerful to make analysis.

The Data Loading applications were configured so that every day at midnight (UTC) their statistics measurements are written to the database. Since the statistics measurements come from applications running on several distributed application servers, the information is grouped within the database and global statistics measurements are computed

based on the load of each server. Still, the information sent from each server is kept for some days in case an in depth analysis is necessary. The statistics measurements stored in the database are able to give feedback on questions such as:

- Is the system performing better now with respect to yesterday / last week / last month / last year...?
- In which month during the year have clients sent the most data?
- Has a client improved his application configuration (reducing duplicate data)?
- Has the behaviour of a client changed over time?
- ...

6.3 Design and Implementation

6.3.1 File

A first try of this kind of approach was already put in place in a debug mode, however it was not complete and difficult to exploit because of problems explained in 2.2.3.1 (Concurrency, huge amount of information, etc...). The following piece of code shows how a logging file looked like:

```
18:01:21 A - validation 250 ms //Client A makes a request and his file is validated
18:01:21 A - parsing 300 ms // Then his file is parsed
18:01:22 B - validation 300 ms // Client B has joined
18:01:22 A - checking 30 ms // Back to Client A for the checking time
18:01:22 C - validation 20 ms // Client C has joined
18:01:22 B - parsing 300 ms // Back to client B for parsing time
18:01:22 C - parsing 20 ms // Back to client C for parsing time
18:01:23 B - checking 30 ms // Back to client B for checking time
18:01:23 C - checking 6 ms // Client C has joined
WARNING  B - duplicate data found in client B records // Client B
18:01:23 A - writing 1200 ms // Only now Client A has his records written in the database
...
```

The reading of this file becomes increasingly complicated as the number of concurrent clients increases. The solution implemented to enhance this file structure is based on two main ideas:

1. The different phases corresponding to a single client request must be grouped together. This makes the concurrency problem transparent. Additionally, the file is written only one time per client request, instead of being written at each operation.
2. Since the quantity of information is still huge and reading many records is a painful task, the information must be organized in a way that it can be more easily analysed.

The following code shows how the same information is written with the new instrumentation aspect.

```
Hour:Minute:Second|client|validation, parsing, checking, writing....
18:01:21|A|250,300,30,1200... //Information of the request made by client A
18:01:22|B|300,300,30... //Information of the request made by client B
18:01:23|C|20,20,6... //Information of the request made by client C
```

This file can be easily opened in a spreadsheet application taking into account the delimiters. This approach is used for specific situations, such as researching patterns and correlations between variables or identifying specific situations like the properties of a single request or peaks of the database activity in a short period of time. However it is still difficult to make analyses of this data over long periods of time. Writing in a file always introduces an overhead because of I/O operations. In order to overcome this, one added the possibility to switch on/off this kind of instrumentation at any time using manageable resources. Also, a limit of 10MB is defined for each file, from that limit a pre-defined number of backup files are created. In fact, the instrumentation file is the only approach that store detailed information about each request. Nevertheless, the file instrumentation has many disadvantages:

- **Execution time** — The process of writing in a file is slow.
- **Storage volume** — The amount of storage required is enormous.
- **Analysis facilities** — The file must be integrated with other applications to be easily analysed and even like this the analysis can only be performed for a short period of time.

6.3.2 Memory

The main challenge is to overcome the drawbacks of the file writing, by reducing the execution time, reducing the storage required and facilitating the analysis. In order, to reach these requirements, derived statistics measurements (such as arithmetic mean, standard deviation, max, min, etc...) are created for different properties (times, files, records, etc...) in main memory and integrated with manageable resources to be monitored by the OEM. Because they are statistics the storage required is constant with respect to the number of client requests.

Three type of statistics have been clearly identified:

- **System** — Statistics that give an insight about the overall system behaviour and performance.
- **Client** — Statistics that give an insight about behaviour and performance for a specific client.
- **Client in a specific machine** — Statistics that give an insight about how a client behaves in a specific host machine.

6.3.2.1 Data structure

To be able to access any statistics in a fast way and with a constant time complexity, the structure is based on hashtables in the main memory. The Figure 6.4 on page 44 shows the schematic of this data structure. The root of the structure contains a *system statistics object* (in yellow) and a hashtable. The *system statistics object* has all the information (properties and corresponding measurements defined in 6.3.2.2) regarding the overall system behaviour and performance. In turn, the hashtable contains information concerning each client who uses the system. The information of each client is similar to the root. It is composed by the *client statistics object* (in green) and another hashtable. Finally, this last hashtable stores directly *machine statistics objects* which refers to a client running from a specific host machine (in blue). The *system statistics object* is created when the application is launched for the first time, but the other statistics objects (*client and machines*) are created when requests are received from new clients, since it is not known in advance which clients will use the system. When every client has made at least one request, all the statistics objects are created and the required storage is constant.

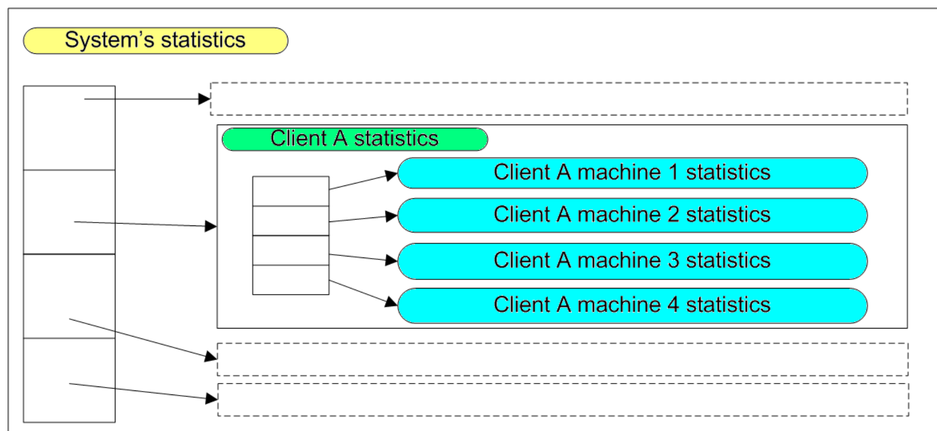

Figure 6.4: Data Structure

The total number of statistics objects depends on the number of clients and on how many host machines a client is running:

```
Number of statistics = 1 + Number of clients + Sum of machines used by each client
```

If the system is composed by a *client A* running on 4 machines and a *client B* running on 15 machines. A total of 1+2+(4+15) = 22 statistics objects are created. 1 statistics object for the system, 2 statistics objects for the clients and 19 statistics objects for the clients running on different machines.

6.3.2.2 Properties and statistics measurements

Statistics objects can be related to many properties, such as the time of operations, the file size, the number of records, etc... These properties are defined in an enumeration class.

The following Java code shows the properties defined.

```
enum Property {
 VALIDATION_TIME, //The time spent to validate the file
 PARSING_TIME, //The time spent to parse the file
 CHECKING_TIME, //The time spent to check the variables of the file
 WRITING_TIME, //The time spent to write the records of the file
 TOTAL_TIME, //The total time spent in the session
 SIZE_FILE, //The size of the file
 VARIABLES, //The number of variables contained in the file
 RECORDS, //The number of records contained in the file
 NUMERIC_RECORDS, //The number of numeric records in the file
 VECTOR_NUMERIC_RECORDS, //The number of vector numeric records in the file
 NUMERIC_STATUS_RECORDS, //The number of numeric status records in the file
 STRING_RECORDS //The number of string records in the file
}
```

In order to maintain low storage requirements, statistics measurements must be constantly worked out. Therefore, for each property six measurements are defined. The following enumeration class shows the statistics measurements that can de derived.

```
enum Measurements {
 MAX, //Represents the maximum value
 MIN, //Represents the minimum value
 INT, //Represents the sum of the values
 AVG, //Represents the arithmetic mean
 STDV, //Represents the standard deviation
 LAST_VALUE //Represents the last value
}
```

6.3.2.3 Update statistics

Each time a client makes a request, the properties are tracked and the three type of statistics must be updated. The following code exemplifies the process.

```
//Gets the statistics of the overall system, client and machine
//If a client or machine still doesn't exist the corresponding object is created

Statistics statSystem = getStatistics();
Statistics statClient = getClientStatistics(username).getStatistics();
Statistics statMachine = statClient.getMachineStatistics(host);

statSystem.add(properties); //Update the overall system statistics
statClient.add(properties); //Update the client statistics
statMachine.add(properties); //Update the machine statistics
```

The number of requests is common to all the properties therefore they are kept only in the statistics class. The simplified class diagram in Figure 6.5 on page 46 shows the relationships between Statistics, Property and Measurements classes. In the class *measurements* there are no attributes referent to the standard deviation or average, in fact these values are calculated based on the sum, sum squared and number of requests in order to reach a better precision.

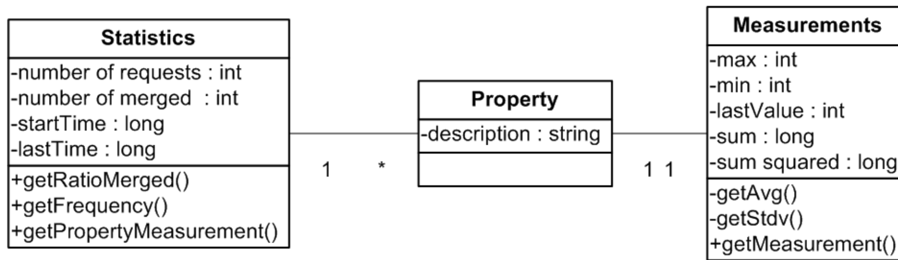

Figure 6.5: Statistics, Property and Measurements

The following code shows how part of the Measurement class is programmatically implemented:

```
public void add(int value){
        lastValue = value;
        max = Math.max(max, value);
        min = Math.min(min, value);
        sum += value;
        sumSquared += Math.pow(value,2);
}

Object getMeasurement(Measurement m)
{
  switch(m){
    case MIN: return min;
    case MAX: return max;
    case LAST: return lastValue;
    case SUM: return sum;
    case STDV: return getStandardDeviation();
    case AVG: return getAvg();
 }
}

private double getAvg() {
        int n = statistics.getNumberRequests();
        if (n < 1)
            return 0;
        double avg = (((double) sum )/ n);
        return avg;
}

private double getStandardDeviation() {
    int n = statistics.getNumberRequests();
    if (n < 1)
        return 0;
    double avg = getAvg();
    double avgSquared = avg * avg;
    double avgOfSquares = sumSquared / n;
    double variance = avgOfSquares - avgSquared;
    if (variance < 0)
        return 0;
    return Math.sqrt(variance);
}
```

The Statistics class can extract any measurement derived, base on the two enumeration classes previously defined. This method is extremely important for the management performed by the MBeans and for writing statistics measurements in the database.

```
double getProperyMeasurement(Property p, Measurement m)
{
 switch(p){
  case p.WRITING_TIME: return writing_time.getMeasurement(m);
  case p.FILE_SIZE: return file_size.getMeasurement(m);
   ...
 }
}
```

6.3.2.4 Daily Summaries

Because the system performance is not constant over time, and because the client behaviour is also extremely influenced by the state of the LHC, and system / client configurations, statistics measurements need to be periodically written to the database, and then reset on the application servers. Not all the statistics measurements are selected, only measurements of interest are written. The selected statistics measurements are also defined using specific features of the Java Enumeration class. The following code shows the definition of three measurements to be written in the database:

- The average writing time (WRT_TIME_AVG) corresponds to the property WRITING and the AVG measurement.
- The maximum number of variables (REQ_VAR_MAX) corresponds to the property VARIABLES and the MAX measurement.

```
enum StatDB
{
 WRT_TIME_AVG(WRITING, AVG),
 REQ_RECORDS_INT(RECORDS, INT),
 REQ_VAR_MAX(VARIABLES, MAX),
  ...

 Property p;
 Measurement m;
 StatDB(Property _p, Measurement _m){
  p = _p;
  m = _m;
 }
 getProperty(){
  return p;
 }
 getMeasurement(){
  return m;
 }
}
```

This enumeration class combined with the *getPropertyMeasurement* function in the *statistics* object enables to extract in a scalable way the information about all the measurements to be written to the database.

Statistics measurements are written to the database using the batch loading technique. Other techniques such as connection pooling, use of prepared statement and statement caching are also employed to guarantee the best performance.

6.3.2.5 Management

MBeans have been integrated with the statistics measurements with the aim to be able to monitor and easily analyse the values of any statistics measurement in real time and know what is actually going on. The implementation related to meta information have followed the model proposed in Section 5.3. Three types of MBeans have been defined to manage the system. The Figure 6.6 on page 48 shows the different types of MBeans.

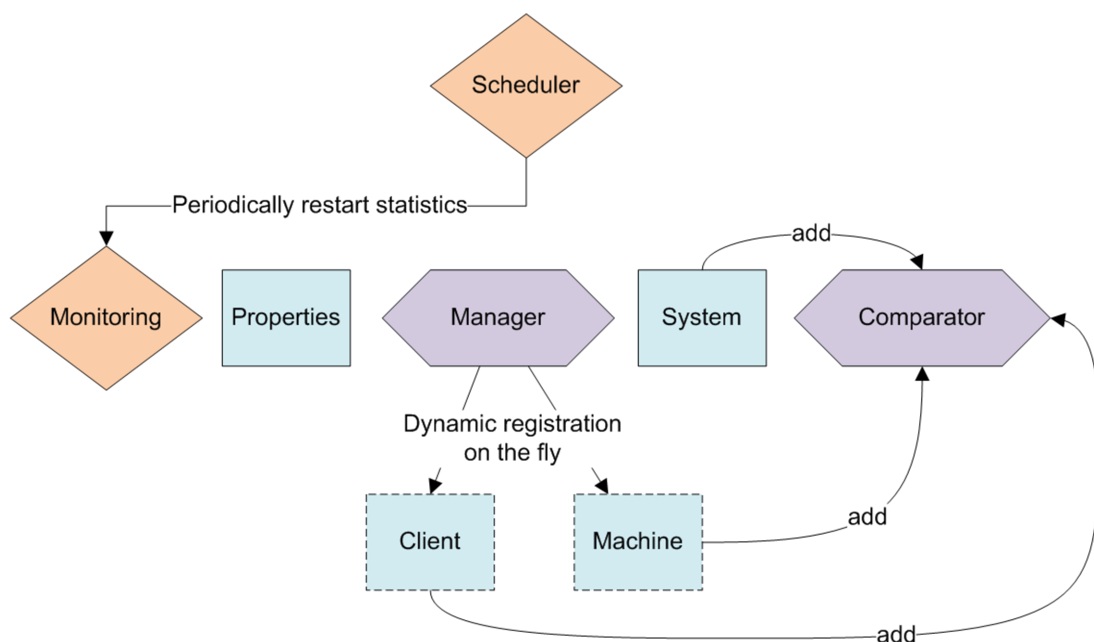

Figure 6.6: MBeans types

1. **Configuration MBeans** (Rhombus/Orange) - These MBeans are the core of the management performed.

   (a) **Monitoring** – This MBean is the main MBean which registers all the other MBeans. This one defines the current state of the instrumentation. It allows the configuration of several thresholds such as times, size of files, etc... and enables the subscription of notifications according to these thresholds. In addition, one can pause/resume and reset statistics invoking its methods.

(b) **Scheduler** – This MBean configures the date when the instrumentation is launched for the first time, and the interval for writing to the database. It provides also some utilities to freeze the statistics after writing to the database and before reseting.

2. **Statistics MBeans** (Rectangle/Blue) — These M(X)Beans are used to extract and monitor information concerning statistics objects. The Client and Machine MBeans are the only MBeans which are not registered automatically when the application is launched. In fact, since the number of clients and machines is numerous, a utility MBean is used to allow any administrator to register any of these MBeans dynamically on the fly. The system, client and machine statistics are MXBeans [30]. MXBeans provide a convenient way to bundle related values together without requiring any configuration from the party who access them. In this way, new properties can be easily added to instrumentation.

   (a) **System** – This is a MXBean which provides statistics about the overall system.

   (b) **Client** – This resource gives information about a specific client. (This type of MXBean must be registered with the manager MBean). One can register as many of these MXBeans as the number of clients using the system.

   (c) **Machine** – This MXBean gives information about a single client in a specific machine. (These manageable objects must also be registered with the manager MBean). One can register as many of these MXBeans as the number of single clients from different machines.

   (d) **Properties** – While System/Client/Machine MXBeans are used to select a specific entity and monitor all the properties about it, this last type of MBeans does the opposite. It takes a property and shows the result for all the clients or machines. The results can be displayed in ascending or descending order, the number of results to display can also be limited. There are as many MBeans of this type as the number of properties defined. Additionally, each one of these MBeans have a number of attributes proportional to the number of statistics measurements derived.

3. **Utils MBeans** (Hexagon/Violet) — Thes manageable resources provides some utilities.

   (a) **Manager** – The manager MBean is used to register/unregister dynamically statistics MXBeans (Rectangle/Blue) corresponding to either clients or machines. This feature is very useful. If those MXBeans would be registered as soon as a new client/machine uses the system, the number of MXBeans would

be extremely high and the majority of them useless. This Manager MBeans enables the possibility for the administrator to register/unregister at any time the client or machine MXBeans that he actually wants to monitor. This MBean validates the client / machine names entered by the administrator to ensure the existence of the corresponding statistics. If it is not a case, an explicit exception is thrown.

(b) **Comparator** – This MXBean is used to add statistics in a structure with the aim to compare values and generate some graphs from different clients/machines or even compare with the overall system statistics.

6.3.3 Logging Database

Statistics of interest created in memory are sent to the database periodically (currently on a daily basis). The database groups the statistics from the different servers and environments and works out global, consistent statistics. The Logging System database is intended to store heterogeneous times series data. Statistics measurements are also time series data, since every day new values are derived. Therefore it is a good idea to store the statistics measurements in the Logging database as well. A statistics measurement is basically a numeric value, the only thing that must be done in advance is the registration of Logging variables for the different measurements. The names of the variables needs to conform to the naming convention defined for all Logging variables. The Logging database brings many advantages. Firstly, the persistence of records is guaranteed over the lifetime of the LHC. Secondly, the TIMBER tool built on top of the database can be used to extract, analyse and correlate Logging variables in an easy way over extended periods of time. The Logging Database schema, the variable mappings and some SQL code corresponding to the work performed can be fully analysed in Annex C.

6.3.3.1 Considerations

During the implementation, important considerations had to be taken into account concerning the architecture and unpredictable events:

1. **The architecture** — The Logging Project has a distributed architecture.

   (a) There are several servers and environments (environment for testing and environment for production).

   (b) Clients can swap from one server to another at any time.

   (c) New clients can pass validation tests (from test environment) and go to a production environment.

   (d) The same server can run both environments.

(e) Each server has a number of requests independent from the other servers.

2. **Unpredictable events** — When these events occur, all the data in memory is lost, therefore the statistics written don't correspond to the total period...

   (a) The application can theoretically be redeployed at any time during the day.

   (b) The container where all the applications are running can be restarted.

   (c) The application server can crash, or need patching or upgrading and thus needs to be restarted.

All these details need to be taken into account in order to reach global and consistent statistics. They are global because the statistics of the system and of clients duplicated in servers/environments have to be grouped. Then it has to be consistent, because the number of requests differs from one server to another. Writing these statistics directly to the corresponding Logging variables records in the database would be extremely difficult to implement and inefficient.

6.3.3.2 Instrumentation table

The solution implemented is based on a temporary instrumentation table. This table facilitates the implementation and the efficiency of the operations to work out global and consistent statistics.

This table is only stored in the production database, because of two reasons. Firstly, it ensures the persistence of data for the LHC's lifetime (the TEST database doesn't). Secondly, regardless of the source enviroment the statistics measurements are considered to be production data. To ensure the the completeness of the statistics they have to be written all in the same database.

Each environment has only one database connection defined, i.e. the test environment can only write in the test database and the production environment only in the production database. To solve this problem a synonym was defined in the test database which maps directly to the corresponding table in the production database. The 6.7 shows how the architecture is defined.

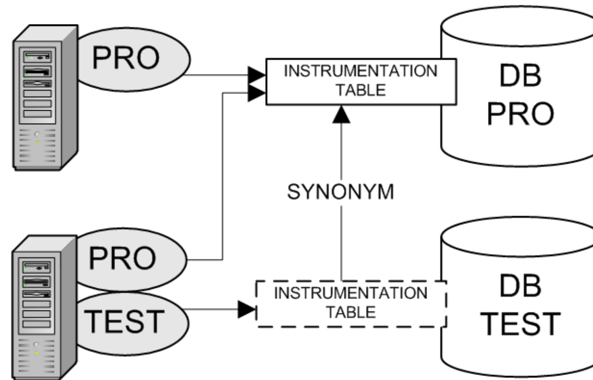

Figure 6.7: Table synonym

The instrumentation table has a primary key made of three fields:

- **SERVER/ENVIRONMENT** — Defines the server and environment, from which the measurement statistics come from.
- **USER_NAME** — Corresponds to the name of the client or to the system if the statistics are referent to the overall system.
- **TIME** — This field contains information about the time the instrumentation begun

This table has also one field for each measurement. The names of these fields have the particularity to have the same name as the suffix of the corresponding mapping Logging variable.

Finally, there is a field for a *scale up* ratio. Although it does not occur frequently, the application can be redeployed. The redeployment of an application causes the loss of any data kept in the main memory referent to the application. Hence, statistics measurements are also lost. It is important to notice, that statistics are used to give a perception of the system's performance and clients behaviour and therefore it is not necessary to have measurements extremely accurate. As long as the proportions are kept, the perception is the same. Hence, a scale up ratio is defined according to the time the statistics are actually running and the time statistics are periodically restarted and written in the database. For instance, if an application is deployed at midday and the measurements are written everyday at midnight the ratio is 0.5. If a client makes 1000 requests during that time (12hours), the value scaled up is (1000/0.5) 2000 requests which represent more reasonably a value of requests for the whole day.

This table also has the advantage to keep statistics gather by a specific server or environment for some days. Currently statistics of this table are kept for 30 days.

6.3.3.3 Instrumentation view

A view was created to group the information contained in the instrumentation table and to work out the statistics values based on the number of requests made on each server - taking into account the scale up ratio. The view has exactly the same columns has the underlying table, with the exception of the server/environment and the scale-up-ratio columns. With the global and consistent information contained in the view it is possible to write directly the time series data for the corresponding Logging variables.

6.3.3.4 Job

A database job is scheduled to periodically select the information from the view and write the equivalent numeric time series data for the corresponding Logging variables. It is executed every day at a fixed time.

6.4 Testing

The reliability of the statistics process is crucial. The instrumentation was subjected to many tests in a development environment, the most important tests are described below:

1. **Memory resource** — One tested the additional memory resource consumed by the instrumentation based on memory. The test was performed with three times the number of known clients and machines, because it is expected that more clients will use the system in the future. Even like that the additional memory usage was not significant.

2. **Time consuming** — The time spent to derive new statistics measurements with the instrumentation based on memory was also measured. Again, the time and CPU usage was not significant.

3. **Accelerated simulation** — This test consisted of generating random values for every property (validation time, parsing time, size file, etc...) from different client and machines. A minimum and maximum was defined differently for each property, so that one could know if the statistics were correct or not. Only once the accuracy of the output was yielded, one could pass to the next test.

4. **Simulation with files** — An application was created to send files to the servlets. The content of each file was already known. Through the JMX browser one could notice if the statistics produced were correct. Once this step was achieved, one could perform the following test.

5. **Threshold and notifications** — The notifications were also tested in case of a measured property value exceeding a defined threshold. In fact, one could be informed by the OEM of new notifications.

6. **Writing in the database** — Many writings were performed from different application servers and environments at the same time and with different periods. One tested with periods of 1 minute, 10 minutes, 1 hour and 24 hours.

7. **Global consistent statistics** —- Several tests were performed in the database to test the consistency of the global statistics. Many tests were made involving the scale up ratio, the case in which a client swaps from environment / server to another and the disproportion between the load of the application servers.

Only once all the tests were successfully passed, one could deploy the application in a test environment. Finally, as everything went fine in the test environment, the application could be deployed in production.

6.5 Interfaces and direct results

6.5.1 Specific requests

The integration of a spreadsheet application with the file level instrumentation makes it possible to select specific requests for a short period of time and analyse many interesting features. The figure 6.8 shows a peak on the writing time. The size file kept nearly the same, which means that the number of records to write is approximately the same. Therefore, if the writing time has raised and the number of records were more or less the same, there was probably a database related issue during that period.

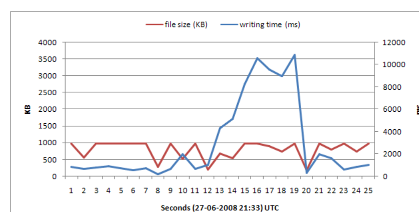

Figure 6.8: Database problem?

6.5.2 JMX browser in OEM

The statistics are integrated with manageable resources for any statistics measurement available in the data structure. In fact any JMX console can be used to monitor the values. Since the OEM is the unique interface that is used to manage the Logging System it makes sense to use it. However the OEM 10*g* does not offer yet the possibility to create

custom charts. Some prototypes (Annex B) have been envisaged in order to overcome some limitations.

Duplicate data

Interacting with the JMX browser of the OEM console is rather intutive. The 6.9 shows the ratio of merged requests (rectangle). Clicking on the *RequestsMergedGroups* (circle) it is possible to see an ordered list of clients with merged data. Many other information can be accessed from the interface.

Name △	Description	Access	Value
AllGroups	AllGroups	R	AtlasCryoLog CL165-cryo CMSLog...
AllUsers	AllUsers	R	AtlasCryoLog@ At...
NumberOfGroups	NumberOfGroups	R	10
NumberOfUsers	NumberOfUsers	R	20
RatioMerged	RatioMerged	R	0.030377995
RequestsGroups	RequestsGroups	R	javax.management.openmbean.Com...
RequestsMergedGroups	RequestsMergedGroups	R	javax.management.openmbean.Com...
RequestsMinute	RequestsMinute	R	17
Statistics	Statistics	R	javax.management.openmbean.Com...
Total_Minutes_Processing	Total_Minutes_Processing	R	18367
Total_Requests	Total_Requests	R	653236
Total_Size_MB	Total_Size_MB	R	185257

Figure 6.9: Interaction

Size of file

In Figure 6.10 on page 55, it is possible to see the representation of an MBean attribute and a graph corresponding to some of the attribute values. In this specific scenario the attribute represents the size of the last file sent by the clients. By selecting the checkbox it is possible to monitor this value in a graph. The graph shows that the client *cryo* send many times a file with the max limit defined while the user *LowBetaLogging* does not reach that limit so often. In fact, it was quite surprising to see many requests with the maximum size file reached.

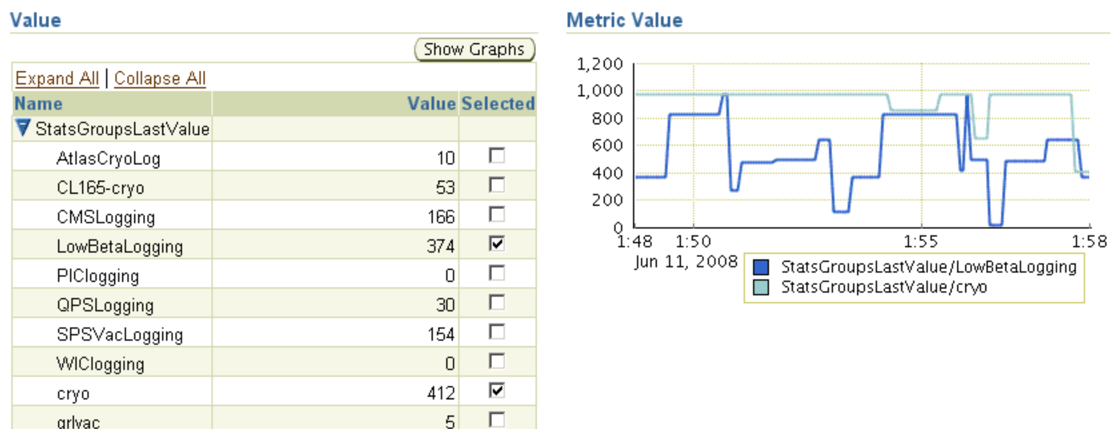

Figure 6.10: Graphs in OEM

6.5.3 TIMBER and historical data

The TIMBER tool (1.3.3) built on top of the Logging Database provides a user-friendly interface to select and analyse time series data. In Figure 6.11 on page 56, a graph shows the number of records sent each day in the production server, during the month of June 2009.

A Logging variable is mapped for each kind of measurement statistic:

- **DATA_NUMERIC** — SYS.ALL.SUMMARY:DN (in green)
- **DATA_NUMERIC_STATUS** — SYS.ALL.SUMMARY:DNS (in yellow)
- **DATA_VECTOR_NUMERIC** — SYS.ALL.SUMMARY:DVN (in orange)
- **DATA_STRING** — SYS.ALL.SUMMARY:DS (in red)

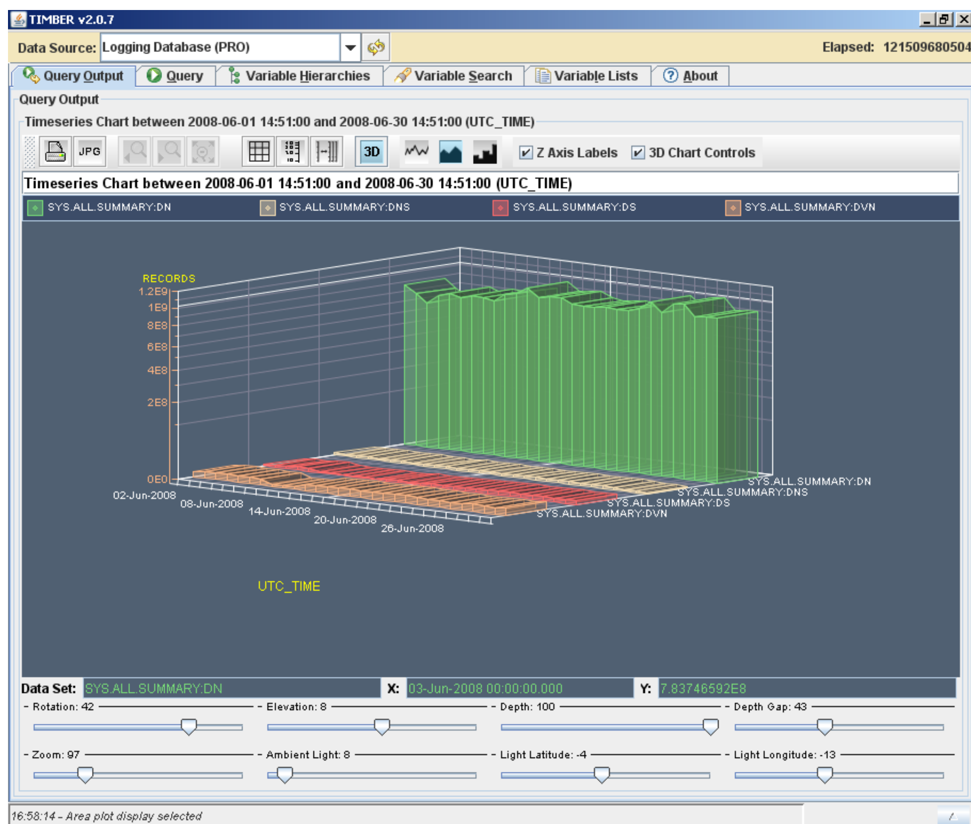

Figure 6.11: TIMBER interface

One can clearly see the disproportion of DATA_NUMERIC records compared with the other data types. In fact, this data type by itself can reach 1 billion records in only one day. DATA_VECTOR_NUMERIC records is the second most used data type, with

a value between 1 and 2 million records each day. Finally, the other data types have below 1 million records per day .

From that tool one can select any Logging variables and compare their data over time. The Logging variables refer to global statistics for clients and system, i.e., the statistics of each machine or from different servers have been grouped. In addition, administrators can write SQL queries directly against the database to analyse more specific instrumentation values, such as temporary statistics measurements from a single server (statistics kept for 30 days).

6.6 Chapter Overview

This chapter has shown three levels of abstraction to instrument client behaviour and system performance. Each one of the levels has its own purposes, advantages and disadvantages. This instrumentation is extremely valuable for the system. In a test environment it helps quickly identify how new clients behave. In fact, it will replace many previous auditing tasks that were resource intensive or not easy to exploit. Clients need also to be instrumented in production environments since behaviours are subject to change over time. In addition, the instrumentation is a tool that can be used to analyse the evolution of system performance and indentify potential for enhancements or the need for hardware upgrades. The next chapter focuses on enhancements that can be done after some interesting instrumentation analyses.

Chapter 7

System analyses for performance tuning

In this chapter some important analyses related to system performance are described. Firstly, direct analysis shows the times spent on each operation. Then, two of them concern the distribution of data within the client's data loading files; and the importance of not having duplicate data. These analyses could be carried out due to the newly implemented instrumentation. Also, a real case could be performed to improve the system performance.

7.1 Operational times

The instrumentation based on memory offers a straightforward approach, through the JMX console to select some statistics of interests. The following measurements were extracted after some hours of instrumentation in one of the production servers. The Table 7.1 on page 59 represents the average value of times spent on each operation (validation, parsing, checking and writing). Then Table 7.2 on page 60 shows on average the number of variables and records sent in each XML file. It was also extracted that 3.15 % of the files contained duplicate data.

After consulting database statistics for the previous days for the same production server, one could notice that the average times changes a little over the days, but the percentage of time used by each operation keeps nearly the same.

One thing that could be immediately noticed was the efficiency of the checking operation. In fact, it is the faster operation with an average of only 15ms per requests, which

	validation	parsing	checking	writing
Average (ms)	478	373	15	639
Average (%)	31.8 %	24.8 %	1.0%	42.5%

Table 7.1: Operational Times

	Variables	Records
Average number in a file	110	6878

Table 7.2: Variables and Records

means 1% of the total time. One could think that database access could require a much higher value than 15ms to check 110 variables on average, but statistics measurements confirmed the usage of good techniques already implemented.

The writing time takes 42.5% of the total time. This value is in fact the highest, however writing around 6878 records in 639 ms, means that on average each record is written in less than 0.1 ms. This is quite good considering the fact that records are sent from the application server and 3.15 % of requests contained duplicate data. Once again this shows the good usage of techniques such as sending records in batch and using statement caching.

As far as validation and parsing are concerned, the values seem quite high. The two operations use more than 50 % of the time in total. These operations may be much more analysed to identify any manner to enhance them. In fact, if the system could trust enough all the clients, the validation process wouldn't be necessary and 31.8% of the time could be saved. However, since the clients behaviour can change over the time, this solution still has to be discussed.

7.2 Data Distribution

The analysis made in this section is related specifically to database access. As described in Section 6.1, the data loading servlet makes two accesses to the database:

- **The checking process** — Simultaneously checks if the variable names contained in the client's data loading file exist in the database and retrieves their unique identifiers (necessary for writing to the database).
- **The writing process** — Writes the records (time series data) corresponding to the variables in the database.

It is important to mention that only requests with no duplicate data were selected to perform this analysis.

7.2.1 Checking Time

The checking operations reads a number of variables, therefore it was expected to have a correlation between the time spent on checking and the number of variables. In order to analyse this situation, a sample of 3000 requests have been captured through the file instrumentation. After processing that file, a scatter graph has been created. The Figure

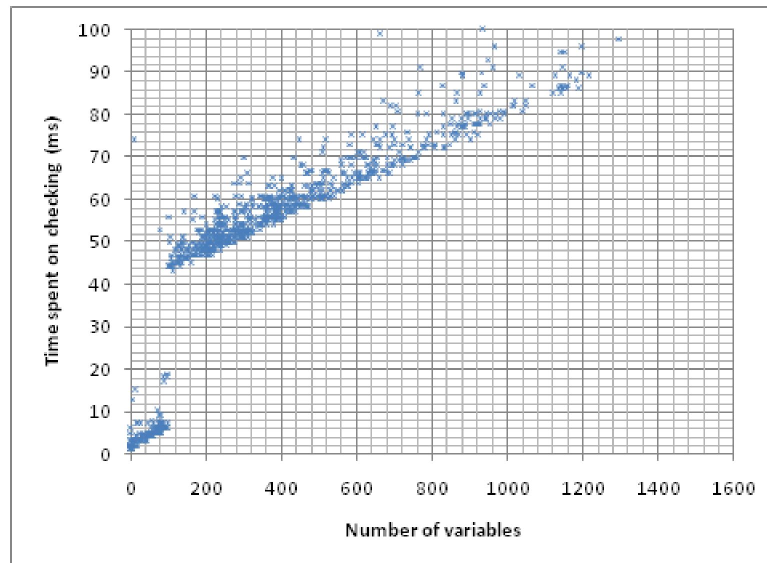

Figure 7.1: Checking time vs Number of variables

7.1 on page 61 shows the graph, where the X axis is the number of variables and the Y axis is the checking time in milliseconds (ms). The graph showed an interesting event: the relation between variables checking times is directly proportional, but two linear trend lines could be drawn. It seems that there is a frontier between number of variables inferior and superior to 100. For a number of variables inferior to 100, the operation takes approximately less than 8ms. However, when the number of variables is greater than 100, the time to check jumps more than 35 ms. It was later discovered that this was due to a change in the database query execution plan when the query optimizer detected that the number of variables crossed the threshold of 100.

7.2.2 Writing Time

In case of process writing the records, the time was expected to be proportional to the number of records, because it is reasonable to predict that the more records are written, the more time the operation will take. Using the same sample, another graph was created. The Figure 7.2 on page 62 shows the graph with the number of records and the corresponding time to write. In this case, no surprises were found. In fact, a trend line can be drawn which represents the average writing time in function of the number of records. At present the current time is around 0.035 ms per record.

An interesting and very important aspect needs to be taken into account. How is the writing process influenced by the ratio of the number of Logging variables to the number of records contained in the file? The time spent can be different to write many records for the a few variables, or to write only a few records for many variables. To analysis this relation, two ratios were defined. A first ratio α, based on the previous linear correlation

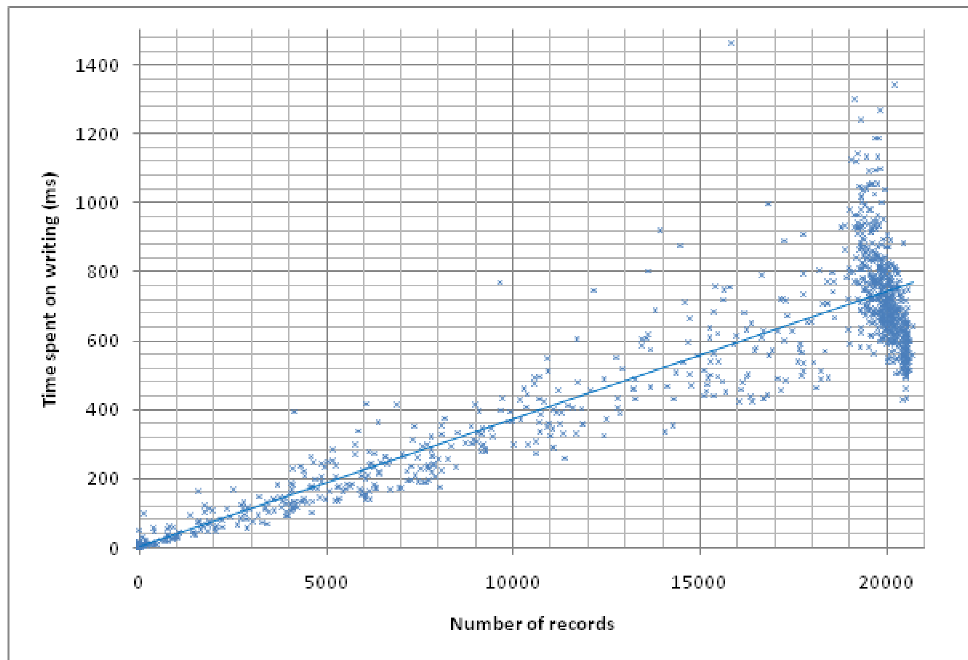

Figure 7.2: Writing time vs Number of records

coefficient analysis, can define the writing time for a single record. The second ratio β defines the proportion of records in a file with the number of variables. Having no previous idea of the relation between both variables, it was really interesting to see the results. In Figure 7.3 on page 63, shows the results in a logarithmic scale between both value. At a first glance, it's possible to notice the higher is β , the lower is α, which means that having a file with many records and few variables is better than having a file with many variables and only a few records. A file having less than 10 records per variable, has a writing time very unstable, it can vary from 0.1ms to 30ms per record. However, a file having $\beta > 10$ results in much more efficient and constant writing time. The chart shows that there is no value below 1ms per record, actually most of the values are concentrated between 0.03ms and 0.1ms. After that, when β reaches approximately 130, the optimum writing time has been reached, which is approximately around 0.027 ms per record.

In an optimal scenario where the ratio between the number of records and variable would be higher than 130, the average writing time would be 0.028ms per record, which is close to the expected optimum for this system. If, it could be possible to receive all the data with a ratio of 130 records per variables, the writing time would be $(0.035/0.028 = 1.25)$ 25% faster.

7.2.3 Interpretation

These two checking and writing analyses can be interpreted in a complementary manner. Checking analysis says that the time is more efficient with number of variables inferior to

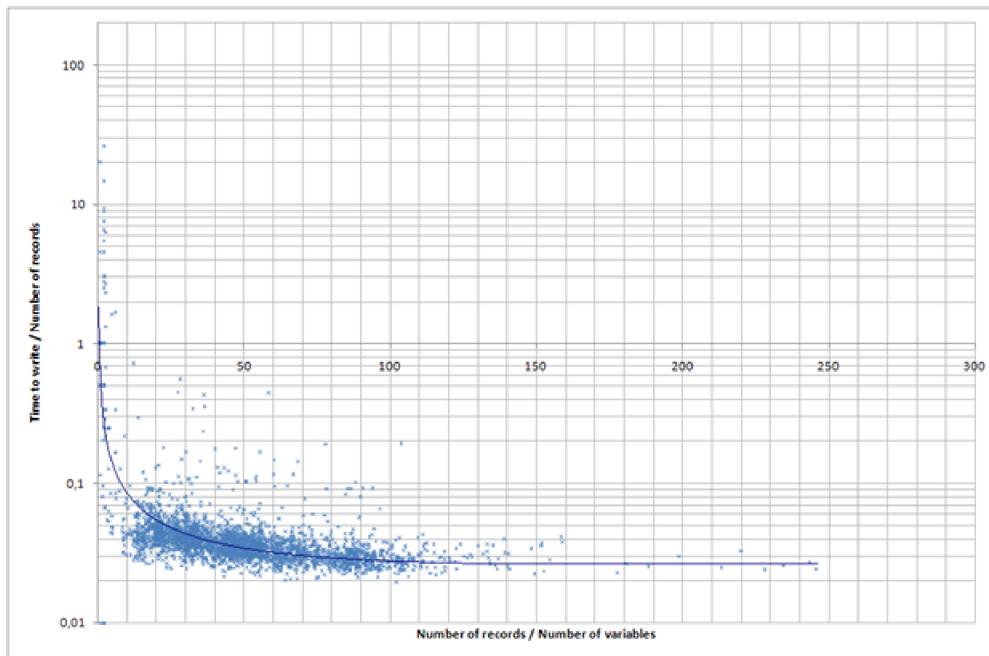

Figure 7.3: Ratio Variables/Records vs Writing Time

100. Then, the writing analysis shows that at least 10 records per variable should be sent. If it could be possible to have more than 130 records per variable it would be very close to the optimum point. If a client could send less than 100 variables in each file, with at least 10 records in each one of them it would be good. The optimum would be closely reached if the clients could be able to send at least 130 records per variable and the number of variables would be lower than 100 in each data loading request.

7.3 Duplicate Data

An aspect already mentioned in this document is the reference to duplicate data. When the OAS receives data loading files containing duplicate time series data a MERGE statement must be performed, instead of doing a direct operation with an INSERT. Through the conducted analysis, it was possible to know, how much extra time a MERGE statement would take. The Figure 7.4 on page 64, shows the trend lines with the following correlation coefficients:

- INSERT $= \frac{700}{20000} = 0.0350$ (ms per record)
- MERGE $= \frac{2850}{20000} = 0.1425$(ms per record)

The ratio between both is approximately 4. It means that a MERGE statement takes 4 times longer than an INSERT.

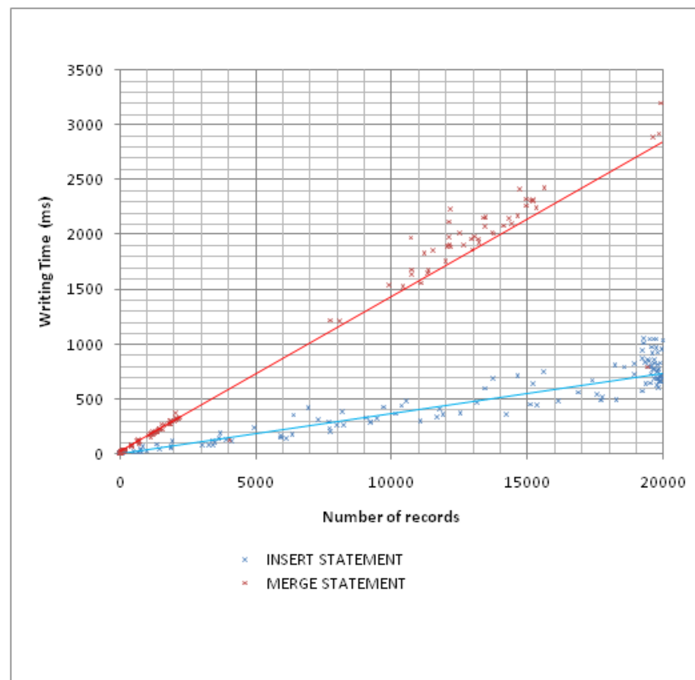

Figure 7.4: INSERT vs MERGE statement

For the total file processing time, this can be represented by a graph showing the file size and the total time spent on in. The results presented in Figure 7.5 on page 65 can be interpreted as follows:

- NO DUPLICATE DATA $= \frac{1900}{976} = 1.947$ (ms per byte)

- DUPLICATE DATA $= \frac{4400}{976} = 4.508$ (ms per byte)

The ratio between both is 2.315. This means that the fact of having duplicate data in a file takes more than two times long per data loading request.

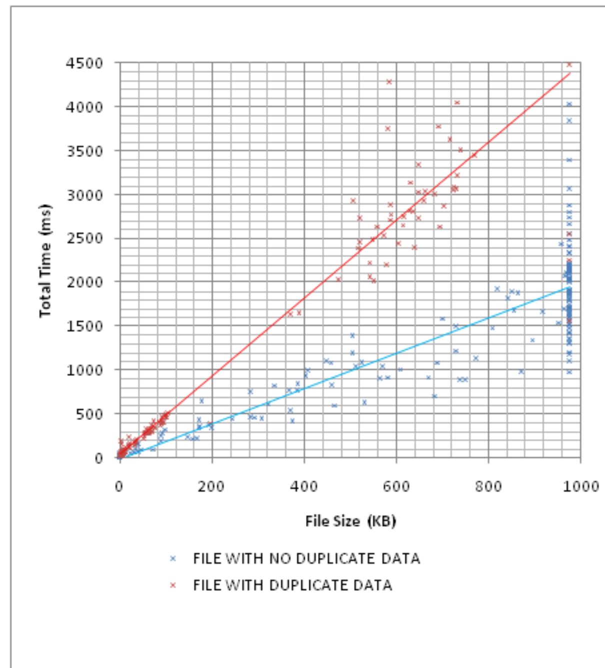

Figure 7.5: File Size vs Total Time

It is important not forget that before a merge statement, the direct operation has been tried (with an INSERT). Therefore, in addition to the merge statement time, the figure takes into account the insert and the resulting rollback operation times.

The instrumentation data stored in the database facilitated a quick analysis of the clients sending duplicate data over the last few days. The results showed that there are some users with a peak of 30% of duplicate data in the Logging System Test environment, and 21% in the Logging System Production environment. Fortunately, most of the clients are well-behaved and have no duplicate data. The global consistent statistics showed a value of 1.4% of duplicate data for the global system.

A Coefficient Time related to the Duplication of Data (CTDD) can be given by the following equation, where PDD is the proportion of duplicate data, TWODD is the time per byte to process a file without duplicate data and TWDD is the time per byte to process a file with duplicate data:

$CTDD = ((1 - PDD) * TWODD) + (PDD * TWDD)$

Therefore the coefficient of the system is currently expressed by:

$CTDD = 0.986 * 1.947 + 0.014 * 4.508 = 1.983$

The optimum CTDD is given when there is no duplicate data at all, which is:

$CTDD = 1.00 * 1.947 = 1.947$

The Performance of the System related to the Duplicate Data (PSDD) can be calculated by the ratio between the actual and optimum CTDD. At present, the PSDD is at 98.16 % ($\frac{1.983}{1.947}$). This value can be considerate good because the whole clients together do not send much duplicate data on average. However, after a sensitivity analysis of the

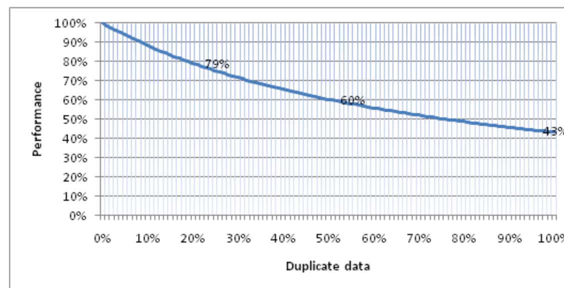

Figure 7.6: Sensitivity analysis

performance showed in Figure 7.6 on page 66, one can clearly notice that the performance decreases significantly with a higher proportion of duplicate data. If 50% of the files contained duplicate data, the PSDD would be only 60%. The worst case happens when receiving all files containing duplicate data, in this case the PSDD would be 43%. This sensitivity analysis emphasises even more that clients must be periodically evaluated to identify wrong behaviour. This evaluation can now be performed easily based on the historic or real-time instrumentation.

7.4 Real case tuning

Like in any other system, the application servers as well as the database have limited resources. To protect the application servers against memory overloading and the database against the increasing table spaces used to rollback operations, the XML files have a limited size. Hence, no clients can send a file bigger than the limit imposed. On the other hand, the clients need to have their data online (in the production database) after a determined period of time, therefore they accumulate records during this period of time and write the information to the XML file. If in case the file is bigger than the limit, they split the file until the size is admissible and send as many requests as the number of files. A limit of size was estimated in order to suit the clients and the system.

Before the instrumentation aspect, there was no clue about how the clients were sending the file. With the instrumentation put in place, it was possible to monitor the size of the last file sent by the clients via the JMX browser. Surprisingly, one could notice that some clients were at almost each request sending the size limit imposed and then a file with a smaller size. This was due to file splitting (if the limit is 1MB, and the file has 1.2MB then the file has to be split in one file of 1MB and another of 0.2MB). It was clear that those clients needed a limit size higher. Therefore the idea of increasing the limit for those clients seemed reasonable to reduce the number of requests.

To see how the system was reacting, a specific client was chosen among those who were often sending files in these conditions and was asked to increase its size limit to double. In fact, one could notice that the number of requests was reduced as expected and the total

time as well, but something much more important immediately jumped out. The ratio of duplicate data decreased 50 times.

The ratio of duplicate data of that client was on average around 2% and suddenly with this changing his new ratio became 0.04%. One could understand that the file splitting was introducing duplicate records among the files. In Section 7.3, it was clearly showed that a request containing duplicate data takes more than two times longer. Since the server memory and the database table spaces didn't grow significantly the same action will be performed for other clients having the same problem. The performance of the system related to duplicate data expects to jump from 98.16% to more than 99% with these incoming changes.

Nevertheless, duplicate data is not only due to file splitting. It is possible to identify from the database instrumentation, that files coming from some clients have never reached the maximum size, therefore they never had to split files. However, the ratio of duplicate data was high for some of those clients, particularly for clients running in a test environment.

7.5 Summary

The system already uses good techniques such as writing database records in batch mode, connection pools and statement caching. However the above analyses show that the system performance could be even more improved. In fact, if every file could have at least 130 records per variable, the data writing process would be 25% faster. It was also possible to identify that the system has a performance of 98.6% related to the number of requests with duplicate data. The actual value is actually quite good, but the sensitivity analysis showed that this performance could decrease dangerously if the proportion of duplicate data increases. Therefore, the ratio of duplicate data of each client must be periodically checked.

Through the memory instrumentation, it was possible to monitor the size of the last file received and notice that some clients needed to send bigger files. The increase of size limit for some clients, caused not only less requests, but also the ratio of duplicate data for those clients decreased significantly.

The introduction of an agent layer in the core of the system could be used to tune the system as well. The information of the records sent by each client is remotely accessible via the MBeanServer. In a SOA (Service Oriented Architecture) approach, a service could access the data of the manageable resources in order to balance optimistically the clients across multiple OAS instances based on a scheduling algorithm. If the data loading activity is distributed the performance of the system definitely increases.

Chapter 8

Conclusion and outlook

The author was given the task of maintaining the security, manageability and high performance on a mission critical operational system. Having analyzed the potential options, the solutions were implemented with an appropriated technology, efficient techniques and in scalable manner to provide a facilitated maintenance for any changes occurring in the next years.

The first aim of the work was related to overcoming problems of obsolescence in order to run the application in the latest supported platform to benefit from the newest enhancements. In fact, the application is currently running in the upgraded version of the OAS in a secured manner. The resources continue to be accessed only by authenticated and authorized clients. The application maintained this security level using the JAAS technology. This technology, used for the both required functions, has showed to be very well appropriated. As far as the manageability is concerned, the functionalities services have also been restored. The parameters can currently be modified with the same interface, the OEM. In the upgraded technology, the OEM provides a complete JMX browser. Since the JMX technology has proof to be really adapted for manageable systems and because the OEM 10g provides a complete JMX browser, this technology was chosen to substitute the context servlet parameters.

This mission-critical operating system, writing approximately one billion records every day, had a very limited diagnostic tool before the current work. Some coarse-grained estimations and heavy operations had to be performed in the Logging Database to extract some information about what was going on. An instrumentation based on three levels of abstraction has been fully designed and implemented with the aim to solve this problem and identify potential to reach a higher performance. Each level has different its own purposes, pros and cons. The first level based on files makes possible to identify and analyse new patterns and correlation between variables of the system. This information is extremely valuable because it allows knowing how efficient are some operations compared with an optimum level of performance. Moreover, it indicates what and how can be tuned

in order to be close of a top high-level f performance. The second level, based on the main memory, turns possible to read in real-time many statistics measurements, monitor in a comprehensive way the incoming new values and be notified in case of unexpected values. This approach was implemented in a manner that overcomes many challenges of the first level, such as the operational time, the analysis facilities and the required storage. There is a clear differentiation between three types of statistics in this level, the overall system statistics, the single client statistics, and statistics for a single client from a specific machine. This important discrimination allows exactly knowing who are the most active clients and the misbehaving of each one of them. Since the system performance changes over time, as does the client behaviour, a third level of instrumentation was defined, the database. Statistics summaries are written to the Logging System database on a daily basis, since they come from several servers, they are gathered and global consistent statistics are derived mapped to Logging variables. This smart integration with the variables in the Logging database has a duplicate advantage. On the one hand it allows extracting statistics measurement directly with the TIMBER tool, which was built on top of the database to provide a user-friendly way to analyse variables. On the other hand, the Logging Database ensures the persistence of data, which will permit to compare statistics for many weeks, months or even for the entire LHC's time, i.e. 20 years.

The choice of the JMX technology has showed to be a good option not only in the case of configurations in real time, but also for the instrumentation. In addition, the JMX agent layer which let statistics values to be remotely accessible can open the way to many other interesting extensions.

The interpretation of the analyses carried out in Chapter 7, should not be misinterpreted. In fact there is ratio between the number of records and the number of variables contained in a file from which the writing time is close to optimal. The ratio was 130, which means that if, on average, each variable has at least 130 records, the writing would be near the optimal. Unfortunately, there is a limitation between theoretical optimal values and what happens actually in a real case. This scenario could be not feasible in practice, only after some discussion with potential clients one could actually see until how much this ratio can be pushed. Actually, a ratio of 10 records is not as good as a ratio of 130, but the difference is not so meaningful, however below this value the time to write is totally unstable and can reach almost 30ms per record. It would be preferable in a first approach to require a ratio of 10 ratios per client, than actually a ratio of 130 would not be so worth it. But once again, only after some discussions with potential clients, one could actually know if this tuning can be performed.

A similar case happens with duplicate data. The analysis showed that even though the insertion of duplicate data is four times heavier, the performance of the overall system was not so much affected. This was due to the fact that for the overall system, the ratio of

duplicate data was only 1.4%. It can be hard for a client to send no duplicate data at all, so instead of claiming for a perfect system with no duplicate data at all, one must focus on the clients who make the overall system ratio of duplicate data raise and ask the to check their system implementation or configuration in order to improve the overall performance.

With the instrumentation in place, one can also help the clients to tune their system. In fact, an enhancement could be done thanks to it. A simple adjustment of the file size of one client reduced considerably the proportion of duplicate data from an average of 2% to 0.04% for that client. This adjustment will be performed for other clients suffering the same problem.

The system is now running in a supported technology and provides a sophisticated instrumentation. Many other analyses can still be made with the instrumentation to find bottleneck and potential for an always higher performance over the years of the LHC living.

Bibliography

[1] CERN : European organization for nuclear research. http://www.cern.ch/.

[2] LHC - THE LARGE HADRON COLLIDER. http://lhc.web.cern.ch/lhc/.

[3] Brian Cox. An inside tour of the world's biggest supercollider. In *TED2008 conference*, 2008.

[4] LHC experiments. http://lhc.web.cern.ch/lhc/LHC_Experiments.htm.

[5] LHC logging project. http://lhc-logging.web.cern.ch/lhc-logging/.

[6] R Billen and C Roderick. The lhc logging service: Capturing, storing and using time-series data for the world's largest scientific instrument. Technical Report AB-Note-2006-046. CERN-AB-Note-2006-046, CERN, Nov 2006.

[7] ETM professional control. http://www.etm.at/.

[8] Oracle application server 10g. http://www.oracle.com/technology/products/ias/.

[9] Oracle enterprise manager 10g. http://www.oracle.com/technology/products/oem/.

[10] Oracle® containers for j2ee security guide 10g release 3 (10.1.3). http://download.oracle.com/docs/cd/B31017_01/nav/docindex.htm.

[11] Apache log4j. http://logging.apache.org/log4j/.

[12] Apache software foundation. http://www.apache.org/.

[13] ORACLE corporation. http://www.oracle.com.

[14] Java. http://www.java.com.

[15] Ulrich Neumann J.P.Lewis. Performance of java versus c++. University of Southern California, 2004. http://www.idiom.com/~zilla/Computer/javaCbenchmark.html.

[16] Hank Shiffman. Boosting java performance: Native code & jit compilers. http://www.disordered.org/Java-JIT.html.

[17] David Murphy. Jdbc connection pooling best practices, 2006. http://www.javaranch.com/journal/200601/JDBCConnectionPooling.html.

[18] Jamie Jaworski and Paul J Perrone. *Java security handbook*. Sams, Indianapolis, IN, 2000.

[19] Brian Pontarelli. J2ee security: Container versus custom choose the appropriate type of security for your application, September 2004. http://www.javaworld.com/javaworld/jw-07-2004/jw-0726-security.html?page=1.

[20] Danny Coward and Yutaka Yoshida. Java(tm) servlet specification 2.4 final release. Java Community Process, November 2003. http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html.

[21] Scott Oaks. *Java security; 2nd ed.* Writting and deploying secure applications. O'Reilly, Beijing, 2001.

[22] Orion server. http://www.orionserver.com.

[23] Heather Kreger, Ward Harold, and Leigh Williamson. *Java and JMX: building manageable systems*. Addison-Wesley, Reading, MA, 2003.

[24] Helmut Reiser Alexander Keller. Dynamic management of internet telephony servers: A case study based on javabeans and jdmk. http://ieeexplore.ieee.org/iel5/6433/17161/00792057.pdf?arnumber=792057.

[25] Java Management Extensions (JMX) Best Practices. http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/best-practices.jsp.

[26] Juha Lindfors and Marc Fleury. *JMX: managing J2EE with Java management extensions*. Sams, Indianapolis, IN, 2002.

[27] Ian Sommerville. *Software engineering; 4th ed.* International computer science series. Addison-Wesley, Reading, MA, 1992.

[28] Eamonn McManus. Defining mbeans with annotations. http://weblogs.java.net/blog/emcmanus/archive/2007/08/defining_mbeans.html.

[29] Oc4j 10g (10.1.3) how-to: Create simple jmx mbeans, 2006. http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-simplembeans/doc/readme.html.

[30] Eamonn McManus. What is an mxbean?, 02 2006. http://weblogs.java.net/blog/emcmanus/archive/2006/02/what_is_an_mxbe.html.

Abbreviations

AB Accelerator and Beams department at CERN

ALICE A Large Ion Collider Experiment - CERN experiment

ATLAS A Toroidal LHC ApparatuS - CERN experiment

CMS the Compact Muon Solenoid - CERN experiment

CO Control group at CERN (AB department)

DM DataManagement section at CERN (AB department and CO group)

GB GigaByte

HTTP Hypertext Transfer Protocol Overview

JAAS Java Authentication and Authorization Service

JDBC Java Database Connectivity

JNDI Java Naming and Directory Interface

JVM Java Virtual Machine

K Kelvin SI base unit of thermodynamic temperature

LHC Large Hadron Collider

LHCb the Large Hadron Collider beauty - CERN experiment

LHCf the Large Hadron Collider forward - CERN experiment

OAS Oracle Application Server

OC4J Oracle Container For Java

OEM Oracle Enterprise Manager

OOL Object-Oriented Language

PL/SQL Procedural Language/Structured Query Language

PLC Programmable logic controller

PVSS The PVSS object-oriented process visualization and control system developed by ETM

SCADA Supervisory Control And Data Acquisition - an industrial control system

SNMP Simple Network Management Protocol, more information in http://ietf.org

TB TeraByte

TOTEM TOTal Elastic and diffractive cross section Measurement - CERN experiment

UTC Coordinated Universal Time

XML Extensible Markup Language

Appendix A

XML File

The XML file contains time series data, for each record there is a value at a moment in time. The following XML code shows the basic representation of a file, with a total of 5 variables. It is possible to see values of the 21st of June 2008 from 00:15:00 until 00:15:05. The file contains two DATA_NUMERIC variables (N_01 and N_02), the first one with 4 records and the second one with only 1. Then there are one DATA_VECTOR_NUMERIC variable (VN_01) with 2 records. Next, comes a DATA_NUMERIC_STATUS (NS_01) variable, with 2 records also. Finally, the last variable is a DATA_STRING (S_01) and has also only one record. This file has only 10 records, the average number of records contained in a file is around 7000.

```
<?xml version="1.0" standalone="no"?>
<archiver_data xmlns="..."
          xmlns:xsi = "..."
          xsi:schemaLocation="...">
        <numeric variable="N_01">
                <row tstamp="2008-06-21␣00:15:00" value="1"/>
                <row tstamp="2008-06-21␣00:15:05" value="2"/>
                <row tstamp="2008-06-21␣00:15:03" value="3"/>
                <row tstamp="2008-06-21␣00:15:04" value="4"/>
        </numeric>
        <numeric variable="N_02">
                <row tstamp="2008-06-21␣00:15:04" value="4"/>
        </numeric>
        <vectornumeric variable="VN_01">
                <values tstamp="2008-06-21␣00:15:02">3.141 34.9 3 7</values>
                <values tstamp="2008-06-21␣00:15:03">3.147 37.2 3 7</values>
        </vectornumeric>
        <numstatus variable="NS_01">
            <row tstamp="2008-06-21␣00:15:02" value="80" status="GOOD"/>
            <row tstamp="2008-06-21␣00:15:05" value="76" status="BAD"/>
        </numstatus>
        <textual variable="S_01">
            <row tstamp="2008-06-21␣00:15:02" value="80"/>
        </textual>
</archiver_data>
```

Appendix B

Prototypes

Some prototypes have been created to explore alternatives to the OEM interface, since this one is still a bit limited as far as graphics are concerned. Two possibilties were envisaged to monitor values: The creation of a custom web application or the development of JConsole plugin. No one of these prototypes was continued because of two reasons. The creation of a complete solution and its mainteance for a long term was not worth it. The OEM has always been the unique interface to manage the Logging System application, therefore it is better to wait for enhanced versions of OEM, instead of having two separated interfaces.

B.1 Custom web application

A custom web application can display any kind of data in a very useful way. The web application can access remotely or locally (if running in the same container) the statistics measurements. In B.1, the values are accessed remotely and bar charts are created.

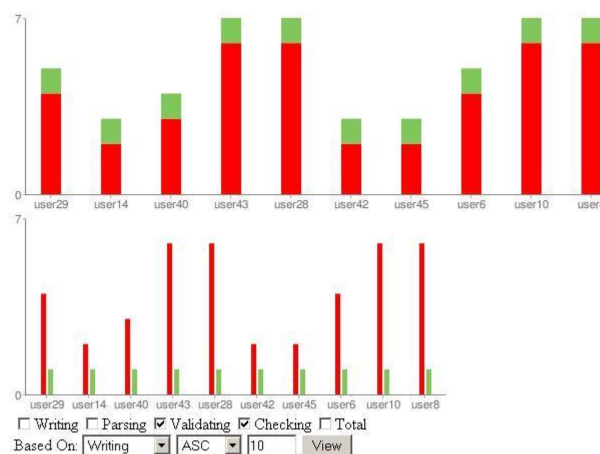

Figure B.1: Custom web application

B.2 JConsole

The Java Monitoring & Management Console provides an API to create custom tabs. In B.2, it is possible to have an idea of how the plugin could look like. The two pie charts represent the operation times of the overall system and a single client. The values are accessed remotely through the agent layer in the OAS.

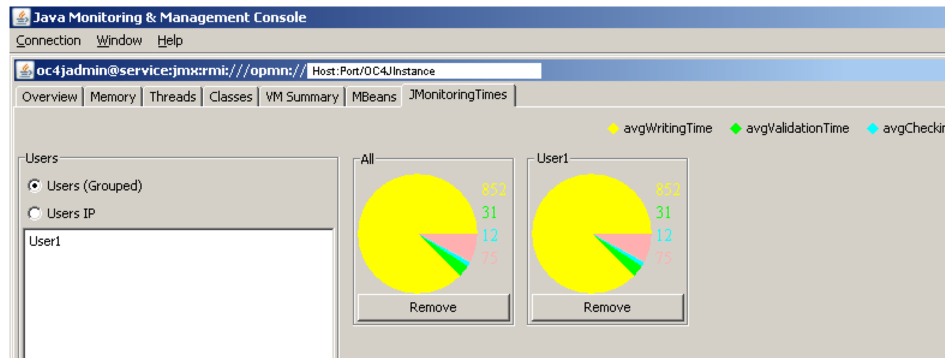

Figure B.2: JConsole plugin

Appendix C

Logging Database

C.1 Schema of the Logging database

In C.1, the diagram of the Logging System is represented.

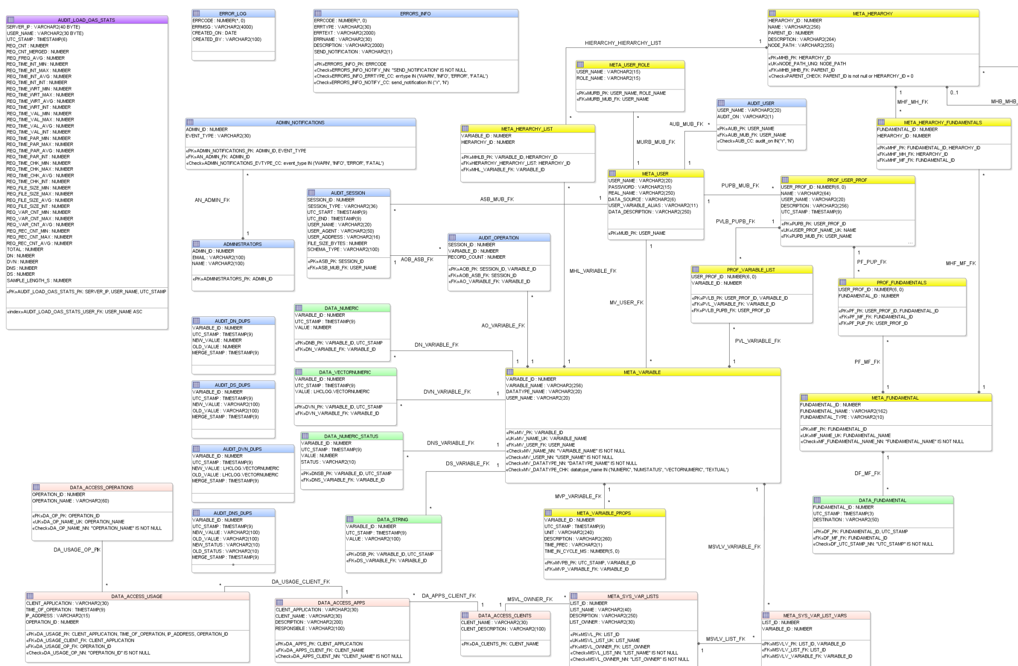

Figure C.1: Loggin data model

There are 5 kind of tables represented by 5 colors:

- **Yellow** — This color represents the metadata tables, which contain the definitions of variables – the entities for which time-series data will be logged, the lists of users of the system, the mappings between users and variables, and some means to group variables for use during data extraction.
- **Green** — Where time series data is stored. Time series data consists of an identifier

– saying what data is being logged. A timestamp – saying when the data was acquired. And the value valid at the time of data acquisition. These tables are index-organized, and range partitioned. With a dedicated tablespace per range – which has the administrative advantage of marking the tablespace 'read-only' once its range has been passed, and thus removing the need for frequent back ups of the tablespace. It is also worth mentioning the use of timestamp data type which allows to stamp the data with up to nanosecond precision. Since LHC measurements are dealing with scientific data, microsecond precision is often required. There is one table for each of the five data types supported by the system:

  - **DATA_NUMERIC_STATUS** — Data with an additional status field – used to qualify the data.
  - **DATA_STRING** — It is data based on a string.
  - **DATA_NUMERIC** — Numeric data, which accounts for the majority of time-series data.
  - **VECTOR_NUMERIC_DATA** — which stores data in a user defined type which is a an array of numbers – this is used for storing 2 dimensional data.
  - **DATA_FUNDAMENTAL** — Data which represents certain events within the machine.

- **BLUE** — In blue are the audit and error tables, which store error definitions, error logs, administrator notifications, and audit data. The introduction of the instrumentation feature added a new table to this schema (AUDIT_LOAD_OAS_STATS). In turn, the previous audit tables don't need to be used anymore because the instrumentation aspect makes things much more efficient and complete.

- **PINK** — The pink tables store the data which defines the end-user applications that have permission to extract data from the Logging database, along with some audit data about their usage.

- **VIOLET** — The violet table (AUDIT_LOAD_OAS_STATS) represents the new aspect of the Logging Database: the instrumentation. This table stores statistics measurements from each server and container for 30 days. The job running everyday in the database selects these values and derives coherent and global measurements which will be mapped to Logging variables. The primary key is defined by three attributes: The location of the statistics (which server? which container?), the name of the client/system and the date. There is a field for the scale up ratio, which represents the percentage of the time period used in instrumentation (in case of new deployments). Finally, the rest of the fields corresponds to the statistics measurements.

C.2 Variables mapping

The name of statsitical variables has to follow a convention name like the rest of the other variables in the Logging database. Nearly 1 million variables are defined in the production database, therefore it's necessary to organize them according to the convention. The name of the variable is defined as following:

```
SYS.user_alias.user_datasource:suffix
```

The *user_alias*, is a short name given to an user. The *user_datasource* defines where the user comes from. At present, all clients come from the *PVSS* system, except the user system which represents the overall system, this one is defined by the data source *SUMMARY*. The *suffix* of each one the statistical variables is represented in Tables C.2,C.4,C.6,C.7,C.8.

Variable suffix	Description
REQ_CNT	Number of data loading requests sent by the user during the time period
REQ_CNT_MERGED	Number of data loading requests sent by the user, which cause a MERGE during the time period
REQ_FREQ_AVG	The average frequency of data loading requests sent by the user during the time period

Table C.2: Counters - Logging variables suffix 1

REQ_TIME_VAL_MIN	Minimum time to validate XML files sent by the user during the time period
REQ_TIME_VAL_MAX	Maximum time to validate XML files sent by the user during the time period
REQ_TIME_VAL_AVG	Average time to validate XML files sent by the user during the time period
REQ_TIME_VAL_INT	Total time to validate XML files sent by the user during the time period
REQ_TIME_PAR_MIN	Minimum time to parse XML files sent by the user during the time period
REQ_TIME_PAR_MAX	Maximum time to parse XML files sent by the user during the time period
REQ_TIME_PAR_AVG	Average time to parse XML files sent by the user during the time period
REQ_TIME_PAR_INT	Total time to parse XML files sent by the user during the time period

Table C.4: Valiation and Parsing time - Logging variables suffix

REQ_TIME_CHK_MIN	Minimum time to check variable ownership and existence for data sent by the user during the time period
REQ_TIME_CHK_MAX	Maximum time to check variable ownership and existence for data sent by the user during the time period
REQ_TIME_CHK_AVG	Average time to check variable ownership and existence for data sent by the user during the time period
REQ_TIME_CHK_INT	Total time to check variable ownership and existence for data sent by the user during the time period
REQ_TIME_WRT_INT	Total time to write data sent by the user, to the database, during the time period
REQ_TIME_WRT_MIN	Minimum time to write data sent by the user, to the database, during the time period
REQ_TIME_WRT_MAX	Maximum time to write data sent by the user, to the database, during the time period
REQ_TIME_WRT_AVG	Average time to write data sent by the user, to the database, during the time period
REQ_TIME_INT_MIN	Minimum time to write data sent by the user, to the database, during the time period
REQ_TIME_INT_MAX	Maximum time to write data sent by the user, to the database, during the time period
REQ_TIME_INT_AVG	Average time to write data sent by the user, to the database, during the time period
REQ_TIME_INT_INT	Total time to write data sent by the user, to the database, during the time period

Table C.6: Checking, Writing and Total time - Logging variables suffix

REQ_FILE_SIZE_MIN	Minimum XML file size sent by the user during the time period
REQ_FILE_SIZE_MAX	Maximum XML file size sent by the user during the time period
REQ_FILE_SIZE_AVG	Average XML file size sent by the user during the time period
REQ_FILE_SIZE_INT	Total size of XML file sent by the user during the time period
REQ_VAR_CNT_MIN	Minimum number of variables per data loading request sent by the user during the time period
REQ_VAR_CNT_MAX	Maximum number of variables per data loading request sent by the user during the time period
REQ_VAR_CNT_AVG	Average number of variables per data loading request sent by the user during the time period

Table C.7: File and Variables - Logging variables suffix

REQ_REC_CNT_MIN	Minimum number of records per data loading request sent by the user during the time period
REQ_REC_CNT_MAX	Maximum number of records per data loading request sent by the user during the time period
REQ_REC_CNT_AVG	Average number of records per data loading request sent by the user during the time period
DN	Number of NUMERIC records sent by the user during the time period
DNS	Number of NUMERIC_STATUS records sent by the user during the time period
DS	Number of TEXTUAL records sent by the user during the time period
DVN	Number of VECTORNUMERIC records sent by the user during the time period
TOTAL	Number of total records sent by the user during the time period

Table C.8: Records - Logging variables suffix

C.3 SQL scripts

C.3.1 View

The view gather the information contained in the AUDIT_LOAD_OAS_STATS table and works out global coherent statistics measurements for the system and clients. It takes into account the number of requests in each server, the duplication of an user in different servers or containers and finally it scales up the values according to the time a client / system has been instrumented compared with the time a normal period takes (The scale up ratio should be always 1, unless a new deployment of the application occured in the middle of a day and statistics in memory were lost).

```
create or replace view v_audit_load_oas_stats_grouped
AS
 SELECT user_name AS user_name,utc_stamp,
 TRUNC(SUM(req_cnt/sample_length_s)) AS req_cnt ,
 TRUNC(SUM(req_cnt_merged/sample_length_s)) AS req_cnt_merged ,
 TRUNC(SUM(req_freq_avg * req_cnt) / SUM(req_cnt),2) AS req_freq_avg ,
 MIN(req_time_int_min) AS req_time_int_min ,
 MAX(req_time_int_max) AS req_time_int_max ,
 TRUNC(SUM(req_time_int_avg * req_cnt) / SUM(req_cnt),2) AS req_time_int_avg ,
 TRUNC(SUM(req_time_int_int/sample_length_s)) AS req_time_int_int ,
 MIN(req_time_wrt_min) AS req_time_wrt_min ,
 MAX(req_time_wrt_max) AS req_time_wrt_max ,
 TRUNC(SUM(req_time_wrt_avg * req_cnt) / SUM(req_cnt),2) AS req_time_wrt_avg ,
 TRUNC(SUM(req_time_wrt_int/sample_length_s)) AS req_time_wrt_int ,
 MIN(req_time_val_min) AS req_time_val_min ,
 MAX(req_time_val_max) AS req_time_val_max ,
 TRUNC(SUM(req_time_val_avg * req_cnt) / SUM(req_cnt),2) AS req_time_val_avg ,
 TRUNC(SUM(req_time_val_int/sample_length_s)) AS req_time_val_int ,
 MIN(req_time_par_min) AS req_time_par_min ,
 MAX(req_time_par_max) AS req_time_par_max ,
 TRUNC(SUM(req_time_par_avg * req_cnt) / SUM(req_cnt),2) AS req_time_par_avg ,
 TRUNC(SUM(req_time_par_int/sample_length_s)) AS req_time_par_int ,
```

```
MIN(req_time_chk_min) AS req_time_chk_min ,
MAX(req_time_chk_max) AS req_time_chk_max ,
TRUNC(SUM(req_time_chk_avg * req_cnt) / SUM(req_cnt),2) AS req_time_chk_avg ,
TRUNC(SUM(req_time_chk_int/sample_length_s)) AS req_time_chk_int ,
MIN(req_file_size_min) AS req_file_size_min ,
MAX(req_file_size_max) AS req_file_size_max ,
TRUNC(SUM(req_file_size_avg * req_cnt) / SUM(req_cnt),2) AS req_file_size_avg ,
TRUNC(SUM(req_file_size_int/sample_length_s)) AS req_file_size_int ,
MIN(req_var_cnt_min) AS req_var_cnt_min ,
MAX(req_var_cnt_max) AS req_var_cnt_max ,
TRUNC(SUM(req_var_cnt_avg * req_cnt) / SUM(req_cnt),2) AS req_var_cnt_avg ,
MIN(req_rec_cnt_min) AS req_rec_cnt_min ,
MAX(req_rec_cnt_max) AS req_rec_cnt_max ,
TRUNC(SUM(req_rec_cnt_avg * req_cnt) / SUM(req_cnt),2)  AS req_rec_cnt_avg ,
TRUNC(SUM(total/sample_length_s)) AS total ,
TRUNC(SUM(dn/sample_length_s)) AS dn ,
TRUNC(SUM(dvn/sample_length_s)) AS dvn ,
TRUNC(SUM(dns/sample_length_s)) AS dns ,
TRUNC(SUM(ds/sample_length_s)) AS ds
 from audit_load_oas_stats
group by utc_stamp, user_name;
```

C.3.2 Procedure

This procedure runs every day and is responsible to write the records of the Logging statistics variables corresponding to the statistics measurements stored in the AUDIT_LOAD_OAS_STAT table. It's statitic structure based on *when ... case* is preferable than a PL/SQL which would not be so efficient.

```
procedure update_audit_load_oas_stats    IS     BEGIN
 merge into data_numeric d using (
   select variable_id, utc_stamp, value
   from (
    select 'SYS.'||upper(mu.data_source) ||'.'||upper(mu.user_variable_alias) ||':'||suffix as var_name,
      (case
        when suffix like 'REQ_CNT'            then req_cnt
        when suffix like 'REQ_CNT_MERGED'     then req_cnt_merged
        when suffix like 'REQ_FREQ_AVG'       then req_freq_avg
        when suffix like 'REQ_TIME_INT_MIN'   then req_time_int_min
        when suffix like 'REQ_TIME_INT_MAX'   then req_time_int_max
        when suffix like 'REQ_TIME_INT_AVG'   then req_time_int_avg
        when suffix like 'REQ_TIME_INT_INT'   then req_time_int_int
        when suffix like 'REQ_TIME_WRT_MIN'   then req_time_wrt_min
        when suffix like 'REQ_TIME_WRT_MAX'   then req_time_wrt_max
        when suffix like 'REQ_TIME_WRT_AVG'   then req_time_wrt_avg
        when suffix like 'REQ_TIME_WRT_INT'   then req_time_wrt_int
        when suffix like 'REQ_TIME_VAL_MIN'   then req_time_val_min
        when suffix like 'REQ_TIME_VAL_MAX'   then req_time_val_max
        when suffix like 'REQ_TIME_VAL_AVG'   then req_time_val_avg
        when suffix like 'REQ_TIME_VAL_INT'   then req_time_val_int
        when suffix like 'REQ_TIME_PAR_MIN'   then req_time_par_min
        when suffix like 'REQ_TIME_PAR_MAX'   then req_time_par_max
        when suffix like 'REQ_TIME_PAR_AVG'   then req_time_par_avg
        when suffix like 'REQ_TIME_PAR_INT'   then req_time_par_int
        when suffix like 'REQ_TIME_CHK_MIN'   then req_time_chk_min
        when suffix like 'REQ_TIME_CHK_MAX'   then req_time_chk_max
        when suffix like 'REQ_TIME_CHK_AVG'   then req_time_chk_avg
        when suffix like 'REQ_TIME_CHK_INT'   then req_time_chk_int
        when suffix like 'REQ_FILE_SIZE_MIN'  then req_file_size_min
        when suffix like 'REQ_FILE_SIZE_MAX'  then req_file_size_max
        when suffix like 'REQ_FILE_SIZE_AVG'  then req_file_size_avg
```

```
        when suffix like 'REQ_FILE_SIZE_INT' then req_file_size_int
        when suffix like 'REQ_VAR_CNT_MIN'   then req_var_cnt_min
        when suffix like 'REQ_VAR_CNT_MAX'   then req_var_cnt_max
        when suffix like 'REQ_VAR_CNT_AVG'   then req_var_cnt_avg
        when suffix like 'TOTAL'     then TOTAL
        when suffix like 'DN'        then DN
        when suffix like 'DVN'       then DVN
        when suffix like 'DNS'       then DNS
        when suffix like 'DS'        then DS
        else -1
      end) as value,
      utc_stamp
    from meta_user mu
    join audit_load_oas_stats_grouped ag on (mu.user_name = ag.user_name),
    (
      select column_name as suffix
      from user_tab_columns
      where table_name = 'V_AUDIT_LOAD_OAS_STATS_GROUPED'
      and column_name != 'USER_NAME'
      and column_name != 'UTC_STAMP'
    )
  ) oasd
  join meta_variable mv on (mv.variable_name = oasd.var_name)
) nd
on (d.variable_id = nd.variable_id and d.utc_stamp = nd.utc_stamp)
when matched then
  update set d.value = nd.value
when not matched then
  insert (d.variable_id, d.utc_stamp, d.value)
  values (nd.variable_id, nd.utc_stamp, nd.value);

--Deletes rows that have more than 30 days
DELETE
FROM audit_load_oas_stats
WHERE utc_stamp < (sysdate - 30);

COMMIT;
END update_audit_load_oas_stats;
```

Appendix D

MBeans Information

In spite of its excellent manageable capacities, JMX technology does not still provide an easy way to write meta information about MBeans as far as descriptions, attributes, methods and notifications are concerned. The workaround done to overcome this limitation is based on subclassing the *javax.management.StandardMBean* class and overriding the *getMBeanInfo* method. For this purpose the meta information of the MBeans is given in a elaborate enumeration class and a middle class is created to retrieve such information by overriding the *getMBeanInfoMethod*. Any MBean created during this work that wants to provide meta-information subclasses this middle class. In this way there is no need to worry about the *getMBeanInfo* (which is not a trivial operation) because the middle class does most of the job. The enumeration class provides methods to retrieve the description of the MBean, the descriptions of the attributes, methods and notifications.

D.1 Attributes

The following code represents the meta information of the MBean related to Auditing. This MBean only provides information related to its description and attributes. The description of each attributes and permissions (write / read only) must be given. Additionnally, the data type and the name of each attribute must also be given, but those are represented in another enumeration class (*AttAuditing*).

```
public enum MBeansMetaInformation {

    /** The Auditing. */
    Auditing("Controls Client Auditing",
     new MBeansAttributeInformation[]{
     new MBeansAttributeInformation(AttAuditing.USERS, "The list of clients to be 
         audited", Boolean.TRUE),
     new MBeansAttributeInformation(AttAuditing.CAPTURE_USERS, "The list of clients
          whose files should be captured", Boolean.TRUE),
     new MBeansAttributeInformation(AttAuditing.STATUS, "Defines if auditing is 
         turned ON or OFF", Boolean.TRUE)}
```

```
        ),
        // Other definitions, constructors and methods
}
```

D.2 Methods

This code shows the example of an MBean that need to provide meta information about its methods as well. Information concerning all the parameters (name, data type and description) of each method needs to be provided. There are even more complex definitions that also need to provide meta information about notifications.

```
/** The Manage displays. */
ManageDisplays("Use to register MBeans, which provides statistics about a specific
    group or user",
        new MBeansAttributeInformation[]{},        new MBeanOperationInfo[]{
                        new MBeanOperationInfo("displayGroup","Register a new MBean
                            which provides statistics of the group specified",
                                    new MBeanParameterInfo[]{
                                            new MBeanParameterInfo("group",
                                                String.class.getName(), "The 
                                                name of the group ex:cryo")
                                    }
                                    ,CompositeDataSupport.class.getName(),
                                        MBeanOperationInfo.ACTION),
                        new MBeanOperationInfo("displayUser","Register a new MBean 
                            which provides statistics of the user specified",
                                    new MBeanParameterInfo[]{
                                            new MBeanParameterInfo("user",
                                                String.class.getName(), "The 
                                                name of the user ex:cryo@1
                                                .2.3.4")
                                    }
                                    ,CompositeDataSupport.class.getName(),
                                        MBeanOperationInfo.ACTION),
                        new MBeanOperationInfo("unregisterAllGroupsAndUsers","
                            Unregister all the MBeans providing statistics about 
                            groups or users",
                                    new MBeanParameterInfo[]{}
                                    ,CompositeDataSupport.class.getName(),
                                        MBeanOperationInfo.ACTION)
                        }

        );
```

D.3 Middle class

This middle class, which is the parent of any MBeans created that need to provide meta information, extends the standard MBeans class. This middle class is constructed with the enumeration and retrieves the MBean information based on the enumeration class.

```
public class MetaInfoMBean extends StandardMBean
{
        String className = ""; //The class name
        MBeansMetaInformation mbd; // Enumeration

        //Constructor that accepts the enumeration
        public MetaInfoMBean(Class<?> c, MBeansMetaInformation _mbd) throws
            NotCompliantMBeanException {
                super(c);
                className = c.getClass().getName();
                mbd = _mbd;
        }

        //The MBeanInfo methods
    public MBeanInfo getMBeanInfo() {

        // Create meta-data for the attributes of this bean
        MBeanAttributeInfo[] mbas;
        mbas = new MBeanAttributeInfo[mbd.getAttributesLength()];

                //Look into the enumeration (mdb)
        for(int i=0; i<mbd.getAttributesLength(); i++)
          mbas[i] = new MBeanAttributeInfo(mbd.getAttributes()[i].getAttribute().
              getAttributeName(),
           mbd.getAttributes()[i].getAttributeClass().getName(),
           mbd.getAttributes()[i].getDescription(),
           true,
           mbd.getAttributes()[i].isWritable(),
           false);

        // Create meta-data for the operations of this bean
        MBeanOperationInfo[] mops = mbd.getOperations();

      // Create meta-data for the constructor, use reflection
      Class<?> c = this.getClass();
      Constructor<?>[] theConstructors = c.getConstructors();
      MBeanConstructorInfo[] mcons = {
        new MBeanConstructorInfo("The default constructor",theConstructors[0])
      };

          //Creates the MBeanInfo
      MBeanInfo myInfo = new MBeanInfo(className,
        mbd.getDescription(),
        mbas,
        mcons,
        mops,
        mbd.getNotifications());

      return myInfo;
    }
}
```