A Distributed Analysis and Monitoring Framework for the Compact Muon Solenoid Experiment and a Pedestrian Simulation

A thesis submitted for the degree of Doctor of Philosophy

by
Edward Karavakis

School of Engineering and Design
Brunel University
January 2010

Abstract

The design of a parallel and distributed computing system is a very complicated task. It requires a detailed understanding of the design issues and of the theoretical and practical aspects of their solutions. Firstly, this thesis discusses in detail the major concepts and components required to make parallel and distributed computing a reality. A multi-threaded and distributed framework capable of analysing the simulation data produced by a pedestrian simulation software was developed. Secondly, this thesis discusses the origins and fundamentals of Grid computing and the motivations for its use in High Energy Physics. Access to the data produced by the Large Hadron Collider (LHC) has to be provided for more than five thousand scientists all over the world. Users who run analysis jobs on the Grid do not necessarily have expertise in Grid computing. Simple, user-friendly and reliable monitoring of the analysis jobs is one of the key components of the operations of the distributed analysis; reliable monitoring is one of the crucial components of the Worldwide LHC Computing Grid for providing the functionality and performance that is required by the LHC experiments. The CMS Dashboard Task Monitoring and the CMS Dashboard Job Summary monitoring applications were developed to serve the needs of the CMS community.

Contents

List of Figures

List of Tables

Listings

Acknowledgements

There are many people that contributed the financial, technical and moral support that made this thesis possible. At Brunel I am grateful for the supervision and guidance that I received from my supervisor, Prof. Akram Khan. The Engineering and Physical Sciences Research Council (EPSRC) provided three years of funding for this research.

The majority of the work for this thesis was performed as part of the IT-GS MND group at CERN. As such, I am extremely grateful to Julia Andreeva for allowing me to work in the Experiment Dashboard group and for being a constant source of guidance and inspiration. Julia provided much valuable support, supervision and punctuation. The rest of the IT-GS MND group also deserve thanks for their help and humour over the years; in particular Benjamin, Pablo, Ricardo, Gerhild and William. Of course, users were essential to the success of the Task Monitoring and Job Summary applications. More than fifty LHC physicists and Stefano Belforte provided valuable feedback throughout the years and deserve special thanks.

The multi-threaded and distributed framework for pedestrian simulation analysis discussed in Chapter 4 would not have been possible without the support of the developers of the Legion pedestrian simulation software. Alex, Martin and James thank you.

I would also like to thank Irene for her endless love and support and my cousin, Eddie, for his proofreading. Finally, I would like to dedicate this research work to my family for their continuous love, support and guidance.

CHAPTER 1.

INTRODUCTION

The Large Hadron Collider (LHC) at CERN on the Franco-Swiss border will operate at energies which have been out of reach from previous High Energy Physics (HEP) experiments. Two beams of subatomic particles will travel in opposite directions inside the circular accelerator, gaining energy at every lap. Physicists will then use the LHC to recreate the conditions just after the Big Bang by colliding the two beams at very high energy at each of four collision points. Teams of physicists from around the world will analyse and examine the particles created in the collisions using a detector trying to find evidence of new physics. There are many scientific, engineering and computational challenges that must be overcome before any answers can be delivered.

Previous High Energy Physics experiments were able to satisfy their computational needs by building a single computing centre close to the detector. This is no longer realistic for the LHC since the LHC will produce approximately 15 Petabytes (15 million Gigabytes) of data annually for ten to fifteen years. The solution is Grid computing which makes use of the infrastructure, expertise and facilities that exist at computing centres around the world. Grid computing is making big contributions to scientific research by helping scientists around the world to analyse and store massive amounts of data.

The first pioneering steps in Grid computing were taken in the US. The term “Grid computing” was first used by Grid pioneers Ian Foster and Carl Kesselman, as a metaphor for making computing power accessible in the similar way to electrical power. The Worldwide LHC Computing Grid Project, led by CERN, uses resources contributed by Grid projects around the globe. The Enabling Grids for E-sciencE project in Europe, the Open Science Grid in the US, GridPP in the UK and the INFN Grid in Italy are

some of the independent Grid projects that provide support for the computing needs of many areas of research and contribute to the Worldwide LHC Computing Grid.

This thesis is divided into two parts; first it discusses the development of a parallel and distributed framework for pedestrian simulation analysis. It then takes distributed computing on a worldwide and global scale by discussing the development of monitoring applications to be used to enable physicists working on the CMS collaboration to monitor their distributed analysis using the Grid. First, as motivation, a more detailed look will be taken at the evolution of computing in Section 1.1. Distributed and High Performance Computing will be discussed in more detail in Section 1.2. The birth of the Internet and its evolution will be discussed in Section 1.3. The final sections are focused on the Grid and the Worldwide LHC Computing Grid.

1.1 Birth of Computing

Charles Babbage produced a prototype of the “difference engine” [1] by 1822, a calculating machine which could do many long computations automatically that was intended to be steam-powered; fully automatic, even to the printing of the resulting tables; and commanded by a fixed instruction programme but in 1833, Babbage stopped working on the difference engine and he never successfully built the machine. In 1890, Herman Hollerith, the founder of IBM, developed a device which could automatically read census information which had been punched onto a card and as a result, reading errors were consequently greatly reduced, work flow was increased, and stacks of punched cards could be used as an accessible memory store [2].

In 1936, the British mathematician Alan Turing wrote a paper [3] in which he described a hypothetical device, a Turing machine, that formed the basis of programmable computers. The Turing machine was designed to perform logical operations and could read, write and erase symbols written on squares of an infinite paper tape. This kind of machine came to be known as a “finite state machine” because at each step in a computation, the machine's next action was matched against a finite instruction list of possible states. Then, in 1941, Konrad Zuse [4] released the first programmable computer designed to solve complex engineering equations. It was the first machine to work on the binary system.

In 1944, Howard Aiken finished the construction of a large automatic digital computer based on standard IBM electromechanical parts. Aiken's machine, called the Harvard Mark I [5] was the first fully automatic, general purpose electro-mechanical computer and was capable of 5 operations a second. In 1945, mathematician John von Neumann undertook a study [6] of computation that demonstrated that a computer could have a simple, fixed structure, yet be able to execute any kind of computation given properly programmed control without the need for any hardware modification. Von Neumann contributed a new understanding of how practical fast computers should be organised and built and these ideas, often referred to as the “stored-programme technique”, became fundamental for future generations of high-speed digital computers and were universally adopted.

The Electrical Numerical Integrator and Computer (ENIAC) [7] was the first machine to use more than 2,000 vacuum tubes and it was capable of 5000 operations a second. Nonetheless, it had punched-card input and output. ENIAC is acknowledged to be the first successful high-speed “Electronic Digital Computer” (EDC) and was productively used from 1946 to 1955.

The Electronic Discrete Variable Automatic Computer (EDVAC) [5] was to be a vast improvement upon ENIAC. Mauchly and Eckert's idea was to have the programme for the computer stored inside the computer. EDVAC had more internal memory than any other computing device to date.

In the late 1940s and 1950s, two devices would be invented which would improve the computer field and cause the beginning of the computer revolution. The first of these two devices was the transistor [8]. Invented in 1947 by William Shockley, John Bardeen, and Walter Brattain of Bell Labs, the transistor was fated to oust the days of vacuum tubes in computers, radios, and other electronics. Vacuum tubes were inefficient, required a lot of room space, and needed to be replaced often. The transistor promised to solve all of these problems but transistors had their problems too; transistors needed to be soldered together. In 1958, Jack Kilby and Robert Noyce manufactured the first integrated circuit. An integrated circuit (IC) [9, 10] is a small electronic device made out of a semiconductor material. In addition to saving space, the

speed of the machine was now increased since there was a diminished distance that the electrons had to follow.

In 1971, Intel released the first microprocessor [11]. The microprocessor was a specialised integrated circuit which was able to process four bits of data at a time. The chip included its own arithmetic logic unit, but a sizeable portion of the chip was taken up by the control circuits for organising the work, which left less room for the data-handling circuitry. The MITS Altair 8800 [12] was the first commercial personal computer in 1974. However it was not until the eighties that home computing began to become desirable and affordable.

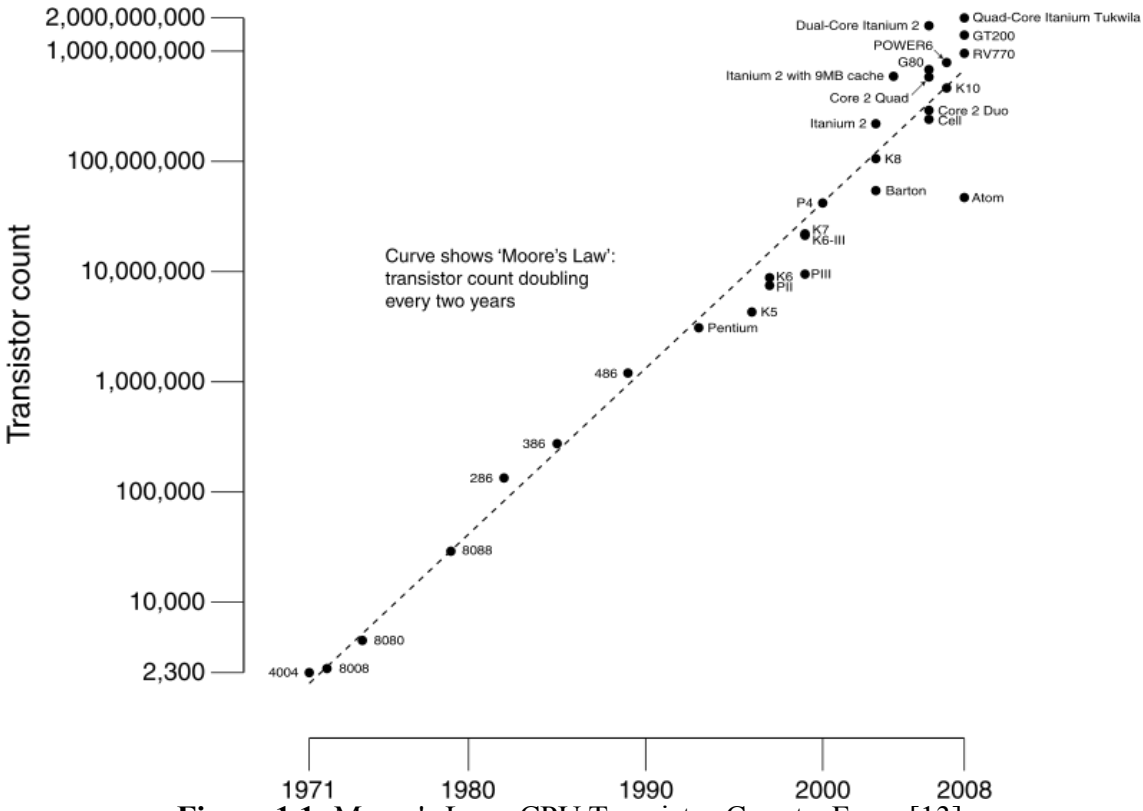


Figure 1.1: Moore's Law: CPU Transistor Counts. From [13].

In 1965, Gordon Moore predicted that the number of transistors on a chip would double every two years [14]. Figure 1.1 illustrates and confirms Moore's famous law; the density of transistors on a chip doubles every 24 months. Moore made his prediction based on the empirical evidence that was available and has so far remained accurate. However, even as performance increases, there will always be a set of problems with requirements beyond those that can be satisfied by a single CPU chip.

1.2 Distributed and High Performance Computing

The speed of light and heat limit the speed of a CPU chip. Furthermore, Lev Levitin and Tommaso Toffoli devised an equation [15] which sets a fundamental limit for quantum computing speeds; a perfect quantum computer can generate 10 quadrillion more operations per second than fastest current CPUs. They estimate that the maximum speed will be reached in approximately 75 years. A quantum computer is a device for computation that makes direct use of quantum mechanical phenomena, such as superposition and entanglement, to perform operations on data. When Moore's Law can no longer meet computational needs, the solution is to introduce some form of a parallelism in the execution of a programme; multiple CPUs or computers can execute and process different parts of a programme simultaneously.

High Performance Computing (HPC) uses supercomputers and computer clusters to solve advanced and complex scientific computation problems. Today, computer systems approaching the teraflops-region are counted as high performance computers. The TOP 500 [16] list ranks the world's 500 fastest high performance computers, as measured by the HPL benchmark [17]. The projected performance graph can be seen in Figure 1.2; it provides an important tool to track historical development and also to predict future trends.

The development of these machines is driven by scientific computational problems with demands that exceed the performance of a single computer; it would take too long to compute and/or the problem may not fit into the memory or the storage of a single computer. The problems can be divided into different tasks and processed simultaneously across multiple processors or computers.

In a shared memory system, there is a common shared address space throughout the system and the communication between the processors occurs using shared data and control variables for synchronisation among the processors using a library such as the OpenMP [18]. In a distributed memory system, there is no shared address space and all the multicomputer systems communicate by passing messages between them using a library such as the MPI [19]. When the tasks are completely independent and there is no dependency between them, the performance benefit is significant.

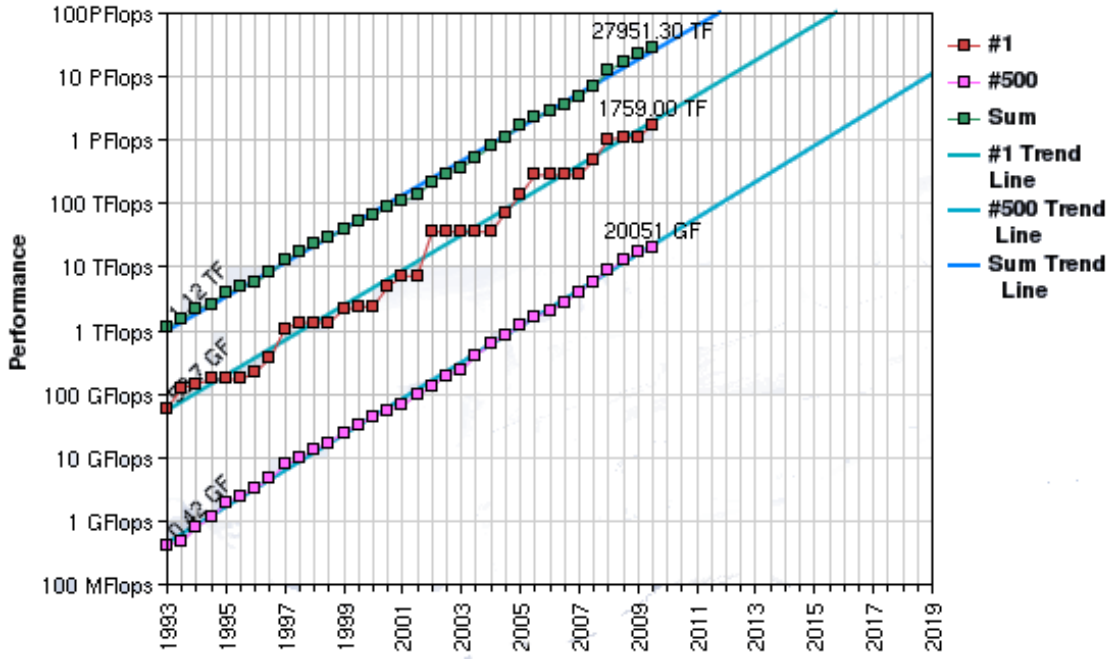


Figure 1.2: Projected Performance Graph. Data from: http://top500.org

1.3 Internet

The origins of the Internet reach back to the 1960s when the United States funded research projects of its military agencies to build robust, fault-tolerant distributed computer networks. This research spawned worldwide participation in the development of new networking technologies and led to the commercialisation of an international network in the mid 1990s, and resulted in the following popularisation of countless applications in virtually every aspect of modern human life. As of 2009, an estimated quarter of Earth's population uses the services of the Internet. The exponential growth of the total number of the internet hosts can be seen in Figure 1.3.

The Internet has no centralised governance in either technological implementation or policies for access and usage; each constituent network sets its own standards. Only the overreaching definitions of the two principal name spaces in the Internet, the Internet Protocol (IP) address space and the Domain Name System (DNS), are directed by the Internet Corporation for Assigned Names and Numbers (ICANN) [20]. The technical standardisation of the core protocols (IPv4 and IPv6) is an activity of the Internet Engineering Task Force (IETF) [21].

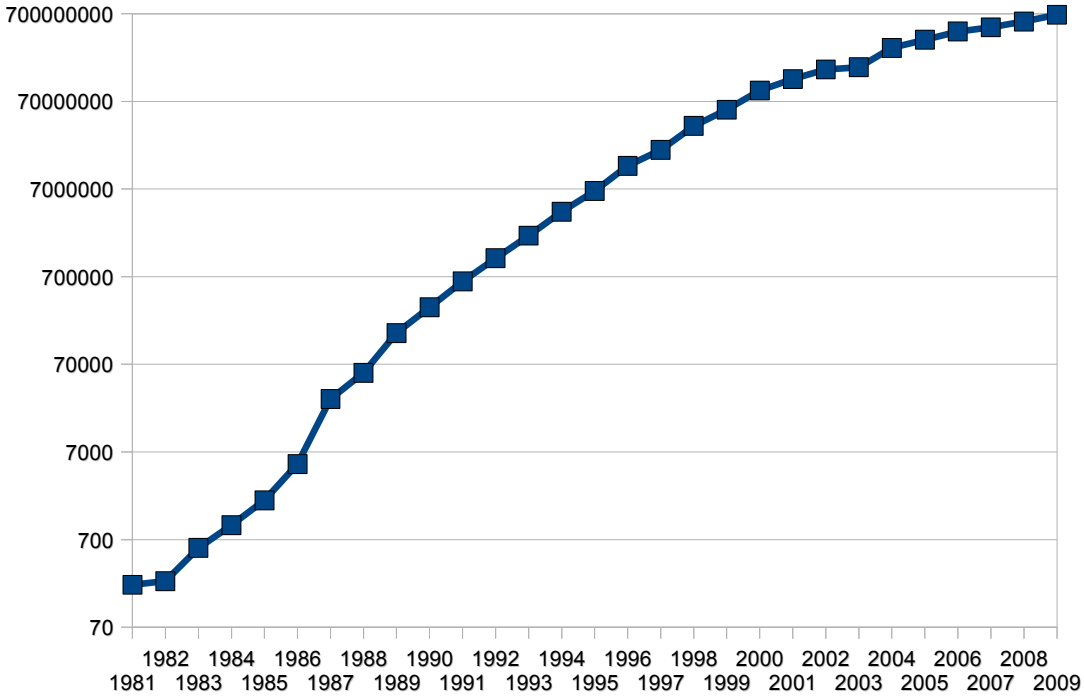


Figure 1.3: Internet Host Count History. Data from: http://www.isc.org/solutions/survey/history

As internet access became commonplace more advanced applications began to emerge. Standards at every level such as TCP/IP, HTTP, HTML, SOAP and XML make the internet a reality. Multiple independent networks can be combined to form a single, global, fault tolerant network, over which applications can request and receive data.

1.3.1 World Wide Web

The World Wide Web (WWW) was developed at CERN as a new form of communicating text and graphics across the Internet using the hypertext mark-up language (HTML) [22] as a way to describe the attributes of the text and the placement of the graphics. Using concepts from earlier hypertext systems, the World Wide Web was invented [23] in 1989 by the English computer scientist Sir Tim Berners-Lee, now the Director of the World Wide Web Consortium (W3C) [24], and later assisted by Robert Cailliau, a Belgian computer scientist, while both were working at CERN in Geneva, Switzerland.

Unlike predecessors such as HyperCard [25], the World Wide Web was non-

proprietary, making it possible to develop servers and clients independently and to add extensions without any licensing restrictions. On April 30, 1993, CERN announced [26] that the World Wide Web would be free to anyone, with no fees due. Since it was first introduced, the number of users has blossomed and the number of sites containing information and searchable archives has been growing at an unprecedented rate. The World Wide Web enabled the spread of information over the Internet through an easy-to-use and flexible format.

1.3.2 Web Services

According to the W3C, a Web Service [27] is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface that is described in a machine-processable format such as the Web Services Description Language (WSDL) [28]. Other systems interact with the Web Service in a manner prescribed by its interface using messages, which are enclosed in a SOAP [28] envelope. These messages are typically transferred using HTTP, and normally comprise XML in conjunction with other Web-related standards.

Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability is due to the use of open standards. The Organisation for the Advancement of Structured Information Standards (OASIS) [29] and the W3C are the committees responsible for the architecture and the standardisation of the Web Services. A typical Web Service invocation can be seen in Figure 1.4.

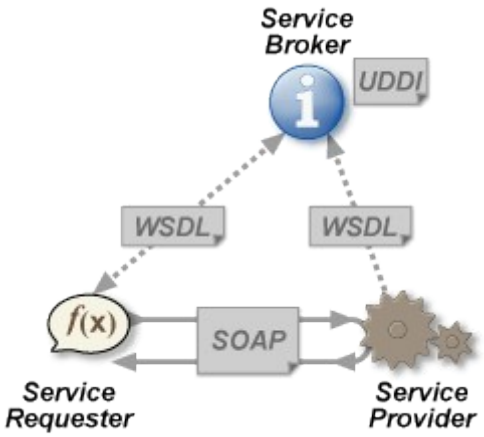


Figure 1.4: Web Service Invocation. From [30].

The architecture of the Web Services is divided into the following parts [31]:

- Service Processes: This part of the architecture generally involves more than one Web Service. Discovery belongs in this part of the architecture since it allows to locate one particular service within a collection of Web Services.
- Service Description: The most interesting feature of the Web Services is that they are self-describing. Once a user has located a Web Service, he/she can ask it to "describe itself" and tell the user what operations it supports and how to invoke it. This is handled by the WSDL.
- Service Invocation: Invoking a Web Service involves passing messages between the client and the server. The Simple Object Access Protocol (SOAP) specifies how to format the client's requests to the server, and how the server should format its responses.
- Transport: All these messages must be transmitted between the server and the client. The protocol of choice for this part of the architecture is HTTP but in theory any other transferring protocol can be used instead.

1.4 The Grid

The first pioneering steps in grid computing were taken in the US. The term "grid computing" was first used in a book [32] by Grid pioneers Ian Foster and Carl Kesselman, as a metaphor for making computing power accessible in the similar way as electrical power.

Grid computing was first proposed as Metacomputing [33] in 1992, but it was not until the Information Wide Area Year (I-WAY) [34] project in 1995 that it really began to emerge by linking together US supercomputing centres, databases and visualisation devices. The experience and software that was developed was later used as the basis for the Globus project [35].

Ian Foster defines a Grid as "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organisations" [36] and this statement defines what distinguishes a Grid from other forms of distributed computing. A Grid is not a single

cluster or within a single site or institution. A Grid can be categorised as a Computational Grid, a Data Grid and an Access Grid [36]. This classification is based on whether a scientific programme requires intensive computation or whether it needs to handle and store a large amount of data or whether it requires a collaboration environment for achieving a common goal.

A Computational Grid [32] supports high computation intensive scientific applications by pooling large scale and distributed resources together. The aim of using a Computational Grid is to solve big computation problems that can not be solved by using a single computer or a cluster of computers and also, to reduce the total computation time of these large scale scientific programmes. Weather forecasting [37] and Earthquake simulation [38] are typical computation intensive applications on a Computational Grid. A Data Grid [39] is a distributed data processing and management centric infrastructure for data intensive scientific applications that is concerned with the issues of data generation, management, storage and transmission in distributed data resources. Finally, an Access Grid [40] is being used in a collaborative environment in which Grid users all over the world are able to participate in a virtual world for collaborated information integration and processing. Interactions are the core of an Access Grid. A multimedia video conference system is a typical application of an Access Grid.

1.5 e-Science

The “e-Science” term was created by John Taylor [41] in 1999 to describe computationally intensive science that is carried out in highly distributed network environments, or science that uses immense data sets that require Grid computing. Examples of the kind of science include social simulations, particle physics, earth sciences and bio-informatics. Particle physics has a particularly well developed e-Science infrastructure due to their need for adequate computing facilities for the analysis of results and storage of data originating from the CERN Large Hadron Collider (LHC) [42].

Due to the complexity of the software and the backend infrastructural requirements, e-Science projects usually involve large teams managed and developed by research

laboratories, large universities or governments. The UK e-Science Programme provides significant funding; the UK e-Science Programme began in 2001 as a coordinated initiative involving all the Research Councils and the Department of Trade and Industry. The e-Science Core Programme [43], managed by the Engineering and Physical Sciences Research Council on behalf of the communities of all the Research Councils, has supported the development of generic technologies, such as the middleware that is needed to link up varying hardware resources across the Grid in a compatible way allowing scientists to access these resources in a uniform and secure way from anywhere in the world by turning the diverse and locally managed computing centres into a single massive virtual resource. Each Research Council has funded its own e-Science activities to develop techniques and demonstrate their use across a broad range of research and applications.

1.6 Computing for the LHC: The Worldwide LHC Computing Grid

The Large Hadron Collider (LHC) at CERN on the Franco-Swiss border is the largest scientific instrument on the planet. The LHC was built to help scientists to answer key unresolved questions in fundamental physics. It consists of a 27 km ring of superconducting magnets with a number of accelerating structures to boost the energy of the particles along the way. Two beams of particles travel inside the accelerator, at close to the speed of light with very high energies before colliding with one another. The beams travel in opposite directions in separate beam pipes and they are guided around the accelerator ring by a strong magnetic field, achieved using superconducting electromagnets.

The six experiments at the LHC are all run by international collaborations, bringing together scientists from institutes all over the world. Each experiment is distinct, characterised by its unique particle detector. The two large experiments, the “A Toroidal LHC ApparatuS” (ATLAS) [44] and the “Compact Muon Solenoid” (CMS) [45], are based on general-purpose detectors to analyse the myriad of particles produced by the collisions in the accelerator. They are designed to investigate the largest range of physics possible. Two medium-size experiments, the “A Large Ion Collider Experiment” (ALICE) [46] and the “LHC-beauty” (LHCb) [47], have specialised

detectors for analysing the LHC collisions in relation to specific phenomena. The remaining two experiments, the "Total Elastic and Diffractive Cross Section Measurement" (TOTEM) [48] and the "LHC-forward" (LHCf) [49], are much smaller in size and often not mentioned at all. They are designed to focus on "forward particles"; particles that just brush past each other as the beams collide, rather than meeting head-on.

The ATLAS, CMS, ALICE and LHCb detectors are located around the ring of the LHC as illustrated in Figure 1.5. The detectors used by the TOTEM experiment are located near the CMS detector and those used by the LHCf are near the ATLAS detector.

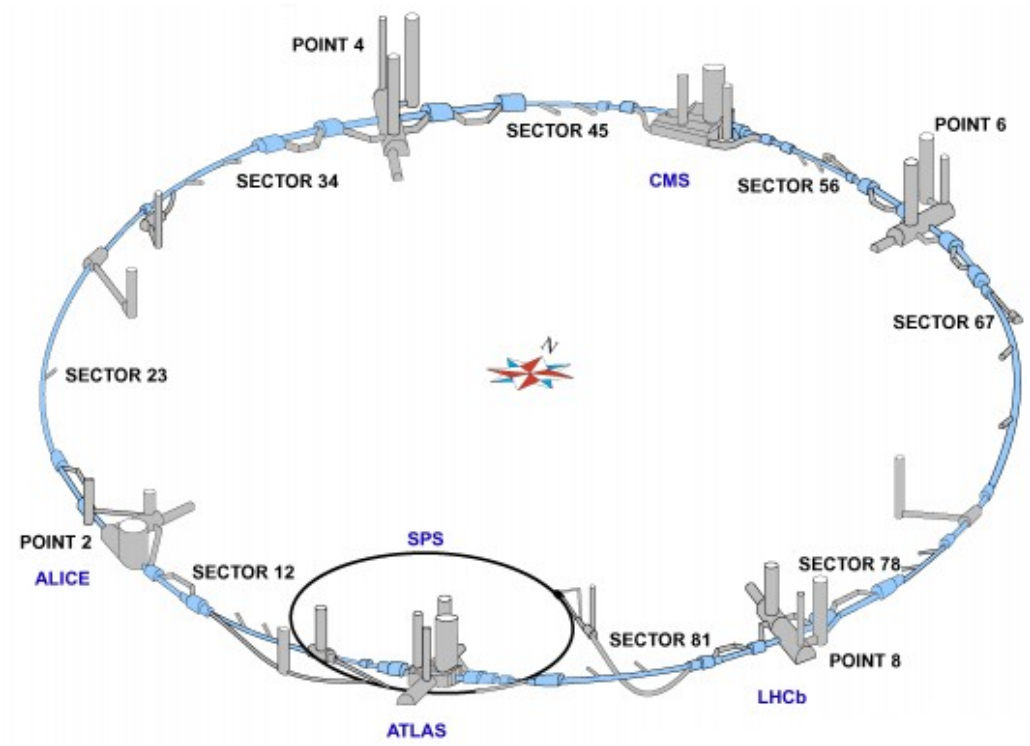


Figure 1.5: The Large Hadron Collider. From [42].

In late 2009, when the LHC restarts operations, it will produce approximately 15 Petabytes (15 million Gigabytes) of data annually for ten to fifteen years, which thousands of scientists around the world will access and analyse. If the LHC data were to be burned to a CD, a tower of CDs around 20 kilometres high would be created within a year; twice as high as the Mount Everest. The Worldwide LHC Computing Grid (WLCG) [50] anticipates running between 500,000 to 1,000,000 tasks per day and

this number will increase as time goes on and as computing resources and new technologies become ever more available across the world. It is no longer practical to use only resources that are co-located with the experiment. Apart from the financial and political implications of financing such infrastructure at a singe location, it also provides a single, critical point of failure.

The mission of the WLCG project is to build and maintain a data storage and analysis infrastructure for the entire High Energy Physics (HEP) community that will use the LHC. The WLCG combines the computing resources of more than 170 computing centres in 34 countries, aiming to harness the power of more than 100,000 CPUs to process, analyse and store data produced from the LHC making it equally available to all partners, regardless of their physical location in order to sift through data, looking for new particles that can provide clues to the origins of our universe.

The computing centres providing resources for WLCG are embedded in different operational Grid organisations, in particular the Enabling Grids for E-SciencE (EGEE) [51] and the Open Science Grid (OSG) [52], but also several national and regional Grid structures such as GridPP in the UK, INFN Grid in Italy and NorduGrid in the Nordic region. Europe and Asia use the gLite middleware [53], developed by the EGEE and co-funded by the European Commission, the Nordic Grids are based on the Advanced Resource Connector (ARC) [54] software and the US contribution to the WLCG relies on the Virtual Data Toolkit (VDT) [55] provided by the OSG middleware distribution. All the middleware systems have been influenced by the Globus Toolkit and many core components still originate from it.

The data from the LHC experiments will be distributed around the globe, according to a four-tiered model as proposed by the MONARC project [56] as illustrated in Figure 1.6. A primary backup will be recorded on tape at CERN, the “Tier-0” centre of LCG. After initial processing, this data will be distributed to a series of Tier-1 large computer centres, through dedicated 10 gigabit per second connections, with sufficient storage capacity and with 24/7 support for the Grid. The Tier-1 centres will make data available to the Tier-2 centres, each consisting of one or several collaborating computing facilities, which can store sufficient data and provide adequate computing power for

specific analysis tasks. Individual scientists will access these facilities through the Tier-3 computing resources, which can consist of local clusters in a University or a national research centre.

By taking advantage of the hardware and personnel distributed throughout the collaborations, it is possible to deliver enough aggregate computing power without locating the resources at a single point. Of course, moving to a completely chaotic distributed architecture introduces many additional problems and complexities.

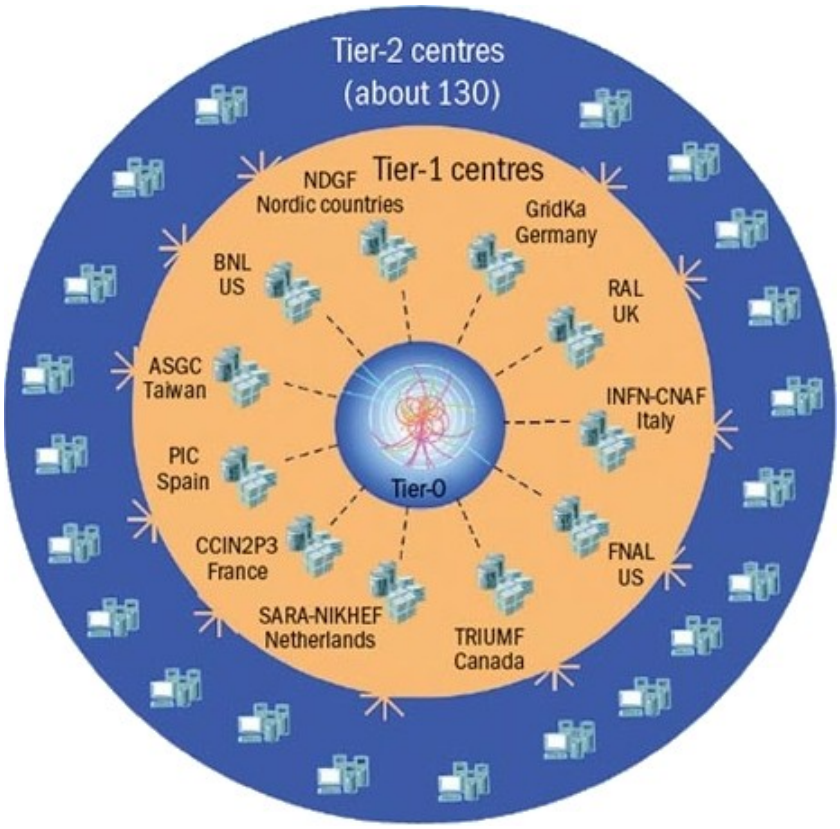


Figure 1.6: The Four-Tiered Model as Proposed by the MONARC Project. Tier-3's are smaller centres connected to Tier-2 sites. From [57].

Reliable monitoring is an aspect of particular importance; it is a vital factor for the overall improvement of the quality of the WLCG infrastructure. In addition, monitoring of the computing activities of the communities using the WLCG infrastructure provides the best estimation of its reliability and performance.

The distributed analysis on the WLCG infrastructure is currently one of the main challenges of the LHC computing. Access to the LHC data has to be provided to more

than five thousand scientists all over the world. Users who run analysis jobs on the Grid do not necessarily have expertise in Grid computing. Simple, user-friendly, reliable monitoring of the analysis jobs is one of the key components of the operations of the distributed analysis.

1.7 Summary

This chapter sets the scene for the more detailed discussion to come. Parallel, distributed computing and solving the computing challenges in the Large Hadron Collider experiments, in particular the computing demands of the CMS experiment, provide the main motivation for this thesis. A general introduction to the area was provided outlining the birth of the computing and the computer revolution that took place in the eighties.

The LHC experiments will produce huge volumes of data which require extensive computing resources to store, transfer and analyse. The Grid is the solution chosen to meet these computational requirements. Grid computing evolved as a key technology enabling scientists in research and industry to solve challenging problems, master complex heterogeneous environments and collaborate in unprecedented ways.

The Grid integrates distributed computing resources and data created through simulations storing them in archive tapes or databases. Grid technology combines high performance and high throughput computing, data intensive and on-demand computing and collaborative computing through a set of service interfaces based on common protocols.

The next chapter discusses in detail the main concepts and components required to make parallel and distributed computing a reality, outlining the design issues and the techniques to avoid non-intuitive behaviours. Chapter 3 identifies and discusses in detail the major concepts and components that are required to make Grid computing a reality. Chapter 4 describes the development of a multi-threaded and a distributed version of a commercial pedestrian simulation software and presents benchmark results demonstrating how the use of a multicomputer or of even a multi-core computer can

greatly accelerate the speed of a pedestrian movement software. Chapter 5 discusses in depth the CMS Dashboard Task Monitoring application focusing on the CMS analysis of the user activities and Chapter 6 discusses the CMS Dashboard Job Summary application that provides a more generic monitoring application to a wide variety of users in the CMS collaboration. Chapters 2 and 4 are focused on the parallel and on the distributed computing whilst Chapter 3, 5 and 6 are focused on the Grid computing. Finally, Chapter 7 summarises this research work and discusses future directions.

Different parts of the research presented in this thesis have been published in [58], [59], [60], [61], [62] and [63]. The first publication, [58], focuses on the design and the implementation of the parallel and distributed version of the commercial pedestrian simulation software presented in Chapter 4.

The second publication, [59], focuses on the Distributed Analysis demands in the CMS experiment and on the CMS Computing Model in general as presented in Section 3.5. The third publication, [60], focuses on the Experiment Dashboard monitoring system for the LHC experiments and its framework as presented in Section 3.6.

The remaining three publications, [61], [62] and [63] focus on the work presented in Chapter 5 for the CMS Dashboard Task Monitoring application, and in Chapter 6 for the CMS Dashboard Job Summary application.

CHAPTER 2.

PARALLEL AND DISTRIBUTED COMPUTING

A distributed computing system is a collection of computers that cooperate to solve a problem that cannot be individually solved. The notion of a distributed computing system as a useful and widely-used tool is already a reality due to the widespread proliferation of the Internet and the emerging global village.

This chapter discusses the main concepts and design issues of parallel and distributed computing.

2.1 Introduction

John von Neumann proposed in 1945 the creation of an Electronic Discrete Variable Automatic Computer (EDVAC). In his paper [6], von Neumann suggested a stored-programme model of computing known as the von Neumann architecture. In the von Neumann architecture [64], a programme is a sequence of instructions stored sequentially in the memory of the computer. The programme's instructions are executed one after the other in a linear and single-threaded way.

The ideas presented by von Neumann were expanded due to the advancements in the mainframe technology and the arrival of the time-sharing operating systems in the 1960s. These operating systems first introduced the concept of the concurrent programme execution. A mainframe computer could be accessed simultaneously by multiple users. The users submitted jobs for processing and the operating system handled the details of allocating CPU time for each individual programme. This concurrency existed at the process level.

Only one programme would run at a time in the early days of personal computing. User interaction occurred via text-based interfaces and the programmes followed the standard model of instruction execution proposed by von Neumann. However, the exponential growth in CPU and graphics performance, quickly led to more sophisticated computing systems. This rapid growth increased the user expectations. Users expected their computing platform to be quick and responsive and their applications to start up quickly and handle background tasks with minimal disruption.

2.2 Threads

A thread is a discrete sequence of related instructions that is executed independently of other instruction sequences [65]. Every programme has at least one thread, which is the main thread, that initialises the programme and starts the executions of the first instructions [66]. This main thread can then create no new threads and do everything by itself or it can create other threads to perform various tasks. A thread is contained inside a process as illustrated in Figure 2.1. Unlike different processes, multiple threads within the same process can share resources such as the computer's memory.

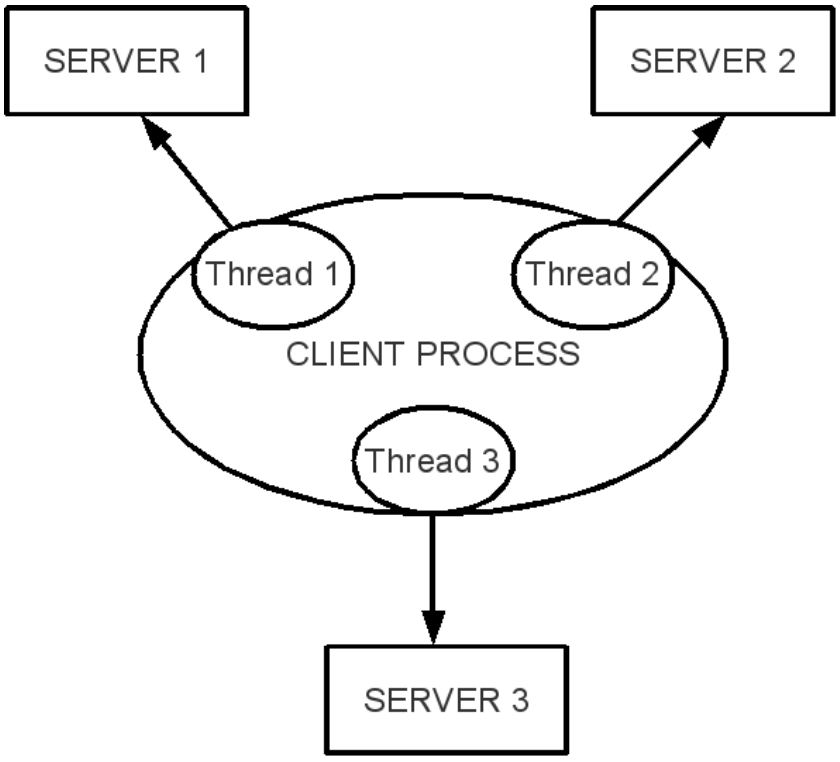


Figure 2.1: A Multi-threaded Process where the client can issue calls to three servers simultaneously.

There are three layers for threading [65]:

- User/application threads. Threads created and destroyed in the application.
- Kernel threads. Used by the kernel of the Operating System (OS).
- Hardware threads. Used by each processor.

One programme thread passes from all the three levels. A programme thread is implemented by the OS as a kernel-level thread and executed as a hardware thread. The interfaces between these layers are handled automatically by the executing system.

As illustrated in Figure 2.2, every newly created thread starts in the "Ready" state, when it is attempting to execute a task it is in the "Running" state and when the work is done, it is either terminated or it returns back into the initial "Ready" state.

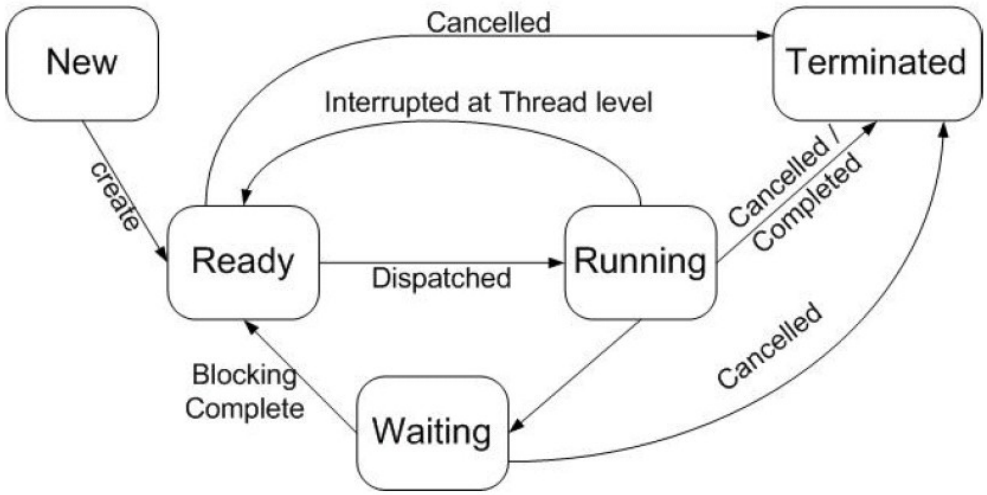


Figure 2.2: State Diagram for a User-level Thread.

Multi-threading on a single processor occurs by time-division multiplexing, thus, the processor switches between different threads. The context switching happens frequently enough that the user perceives that the threads are running at the same time. On the other hand, threads on a multiprocessor or multi-core system will run at the same time, with each processor or core running a particular thread.

Multi-threading occurs when multiple threads exist within the context of a single process. These multiple threads share the resources of the process but are being

executed independently. Multi-threaded programming allows a programme to operate faster on computer systems with multiple CPUs, CPUs with multiple cores or on a cluster of computers due to the fact that the threads of the programme naturally run in a concurrent execution. In such case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviours. The improper use of threading can degrade the performance of the programme as described in Section 2.8. In order for data to be correctly manipulated, threads will often need to synchronise in time to process the data in the correct order. Threads may also require atomic operations in order to prevent common data from being simultaneously modified, or read while being modified by another thread.

Another feature of having multiple threads in a single process is the ability for an application to remain responsive to the user. In a single threaded programme, if the main execution thread blocks on a big task, the entire application can appear to be non-responsive to the user's input. It is possible for an application to remain responsive to the user by moving background long running tasks to another thread that runs in parallel with the main execution thread. Operating systems schedule threads in one of two ways [65][66]:

1. Pre-emptive multi-threading allowing the operating system to determine when a context switch should happen.
2. Cooperative multi-threading relying on the threads themselves to release control once they are at a stopping point.

2.3 Flynn's Taxonomy

In 1966 Flynn produced a taxonomy [67] for computer architectures based on the number of concurrent operations that the architecture can support. A hardware may support a single instruction stream or multiple instruction streams working on a single data stream or multiple data streams.

- Single instruction stream, single data stream (SISD). Are the traditional processors which execute one instruction on one piece of data and they

correspond to the conventional processing in the von Neumann architecture with a single CPU, and a single memory unit connected by a system bus.

- Single instruction stream, multiple data stream (SIMD). Implements data level parallelism where the same instruction operates on an array of data. Corresponds to the processing by multiple homogeneous processors.
- Multiple instruction stream, single data stream (MISD). Corresponds to the execution of different operations in parallel on the same data. According to Flynn, an MISD computer is "a pipeline of multiple independently executing functional units operating on a single stream of data, forwarding results from one functional unit to the next" [68].
- Multiple instruction stream, multiple data stream (MIMD). Different CPUs can simultaneously execute different instruction streams working on different data streams. Multiprocessors and multicomputers fall into this category and this is the mode of operation in distributed systems as well as in the vast majority of parallel systems.

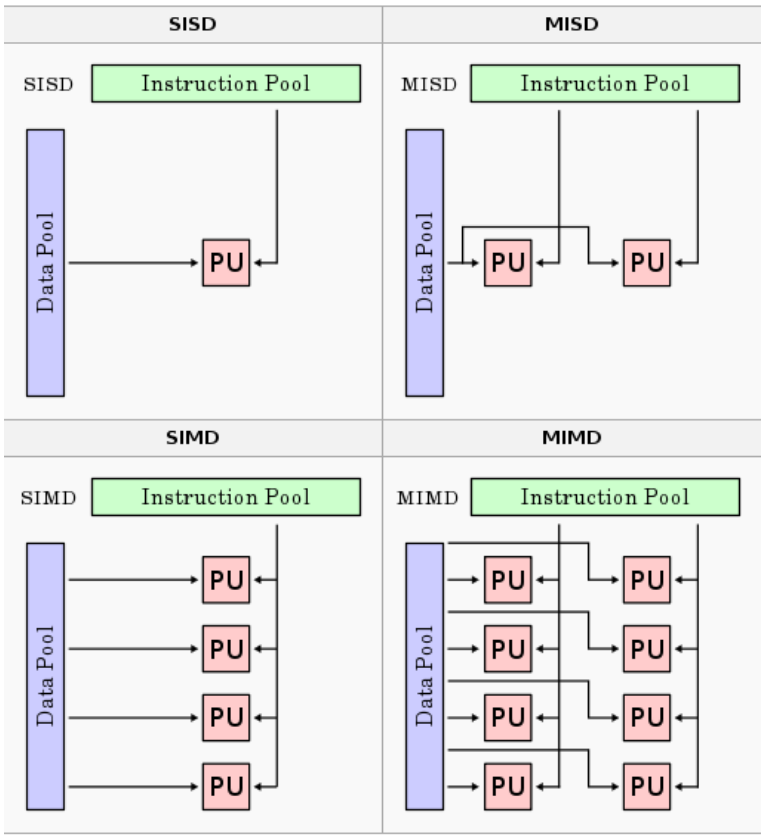


Figure 2.3: Flynn's Taxonomy.

SISD, SIMD, MISD, and MIMD architectures are illustrated in Figure 2.3. Most contemporary parallel and distributed computers fall into the MIMD category. The MIMD architectures allow much flexibility in partitioning the code and the data to be processed among the processors.

2.4 Characteristics of a Parallel System

A parallel system may be classified as belonging to one of the three following types [69]:

1. A multiprocessor system. It is a parallel system in which the multiple processors have direct access to a shared memory which forms a common address space. They can be built out of commodity CPUs.
2. A multicomputer parallel system. It is a parallel system in which the memory of the multiple processors may or may not form a shared address space. Each processor has direct access to its own local memory. Without a shared address space, the multiple processors interact with each other by passing messages.
3. Processor Arrays. This is a class of parallel computers that are physically co-located, are very tightly coupled and have a common system clock but may not share memory and communicate by passing data using messages.

2.4.1 Coupling

The degree of coupling can be measured [69] in terms of the interdependency and binding and/or homogeneity among the modules. When the modules are tightly coupled, a particular module might be harder to re-use or test because dependent modules must also be included.

2.4.2 Parallelism

There are two types of parallelism in a programme [69]:

- Parallelism or speed up of a programme on a specific system. This is a measure of the speed-up of a specific programme running on a given machine. It depends on the number of processors and the allocation of the processing instructions to the processors. It is expressed as the ratio of the time $T(1)$ with a single

processor, to the time $T(n)$ with n processors.

- Parallelism within a parallel / distributed programme. This is an aggregate measure of the time percentage that all the processors are executing CPU instructions in contrast to waiting for any communication operations to complete. The communication operations might involve either accessing a memory block via shared memory or passing data via message-passing.

2.4.3 Concurrency

The concurrency in a distributed programme can be measured [70] by the ratio of the number of local operations excluding the communication and the shared memory access operations to the total number of operations including the communication operations via message-passing or the access to the shared memory operations.

2.4.4 Granularity

Granularity is the ratio of the amount of computation in relation to the amount of communication within a parallel programme. In a fine-grained parallelism, individual tasks are relatively small in terms of execution time. On the other hand, in a coarse-grained parallelism the data are communicated infrequently, after larger amounts of computation. The finer the granularity, the greater the potential for parallelism and hence the speed-up, but the greater the overheads of synchronisation and communication [71].

The best balance between the communication and the computation load overhead needs to be found for a programme to achieve the best parallel performance. In a fine-grained granularity, the performance can suffer from the increased communication overhead by frequently exchanging data via message-passing. On the other hand, in a coarse-grained granularity, the performance can suffer from load imbalance; the system workload will not be evenly distributed across all physical processors in the system. Programmes with fine-grained parallelism are best suited for tightly coupled systems including the SIMD and the MISD architectures, the tightly coupled MIMD multiprocessors that have shared memory, and the loosely-coupled multi-computers without shared memory that are physically located in the same room. Programmes with

fine-grained parallelism running over loosely-coupled multiprocessors that are physically remote experience a significant degrade of the overall throughput due to the latency delay for the frequent communication over the network.

2.5 Performance Analysis of Parallel Programming

A large performance increase can be seen by subdividing different tasks and by processing them simultaneously. When the tasks are completely independent, the performance benefit is significant. The speed-up ratio characterises how much faster a programme runs when parallelised by comparing the elapsed run time of the best sequential algorithm to the elapsed run time of the programme running in parallel.

$$Speed-up(n_t)=\frac{Time_{BestSequentianalAlgorithm}}{Time_{ParallelImplementation}(n_t)}$$

The $Speed-up$ is defined in terms of the number of physical threads (n_t) used in the parallel implementation.

The theoretical limit on the performance benefit of increasing the total number of the CPU cores can be determined using the Amdahl's Law [72], also known as Amdahl's Argument, that examines the maximum theoretical performance benefit of a parallel solution relative to the best case performance of a serial solution.

$$Speed-up=\frac{1}{S+(1-S)/n}$$

The S is the time spent whilst executing the serial portion of the parallelised version of the programme and n is the total number of the processor cores of the system. The numerator assumes that the programme takes 1 unit of time to execute the best sequential algorithm.

Setting $n=\infty$ and assuming that the best sequential algorithm takes 1 unit of time leads to the following equation to find the upper bound of an application with S time spent in sequential code.

$$Speed-up = \frac{1}{S}$$

An alternative formulation for speed up referred to as “scaled speed-up” was developed by E. Barsis and it is known as the Gustafson-Barsis's Law [73].

$$Scaled\ speed-up = N + (1-N) * s$$

Where N is the total number of CPU cores and s is the ratio of the time spent in the sequential version of the programme versus the total execution time.

The Amdahl's Law and the Gustafson-Barsis's Law can overestimate the speed-up or the scaled speed-up performance because they both ignore the parallel overhead term. Karp and Flatt have proposed another metric, called the experimentally determined serial fraction, which can provide valuable performance insights [74].

Given a parallel computation exhibiting speed up ψ on p processors, where $p > 1$, the experimentally determined serial fraction e is defined to be the Karp - Flatt Metric:

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

The less the value of the experimentally determined serial fraction e, the better the parallelisation of the algorithm. By using the experimentally determined serial fraction, we can determine whether the efficiency decrease is due to limited opportunities for parallelism or increases in algorithmic overhead.

2.6 Message Passing Communication

In this section, the message passing communication technique will be discussed in detail based on the messages used in a communication and the mechanisms used to send and receive a message. A message is an accumulation of data consisting of a header and

a body which can be managed by a process and delivered to its destination.

2.6.1 Message-Passing Systems versus Shared Memory Systems

Shared memory systems are those in which there is a common shared address space throughout the system. The communication between the processors occurs using shared data and control variables for synchronisation among the processors. In a shared memory system, synchronisation can be achieved by using semaphores and locks that were designed for shared memory uniprocessors and multiprocessors. All multicomputer systems without a shared address communicate by passing messages. It is considered easier to programme using shared memory than by passing messages between the computers. It is possible to simulate a shared address space for a distributed system with the Distributed Shared Memory (DSM) abstraction.

Emulation of the message-passing technique on a shared memory system

The shared address space can be partitioned into distinct parts, one part being assigned to each processor. The "send" and "receive" operations can be implemented by writing to and reading from the destination/sender processor's address space. Finally, synchronisation primitives are used to control the write and read operations.

Emulation of a shared memory space on a message-passing system

This type of emulation involves the use of "send" and "receive" operations for "write" and "read" operations. Every shared location can be modelled as a separate process. The "write" to a shared location operation is emulated by sending an update message to the corresponding owner process and the "read" from a shared location operation is emulated by sending a query message to the owner process. This type of emulation is quite complicated as it requires "send" and "receive" operations to access the memory of another processor. The latencies involved in the "read" and "write" operations will be most probably high because these "read" and "write" operations are implemented using a network communication underneath.

2.6.2 Primitives for Distributed Communication

The message "send" and the message "receive" communication primitives are

denoted as “Send()” and “Receive()”. A “send” primitive has two parameters: the destination and the buffer in the user space containing the data to be sent to the destination. Likewise, a “receive” primitive has two parameters: the source from which the data is to be received and the user buffer into which the data is to be received. There are two ways of sending data when the “send” primitive is invoked [75]:

- The buffered option which is the default option that copies the data from the user buffer to the kernel buffer and the data then gets copied from the kernel buffer onto the network.
- The unbuffered option where the data gets copied directly from the user buffer onto the network.

The “receive” primitive usually requires the buffered option because the data may already have arrived when the primitive is invoked and needs to be stored in the kernel. There are blocking / non-blocking and synchronous / asynchronous primitives for a distributed communication between two machines [75]:

- Synchronous primitives. A “send” or a “receive” primitive is synchronous when there is a handshake between both the “Send()” and “Receive()” operations. The invoking machine first learns that the other corresponding “receive” primitive has also been invoked and that the “receive” operation been completed and then the processing for the “send” primitive completes. The processing for the “receive” primitive completes when the sending data is copied into the receiver’s user buffer.
- Asynchronous primitives. A “send” primitive is asynchronous when the control returns back to the invoking process after the sending data has been copied out of the user-specified buffer. There is no asynchronous “receive” primitive defined.
- Blocking Primitives. A blocking primitive occurs when the control returns to the invoking process after the processing for the primitive completes either in the synchronous or the asynchronous mode.
- Non-Blocking Primitives. A non-blocking primitive occurs when the control returns back to the invoking process immediately after the invocation. A non-

blocking “send” occurs when the control returns to the process even before the data is copied out of the user buffer. Likewise, a non-blocking “receive” occurs when the control returns to the process even before the data may have arrived from the sender.

From the programme's point of view, a synchronous “send” is easier to implement and to use because of the handshake between the “send” and the “receive” primitives but the truth is that a synchronous “send” lowers the overall efficiency within the process and in fact, the “receive” may not get issued until much after the data arrives at the destination, in which case the data arrived would have to be buffered in the system buffer at the destination and not in the user buffer. At the same time, the sender would remain blocked and non-responsive.

The non-blocking asynchronous “send” is quite useful when sending a large data item over the network because it allows the sender to perform other instructions in parallel with the completion of the “send” and hence, it avoids any potentially large delays for the handshaking process. Likewise, the non-blocking synchronous “send” also avoids any large delays caused by the handshaking process, particularly when the receiver has not yet issued the “receive” call.

The non-blocking “receive” is useful when large amount of data is being received or when the sender has not yet issued the “send” call. This is true because it allows the process to execute other instructions in parallel with the completion of the “receive”. If the data item has been received, it is stored in the kernel buffer and it may take a while to copy it to the user-specified buffer. The hassle on the programmer increases for the non-blocking calls because the programmer needs to keep track of the completion of such operations in order to write to or read from the user buffers and this is the reason why it is easier to use blocking primitives from the programmer's perspective.

The blocking and non-blocking send primitives can be seen in Figure 2.4. When using a non-blocking “send”, the sending process is blocked only for the time period of copying the message in the kernel buffer. Therefore, the block of code after the “send” primitive can be executed before the message is actually sent. On the contrary, the

sending process is blocked completely when using the blocking "send" primitive and thus, the block of code after the "send" primitive is not executed until the sending message has been completely sent. When using the blocking "receive", the process issued this primitive remains entirely blocked until the message arrives and is stored in the buffer.

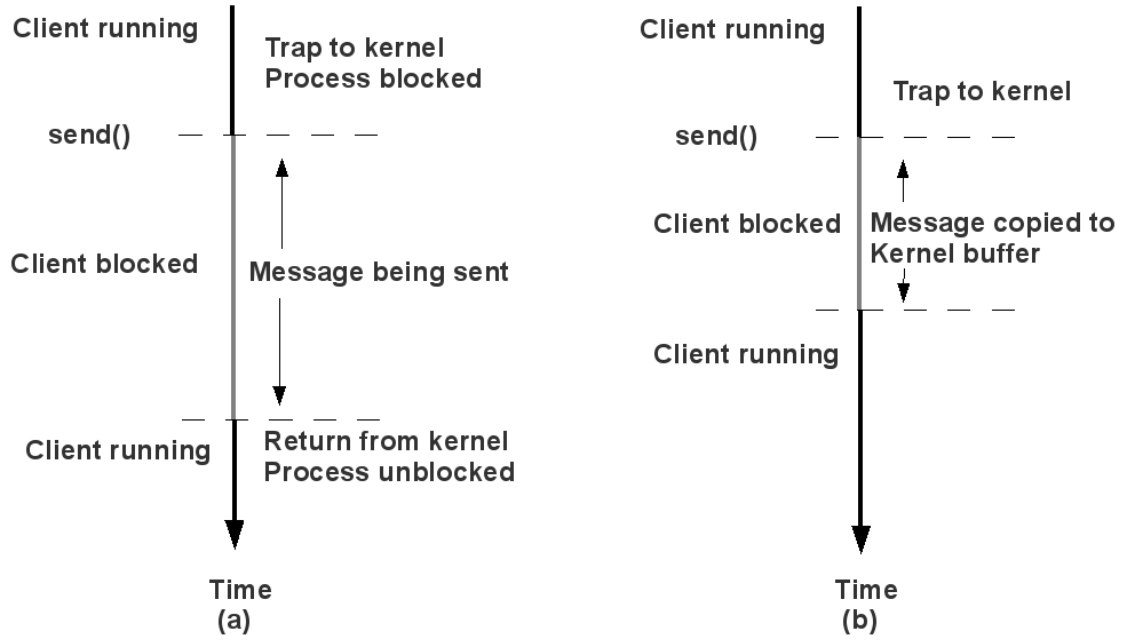


Figure 2.4: Send Primitives. (a) blocking; (b) non-blocking.

2.6.3 Buffered versus Unbuffered Message Passing Primitives

The messages are buffered between the time they are sent by a client and received by a server in most message-based communication systems. There are two possible outcomes when a send is executed and the buffer is full [75]:

- The "send" will delay until there is a space in the buffer for the message.
- The "send" will return to the client indicating that the message could not be sent because the buffer was full.

The outcome on the receiving server is slightly different, the "receive" primitive informs the OS about a buffer where the server needs to store the arrived message and the problem appears when the "receive" primitive is issued after the message arrives. One approach is to discard the entire message from the server's side and the client could

time-out and re-submit the message. Another approach is to save the message in the OS area for a limited time period and then the message will be copied to the invoking server-space only when the “receive” primitive is invoked. Otherwise, the message will be discarded.

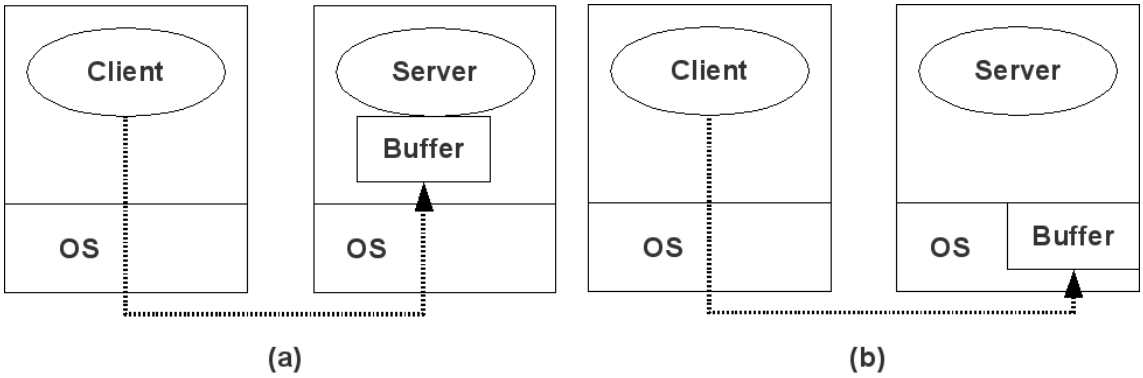


Figure 2.5: (a) Unbuffered and (b) buffered message passing.

The unbuffered message where the message is discarded when the server buffer is fully used can be seen in Figure 2.5 (a). The buffered message where the message is buffered in the buffer of the OS for a limited time period can be seen in Figure 2.5 (b).

2.6.4 The Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a standard for the communication between the nodes running a parallel programme on a distributed memory system. MPI is a library of routines that can be called from Fortran, C, C++, Java and Python programmes. It is a widely used message-passing standard for parallel programming and it is also the dominant model used in the high-performance scientific computing [76] both in the academia and in the industry.

The MPI library supports both point-to-point and collective communication and according to its founder, it "is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation" [19]. The goals of the MPI are high performance, scalability, and portability.

MPI has Language Independent Specifications (LIS) for the function calls and

language bindings. There are two versions of the standard that are currently available [77]:

- The MPI-1 implementation which emphasises message passing and has a static runtime environment.
- The MPI-2 implementation which includes some new features such as parallel I/O and remote memory operations.

The MPI-1 model has no shared memory concept and MPI-2 has only a limited distributed shared memory concept. MPI-1 programmes still work under MPI implementations compliant with the MPI-2 standard.

MPI is the widely used message-passing library because it is both portable and fast. It is portable because MPI has been implemented for almost every computer hardware architecture and it is fast because each implementation is intensively optimised for the hardware it runs on. MPI can be used in low latency networks for inter-node communication using a computer cluster, as illustrated in Figure 2.6.

Figure 2.6: MPI Cluster. A well designed application can scale almost linearly with the addition of more nodes allowing increases in accuracy and speed for scientific applications. From [78].

Shared memory programming models such as the Pthreads [79] and the OpenMP and message-passing programming models such as the MPI and the Parallel Virtual Machine (PVM) [80] can be both utilised and used together in scientific computing programmes.

2.6.5 MPI and OpenMP

There is a lot of interest in how to appropriately utilise both the distributed and shared-memory models due to the growth of the distributed shared-memory machines in the scientific computing community [81]. The MPI library provides an efficient medium for the parallel communication among a distributed collection of computers but no MPI implementation takes advantage of the shared memory when it is available between multiple processors.

The Open Multi Processing (OpenMP) was introduced to provide a shared-memory parallelism in FORTRAN, C, C++ and Python programmes. It specifies a set of environment variables, library routines and compiler directives to be used for parallelisation in a shared memory environment as illustrated in Figure 2.7.

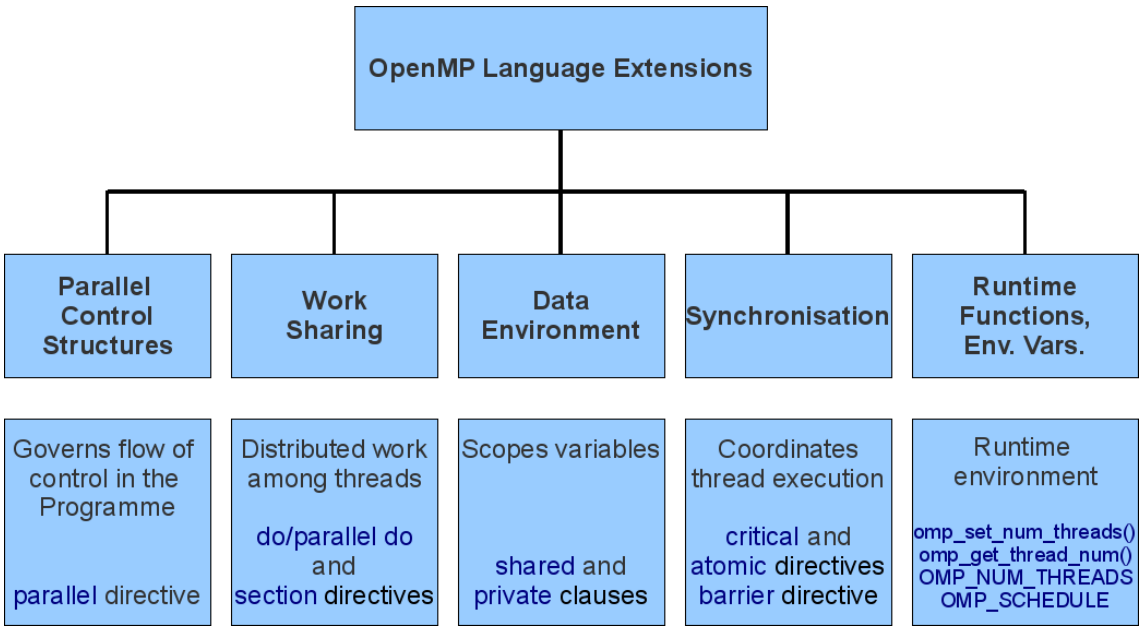


Figure 2.7: The OpenMP Language Extensions.

OpenMP was designed to directly access the memory of the system with low latency

and very fast shared memory locks. Some of the advantages of using the OpenMP library instead of using simple threading software libraries are [82]:

- It is intuitive and comparatively easy to introduce into a programme.
- It is portable across different operating systems, architectures and compilers.
- The compiler is able to make architecture-specific optimisations.

MPI and OpenMP are both extensively used for parallelisation in scientific computing [81]. In a distributed shared memory environment, MPI is used for the "inter-node" communication between a distributed collection of computers and OpenMP is used for the "intra-node" communication between a collection of processors that share the same memory system.

2.7 Parallel Programming Constructs

This section describes the theory and practice of the parallel programming constructs that focus on threading.

2.7.1 Synchronisation

Synchronisation is a mechanism used to manage and control the order of the execution of a thread and it is also used to manage shared data. Synchronisation resolves any conflict between the threads that might produce a misbehaviour [83].

2.7.2 Critical Sections

A section of a code block is called a Critical Section when shared dependency variables reside and those shared variables have dependence between multiple threads [83]. Only one thread is allowed to access a critical section at a time by using proper synchronisation techniques. Critical Sections should be implemented in a way that multiple threads execute mutually exclusive operations for Critical Sections avoiding the simultaneous use of the Critical Sections.

2.7.3 Semaphores

The Semaphores were introduced by Edsger Dijkstra in 1968 [84] and were the first primitives to accomplish mutual exclusion of parallel process synchronisation. A semaphore can be represented by an integer sem and can be bounded by two basic atomic operations, P and V . These atomic operations are referred to as the synchronisation primitives [83]. P represents the "delay" or "wait" and V represents the "barrier removal" or "release" of a thread.

2.7.4 Locks

Locks and Semaphores are similar in concept except that when using the Locks, a single thread can handle a lock at one instance. Two simple atomic operations get performed on a lock [85]:

- "Acquire()" or "Lock()": Atomically waits for the lock state to be unlocked by another thread and sets the lock state to lock.
- "Release()" or "Unlock()": Atomically changes the lock state from locked to unlocked.

At most one thread can acquire a lock. A thread has to acquire a lock prior to the use of a shared resource otherwise it waits until the lock becomes available. When a thread wants to access a shared data item, it acquires the lock, then it performs the required operations on the shared data item and finally, releases the lock for other threads to use.

An application can have different types of locks according to the constructs required to accomplish the task. There are four different types of locks [85] and they are briefly described below.

Mutexes

The mutex is a simple lock implementation and it is often the basis to describe locks in general. A timer attribute can be also added with a mutex and if the timer expires before a release operation, the mutex releases the locked code block to any other running threads.

Recursive Locks or Recursive Mutexes

Recursive Locks may be acquired several times by a thread that currently owns the lock without causing the thread to deadlock. No other thread can acquire this type of lock until the owner releases it once for each time it has acquired it.

Read / Write Locks

The Read / Write Locks are also known as multiple-read/single-write locks. This type of lock allows simultaneous read-only access to multiple threads but limit the write access to only one thread.

Spin Locks

Spin Locks are non-blocking locks owned by a thread. The waiting threads must poll the state of a lock rather than get blocked. This type of lock is commonly used on multiprocessor systems.

2.7.5 Barrier

The Barrier mechanism is a synchronisation method where a thread from an operational set has to wait for all the other threads in that set to complete in order to be able to proceed to the next code block. The Barrier mechanism guarantees that no thread proceeds beyond an execution point until all threads have arrived at that point.

2.8 Common Parallel Programming Problems

This section describes the most common problems and their symptoms in parallel and distributed programming.

2.8.1 Number of Threads

Having a large number of threads running simultaneously can seriously degrade the performance of a parallel programme [86]. The partitioning of a fixed amount of work among a large number of threads gives each thread too little work and thus, the overhead of starting and terminating the threads increases. Also, having a large number of concurrent threads results in an overhead from having to share fixed hardware

resources.

The overhead of the initialisation and destruction process of having a large number of threads for short lived tasks can be eliminated by using a thread pool [87]. A thread pool is a collection of tasks which are serviced by the software threads in the pool. Each software thread finishes a task before taking on another.

2.8.2 Parallel Slowdown

Parallel slowdown occurs when the parallelisation of a parallel computer programme beyond a certain point causes the programme to run slower typically due to a communications bottleneck [88]. As more processing nodes are added, each processing node spends more and more time communicating than performing useful processing.

2.8.3 Race Conditions

Unsynchronised access to shared memory resources can introduce race conditions [89]. A race condition occurs when the programme results depend non-deterministically on the relative timings of two or more threads. Operations on shared states are critical sections that must be atomic to avoid any collision between the threads sharing those states. Race conditions are effectively avoided by adding a lock that protects the invariant that might otherwise be violated by interleaved operations.

2.8.4 Deadlock

Deadlocks occur when a thread is blocked waiting on a resource of another thread that will never become available [69]. A deadlock is often associated with the incorrect use of locks but it can also happen any time a thread tries to acquire exclusive access to two or more shared resources. Deadlock can occur only when the following four conditions are met:

- Access to each resource is exclusive.
- A thread is allowed to hold one resource while requesting another.
- No thread is willing to relinquish a resource that it has acquired.

- There is a number of threads trying to acquire resources and where each resource is held by one thread and requested by another.

The most effective technique to avoid deadlocks is to replicate a resource that requires exclusive access so that each thread will have its own private copy of the data item. Hence, each thread will access its own copy without the need to lock it and if necessary, the copies can be merged into a single shared copy at the end.

2.9 Summary

This chapter introduced the major concepts and components required to make parallel and distributed computing a reality. The design of a distributed computing system is a very complicated task. It requires a solid understanding of the design issues and of the theoretical and practical aspects of their solutions.

Distributed Computing covers the area formerly known as Meta-computing and is the pre-cursor to what we would currently call the Grid. The Grid is typically used to solve problems that would traditionally have run on a single High Performance Computer, but due to memory, storage and/or computational demands it is forced to execute across multiple resources.

CHAPTER 3.

GRID COMPUTING

Computational Grids combine heterogeneous, distributed resources across geographical and organisational boundaries. Grids may be formed to provide computational power for CPU-intensive simulation, high-throughput computing for analysing many small tasks or for data intensive tasks such as those required by the LHC Experiments.

This chapter discusses in detail the main concepts and components that combined make computational Grids possible.

3.1 Introduction

In 1998 Ian Foster and Carl Kesselman provided the first definition of what a Grid is: "*A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities*" [32].

There have been many other attempts to define what a Grid is: "*a grid is a software framework providing layers of services to access and manage distributed hardware and software resources*" [90] or a "*widely distributed network of high-performance computers, stored data, instruments, and collaboration environments shared across institutional boundaries*" [91].

In 2001, Foster, Kesselman and Tuecke refined their definition of a Grid to "*coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organisations*" [36]. The latter definition is the most commonly used today to define a Grid.

Foster later provided a three point check-list that could be used to understand what can be identified as a Grid system. A Grid, according to Ian Foster [92]:

1. Coordinates resources that are not subject to centralised control
2. ...using standard, open, general purpose protocols and interfaces
3. ...to deliver non-trivial Quality of Service (QoS).

Without a single centralised point of control, networks of trust must be established. Collaborations create Virtual Organisations (VOs) which span traditional organisations and can be formed dynamically. Users and resources can then be authorised on the Grid based on their membership of a particular VO.

All of the above are only possible through the adoption of standard, open and general purpose protocols and interfaces otherwise it will be impossible for all the different system components to interoperate. The wide range of hardware and software available on a Grid means that the only hope for interoperability is that an application written for one middleware platform can communicate in the same language as another.

The delivery of non-trivial Quality of Service (QoS) provides the motivation to overcome all of these hurdles. As network speeds have increased, it has become feasible to harness massive amounts of computing power across multiple domains utilising resources that might otherwise be idle. Hence, we are considering how the components that make up a Grid can be used in a coordinated way to deliver combined services, which are appreciably greater than the sum of the individual components.

There are three main characteristics that distinguish a Grid from other common distributed systems [93]:

- Heterogeneity: A multiplicity of Grid resources are heterogeneous and might span numerous administrative domains across geographically distributed distances.
- Scalability: A Grid is able to grow from few resources to a huge global infrastructure.

- Adaptability: With so many resources and services contributed by multiple geographically distributed organisations, the probability of resource and service failures is extremely high. The Grid applications and the resource managers must dynamically adapt their behaviour to extract the maximum performance from the available resources and services.

3.2 Architecture

The Grid is composed of multiple layers with higher layers making use of the functionality provided by lower layers. This is also referred to as the "hourglass model" [94], where the neck defines a limited number of key protocols, which can be used by a large number of applications, to access a large number of resources. The key layers that are required in a Grid are shown in Figure 3.1 and are discussed in the following subsections.

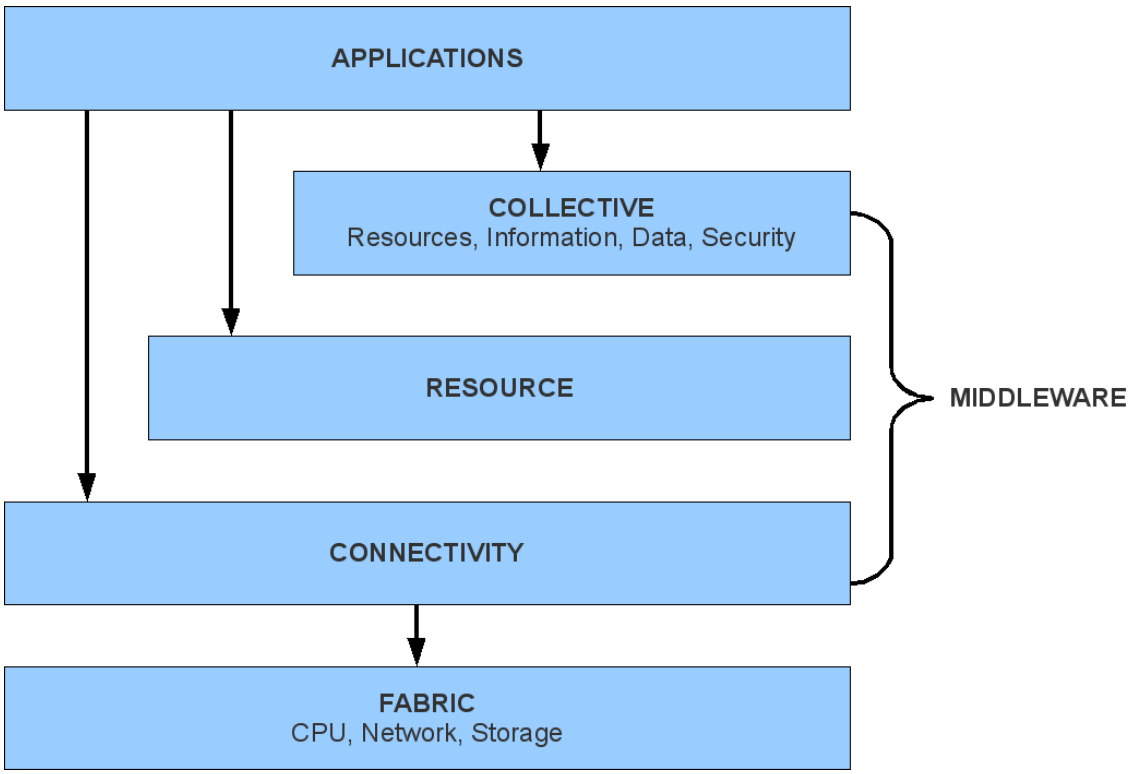


Figure 3.1: The layered architecture of the Grid. From [32].

3.2.1 Fabric

The fabric layer comprises all the resources geographically distributed across the world and accessible from anywhere on the Internet. These "resources" are logical

entities such as a distributed file system, computer cluster, PCs or Workstations, storage devices and databases. Hence, computational resources, high performance networks, storage devices and scientific instruments all combine to form the underlying fabric which forms a Grid. The fabric layer provides the resource specific implementations of operations that will be required by the resource layer.

All the available resources on a Grid should implement introspection and resource management mechanisms. The introspection mechanisms permit the discovery of their structure, of their capabilities and of their state and the resource management mechanisms provide control over the delivered quality of service.

3.2.2 Connectivity

The connectivity layer defines the core communication and authentication protocols required for the Grid. The communication protocols enable the exchange of the data between the resources of the fabric layer while the authentication protocols provide secure mechanisms to identify the users and the resources. Thus, this layer binds the fabric resources together by providing the core communication and the security protocols to support the information exchange between the Grid resources in the fabric layer.

In order to support transparent access to the resources, a single sign-on authentication mechanism is required and without it, the users would have to verify their identity before using every single resource on a Grid.

3.2.3 Resource

The role of the resource layer is to allow the user of a Grid to interact with the remote resources and services. It defines the protocols for the secure negotiation, initiation, monitoring and control of the sharing operations on the individual resources. The resource protocol layers form the “neck” of the “hourglass model” architecture and thus, should be limited to a small and focused set. Secure connections are established through the connectivity layer to the resources in the fabric layer.

There are two classes of protocols in the resource layer: the information and the management protocols. The information protocols are used to obtain information from a Grid regarding the state and the structure of a resource and the management protocols are used to negotiate the access to a shared resource by specifying a set of resource requirements such as the QoS.

3.2.4 Collective

The collective layer provides services that combine all of the resources, represented by the resource layer, into a single global image. The collective layer provides protocols and services associated with a collection of resources and it defines the protocols for coordinating the utilisation of multiple resources.

3.2.5 Applications

The applications layer comprises the user applications that operate within the environment of a Virtual Organisation. Developers can use the services offered at the lower levels to compose applications that can take advantages of the resources within the Grid.

The application layer includes the high-level user applications in a Grid. The applications are able to utilise the implementations of protocols defined within each lower layer by using the appropriate APIs provided by a Grid middleware. This layer is the one that the users of a Grid interact with.

3.3 Open Standards

Open standards are essential to ensure the interoperability and the re-use of the components in a Grid environment. The Open Grid Forum (OGF) [95] is leading the global standardisation effort for the Grid computing and trying to accelerate the adoption of the Grid computing worldwide. The OGF was formed in 2006 by a merge of the Global Grid Forum (GGF) [96] and the Enterprise Grid Alliance [97].

3.3.1 OGSA

The Open Grid Services Architecture (OGSA) [98] was the first Grid standard proposed in 2002. The OGSA defines a set of standard protocols and interfaces for managing the resources as part of a Service Orientated Architecture (SOA). The goal of the OGSA is to standardise all the common services aiming to boost the interoperability between the services by specifying a set of standard interfaces for these services.

The OGSA stretches the existing Web Services framework to provide additional functionality required by a Grid Service, such as creation, destruction, discovery and notification. A Grid service is *"an extended web service that provides a set of well-defined interfaces and that follows specific conventions"* [98].

The Open Grid Services Infrastructure (OGSI) [99] defines a set of conventions and extensions on the use of the Web Service Definition Language (WSDL) and the XML Schema to enable stateful Grid services. The WSDL is used to describe the Web Service interfaces and the XML Schema is used to complete those descriptions between a service and a client. The OGSI was replaced by the Web Services Resource Framework (WSRF) in 2004.

3.3.2 WSRF

In 2004, the standard was proposed by the Globus Alliance [100], IBM [101] and HP [102] and was standardised by the Organisation for the Advancement of Structured Information Standards (OASIS). The Web Services Resource Framework (WSRF) [103] has been designed to solve the disadvantages of the OGSI specification [104]: it is too large, it does not work well with the existing Web Services and it is too object-oriented.

The WSRF is concerned with the creation, addressing, inspection, and lifetime management of the stateful resources. The WSRF retains most of the functionality of the OGSI, but it is repackaged into six standards using existing Web Service standards. The WSRF uses the WSDL version 1.1 for the interface definition and it explicitly separates a stateless Web Service from a stateful Grid resource wrapped by a web service. The resource or state information of an interaction is specified explicitly by the client during an interaction.

A normal Web Service is stateless; it contains no data between invocations. On the other hand, a client of a stateful Grid service can communicate with the resource services which allow data to be stored and retrieved. The composition of a stateful resource and a Web service that participates in the implied resource pattern is termed a WSResource. The framework describes the WS-Resource definition, and describes how to make the properties of a WS-Resource accessible through a Web Service interface, and how to manage it during the WS-Resource's lifetime.

3.4 Grid Middleware

A Grid Middleware implementation can be seen as a layer between an application programme and a network, managing all the interactions between different programmes across heterogeneous computing platforms distributed around the world. It enables the sharing of heterogeneous resources and it is installed and integrated into the existing infrastructure of the involved Virtual Organisations, providing a special layer placed among the heterogeneous infrastructure and the specific scientific programmes.

The Grid middleware provides users with seamless computing ability and uniform access to the available resources in a heterogeneous Grid environment overcoming several challenges inherited from the nature of the Grid as described in Section 3.1; the heterogeneity in grid environments, the multiple administrative domains and autonomy issues and the scalability issues.

Several Grid middleware systems have been developed as a result of various academic research projects led by different organisations. These Grid middleware systems provide a grid-computing infrastructure where users access computer resources without knowing where these resources are coming from. Some of the most common middleware implementations are discussed in this section.

3.4.1 Globus Toolkit

The Globus project, started in the late 1990s, originated from the I-WAY project [105] in the United States. The Globus Toolkit (GT) has produced many of the fundamental standards and components that underly many of the Grids today. Version 2

of the toolkit, released in 2002, provides "non-Web Service" implementations of features such as GridFTP, which still form the basis of many Grids today. GT version 3 includes OGSA-compliant services, called the Web Services (WS) components, and many other services, programmes, utilities, which are non-OGSA services and are called the pre-WS components, such as the function modules in GT2.

The version 4 of the toolkit, released in 2005, was the first implementation of OGSA and WSRF compliant version for supporting the Web Services. It includes a complete implementation of the WSRF standard and containers are provided for Java, Python and C which implement all the standard requirements such as the security, discovery and management. GT4 provides service components in common runtime components, security, information management, execution management and data management. The components of the GT4 can be seen in Figure 3.2 and the most important components are described below.

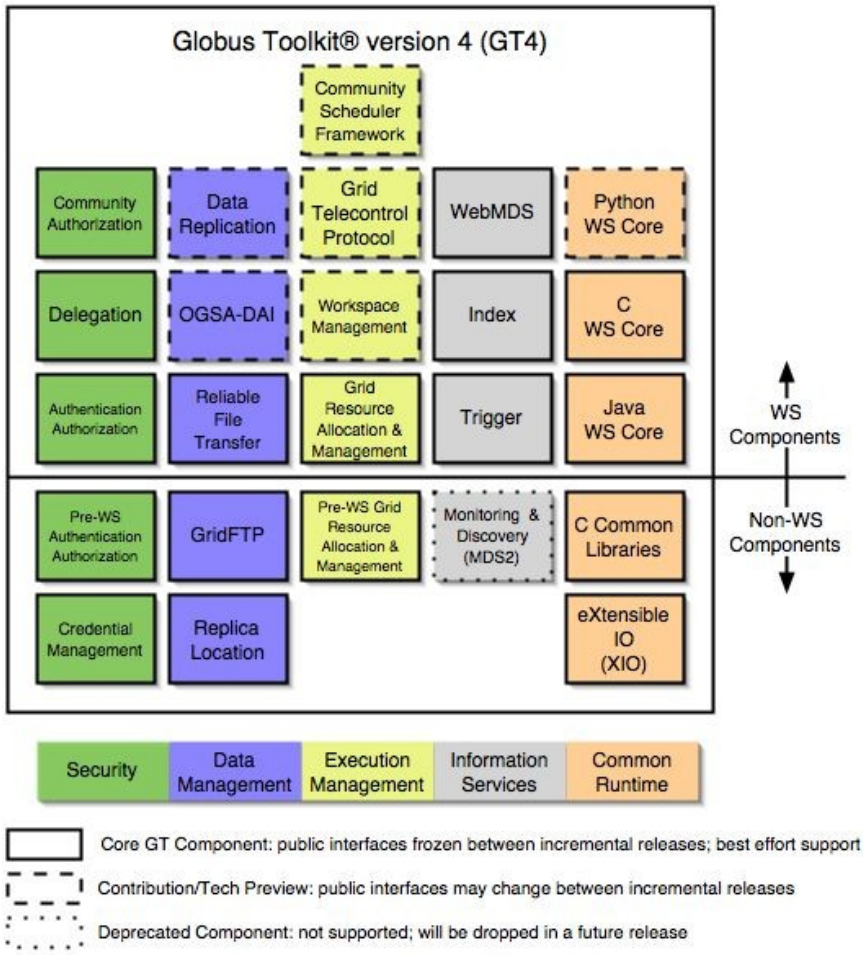


Figure 3.2: Globus Toolkit 4 Architecture. From [106].

Globus Resource Allocation and Management

Mechanisms to start and monitor jobs on remote machines are needed in order to be able to run jobs on a Grid. By using the Globus Resource Allocation and Management (GRAM), it is possible to submit, cancel and check the status of remote jobs. A GRAM client is used to interact with the remote machines.

The remote machines run the server component of GRAM, also known as the "Gatekeeper", allowing the clients to connect. When a client is connected, GSI is used for authentication and once a client is authenticated and authorised, the Grid user who submitted the job is mapped to a local user, which runs a GRAM job manager managing the given job. GRAM does not include any scheduling logic and therefore, it interacts with a scheduling system such as Condor. The job manager runs while the job is active and can be queried by the client regarding any changes in the status of the job.

GRAM also handles the staging of the files, which is the transfer of files to and from the remote machine, using the Global Access to Secondary Storage (GASS). GASS is designed to enable easy access to remote files possibly stored on the submitting machine.

GridFTP

GridFTP is a secure, reliable and high performance data transfer protocol designed for wide-area networks with a high bandwidth Grid environment. It is based on the File Transfer Protocol (FTP) with extended functionality to offer features specifically needed in a Grid environment. The Globus Security Infrastructure (GSI) is used to secure both the control and the data channel of the FTP communication.

GridFTP supports parallel data transfers that involves splitting a given file into chunks and transferring the chunks simultaneously from different servers that store the same file. When only one copy of the data is available, parallel data transfer still has the potential to offer increased performance because the individual data streams can be routed individually. Reliable data transfers are also needed in a Grid environment and therefore, GridFTP is able to restart and to resume the failed transfers.

Replica Management

Multiple copies of the same data, called replicas, are stored in a Grid environment for the sake of performance, robustness and scalability. The Replica Management manages the data and it can select the best suited replicas for a given scenario. By using the Replica Management, the users can create, delete and find the requested replicas and they can also obtain information regarding the resources storing them.

The Replica Management uses the Replica Catalogue in order to store meta-data information. The Replica Catalogue stores the association between logical file names (LFN) and the physical file names (PFN) which are often stored as URLs that can be used to access the files. This scheme allows the PFN to change without any misbehaviour and also, the best replica is chosen during the runtime, thus, applications are not bound to a specific instance of a data set. The Replica Catalogue also stores information about the resource that stored the replica and this information is used by the Replica Management to select the best suited replicas for a given scenario. Unfortunately, the Replica Management system has many scalability issues due to its non-distributed nature.

Grid Security Infrastructure

The Grid Security Infrastructure (GSI) is the most widely used component of the Globus Toolkit. It provides the tools and the services for the authorisation and authentication of the users using a “Public Key Infrastructure” (PKI). The users create a short-lived proxy which is then used to authenticate with the resources. Organisation policies normally limit the lifetime of the proxy to one day or even less. Since it is unacceptable to enter a password every time a communication is initiated in a Grid environment, GSI supports “single sign-on” authentication. With the single sign-on feature, a user enters his password only once and then remains authenticated for all the Grid elements. The MyProxy credential store provides a secure location to store long-lived credentials which can then be retrieved by authorised services.

Monitoring and Discovery Service

The Monitoring and Discovery Service (MDS) is a system for publishing and querying the status of resources and their configuration and it can be used with the

GRAM service to create a scheduler for a Grid.

MDS consists of three major components: the “Grid Index Information Service” (GIIS), the “Grid Resource Information Service” (GRIS) and the “Information Providers” (IPs). IPs are the interfaces that receive information about a resource from resource-specific monitoring systems. GIIS collects the information from several GRIS enabled resources to allow searching through the information to find a suitable resource. A GIIS can connect to another GIIS forming several levels of GIISs; a Grid could have one GIIS per site and one global Grid-level GIIS.

MDS supports two schemas: the MDS Monitoring and Discovery Service core schema containing basic information and the “Grid Laboratory Uniform Environment” (GLUE) [107] schema, which is an effort between a lot of Grid projects to define the information needed to represent the Grid resources. The information offered by MDS about the resources could be the load status, the CPU, the disk, the memory and the network information.

The Globus Alliance has announced the release of the Globus Toolkit 5 in late 2009 [108]. A major change will be the abandonment of GRAM4 in favour of an enhanced version of GRAM2, called GRAM5, which will solve scalability issues and add new features. Also, the monitoring and discovery tasks currently performed by MDS will be replaced by a Crux-based Integrated Information Services (IIS) [109].

3.4.2 Condor

The Condor project, developed at the University of Wisconsin-Madison, started in late 1980s. It is a freely available project designed to encapsulate and run large collections of distributed computing resources with the aim of giving scientists more access to available computing power. Condor is a distributed batch computing system and its main focus is on high-throughput computing (HTC) and on CPU harvesting giving users the ability to run huge numbers of tasks over long periods of time [110].

Condor provides services for Job Queuing, Job Scheduling, Resource Monitoring and

Resource Management. When the users want to submit a job, they have to specify their requirements in a small file called a ClassAd and the Condor system will take care of the rest. A typical Condor installation might exploit and use all the wasted computing power in idle workstations.

Condor's architecture consists of three main components: the Agents, the Matchmakers and the Resources. The core of the system is the Matchmaker. Users specify their requirements using the "Classified Advertisement" (ClassAd) language and submit them to the Agents that will find Resources suitable for the jobs via a Matchmaker. ClassAds allow users to define custom attributes for resources and jobs such as the memory and the CPU. On the other side, the Resources publish their information to the Matchmaker and the Matchmaker then matches job requests with the available Resources. Every community of Agents and Resources that is served by a Matchmaker is known as a "pool". Every single "pool" will typically be administered by a different institution or organisation.

An important feature of Condor is that it saves the entire state of a programme with checkpoints and in the event of a resource failure, the job will be migrated to another available resource and it will be restarted from the saved checkpoint.

Condor-G is the combination of Condor and Globus Toolkit as illustrated in Figure 3.3. Condor is used for the local job management while Globus is used to perform the secure inter-domain communication.

Condor-G contains a GASS server, used to transfer the executable, the standard input (stdin), the standard output (stdout), and standard error (stderr) files to and from the remote job execution site. Condor-G uses the GRAM protocol to contact the remote Globus Gatekeeper to request that a new job manager should be started. GRAM is also used to monitor the status of the job and it is also in charge to detect and handle any potential resource crashes.

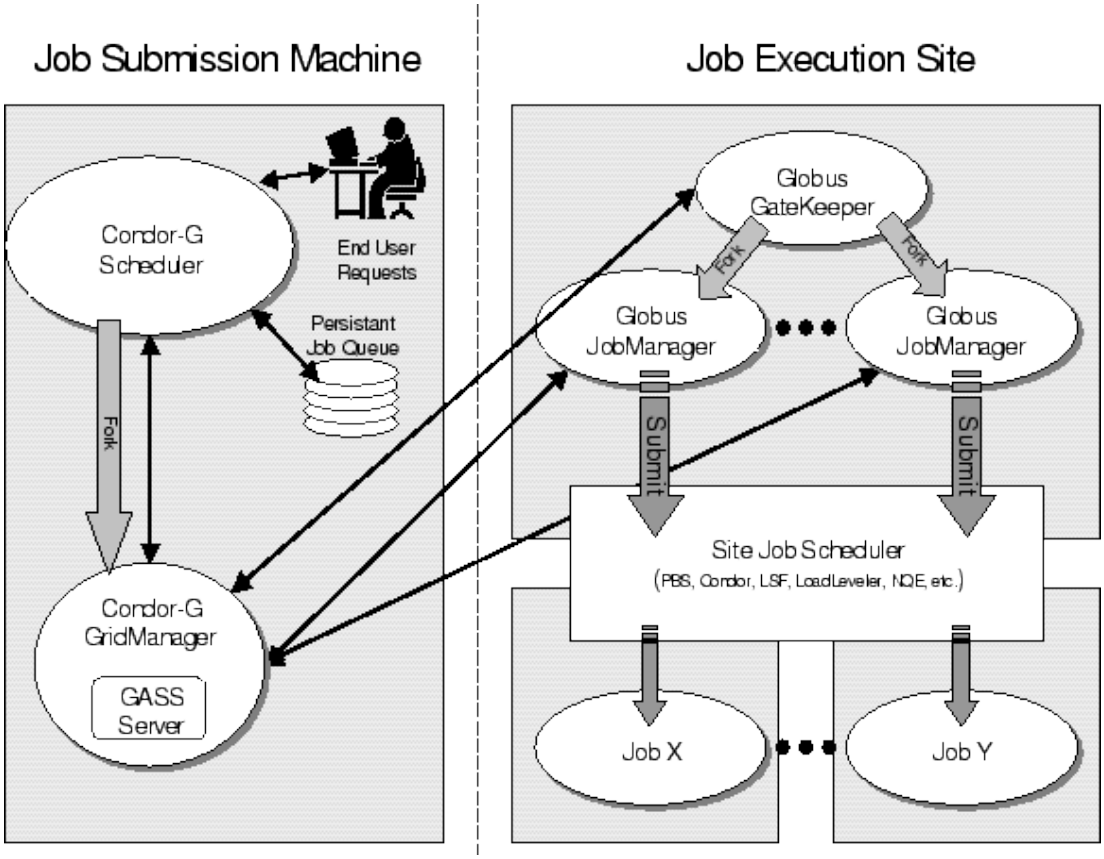


Figure 3.3: Remote Execution by Condor-G on Globus resources. From [111].

3.4.3 LCG

The LHC Computing Grid (LCG) is a worldwide computational Grid targeted at providing computational power and storage space for the requirements of the LHC experiments. To achieve this, the LCG version 2 (LCG-2) takes its software components from multiple middleware projects such as the Globus, the Condor and the European DataGrid (EDG) [112] project. The LCG project is also closely related to the Enabling Grids for E-SciencE (EGEE) project. The most important LCG-2 components [113] will be described below.

Workload Management System

In order to submit jobs to the LCG-2, the users need to log in to a machine with a User Interface (UI) installed that interacts with the Workload Management System (WMS). When the jobs have been submitted, the UI connects to the Resource Broker (RB) and then, the RB handles the scheduling, the submission of the jobs to the remote

machines, the transferring of the files and the logging. The RB, at first, uses the Globus Security Infrastructure (GSI) to authenticate the Grid users and then it copies the input sandbox, which is a collection of files stated by the user on the UI required for the job, to its local storage. The WMS uses the Matchmaker, the Information System (IS) and the Replica Location System (RLS) from the Data Management System (DMS) to find the best suited resources for a given job considering many requirements such as the user's specified constraints and the queue lengths.

The Matchmaker works as the matchmaking mechanism of Condor and it also uses the ClassAds files. Condor-G is used to submit the jobs to the best suited Computing Element (CE) and along with the job, a monitoring job called the “Grid Monitor” is also submitted. The CEs start the Gatekeeper, accepting incoming jobs from Condor-G, and start a GRAM job manager. The Job Manager submits the job to a site-specific batch system.

The Job Manager is only used to submit and to control the jobs but not to query about their status. This task is performed by the Grid Monitor job submitted along with the jobs and the reason for using the Grid Monitor and not the Job Manager to monitor the status of the jobs, is performance; the Job Managers use a lot of resources on the CE since the Gatekeeper starts a Job Manager for every job and these Job Managers run until the jobs finish successfully or unsuccessfully. The Grid Monitor on the other hand can monitor all the jobs from the same user on a CE and can be instructed to exit as soon as jobs have been submitted to the batch system, resulting in a much smaller load on the CE. The machines running the jobs are called the “Worker Nodes” (WNs).

Data Management System

The Data Management System (DMS) is composed of the Replica Location System (RLS) and the Storage Elements (SEs). The RLS is queried to find and retrieve the data, the meta-data and the information about the SE storing the data. The SEs are computers with access to large amounts of data storage. A GridFTP server is running on every single SE in order to make the storage available to the Grid users once they have the PFN of the required data. The PFN is discovered by querying the RLS which contains the correlations between the LFNs and the PFNs.

The RLS system used in the LCG-2 is not the same as the one used in the Globus Toolkit; the Globus Toolkit RLS was developed in collaboration with the EDG but their paths divided and two versions of the RLS were implemented. LCG-2 uses the EDG distributed version of the RLS.

Information System

The LCG-2 Information System (IS) is based on the Monitoring and Discovery Service (MDS) system from the Globus Toolkit, using the GLUE schema to organise the information. The IS is a modified version of the MDS system that deals with scalability and robustness issues. The IS uses the information providers to provide information to a GRIS, and the GRIS relays this information to a site level GIIS. There is no regional or Grid-level GIIS in the IS because the overhead of the regional GIIS-system was decreasing the overall performance and also, the Grid-level GIIS was unstable when collecting information from many sites and being queried by many users and RBs at the same time.

LCG-2 is using the Berkeley Database Information Index (BDII) to serve as the Grid-level information service. The BDII consists of two LDAP-servers where one of them contains a read-only database and the other, a write-only database. The BDII executes queries from the users and the RBs on a read-only database whilst updates a write-only database with information coming from the GIISes.

Another system being used for monitoring and information is the Relational Grid Monitoring Architecture (R-GMA) [114]. The R-GMA makes all the monitoring information appear like one large relational database that may be queried by the users and by Grid applications to find the information required. As illustrated in Figure 3.4, it consists of the Producers which register themselves with the Registry and publish the information into R-GMA, and the Consumers which subscribe.

The Logging and Bookkeeping (LB) database service is updated by the WMS and the CE as jobs progress through the system and the users can query the status of their jobs via the WMS.

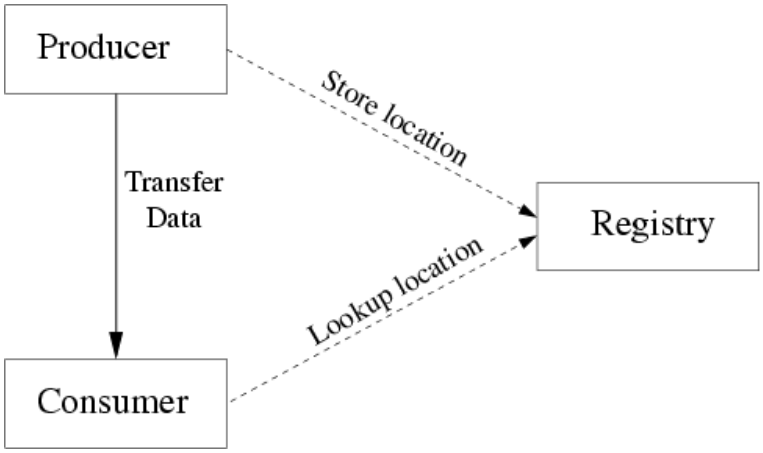


Figure 3.4: Components of the R-GMA. From [115].

Authorisation and Authentication System

The Virtual Organisation Membership Service (VOMS) [116] is being used to manage the membership information about a user's role and privileges within a VO. When a proxy is created, a VOMS server is contacted and it returns a mini certificate known as the "Attribute Certificate" (AC) which is then signed by the VO. The AC contains the user's membership information and any associated roles within the VO.

3.4.4 gLite

The gLite middleware is based on the EDG and the LCG middleware implementations. The convergence with the LCG-2 middleware was reached in May 2006 when gLite version 3.0 was released and became the official middleware for the EGEE project and it is currently the default Grid middleware for the WLCG. The main differences between the gLite 3.0 and the LCG-2 middleware implementations are outlined below.

Workload Management System

In gLite 3.0, there is a Web Service to the gLite WMS, known as the WMProxy, that allows not only single job submissions but also collections of jobs submissions, known as bulk submissions. This is, certainly, a much more efficient way compared to the LCG-2 WMS's single job submissions.

The gLite's RB uses information from the “Information Super Market” (ISM) to match the requirements for a job with the resources. The CE can retrieve and store information to the ISM. Condor-G is used for the job submission to a gLite CE and Condor daemons are used to submit jobs via the Batch Local ASCII Helper (BLAH) [117] abstraction layer. An alternative architecture is available. The Computing Resource Execution And Management (CREAM) [118] service is a simple and lightweight service for job management operation at the CE-level.

Data Management System

The most important difference between the gLite's and the LCG's data management system is that the gLite uses the File Transfer Service (FTS) [119]. The FTS is a low level data movement service where a user can schedule asynchronous and reliable point-to-point file replication from the source to the destination while participant sites can control the network usage. The FTS manages the transfers using the GridFTP.

3.5 The CMS Computing Model

The CMS distributed computing and analysis model [120] is designed to serve, process and store the large amount of data that will be generated when the CMS detector starts taking data. The data will be distributed and processed over many computing centres. A set of CMS-specific Workload and Data Management tools and services have been deployed in order to enable the CMS distributed analysis. These CMS-specific tools and services have been built on top of the existing Grid services [59].

3.5.1 Data Management System

The CMS DMS provides the infrastructure to manage the large amounts of data produced, processed and analysed in a distributed computing environment. Files are grouped together into blocks of files to simplify bulk data management and transfer and these file blocks are then grouped into datasets. A file block contains files that can be processed and analysed together. The tracking of the location of the data is 'file block-based' and the tracking information provides the name of the sites hosting the data but not the physical location of the files nor the composition of the file blocks. In order to

avoid scaling storage issues and to optimise the data transfer, the average file size is at least 1GB and this is accomplished by merging smaller output files produced by individual jobs into larger files. This section describes the CMS-specific DMS tools and services.

Dataset Bookkeeping Service

The Dataset Bookkeeping Service (DBS) [121] catalogues the CMS-specific data definitions, such as the algorithms and the configurations used to process the data, and it provides the means to discover, describe and use the CMS events data. The DBS is used in the analysis and production systems via a DBS API and the users can discover the data via a Web Browser or a Command Line Interface tool (CLI).

The DBS is a multi-tier web application that supports many database systems such as the ORACLE, the MySQL and the SQLite. A single instance Global DBS hosted at CERN is used to describe CMS-wide data and many local DBSs are used to describe data produced by the Monte Carlo production, physics groups or individual physicists.

Local Data Catalogue

A CMS-specific application is aware only of the logical files and relies on a local catalogue service to gain access to the physical files. Every CMS site has a Trivial File Catalogue installed that builds site-specific physical file paths consisting of the logical file name and the access protocol.

Conditions Data

The conditions data describe the alignment and the calibration of the detector. CMS uses a caching system for the conditions data, the Frontier [122], because these conditions data are frequently accessed by many processing jobs worldwide. The Frontier queries a central database located at CERN and then caches the results with the help of the Squid proxy server [123] deployed at every CMS site. The CMS applications then use an instance of a Squid proxy server to read the conditions data.

PhEDEx

All the CMS data placement and transfer operations are performed by the Physics

Experiment Data Export (PhEDEx) [124] system where distinct storage areas are represented as a node and the links between the nodes define the transfer topology. The transfer of the data occurs when a user requests a specific set of data to a node via a web page and this operation has to be approved by the Data Manager of this node. The user has to specify the destination node only; the optimal source node is determined automatically by PhEDEx that calculates the 'least-cost' path according to the available file replicas, the recent transfer rate and the size of the queue over that link.

3.5.2 Workload Management System

The CMS-specific WMS is responsible for the user's processing requests, the creation of the jobs that process the data, the submission of the jobs to a local or to a distributed system, the monitoring of the jobs and the retrieval of their outputs. CMS, uses two WMS tools; the Monte Carlo Production Agent (ProdAgent) [125] and the CMS Remote Analysis Builder (CRAB) [126]. The ProdAgent is optimised to perform the previously mentioned operations in a controlled environment whereas CRAB is optimised for user analysis.

ProdAgent

The architecture of the Monte Carlo (MC) production system consists of the Request System (ProdRequest) that acts as a front-end application for the user production request submissions into the production system; the Production Manager (ProdManager) that manages these user requests, performing accounting and allocating work to a collection of Production Agents (ProdAgents). The Production Agents request for work when resources are available and manage the job submissions and the resubmissions.

CRAB

CRAB has been developed as a user-friendly application to handle the CMS data analysis in a local or a distributed environment, hiding from the user the complexity of the Grid and of the CMS services. CRAB is coded in Python and it provides plug-ins for various Grid middleware implementations such as the gLite [127], the OSG [52] and the ARC [54] used in the NorduGrid.

The user can submit and manage jobs using either a direct CRAB client or an intermediate CRAB Server. The CRAB Server automates the analysis workflow, handling the errors and the resubmissions automatically. The functionalities that CRAB provides, as illustrated in Figure 3.5, are:

- *Data discovery and location*. Queries the CMS-specific data catalogues, DBS and PhEDEx to find which data is needed and where they are located.
- *Job preparation*. Packs the code of the user and the environment and sends it to the remote sites.
- *Job splitting*. Decides how to split the complete set of event collections among several jobs, each of which will access a subset of the event collections in the selected dataset, according to the requirements of the user.
- *Job submission*. Submits the jobs to the CMS sites.
- *Job monitoring*. Monitors the status of the submitted jobs by querying the Grid services. A more elegant approach will be described in the next section.
- *Output data handling*. Copies the produced output to a remote site or, if the output size is small, returns it to the user. Finally, it publishes the produced data into a local DBS to be used by other physicists.

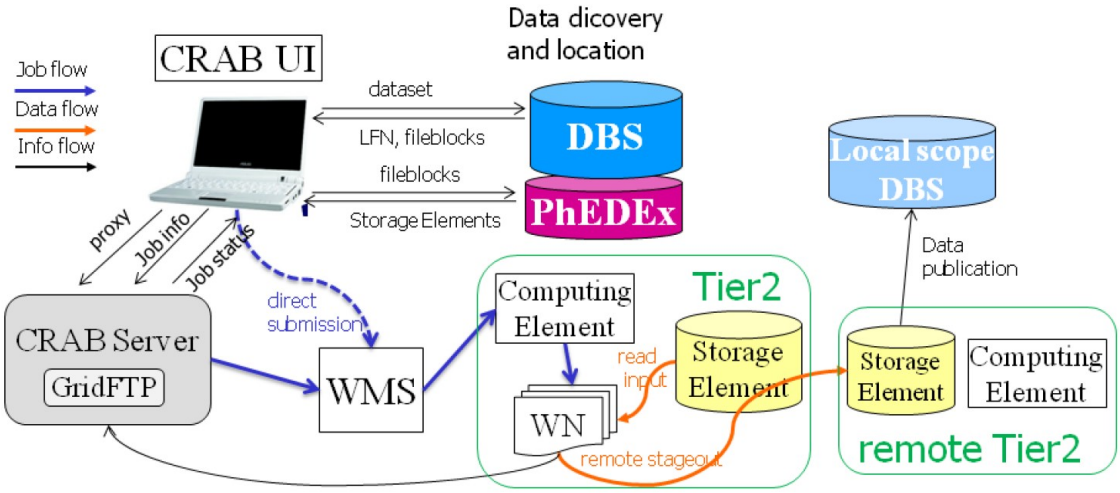


Figure 3.5: The CRAB Workflow Schema. From [59].

3.6 Monitoring with the Experiment Dashboard

The Worldwide LHC Computing Grid (WLCG) provides data storage and computational resources to the High Energy Physics (HEP) community. Operating the heterogeneous WLCG infrastructure, which integrates more than 140 computing centres in 33 countries all over the world, is a complicated task.

Reliable monitoring is a necessary condition for the production quality of the distributed infrastructure. Monitoring of the computing activities of the main communities using this infrastructure in addition provides the best estimation of its reliability and performance.

The importance of flexible monitoring tools focusing on the applications has been demonstrated to be essential not only for “power-users” but also for individual users. For the power users, a very important feature is to be able to monitor the resource behaviour to detect the origin of failures and optimise their system. They also benefit from the possibility to “measure” efficiency and evaluate the quality of service provided by the infrastructure. Individual users are typically scientists using the Grid for analysis data, verifying hypothesis on data sets they could not have available on other computing platform. In this case, reliable monitoring is a guide to understand the progress of their activity, identify and solve problems connected to their application.

This is essential to allow efficient user support by “empowering the users” in such a way that only non-trivial issues are escalated to support teams, for example, jobs on hold due to scheduled site maintenance can be identified as such and the user can decide to wait or to resubmit.

In order to monitor the computing activities of the LHC experiments, several specific monitoring systems were developed. Most of them are coupled with a specific Data Management and a Workload Management System of the LHC Virtual Organisations (VOs), for example with PhEDEx [124], Dirac [128], Panda [129] and AliEn [130]. In addition, there was a generic monitoring framework developed for the LHC experiments; the Experiment Dashboard. If the source of the monitoring data is not VO-specific, the Experiment Dashboard monitoring applications can be shared by several

VOs. Otherwise, the Experiment Dashboard offers experiment-specific monitoring solutions for the scope of a single experiment.

The Experiment Dashboard system provides monitoring of the WLCG infrastructure from the perspective of the LHC experiments and covers the complete range of their computing activities. The goal of the project is to provide transparent monitoring of the computing activities of the LHC VOs across several middleware platforms such as the gLite, the OSG and the ARC.

Currently the Experiment Dashboard covers the full range of the LHC computing activities: job processing, data transfer and site commissioning, and it is used by all the four LHC experiments, in particular by the two largest ones, the ATLAS and the CMS.

The Experiment Dashboard provides monitoring to various categories of users:

- Computing teams of the LHC VOs.
- VO and WLCG management.
- Site administrators and VO support at the sites.
- Physicists running their analysis tasks on the EGEE infrastructure.

The Experiment Dashboard allows to estimate the quality of the infrastructure and to detect any problems or inefficiencies. Furthermore, it provides the necessary information to conclude whether the LHC computing tasks were accomplished. The main computing activities of the LHC VOs are the data distribution, the job processing, and the site commissioning. The Experiment Dashboard covers all these activities.

The Experiment Dashboard is intensively used by the LHC community. According to the Dashboard Web Statistics web page [131], only for the CMS Dashboard, more than 2,500 unique visitors use it per month and approximately 30,000 pages are accessed daily.

3.6.1 Experiment Dashboard Framework

The structure of the Experiment Dashboard monitoring system consists of the information collectors, the data repositories, normally implemented in ORACLE database, and the user interfaces. The Experiment Dashboard uses multiple sources of information such as [60]:

- Other monitoring systems, like the Imperial College Real Time Monitor (ICRTM) [132] or the Service Availability Monitoring (SAM) [133].
- gLite Grid services, such as the Logging and Bookkeeping service (LB) [134] or CEMon [118].
- Experiment specific distributed services such as the ATLAS Data Management services or distributed Production Agents for CMS.
- Experiment central databases such as the PANDA database for ATLAS.
- Experiment client tools for job submission, like Ganga [135] and CRAB.
- Jobs instrumented to report directly to the Experiment Dashboard.

Information can be transported from the data sources via various protocols. In most cases, the Experiment Dashboard uses asynchronous communication between the source and the data repository. For several years, in the absence of a messaging system as a standard component of the gLite middleware stack, the MonALISA [136] monitoring system was successfully used as a messaging system for the Experiment Dashboard job monitoring applications. Currently, the Experiment Dashboard is being instrumented to use the Messaging System for the Grid (MSG) [137] for the communication with the information sources.

A common framework providing components for the most usual tasks was established to fulfil the needs of the dashboard applications being developed for all the experiments. The schema of the Experiment Dashboard framework is presented in Figure 3.6.

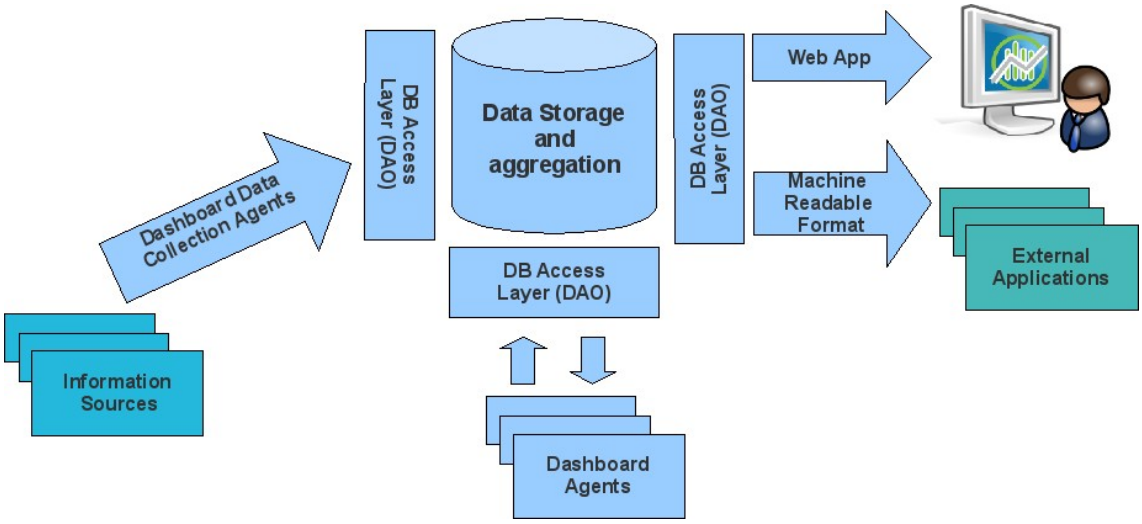


Figure 3.6: The Experiment Dashboard Framework Schema.

The Experiment Dashboard framework is implemented in the Python programming language. The tasks performed on regular basis are implemented by the Dashboard agents. The framework provides all the necessary tools to manage and monitor these "agents", each focusing on a specific subset of the required tasks, such as collection of the input data or the computation of the daily statistics summaries.

To ensure a clear design and maintainability of the system, the definition of the actual monitoring application database queries is decoupled from the internal implementation of the data repository. Every monitoring application implemented within the Experiment Dashboard framework comes with the implementation of one or more Data Access Object (DAO), which represents the "data access interface"; a public set of methods for the update and retrieval of information. Access to the database is done using a connection pool to reduce the overhead of creating new connections, therefore the load on the server is reduced and the performance is increased.

The Experiment Dashboard requests are handled by a system following the "Model-View-Controller" (MVC) pattern. They are handled by the "controller" component, launched by the apache 'mod_python' extension, that associates the requested URLs with the corresponding "actions", executing them and returning the data in the format requested by the client. All actions will process the request parameters and execute a set of operations, which may involve accessing the database via the DAO layer. When a

response is expected, the action will store it in a python object, which is then transformed into the required format (HTML page, plain XML, CSV, image) by the "view" components. Applying the view to the data is performed automatically by the controller.

All the output data produced by the Experiment Dashboard can be retrieved in HTML, so that it can be accessed by any browser. The framework of the Experiment Dashboard also provides the functionality to retrieve information in XML (eXtensible Markup Language), CSV (Comma Separated Values), JSON (JavaScript Object Notation) or image formats. This flexibility allows the system to be used not only by the users but also by other external, third party, applications. A set of command line tools is also available.

The current web page frontends are based on XSL style sheet transformations over the XML output of the HTTP requests. In addition, in some cases the interfaces follow the AJAX model, triggering javascript issues both in debugging and browser support. Recently, support for the Google Web Toolkit (GWT) [138] has been added to the framework which gives many benefits both for the users and the developers such as compiled code, easier support for all browsers and out of the box widgets.

All components are included in an automated build system based on the Python distutils, with additional or customised commands enforcing strict development and release procedures. In total, there are more than fifty modules in the framework, and fifteen of them being common modules offering the functionality shared by all the applications.

3.6.2 Job Processing and the Experiment Dashboard Applications for Monitoring

The LHC job processing activity is divided in two categories: processing raw data and large-scale Monte-Carlo (MC) production, and user analysis. The main difference between the mentioned categories is that the first one is a large scale, well-organised activity, performed in a coordinated way by a group of experts, while the second one is chaotic data processing by members of the distributed High Energy Physics community.

Users running physics analysis do not necessarily have enough knowledge about the Grid and profound expertise in computing in general. Clearly, for both categories of the job processing, complete and reliable monitoring is a necessary condition for the success of this activity.

The organisation of the Workload Management Systems (WMSs) of the LHC experiments differs from one experiment to another. While in the case of ALICE and LHCb the job processing is organised via a central queue, in the case of ATLAS and CMS, the job submission process is distributed without any central point of control as in ALICE or in LHCb. Therefore, the job monitoring task for ATLAS and CMS is much more complicated and it is not necessarily coupled to a specific WMS.

The Experiment Dashboard provides several job monitoring solutions for various use cases, namely the generic job monitoring applications, monitoring for ATLAS and CMS production systems, and applications focused on the needs of the analysis users. The generic job monitoring, which is provided for all LHC experiments, is described in more detail in the next section. Since the distributed analysis is currently one of the main challenges for the LHC computing, several new applications were built recently on top of the generic job monitoring, mainly for monitoring of the analysis jobs.

3.6.3 Experiment Dashboard Generic Job Monitoring Application

The overall success of the job processing depends on the performance and the stability of the Grid services involved in the job processing and on the experiment-specific services and software. Currently, the LHC experiments are using several different Grid middleware platforms and therefore a variety of Grid services. Regardless of the middleware platform, access from the running jobs to the input data as well as saving output files to the remote storage are currently the main reasons for the job failures.

Stability and performance of the Grid services, such as the Storage Element (SE), the Storage Resource Management (SRM) [139] and various transport protocols, are the most critical issues for the quality of the data processing. Further on, the success of the

user application depends also on the experiment-specific software distribution at the site, the Data Management System of the experiment and the access to the alignment and calibration data of the detector known as the “conditions data”.

These components can have a different implementation for each experiment and they have a very strong impact on the overall success rate of the user jobs. The Dashboard Generic Job Monitoring Application tracks the Grid status of the jobs and the status of the jobs from the application point of view. For the Grid status of the jobs, the Experiment Dashboard was relying on the Grid related systems as an information source. In the past, the Relational Grid Monitoring Architecture (R-GMA) and the Imperial College Real Time Monitor were used as information sources for the Grid job status changes.

None of the mentioned systems provided complete and reliable data. The recent development focused on improving this situation, as described later in this section. To compensate the lack of information from the Grid-related sources, the job submission tools of the ATLAS and CMS experiments were instrumented to report any job status changes to the Experiment Dashboard system. Every time when the job submission tools query the status of the jobs from the Grid services, the status is reported to the Experiment Dashboard. The jobs themselves are instrumented for the runtime reporting of their progress at the worker nodes. The information flow of the generic job monitoring application is described in the next section.

Information Flow of the Generic Job Monitoring Application

Similar to the common Dashboard structure, the job monitoring system consists of the central repository for the monitoring data (Oracle database), the collectors, and a web server that renders the information in HTML, XML, CSV, or in an image format. The main principles of the Dashboard job monitoring design are [60]:

- to enable non-intrusive monitoring; the monitoring process should not have any negative impact on the job processing itself.
- to avoid direct queries to the information sources and to establish asynchronous connections between the information sources and the data repository.

When the development of the job monitoring application started, the gLite middleware did not provide any messaging system, so the Experiment Dashboard was using the MonALISA monitoring as a messaging system. The job submission tools of the experiments and the jobs themselves are instrumented to report needed information to the MonALISA server via the 'apmon' library, which uses the UDP protocol. Every few minutes the Dashboard collectors query the MonALISA server and store job monitoring data in the Dashboard Oracle database. The data related to the same job and coming from several sources is correlated via a unique Grid identifier of the job.

Following the outcome of the work of the WLCG monitoring working groups, the existing open source solutions for the messaging system were evaluated and as a result of this evaluation, Apache [140] ActiveMQ [141] was proposed to be used for the Messaging System for the Grids (MSG). Currently, the Dashboard job monitoring application is instrumented to use the MSG in addition to the MonALISA messaging system.

The job status information presented by the Experiment Dashboard is close to the real-time status. The maximum latency is five minutes, which corresponds to the interval between the sequential runs of the Dashboard collectors. Information stored in the central job monitoring repository is being regularly aggregated in the summary tables. The latest monitoring data is made available to the users. For the long term statistics, data is being retrieved from the summary tables which keep aggregated data with hourly and daily time bin granularity.

Instrumentation of the Grid Services for Publishing Job Status Information

As it was mentioned above, information about any job status changes provided by the Grid-related sources is currently not complete and covers only a subset of jobs. This has a bad impact on the trustworthiness of the Dashboard data. Though some job submission tools are instrumented to report any job status changes at the point when they query the Grid-related sources, this query is done from the user's side. For example, when a user never requests the status of his jobs and the jobs are aborted, there is no way for the Dashboard to be informed about the abortion of the jobs. As a result, they can stay in “running” or “pending” status, unless being turned into the “terminated”

status with “unknown” exit code by a so-called “time-out” Dashboard procedure.

To overcome this limitation, the ongoing development aims to instrument the Grid services involved in the job processing to publish any job status changes to the MSG as illustrated in Figure 3.7. The Dashboard collectors consume the information from the MSG and store it in the central repository of the job monitoring data.

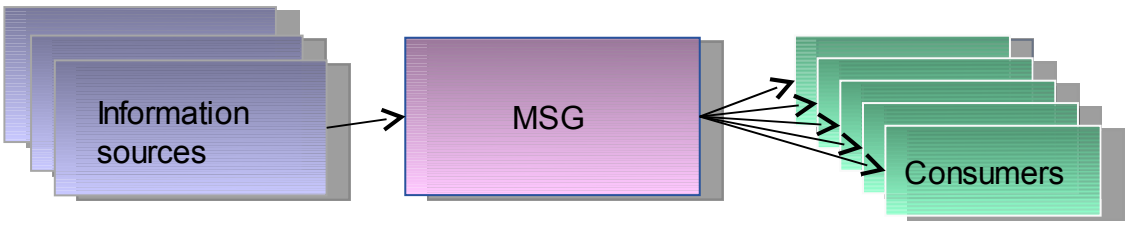


Figure 3.7: Publishing information using the MSG.

The advantages of using the MSG are numerous [62]:

- Common way of publishing information by various information sources.
- Common way of communicating between different components of the WLCG infrastructure.
- Monitoring information is publicly available to all interested parties.
- Decreasing the load of the Grid Services caused by the regular pooling of information regarding any job status changes.

When the jobs are submitted via the gLite Workload Management System (WMS), the LB service keeps full track of the job processing. The LB provides the notification mechanism which allows to subscribe to the job status changes events and to be notified as soon as events matching the conditions specified by the user happen. A new component was developed, the “LB Harvester” [142], in order to register at several LB servers and to maintain the active notification registration for each one. The output module of the harvester formats the job status message according to the MSG schema and publishes it to the MSG.

Currently, the LB does not keep track of the jobs submitted directly to the Computing Resource Execution And Management (CREAM) Computing Element (CE). The

CEMon service plays a role similar to the LB but only for jobs submitted to the CREAM CE. A CEMon listener component is being developed in order to enable job status changes publishing to the MSG. It subscribes to CEMon for notifications about job status changes and publishes this information to the MSG.

Finally, jobs submitted with Condor-G do not use the WMS service and correspondingly do not leave a trace in the LB. The job status changes publisher component was developed in collaboration with the Condor and the Dashboard teams. Condor developers have added a job logs parsing functionality to the Condor standard libraries. The publisher of the job status changes reads new events from standard Condor event logs, filters events in question, extracts essential attributes and publishes them to the MSG. The publisher runs in the Condor scheduler as a Condor job. In this case, Condor itself takes care of publishing job status changes.

3.7 Summary

This chapter introduced the major concepts and components that are required to make Grid computing a reality. The concept of a computational Grid is the idea of accessing vast quantities of computing power and data storage as easy as accessing electricity from a power grid. This idea has not yet been turned into reality but in a relatively short period of time the Grid has been developed and evolved, providing a significant amount of computing power and data storage.

The major components that form a Grid were identified and discussed along with the Grid standards and the most important Grid middleware implementations including the Globus Toolkit, the Condor, the LCG and the gLite.

Finally, the Experiment Dashboard was presented as a reliable monitoring system to monitor all the computing activities in the Worldwide LHC Computing Grid infrastructure. The aim of the project is to provide transparent monitoring of the computing activities of the LHC Virtual Organisations across several middleware platforms such as the gLite, the OSG and the ARC.

Chapter 5 discusses in depth the CMS Dashboard Task Monitoring application focusing on the analysis of the user activities and Chapter 6 discusses the CMS Dashboard Job Summary application that provides a more generic monitoring application to a wide variety of High Energy Physics users.

CHAPTER 4.

MULTI-THREADED AND DISTRIBUTED FRAMEWORK FOR PEDESTRIAN SIMULATION

Legion is the company behind the commercial pedestrian simulation software, Legion Studio and its accompanying 3D visualisation software, Legion 3D [143]. Both are used worldwide to optimise the design and operation of public spaces. Such spaces typically include transport terminals; sport, entertainment and leisure venues; shopping centres; commercial and public buildings; and venues for major international events such as the Olympics.

Their global portfolio includes [144] key organisations in the fields of transport, major events, sports, urban development and government. Legion software is used by many of the leading rail and transit agencies and has been deployed for each Olympic Games from Sydney 2000 right up to London 2012. Legion simulations are also used in many urban developments around the world. Designers, planners, engineers and asset managers have used Legion software and services to evaluate and optimise public spaces in improving safety, efficiency and profitability.

Their customers benefit greatly from the fully validated analyses and visualisations that the software produces [145]. These outputs are used to attain considerable economic benefits for facilities and programmes. Additionally, Legion software and services can improve the efficiency of projects; streamline the decision making process; ensure security; improve risk management and enhance profitability.

Legion's patented simulation technology is the result of many years' inter-disciplinary research into pedestrian behaviour. The accuracy of the simulations has been independently tested against real-world data resulting in endorsements by the Crossrail, London Fire Brigade, London Underground and Santiago Metro.

The company has a keen interest in advancing its science and technology to maintain its competitive edge. Industry trends suggest a continued move towards multiple CPU personal computers. The development of a multi-threaded version of the Legion simulation software is the only way to harness the power of commodity hardware. In addition, distributed computing is an indispensable tool for tackling simulations of ever increasing size and complexity. This research aims to produce state-of-the-art and commercially desirable output.

This chapter describes the development of both a multi-threaded and a distributed version of the software and presents benchmark results demonstrating how the use of a multicomputer or of even a multi-core computer can greatly accelerate the speed of a pedestrian movement software. The work was performed by the author and is published in [58].

4.1 Introduction

The Legion Studio software suite [143] is a widely adopted, powerful and accurate pedestrian simulation software. It comprises of three applications: the Model Builder, the Simulator and the Analyser. In combination, these applications enable the user to simulate pedestrian movement within a defined space, such as a railway station, sports stadium, sports park, airport, tall building, piazza, transport hub, town centre or any place that people assemble in or move through.

The software simulates the behaviour and movement of pedestrians footstep-by-footstep[1] calculating how individuals interact with each other and with the physical obstacles in their environment. The simulations employ a microscopic simulation model [145], which treats space as a continuum, using spacial objects, such as entrances, exits and escalators, to define space utilisation. The simulation navigates entities on the 'least-effort' principle. Each entity chooses its next step in an effort to find the best compromise between directness of path, speed and comfort.

The Model Builder can be used to create an accurate model of the space that we want to simulate. The following actions can be performed in the Model Builder:

1 In a quantitatively verifiable manner.

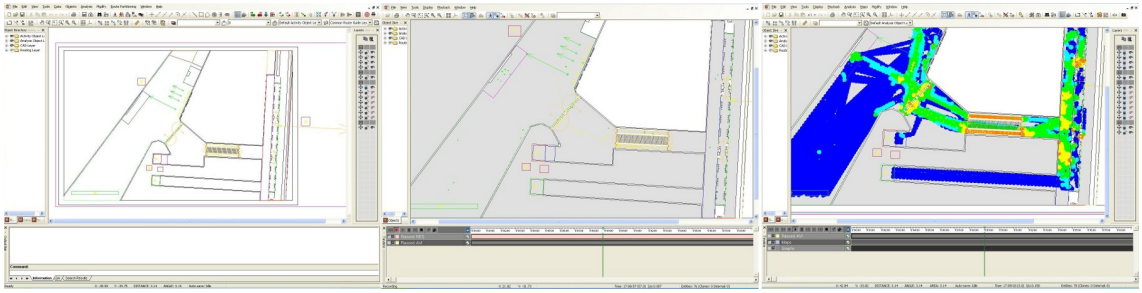
- Import architectural drawings (CAD) that define the physical space.
- Specify the pedestrian demand imposed on the space.
- Designate areas where activities such as queuing or waiting occur.
- Account for different routes.
- Link operational data to the model.
- Export model files for use in the Simulator.

The Simulator can be used to run a simulation of how pedestrians move or circulate within the space defined in the Model Builder. The following actions can be performed in the Simulator:

- Import model files.
- Playback and view the simulation.
- Record appropriate parts of the simulation as a 'results file' (.res) to be analysed.
- Record all or appropriate parts of the simulation as a video file for presentations.

The Analyser can be used to run a series of analyses on the simulated space. The following actions can be performed in the Analyser:

- Import results files and model files.
- Play back selected parts of a recorded simulation, or run a new simulation just like in the Legion Simulator.
- Visualise key metrics in the form of maps.
- Run detailed analyses and display the results as time series, stacked bars or histograms.
- Export the analysis session as graphs, results files, video, pictures or tables for inclusion in presentations, reports and spreadsheets.



a) Model Builder b) Simulator c) Analyser

Figure 4.1: a) Build a precise model of the space to be simulated and analysed based on a set of key inputs, b) run and record step-by-step simulations of pedestrian movement within the space defined in the Model Builder, c) set up and run a user-defined analysis based on the simulator.

Using Legion Studio, we can perform simulations on the design or operation of a space and assess the impact of different physical designs or levels of pedestrian demand. The impact of chance events can be studied, such as the impact of the closure of an exit or the late arrival of a train, and we can also test different evacuation scenarios for speed and safety. The latter can prove vital for compliance with increasingly rigid safety regulations. Legion simulation solutions are well suited for various stages of projects:

- **Capital Planning**

  During the strategic planning or capital planning process is where, economically, the software, data and analysis can have the biggest impact by evaluating early in the process where the clients need to spend money and where they don't, enabling the clients to maximise cost savings at the earliest stage.

- **Design Phase**

  During the design phase for a facility design or refurbishment, a client can minimise design iterations or alternatives by analysing and comparing potential designs before too much time has been spent on the design options. This can help shorten the overall design phase by efficiently removing options with data and analysis. Additionally, by evaluating a design, a client can optimise the design and avoid costly design changes downstream during the build out.

- **Construction Phase**

  Construction in transit, aviation, stadiums or rail stations as part of an upgrade to the infrastructure is a common occurrence. The agency wishes to maximise the

available space for construction and material staging while remaining open to the public with minimal service interruptions. Maximising the speed of construction while accommodating the pedestrian demand is a difficult balancing act. By modelling the proposed construction phasing plan the guess work is taken out of the process. Decisions regarding how much and where to close can be made with facts on what the outcome will be of the different construction staging and operations plans.

- **Daily Operations and Operations Planning**

  Streamline daily operations by identifying more efficient designs or layouts which can drive better pedestrian flow without the need for added personnel or temporary barriers. A client can compare and analyse various operational procedures and traffic demands to help a venue reach and maintain optimum operational efficiencies. In the sports arena and special events situations, simulations can help to identify improvements to pedestrian flow without disrupting existing operations. In the train sector, Legion Software can be used to manage various aspects of train operations which includes train car selection and fit out as well as assessment of timetable efficiency and performance optimisation. At any stage of operations a client can use Legion Simulation to assess and optimise the train schedules and train car capacities.

- **Safety and Security assessment**

  Every rail and metro station, football stadium and airport requires an annual safety certificate. Commercial buildings need to test evacuation scenarios. Every major event needs to establish evacuation and contingency plans. A client can design, simulate and stress test safety measures in an efficient and timely manner. A client can simulate alternative evacuation scenarios where the key variables are modified so that the client can see all results and eliminate the guess work. Safety and security plans can be designed based on clear assessment of risk, calculated predictions thus removing a lot of the guess work and lowering the overall risk associated with security or safety issues.

4.2 Legion Analyser

The Legion Analyser enables us to set up and run a series of rich, user-defined, analyses on our simulation using two methods:

- On-line analysis – analysing while simulating (using an .ora file).
- Off-line analysis – analysing a recorded simulation (using a .res file).

Both methods give access to a wide range of metrics, such as density, speed, flow, journey time and dissatisfaction, and a rich array of display methods and outputs including maps, graphs, tables and raw data. In the Legion Analyser a user can import data and model files, playback all or selected parts of the simulation, track individual entities and visualise their walking paths over time, visualise key metrics in the form of colour-coded maps, analyse any area of the model and display the results as time series, stacked bars or histograms and finally, produce results files, video, pictures or data for presentations, reports and spreadsheets.

The Legion Analyser creates an analysis (.ana) file as a template for storing the settings of all the maps, graphs and analyses generated from an .ora file or the simulation's .res file. In this way, many files using the same analysis template can be analysed, which is a good way to compare different scenarios.

The Legion Analyser enables us to take the whole model, or a defined portion of it, and ask certain questions. The four main objectives that Legion analyses relate to are:

- Feasibility studies.
- Design and construction as illustrated in Figure 4.2.
- Renovation.
- Operations.

The following is a sample of the types of questions we can ask and get an answer using Legion analyses:

- Will the venue cope with projected demand?
- What are the density levels at bottleneck points such as the bottom of stairs, main entrances or stadium vomitories?
- What is the average waiting time at facilities during peak periods?
- Can the venue be evacuated safely in the case of an incident?
- What is the interchange time distribution between lines A, B and C?

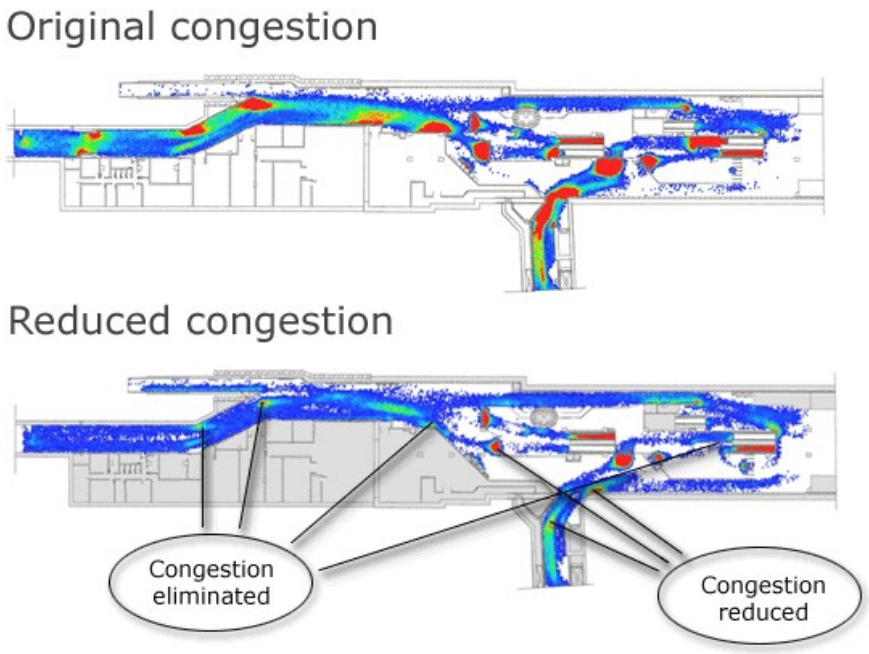


Figure 4.2: Platform Design.

4.2.1 Maps and Value Ranges

Legion Analyser maps provide colour-coded representations of the simulation we are analysing, enabling us to visualise key entity experience and crowd dynamic metrics such as density and space utilisation. They are really good for obtaining an overview of a scheme's performance and they can be applied to the whole of model or restrict them to specific areas defined by Analysis Zones.

The colours displayed in a Legion map are linked to two types of range:

- Value ranges – essentially these are Levels of Service, such as those defined by J. Fruin [146] or the US Highway Capacity Manual [147], used to rate experience-metrics.

- Colour ranges – an ordered list of colours used to describe local conditions that typically range from “excellent” (blue) to “bad” (red).

Colours within a map can represent the following:

- Occupancy – the number of Entities inside an area.
- Anything that can be used to measure Entity experience – examples include speed achieved, density experienced and total distance covered by Entities inside an area.
- Time – the duration inside an area for which a pre-set condition on occupancy or on any Entity experience metric has been met.

The Legion Analyser provides several default maps, as illustrated in Figure 4.3, but we can also create our own using default or custom value and colour ranges.

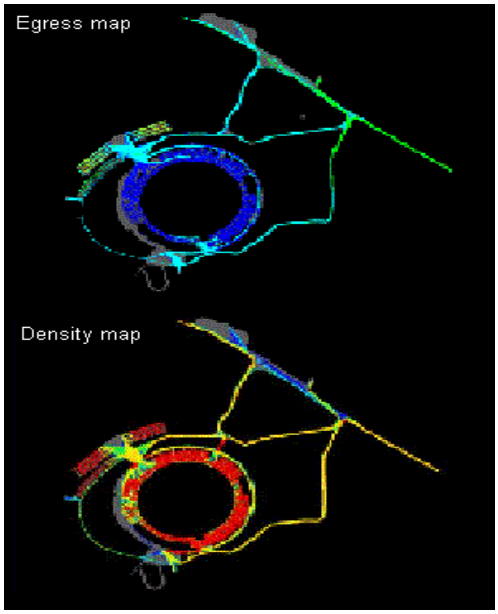


Figure 4.3: Egress and Density Maps

4.2.2 Standard Maps

The following standard maps are available within the Analyser:

- Cumulative High Density
- Cumulative Max Density
- Cumulative Mean Density

- Cumulative Min Density
- Evacuation
- Space Utilisation

Descriptions of each map and their typical uses follow.

Cumulative High Density Map

This map shows how long various areas of a site have registered densities greater than a specified limit. The range of colours represent time. The map is similar to a "temperature" map: areas that have experienced high levels of density for a long time appear red, those that have experienced shorter periods of density appear blue.

This map is best used for identifying "hot-spots" within a site such as areas where high levels of density are sustained. It asks the questions "is this design creating persistently uncomfortable crowd densities?" and "should it be altered to alleviate these problems?".

Cumulative Max/Mean/Min Density Map

These maps display the maximum, mean and minimum levels of density registered in an area from the beginning of playback to the current moment. They are generally used in combination with value ranges corresponding to widely used Levels of Service.

They are best used for measuring the performance of a site against predetermined standards or imperatives such as "the average density within a unit of space must not exceed Fruin's Level of Service x".

Evacuation Map

Evacuation Maps represent the amount of time that has elapsed from the beginning of playback to the most recent moment when an area was occupied. They are useful for safety assessments such as a train on fire or a station on fire, and egress assessments such as time to clear a stadium, as illustrated in Figure 4.4, or office building. They can also be used for platform capacity assessments, to show how quickly platforms clear following the arrival of a train.

Space Utilisation Map

The Space Utilisation Map reveals how much space within a site is being used. It records the location of every step of each Entity over the duration of the simulation. Heavily used areas are coloured red and lightly used areas are coloured blue. Areas of the simulation that are not used at all remain white.

The colour range represents the amount of time a unit of space has been occupied within the simulation. The default setting of this unit of space is 10x10cm. This map is best used for illustrating which areas of a site are used the most and the least. It can support questions such as “if this area is not being used regularly, could it be used for a small kiosk or retail unit?”.

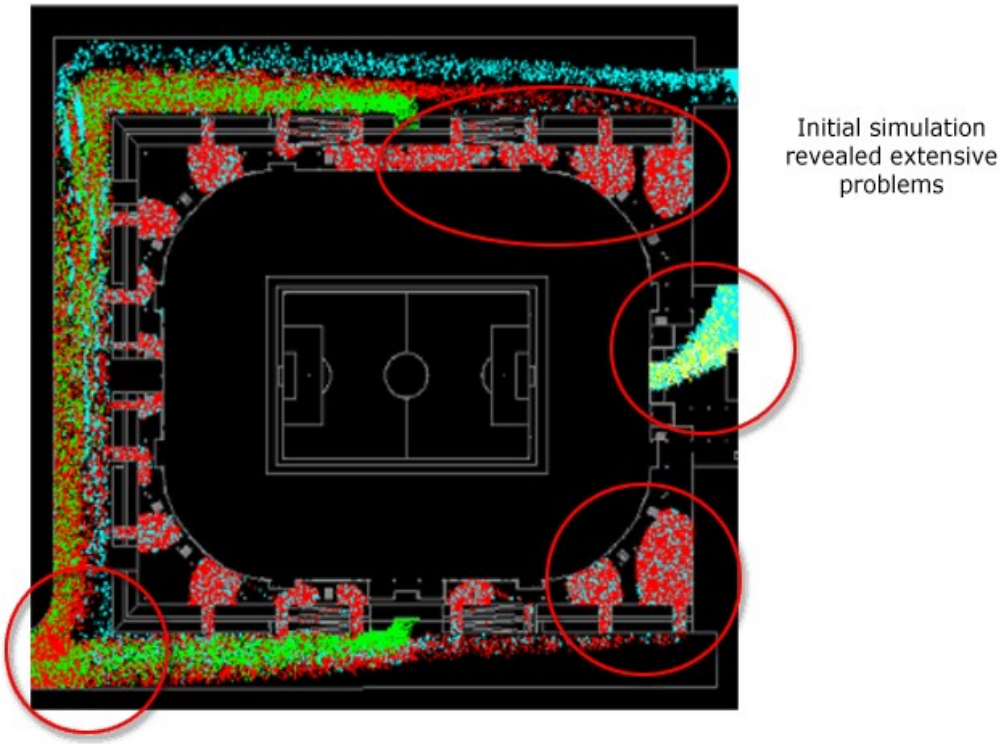


Figure 4.4: Dusseldorf Arena Evacuation Map.

4.3 Multi-Threaded Legion Analyser

The following sections describe in depth the design, the implementation and the benchmark results of the Multi-Threaded version of the Legion Analyser commercial software.

4.3.1 Design

The following sections discuss the requirement that shaped the design of the multi-threaded version of the Legion Analyser.

Objectives

The main objective for re-developing the Legion Analyser is to provide a faster, maintainable release. Industry trends suggest a continued move towards multiple CPU personal computers. The development of a multi-threaded version of the Legion simulation analysis software is the only way to harness the power of commodity hardware.

The main beneficiaries of this activity were the users who have come to rely on the functionality that the Legion Analyser provides. The increased performance was a benefit to them and to new users. In addition, one of the major considerations when redesigning the Legion Analyser was to make maintenance and support easier for the developers of Legion.

Architecture

The most important components of the Legion Analyser are shown in Figure 4.5. The class CReSpaceMapManager is responsible for the list of the enabled maps, for their metrics and for their implementation. The CCellStorageManager class is responsible for the accumulation and for the identification of the data of the cells. The environment is represented by a grid of cells and movement is modelled as cell switching. The storage is a grid full of CCellStorageData class pointers. The CCellStorageData contains a vector of CCellStorageDataItem, one item for each map. The major components of the Cell Accumulation and Identification classes are being illustrated in Figure 4.6. These classes are responsible for resolving the list of affected cells, stepped by the entities, computing those affected cells and then accumulating them. The Statistics and the Entity Map Manager classes are being illustrated in Figure 4.7. The Statistics Manager is responsible for the statistics of the Legion Analyser, keeping a track of the running time of the enabled analysis and of the statistical metrics. The Entity Map Manager is responsible for handling and modifying the entity maps.

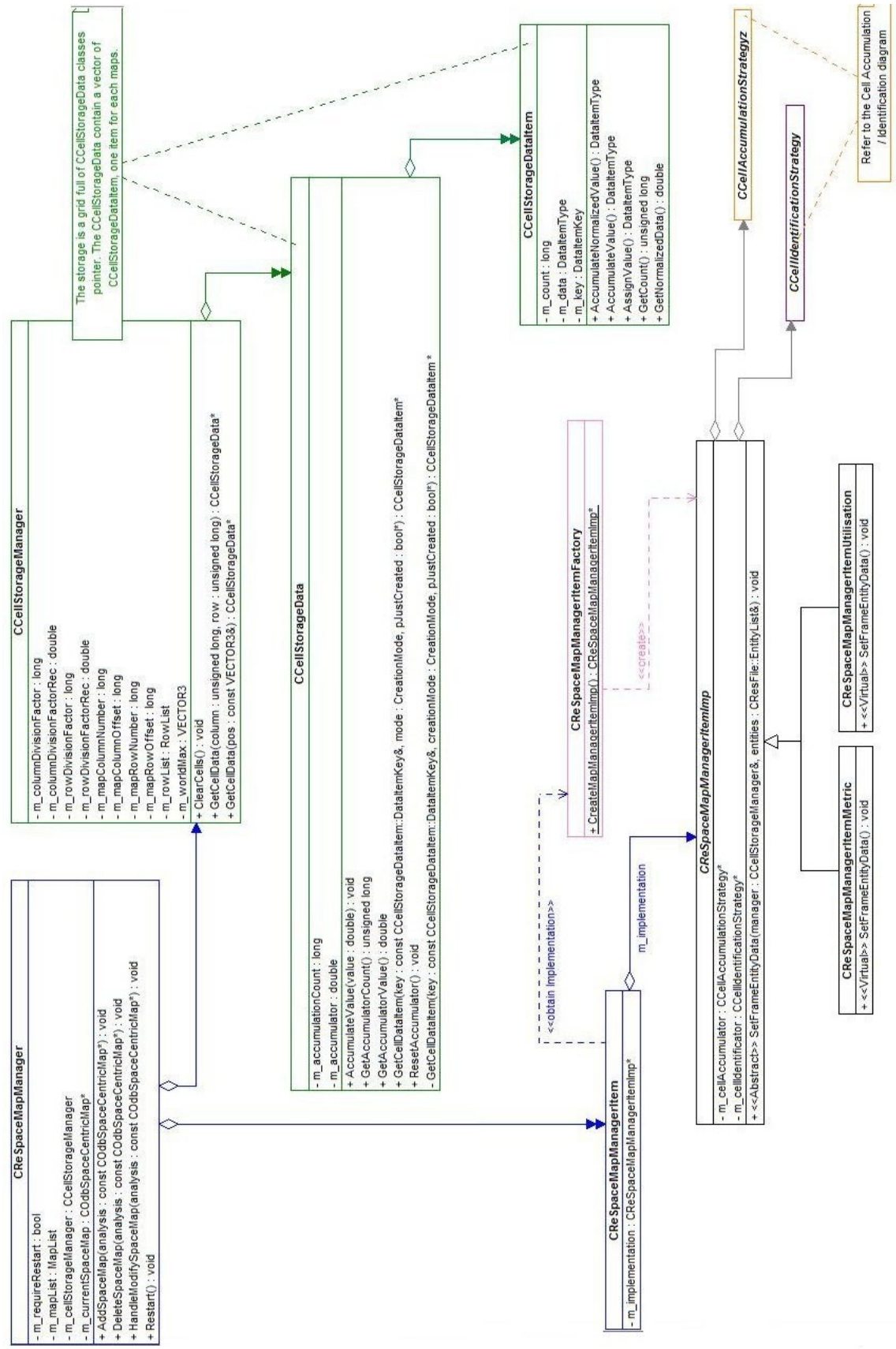


Figure 4.5: The major components of the Legion Analyser and their internal interactions.

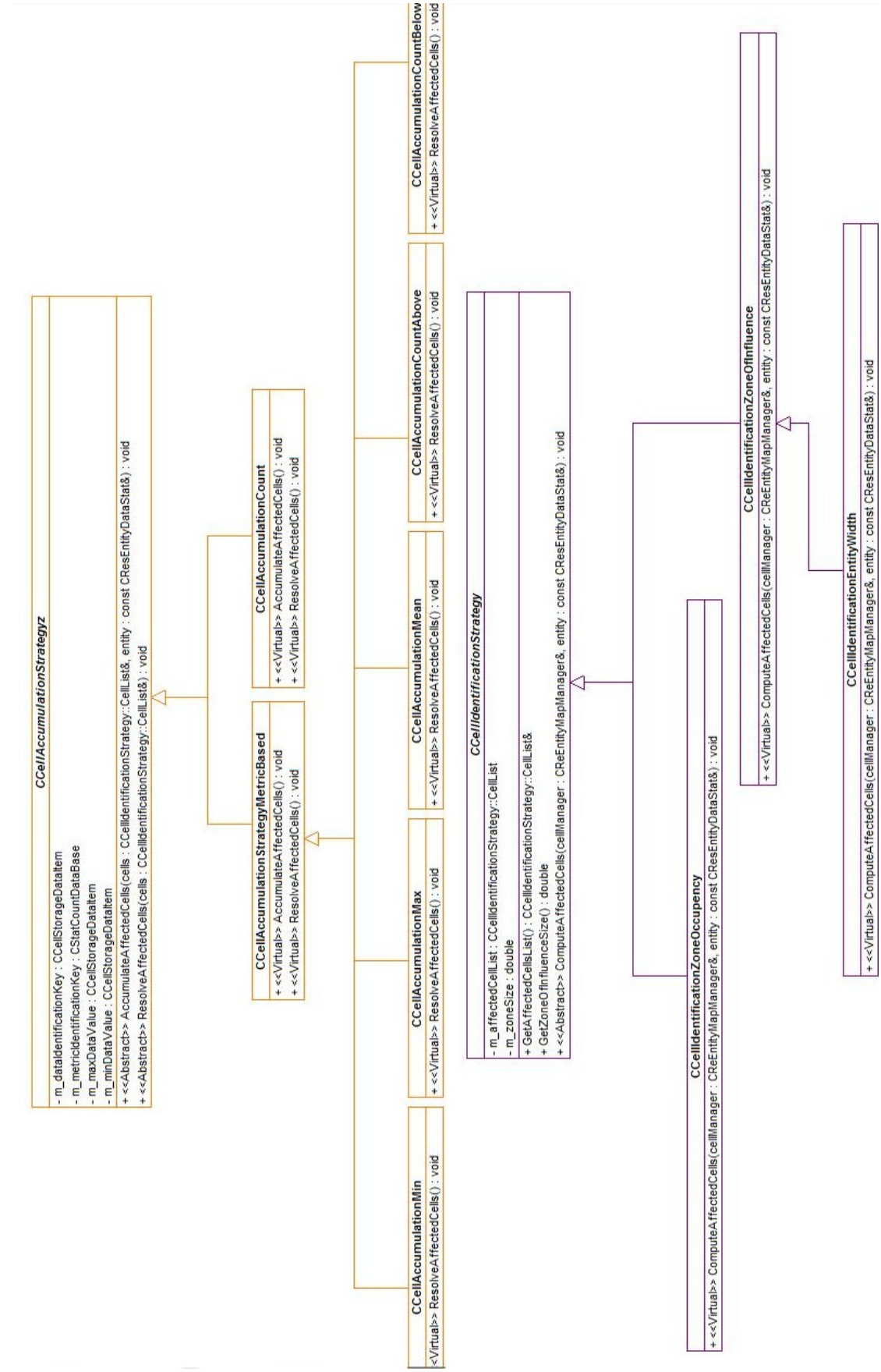


Figure 4.6: The components of the Cell Accumulation & Identification classes and their internal interactions.

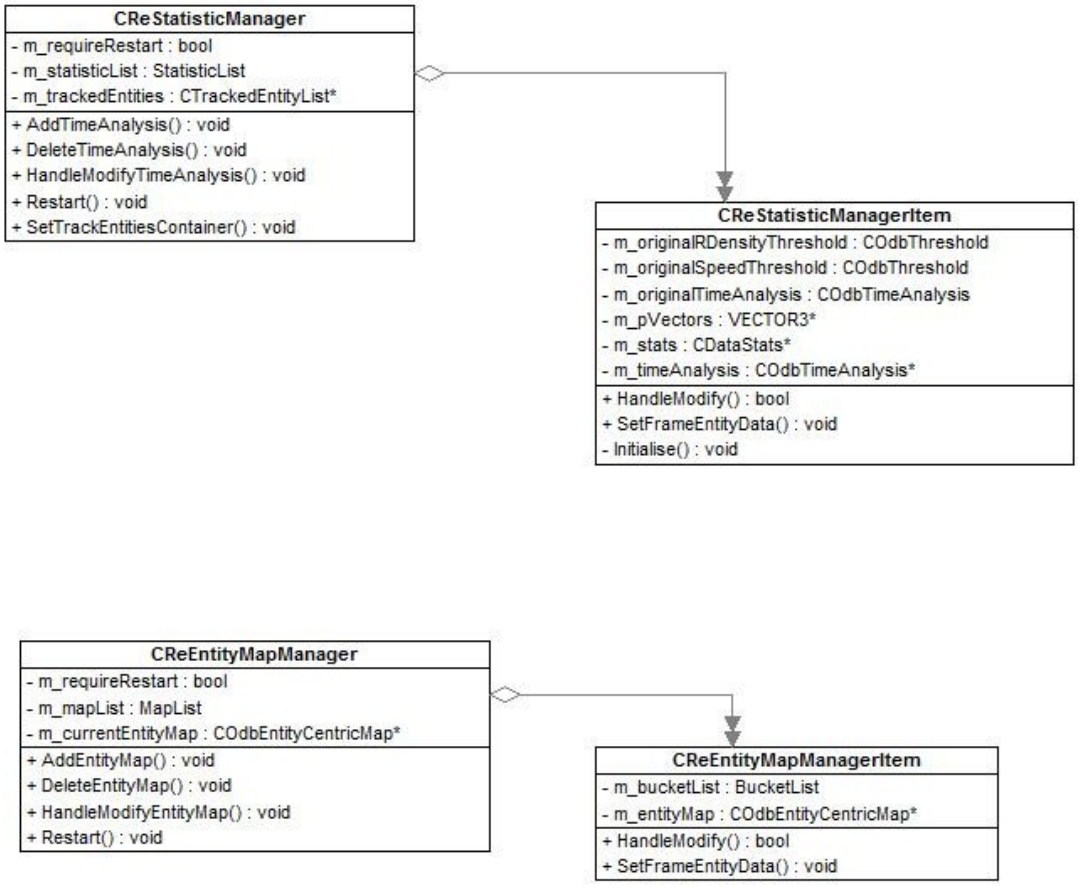


Figure 4.7: The Statistics and the Entity Map Managers.

4.3.2 Implementation

An analysis session comprises of the following tasks:

- Advances the simulation time clock.
- Loads entity list from a ROOT [148] file.
- Calculates the maps by traversing a grid-like structure gathering information from nearby entities[2].
- Renders the maps and the entity movement.
- Computes analyses by traversing a list of analyses.
- Updates the graphs and saves any files that need saving.

The maps are the collection of objects that take care of accumulating various metrics from the entities as they move across the usable space. They are responsible for:

2 The maps are generated from the entities by adding their contribution to the map.

- Internal abstract representation needed for generic rendering.
- Internal memory structure.
- Algorithms needed to identify the space that is stepped on.
- Algorithms needed to accumulate entity's metrics as they move.

The Multi-threaded Analyser creates a thread pool with a size equal to the total number of the CPU cores or processors. The use of a thread pool is proved to be faster than native threads since there is no thread creation and destruction overhead [86]. There is no essential dependency or communication between the parallel tasks since a communication overhead reduces the speed up achievable by the programme. There are no invalid pointers during the execution of the programme since iterators are invalidated during the data insertions and the data removals. The use of Critical Sections to lock the critical region of the OpenGL drawing procedure of the maps was faster than the use of a simple mutex or of a recursive mutex. Listing 4.1 contains the pseudo-code of the process.

1. *Create a thread pool according to the number of the cores*
2. *For each simulation time step*
    - *a. Get the entity list*
    - *b. Traverse the entity list from the beginning to the end or vice versa*
    - *c. Lock the openGL drawing procedure*
    - *d. Wait for the other thread(s) to finish calculating the time step*
    - *e. Remove the lock and draw the maps on the screen*
    - *f. Advance to the next simulation time step*

Listing 4.1: The pseudo-code of the multi-threaded Analyser.

The sequence of the actions performed in an off-line Legion analysis can be seen in Figure 4.8 and the sequence of the actions performed in an on-line Legion analysis can be seen in Figure 4.9.

The only difference between the on-line and the off-line analysis is that during the on-line analysis, the Analyser communicates with the Simulator using the Simulation Wrapper class.

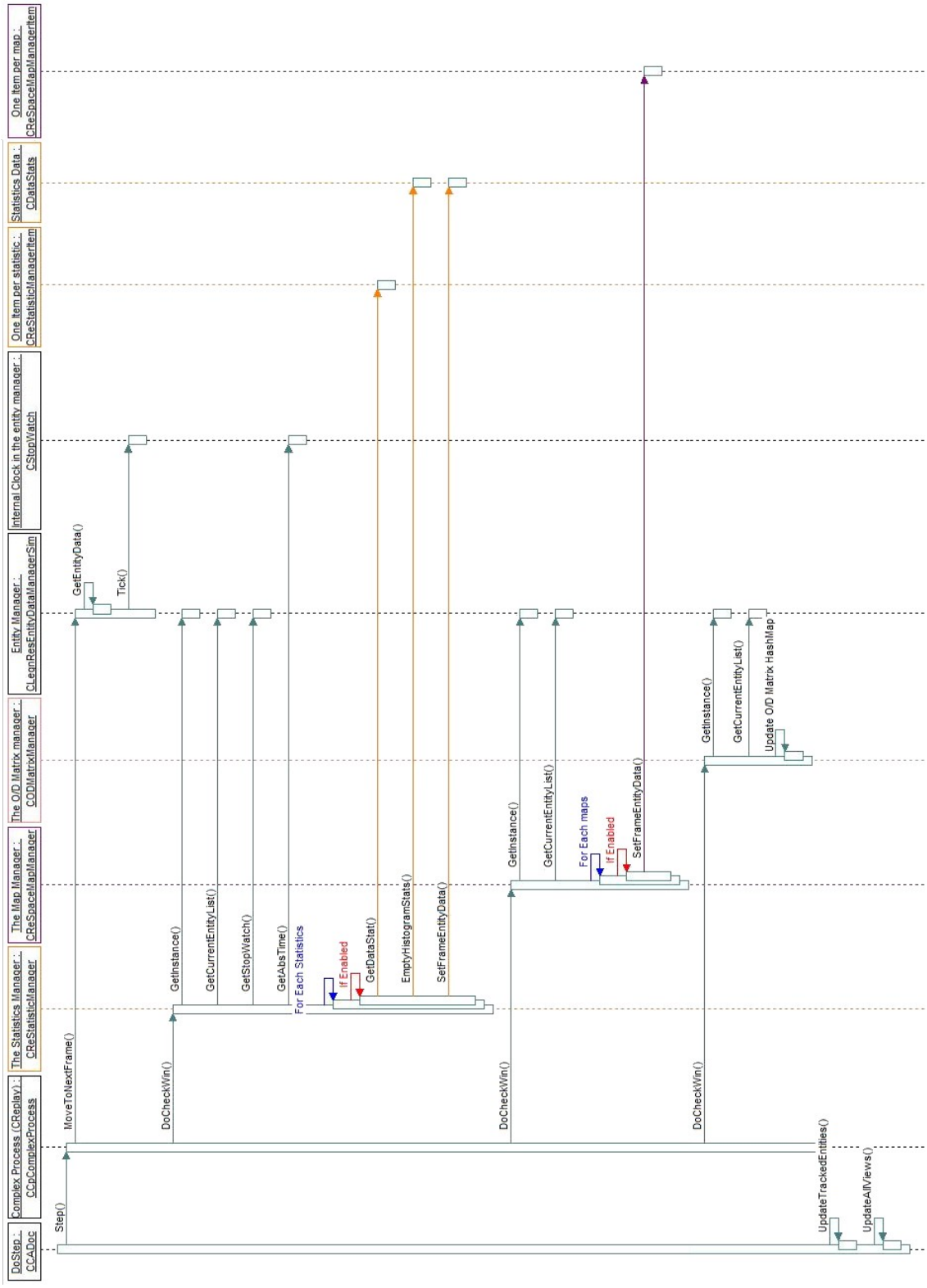


Figure 4.8: The sequence of actions that are performed in an off-line Legion analysis.

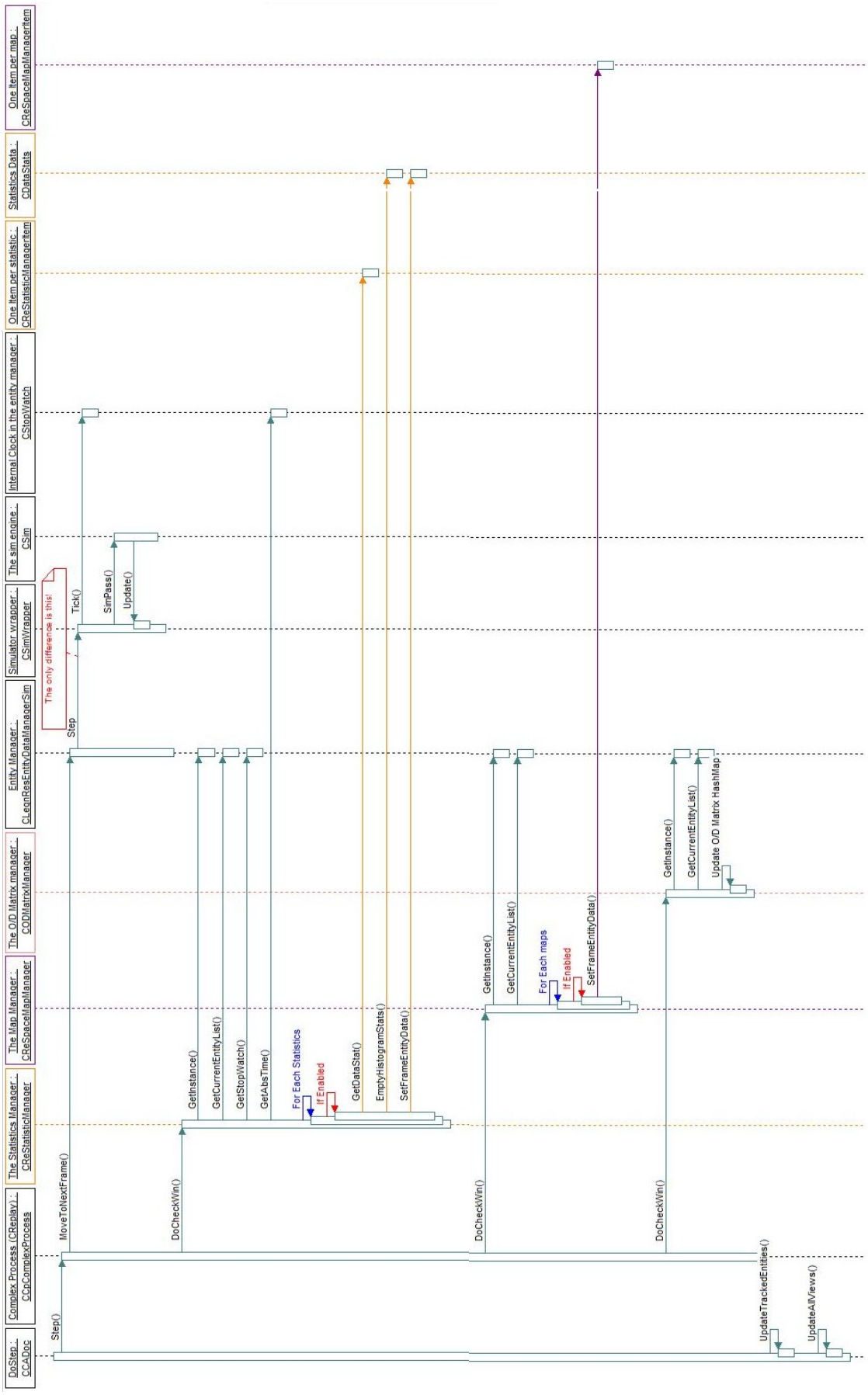


Figure 4.9: The sequence of actions that are performed in an on-line Legion analysis.

The detection of the number of the processors or of the cores in a machine is being illustrated in Listing 4.2.

```
// CLASS CReSpaceMapManager
CReSpaceMapManager::CReSpaceMapManager()
: m_threadPool( )
{
	// Detect the number of processor in the machine, and set it as the default value for the processor property
	SYSTEM_INFO systemInfo;
	::GetSystemInfo( &systemInfo );		// NOTE: the defaut pool is fifo
	m_threadPool.size_controller().resize( systemInfo.dwNumberOfProcessors );
}
```

Listing 4.2: The detection of the total number of processors or of the cores in a machine.

The execution of a thread for each enabled map is being illustrated in Listing 4.3.

```
void CReSpaceMapManager::DoCheckWin( void )
{
	// Get the entities from the entity manager
	Legion::Simulator::IEntityPtrVector& entities =
		CLegnResEntityDataManagerBase::GetInstance()->GetCurrentEntityList();
	MapList::iterator	iter( m_mapList.begin() );
	MapList::iterator	end(  m_mapList.end()   );
	while( iter != end )
	{
	const COdbSpaceCentricMap* pSpaceMap = dynamic_cast<const COdbSpaceCentricMap*>( (*iter)->GetMap() );
				// Only do calculations for enabled maps
				if( pSpaceMap->IsEnabled() )
				{
				CReSpaceMapManagerItem* pSpaceMapItem =
					dynamic_cast<CReSpaceMapManagerItem*>(*iter);
				ASSERT( pSpaceMapItem );
				// Check for the reset interval
				int nResetInterval = pSpaceMap->GetResetInterval();
				if (nResetInterval != COdbSpaceCentricMap::MapResetDisabled)
				{
					double timeStamp = ClegnResEntityDataManagerBase::GetInstance()-
						>GetStopWatch().GetTime().GetTimeSecond();
					double rIntervals = double(int(timeStamp / double(nResetInterval)));
					// stopwatch keeps time-step interval in milliseconds
					double timeTolerance = ClegnResEntityDataManagerBase::GetInstance()-
						>GetStopWatch().GetTimeStepInterval() / 1000.0;
					if( timeStamp - rIntervals*nResetInterval < timeTolerance )
					{
						ResetMap( pSpaceMap );
					}
				}
				// Execute a thread
				m_threadPool.schedule( SpaceMapTask( pSpaceMapItem, entities ) );
				} ++iter; // increase the iterator of the map list
	}
	// Join the thread pool to wait for all the maps to finish the computation
	if( !m_threadPool.empty() )
		{
			m_threadPool.wait();
		}
} // End of DoCheckWin function
```

Listing 4.3: The execution of a thread for every enabled map.

4.3.3 Performance

The memory footprint of the programme has been reduced to the minimum with the use of associative vectors instead of using maps of vectors. The associative vector is a std::map look-alike that uses a sorted vector for storage and such a choice has the advantage of fast binary searches but slow insertions and removals. Iterators are invalidated during insertions and removals, which doesn't happen with std::map's node based storage. The Associative Vector is faster than std::set/map in lookups and more memory friendly, especially for small types, since normally a tree like structure imposes an overhead of three pointers and an integer per node; without counting that memory allocation for a vector has far less fragmentation when using std::allocator.

The memory management has been optimised by changing the structure of the programme. As a result, a lot of unnecessary search procedures at every simulation time step have been removed. The programme uses the same amount of memory as the original single-threaded version in most of the models and in case that the programme uses more memory, the increase is only between 3% to 6%. To benchmark our multi-threaded implementation, six models with different levels of complexity and size have been used on a 2 GHz of CPU dual-core system with 2 GB of memory. The increase in the performance depends on the size and complexity of the model. All the models used for the benchmarking are available in Appendix C.1.

In Table 4.1, we present a 55.43% increase in performance and a 3.16% increase in memory usage is being illustrated using a small-sized model with 350 entities.

Metrics	Original	Multi-threaded
Total Time HH:MM:SS	00:39:45	00:17:43
Memory Usage in MB	190	196
Peak CPU Usage	50.00%	75.00%

Table 4.1: Small-sized model. Name: PM Peak. 350 Entities. Simulation time: 3 Hours.

In Table 4.2, the increase was 34.47% and with a 3.51% increase in memory usage using our second small-sized model with 552 entities.

Metrics	Original	Multi-threaded
Total Time HH:MM:SS	00:07:50	00:05:08
Memory Usage in MB	114	118
Peak CPU Usage	50.00%	75.00%

Table 4.2: Small-sized model. Name: UP Demo v3:1. 552 Entities. Simulation time: 1 Hour.

In Table 4.3, an increase of 57.77% in the performance and a small decrease of 0.82% in memory usage is being presented using a medium-sized model with 1200 entities.

Metrics	Original	Multi-threaded
Total Time HH:MM:SS	00:22:32	00:09:31
Memory Usage in MB	245	243
Peak CPU Usage	50.00%	88.00%

Table 4.3: Medium-sized model. Name: Gatwick Airport Station Re-development. 1200 entities. Sim time: 1 Hour.

Likewise, in Table 4.4, an increase of 65.50% in performance and a 5.93% increase in memory usage is being illustrated using a medium-sized model with 2500 entities.

Metrics	Original	Multi-threaded
Total Time HH:MM:SS	01:41:22	00:34:58
Memory Usage in MB	489	518
Peak CPU Usage	50.00%	85.00%

Table 4.4: Medium-sized model. Name: New WTC Model. 2500 entities. Simulation time: 1 Hour and 30 Mins

In Table 4.5, an increase of 34.15% in performance can be seen in Table 3 together with a 6.38% decrease in memory usage using a large-sized model with 51000 entities. Likewise, in Table 4.6, an increase of 32.19% in performance and a decrease of 1.34% in memory usage is being illustrated using a large-sized model with 52000 entities.

Metrics	Original	Multi-threaded
Total Time HH:MM:SS	02:16:25	01:29:50
Memory Usage in MB	940	880
Peak CPU Usage	50.00%	99.00%

Table 4.5: Large-sized model. Name: London Olympic Park 2012. 51000 entities. Simulation time: 14 Mins.

Metrics	Original	Multi-threaded
Total Time HH:MM:SS	01:25:04	00:57:41
Memory Usage in MB	373	368
Peak CPU Usage	50.00%	98.00%

Table 4.6: Large-sized model. Name: HOS Case3. 52000 entities. Simulation time: 19 Mins.

The performance gained and the memory usage can be seen in Figure 4.10. The performance increase ranges between 35% to 65.5% compared to the original single-threaded Legion Analyser on a dual core system[3]. The programme uses approximately the same amount of memory as the original single-threaded version; the memory increase is only between -6.38% to 6%.

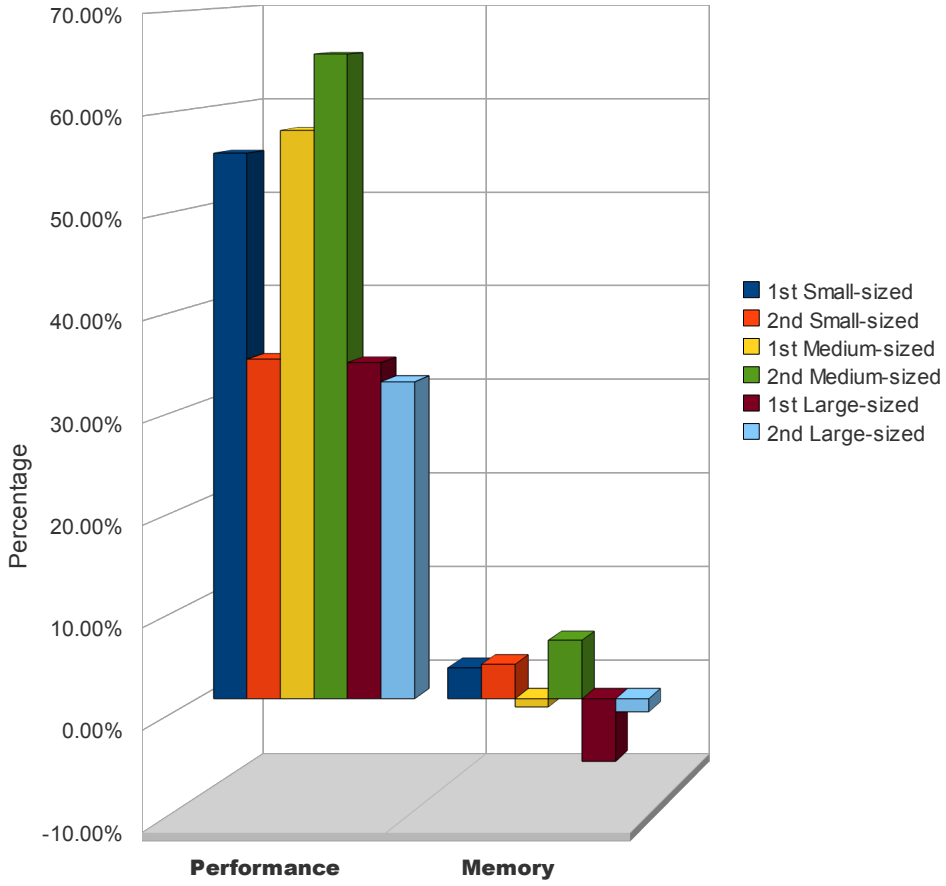


Figure 4.10: Performance and Memory Benchmark.

3 Using a dual core PC. 2GHz of CPU and 2GB of RAM.

4.4 Distributed Legion Analyser

The following sections describe the design, the implementation and the benchmark results of the prototype distributed version of the Legion Analyser commercial software.

4.4.1 Design and Implementation

The following sections discuss the requirements that shaped the design of the prototype distributed version of the Legion Analyser.

Objectives

The main objective for developing a distributed version of the commercial programme is to provide a system capable of tackling simulations of ever increasing size and complexity. This work aims to demonstrate how the use of a multicomputer can greatly accelerate the speed of pedestrian movement software.

The main beneficiaries of this research work were the developers of Legion. The demonstration of the increased performance was a benefit to them and to their customers.

Architecture

In the early stages of the development of the distributed Analyser, the OpenMP standard was considered but such an option was abandoned because OpenMP is limited to be used in a shared-memory environment, i.e. a shared memory cluster [149]. Since we wanted to use the Distributed version of the programme in a network using workstations in a distributed-memory environment, the Message Passing Interface (MPI) library was used to send messages between the nodes and across the network. MPI is the most popular message-passing library standard for parallel programming [82]. The MPICH2 implementation of the version 2.1 (MPI-2) of the standard was chosen together with the Boost.MPI library, part of the Boost C++ library. The Boost.MPI library provides a C++ friendly interface to the MPI standard that better supports modern C++ development styles [150].

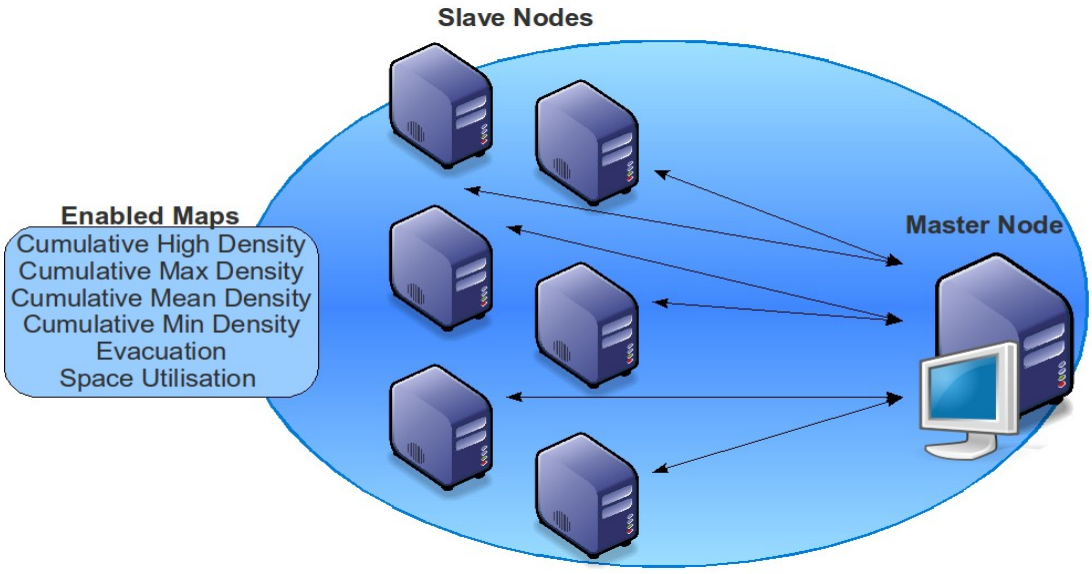


Figure 4.11: The distributed implementation uses a Master/Slave organisation. Each Slave node is responsible for calculating an assigned map. The Master node collects the results and displays the results on the screen.

The prototype version of the Distributed Legion Analyser consists of the Master node and the Slave nodes as illustrated in Figure 4.11. The Master node is responsible for collecting the results from the Slave nodes, drawing the results on the screen and updating the statistics and the graphs. The Slave nodes are responsible for all the calculations of the maps. The work is divided and evenly distributed between the Slave nodes and a load balancing algorithm makes sure that no Slave node will be idle for a long period of time.

All the nodes open a read-only model on the network and begin the Distributed Analysis. The division of the work is done according to the total nodes registered and the total maps enabled for the analysis session. Each node is registered and a list of all the available nodes exists on the MPI_COMM_WORLD. The map list and the entity list is then fetched together with the list of the computers registered in the MPI_COMM_WORLD. Hence, every node is aware of all the registered nodes taking part in the analysis.

Each registered Slave node starts the calculation of the assigned maps and at every simulation time step, it calculates the assigned maps, serialises the results, packs them using MPI_Pack(), sends them to the Master node using a non-blocking MPI_ISend()

and waits for all the other Slave nodes to finish the calculation before advancing to the next time step. The Master node collects the results using MPI_IRecv(), unpacks the packed data using MPI_Unpack(), draws and displays the maps on the screen and updates the graphs and the statistics. The C++ code listings available in Appendix C illustrate the use of the MPI for the communication and the division of the work between the nodes. Listing 4.4 illustrates the initialisation of the MPI communication library.

```
/// Initialise MPI
MPI_Init( NULL, NULL );
// Boost.MPI code
mpi::environment env (NULL,NULL);
mpi::communicator world;
int mynode, totalnodes;
// Assign a rank to each available node
MPI_Comm_rank( MPI_COMM_WORLD, &mynode );
// Get the total size of the available nodes
MPI_Comm_size( MPI_COMM_WORLD, &totalnodes );
```

Listing 4.4: The Initialisation of the MPI.

The work allocation and division can be seen in Appendix C.3. Most of the communication between the Slave and the Master nodes can be seen in Appendix C.4 and Appendix C.5.

4.4.2 Performance

To benchmark our distributed implementation, we have used an evacuation case study. The area is modelled after the London 2012 Olympic Park and we have populated the model with 56500 entities. The model is available in Appendix C.2.

We have benchmarked our prototype distributed implementation on commodity hardware connected by a gigabit Ethernet switch. Figure 4.12 illustrates the performance of the distributed programme in terms of the time it takes in seconds to analyse a simulation second as a function of the number of the Slave processors.

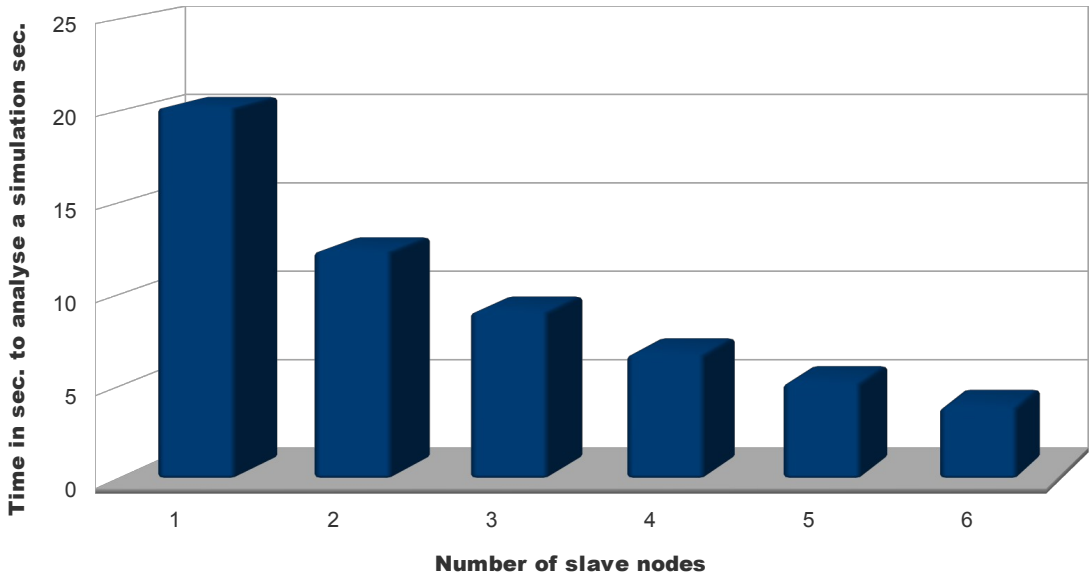


Figure 4.12: Time in seconds to analyse a simulation second. Each Slave node is a processor. An additional processor is allocated to the Master node.

The performance scales well as the number of the processors is increased. With one Slave processor, the prototype system is able to analyse 56500 simulated pedestrians in 20.17 seconds. In 12.33 seconds with two Slave processors, in 9.02 seconds with three Slave processors, in 6.68 seconds with four Slave processors and in 5.13 seconds with five Slave processors. Finally, with six Slave processors the prototype system is able to analyse 56500 simulated pedestrians in just 3.8 seconds.

4.5 Summary

We have faced many challenges and obstacles during this research project, mainly due to the difficulty of understanding the existing code of the Legion Studio software suite, a 6 GB code with more than 26000 C++ files but mostly due to the company's Intellectual Property (IP) rights.

This chapter presented the requirements and implementation of the Legion Analyser commercial programme. A framework capable of analysing the simulation data produced by the commercial Legion Studio pedestrian simulation software has been developed. The programme has been implemented as a multi-threaded and as a

distributed programme written in C++ with calls to the MPI library.

Benchmarking the programme on a dual-core PC and on a commodity cluster of high performance PCs demonstrated the system's increase in performance compared to the original single-threaded analyser. The performance increase for the multi-threaded version ranges between 35% to 65.5% compared to the original single-threaded Legion Analyser on a dual core 2GHz system. The performance of the distributed prototype version of the programme scales well as the number of the processors is increased; with six Slave processors the prototype system is able to analyse 56500 simulated pedestrians in just 3.8 seconds.

CHAPTER 5.

CMS DASHBOARD TASK MONITORING

We are now in a phase change of the CMS experiment where people are turning more intensely to physics analysis and away from construction. This brings a lot of challenging issues with respect to monitoring of the user analysis. The physicists must be able to monitor the execution status, application and grid-level messages of their tasks that may run at any site within the CMS Virtual Organisation.

The CMS Dashboard Task Monitoring project provides this information towards individual analysis users by collecting and exposing a user-centric set of information regarding submitted tasks including reason of failure, distribution by site and over time, consumed time and efficiency. The work was performed by the author and is published in [59], [60], [61], [62] and [63].

5.1 Introduction

The Experiment Dashboard [60] is a monitoring system developed for the LHC experiments in order to provide the view of the Grid infrastructure from the perspective of the Virtual Organisation. The CMS Dashboard provides a reliable monitoring system that enables the transparent view of the experiment activities across different middleware implementations and combines the Grid monitoring data with information that is specific to the experiment.

The scientists must be able to monitor the execution status, application and grid-level messages of their tasks that may run at any site on the distributed WLCG infrastructure. The existing CMS monitoring systems provide this type of information but they are not focused on the user's perspective.

The CMS Dashboard Task Monitoring project addresses this gap by collecting and exposing a user-centric set of information to the user regarding submitted tasks. It provides a clear and precise view of the status of the task including job distribution by sites and over time, reason of failure and advanced graphical plots giving a more usable and attractive interface to the analysis and production user. The development was user-driven with physicists invited to test the prototype in order to assemble further requirements and identify weaknesses with the application.

This chapter discusses the development of the CMS Dashboard Task Monitoring that was performed by the author. In the first section, the concept of the Experiment Dashboard monitoring system and its framework will be described in detail. The next sections provide an overview of the CMS Dashboard Task Monitoring application and its features. The final section focuses on the known issues.

5.2 Design

The following sections discuss the requirements that shaped the design of the CMS Dashboard Task Monitoring application.

5.2.1 Objectives

Most of the CMS analysis users interact with the Grid via the CMS Remote Analysis Builder (CRAB). User analysis jobs can be submitted either directly to the WLCG infrastructure or via the CRAB analysis server, which operates on behalf of the user. In the first case, the support team does not have access to the log files of the user's job or to the CRAB working directory, which keeps track of the task generation.

To understand the reason of the problem of a particular user’s task, the support team needs a monitoring system capable of providing complete information about the task processing. To serve the needs of the analysis community and of the analysis support team, the CMS Dashboard Task Monitoring [61] application has been developed on top of the CMS job monitoring repository.

5.2.2 Use Cases

A use case analysis was carried out based upon the feedback received by the CMS physicist community. The main use cases are described in Appendix A.1 and illustrated in Figure 5.1.

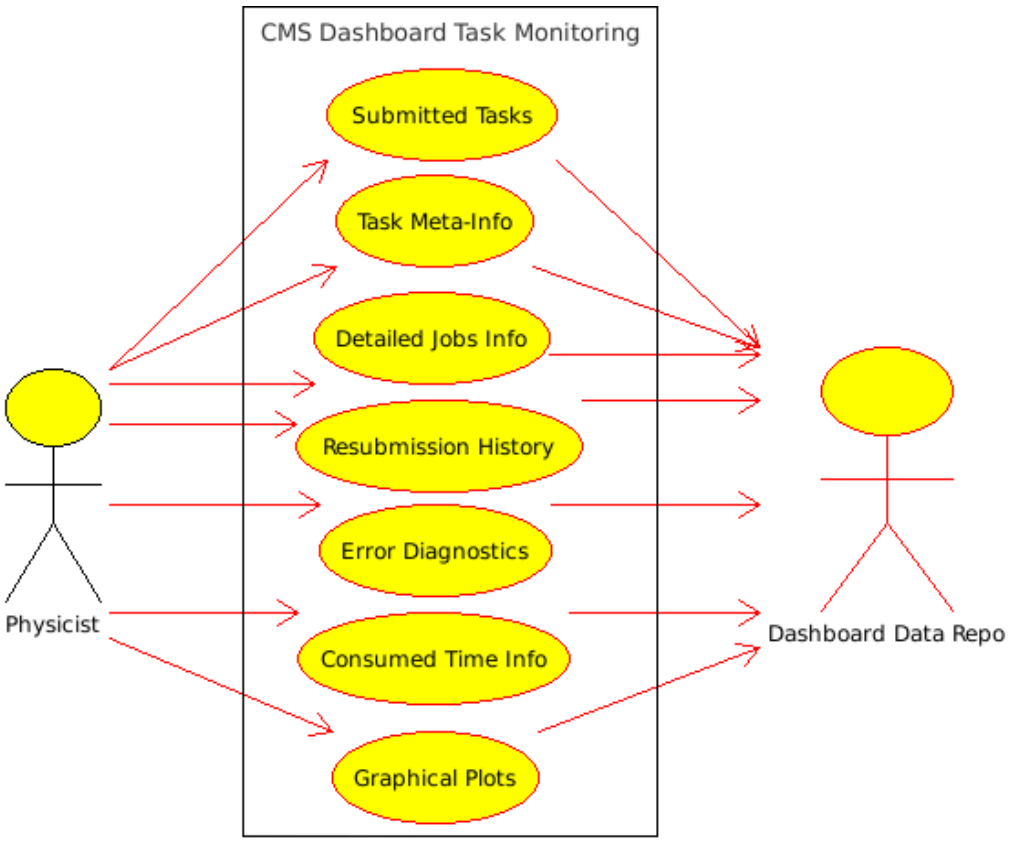


Figure 5.1: The main use cases that the application is expected to implement in conjunction with the CMS Dashboard system and with the CMS Physicist actors.

With the major use cases established it is possible to extract the key requirements that the application has to fulfil. The following points represent the baseline requirements divided into principal areas.

5.2.3 Requirements

Assumptions

1. Users have a grid certificate.
2. Users are members of the CMS VO.
3. Users have submitted jobs to the Grid within one month.

User Interface

1. Users control the application via a web interface using a browser.
2. The application will be focused on the CMS analysis user's perspective.
3. Easy to understand how it works and how to navigate throughout the tool.
4. Compatible with all the recent browsers and operating systems.
5. Simple, clean and intuitive in layout containing no unnecessary information.
6. All of the Grids and the job submission systems that CMS uses will be supported.
7. The user will access a very detailed information of the job processing including every single resubmission that he/she might have performed for each job individually.
8. The application will offer task meta-information.
9. The application will offer consumed time information and processing efficiency.
10. Individual jobs within a task can be selected.
11. Fast with very low latency.
12. Update in 'real-time' from the worker nodes where the jobs are running.
13. The user will be able to bookmark his/her favourite tasks for later use or to share them among his/her colleagues.
14. Offer a wide selection of advanced graphical plots that will visually assist the user.
15. The application will be built on top of the CMS Dashboard Job Monitoring Data Repository.
16. Exceptions should be caught by the application and informative error messages will be provided to the users.
17. Verbose logging should be available to identify any problems.
18. Quick access to the application's manual, help and the meanings of the error exit codes should be provided.

Developer's Requirements

1. Variable level of logging will be built in from the start.
2. Logging will write to stdout and to a file to ease debugging.
3. Low coupling between the components is required.
4. Minimum version of Python that is supported is determined by that installed on

lxplus.cern.ch (currently 2.3).

5.2.4 Architecture

The CMS Dashboard Task Monitoring application is part of the Experiment Dashboard system [60] which is widely used by the four LHC experiments. The framework of the system consists of the following components, as illustrated in Figure 5.2:

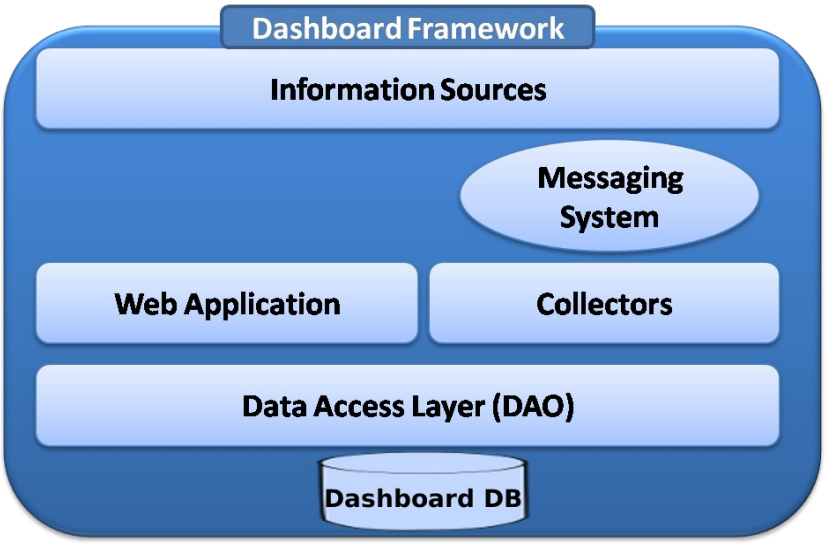


Figure 5.2: Dashboard Framework.

The Data Access Layer (DAO) is responsible for the management of the persistent data stored in a Relational Database Management System (RDBMS). Each component in this layer will provide query/update capabilities for a subset of the stored data. The Web Application is responsible for the HTTP entry point to the available data. It exposes the data to the users in different formats and inserts new records/updates existing ones. It makes heavy use of the DAO. The Collectors layer listens to messages/events coming from the Messaging Infrastructure and it quickly analyses the data and passes it on to the DAO layer for storage. The Information Sources layer sits closely to the services/applications being monitored and listens to interesting events. Finally, the Messaging System is an external component used by the Dashboard to communicate with the Information Sources.

The Controller is the main piece of the web application and is illustrated in Figure

5.3. It receives all client requests and decides what to do with them. For each client request there should be a corresponding Action, which will normally involve some interaction with the model of the application (some business logic that might involve accessing or updating persistent data).

A client request might involve producing some output. This output is identified by its mime/type and will have a View associated with it. The Action will put any data that it collected/produced in a shared area, the ActionContext, so that it can later be taken by the View to produce the output to the client.

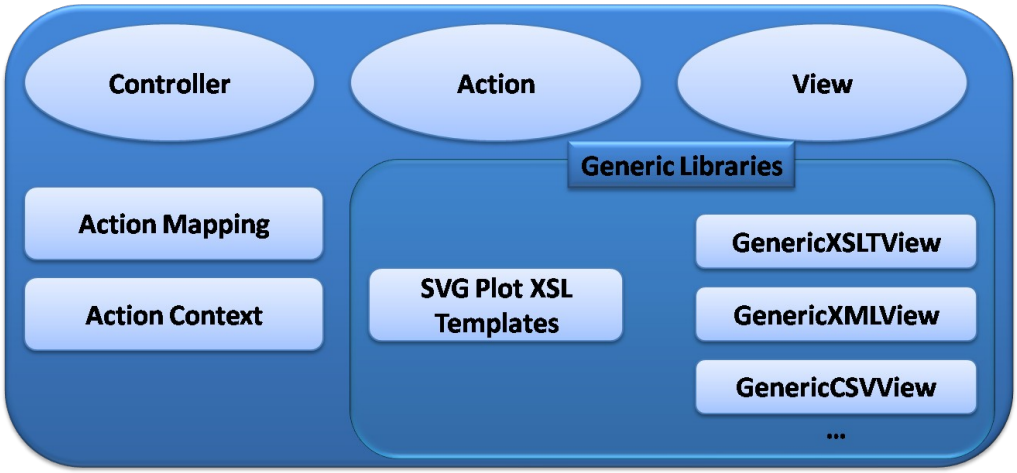


Figure 5.3: Web Application Architecture.

All the relationships between client requests, Actions, Views and their associated mime/types is defined in a single configuration file, the ActionMapping file. A widely used format for data retrieval is HTML but information can also be retrieved in XML, CSV or image formats allowing any third party application to use the system. The sequence of actions of the Web Application are illustrated in Figure 5.4.

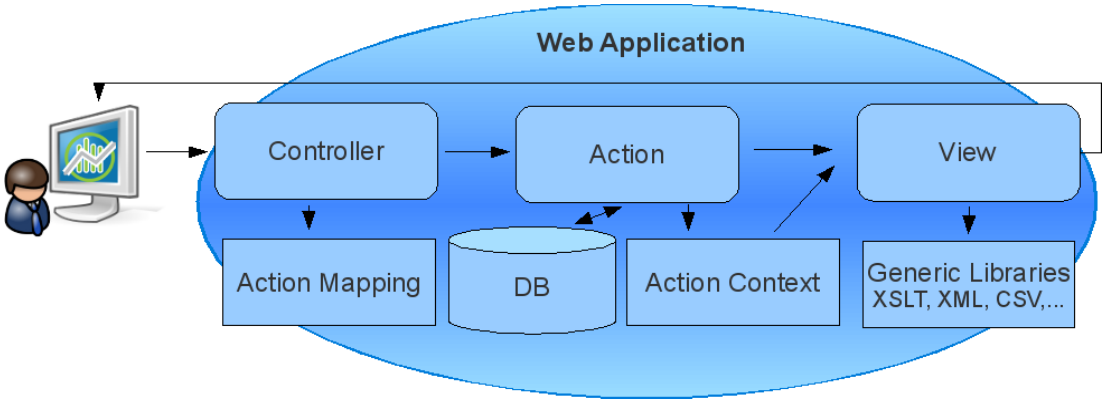


Figure 5.4: The sequence of actions of the Web Application.

The Dashboard Task Monitoring application is built on top of the Dashboard Job Monitoring system which uses multiple sources of information [151]. There are two main architectural principles of the Dashboard Job Monitoring system:

1. Monitoring should not be intrusive to the information source. Thus, it does not pool information from the primary monitoring sources on a regular basis to avoid adding additional load on the services responsible for the job processing.
2. The Dashboard uses a message-oriented architecture. There is no synchronous connection to the primary information producer. The job submission tools as well as the jobs themselves are instrumented to report in real time important events to the MonALISA servers. The Dashboard Collectors regularly consume information published by the MonALISA servers. At the time when the development of the Dashboard started in the summer of 2005, no messaging system was provided as a standard component of the Grid Middleware stack. The MonALISA system was selected to be used as a messaging system for the Dashboard. Currently, the Dashboard development team is integrating the Dashboard with the Messaging System for the Grid (MSG) [137].

The data collectors gather both Grid-related information as well as information specific to the application which is run by the users as illustrated in Figure 5.5. The Grid-related information is obtained in the XML format from the Logging and Bookkeeping Database using the Imperial College Real Time Monitoring publisher (ICRTM). The application-specific information is gathered throughout a job's lifetime via the MonALISA monitoring system.

The job submission tools of the CMS experiment and the job wrappers generated by these tools are instrumented to report meta-information about a user's tasks and the progress of a user's job to the MonALISA server. The Dashboard then presents all this information in a coherent way, as if all of it came from one source [152].

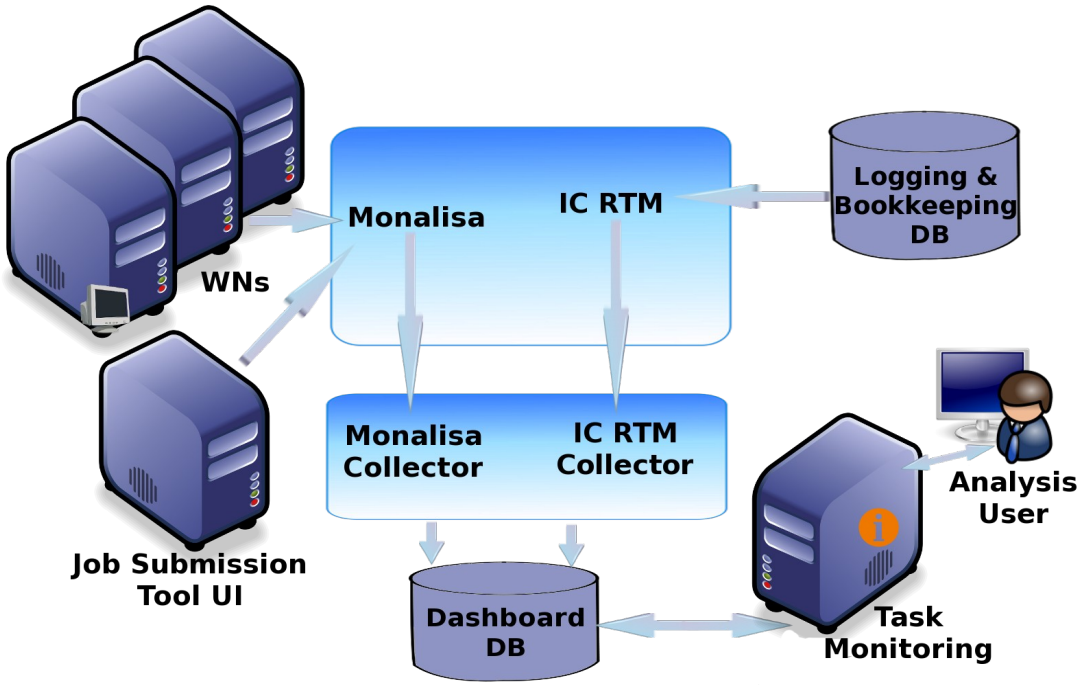


Figure 5.5: Job Information Gathering.

5.3 Implementation

The Python language was chosen for the development of the CMS Dashboard Task Monitoring due to the power, flexibility and speed of development that it offers. It is also widely used within the High Energy Physics community. Apache 2.0.52 (as of November 2009) was chosen to provide the client interface as it has a history of being flexible, secure and performant. The dojo javascript toolkit was used to connect the web interface with the database. Finally, the Graphtool [153] python library was used for the creation of all the plots.

The major components that were identified in the requirements are illustrated in Figure 5.6 and are discussed in more detail in the following sections. The client revolves around the concept of a task which coordinates all of the actions required to satisfy the user requirements.

The relation between the Action and the View python classes and their generated output files is being illustrated in Figure 5.7. All the Action classes access the database to collect the data and if a calculation in the results is needed, they forward the data to the appropriate View class for the calculation and then the data is returned to the user in

the appropriate output format. There are also 40 Action and View python classes and 20 Output image files for the 20 available plots generated by the application. These python classes are not shown in Figure 5.7 for clarity reasons.

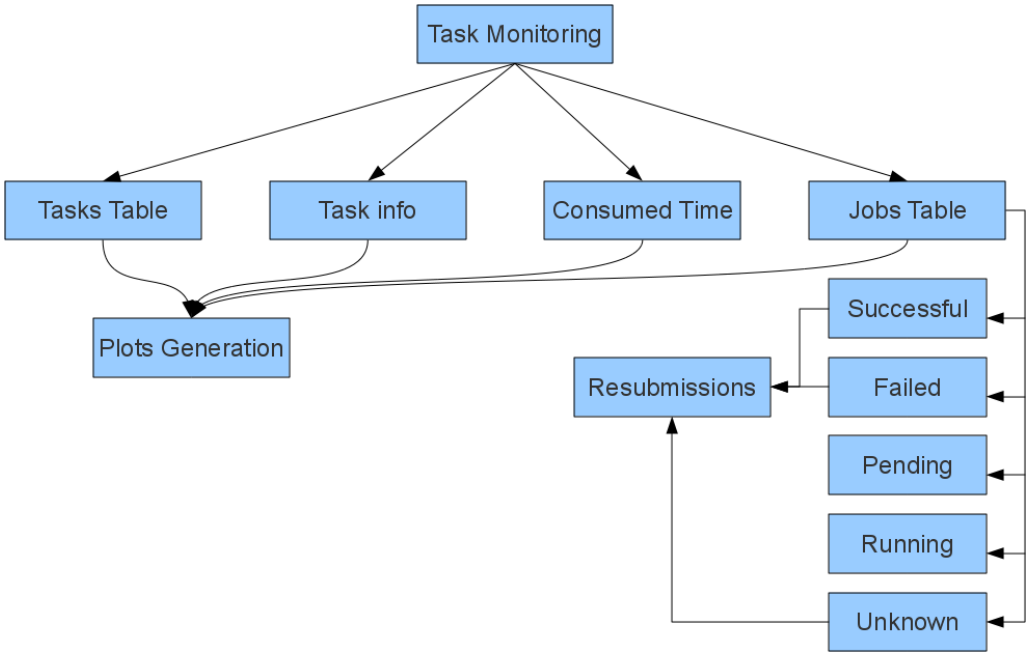


Figure 5.6: The major components of the application.

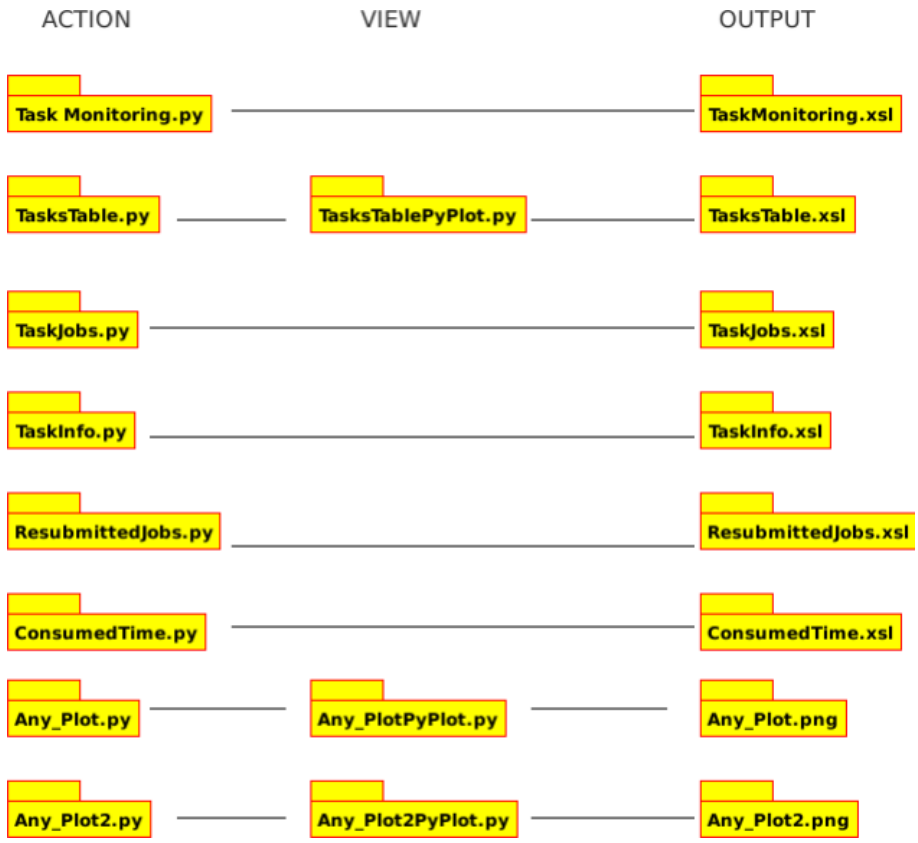


Figure 5.7: The relationship between the Action and the View python classes and their generated output files.

5.3.1 CMS Dashboard Database Schema

The CMS Dashboard Task Monitoring application is built on top of the CMS Dashboard Job Processing Data Repository. To ensure a clear design and maintainability of the application, the actual monitoring queries are decoupled from the internal implementation of the data storage.

The CMS Dashboard Task Monitoring application comes with a Data Access Object (DAO) implementation that represents the data access interface. Access to the database is done using a connection pool to reduce the overhead of creating new connections and therefore, the load on the server is reduced and the performance is increased. A flowchart illustrating all the major paths for a client request is shown in Figure 5.8.

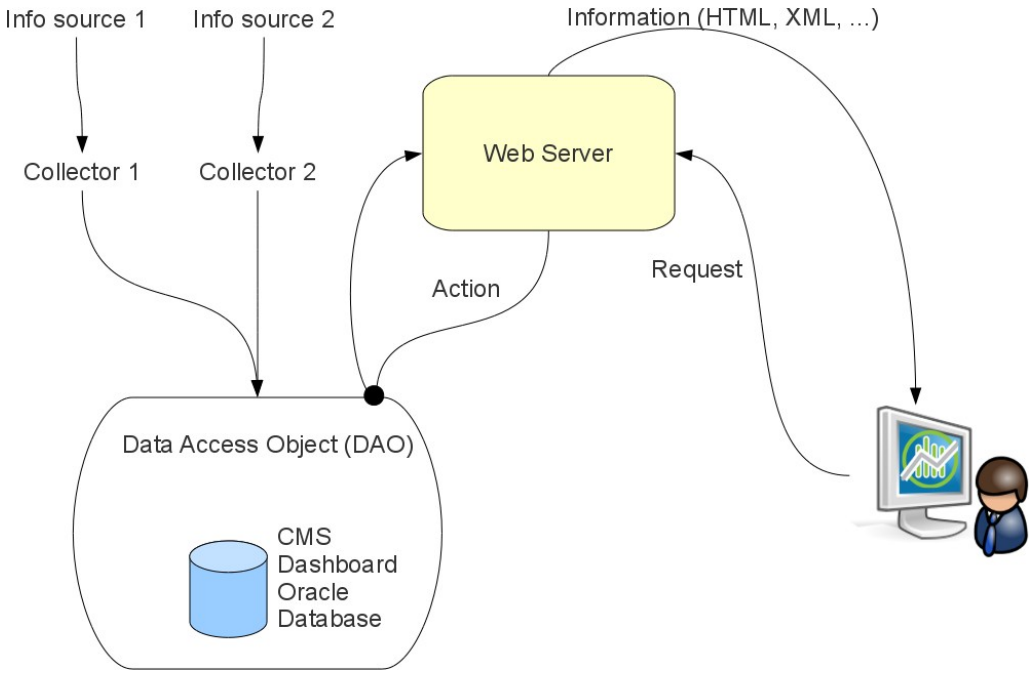


Figure 5.8: Client Request Flowchart.

Figure 5.9 illustrates the entity relationship diagram between the most important tables of the database used by the CMS Dashboard Task Monitoring application. The job table contains information regarding the job itself such as the number of events to be analysed, the task to which it belongs, the site at which the job is running and various submission timestamps. The task table contains task-specific information such as the task creation timestamp, the name of the task, the submission method used, the user that has submitted this task, the input collection and the target Computing Element (CE). The site table contains site-specific information such as the site name, the country that

the site belongs to, the Computing Elements of the site and the worker nodes of the site.

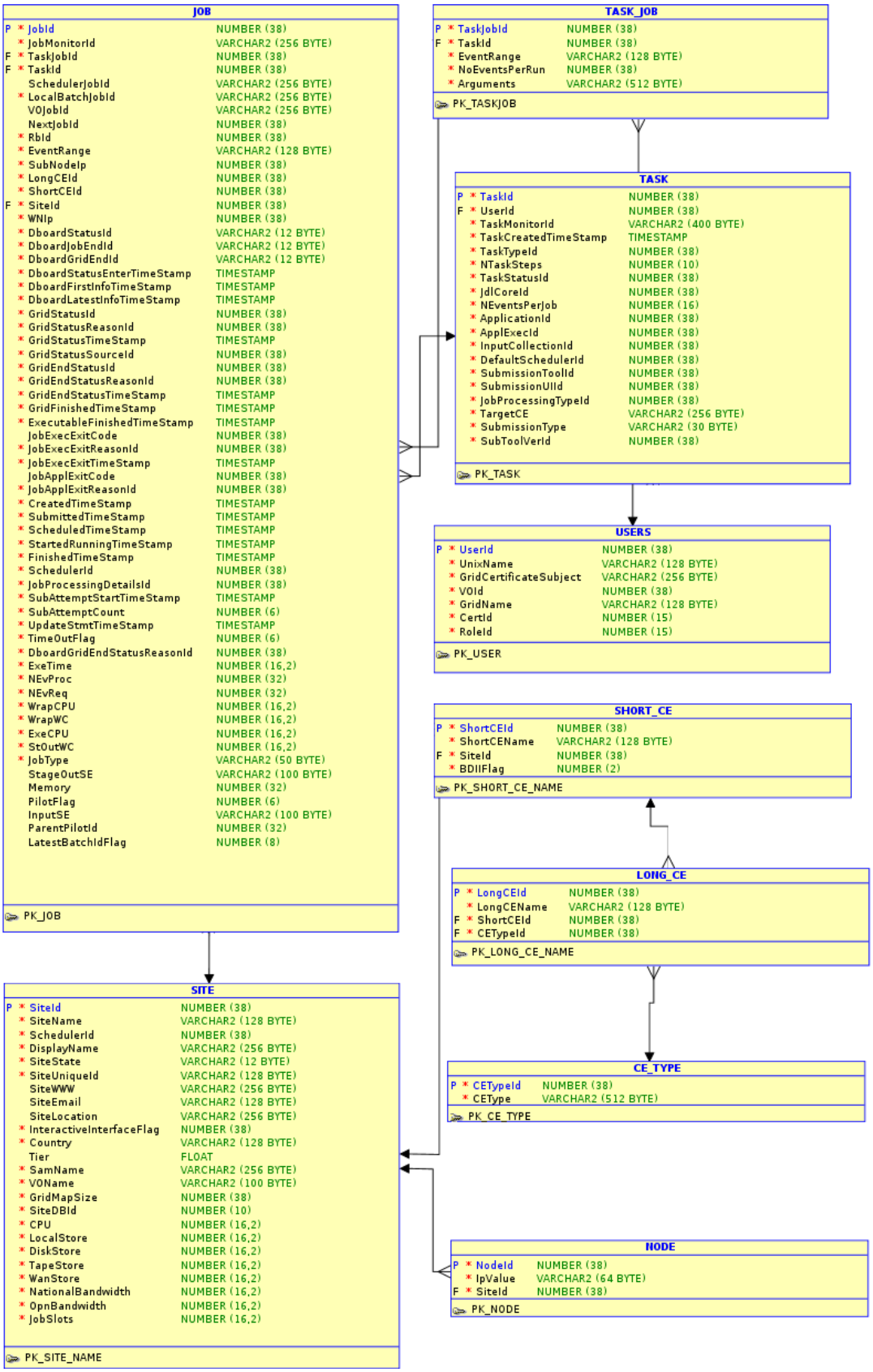


Figure 5.9: The Entity Relationship Diagram.

The connection to the database is defined in a single configuration file, the dashboard-dao-oracle-job.cfg as illustrated in Listing 5.1.

```
### ORACLE SPECIFIC CONFIGURATION
[oracle]
# Home of the oracle libraries
oracle_home  = /var/www/tmp
# Connection parameters
# You can either specify a set of 'user', 'password', 'host', 'port', 'sid'
# or set the full connection string in the 'connect_string' property
user        = <username>
password    = <password>
host        = <hostname>
port        = <port>
sid         = <sid>
connect_string =
(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
(HOST=<hostname>)(PORT=<port>)))(CONNECT_DATA=(SID=<sid>)))
# Pool configuration parameters
pool_min_size = 1
pool_max_size = 2
pool_increment = 1
pool_mon_interval = 600
```

Listing 5.1: The configuration file for the database connection.

5.3.2 SQL Queries

The most important SQL database queries of the application can be seen in Appendix A.6.

5.3.3 Gridsite Authentication

We have integrated the CMS Dashboard Task Monitoring with the Gridsite library [154] to enable secure access to the information based on the X509 authentication. GridSite was originally a web application developed for managing and formatting the content of the GridPP website. Over the past three years it has grown into a set of extensions to the Apache web server and a toolkit for Grid credentials, GACL access control lists and http(s) protocol operations. The sequence of actions can be seen in Figure 5.10.

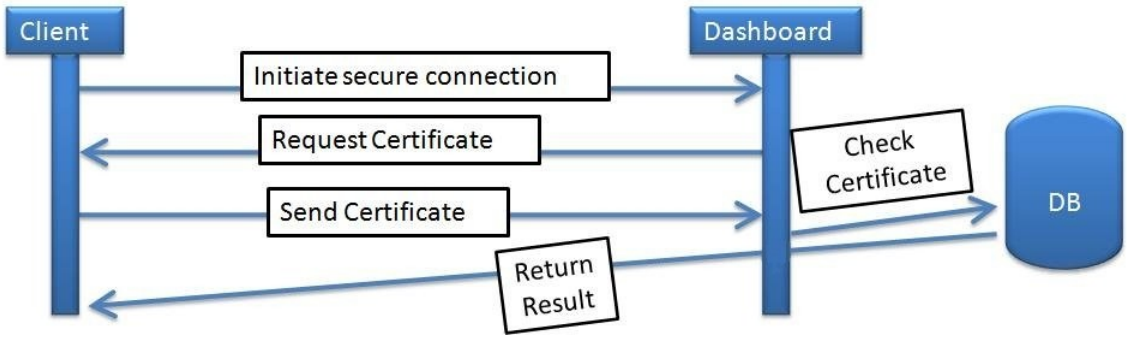


Figure 5.10: Sequence of Actions for the Authentication Mechanism.

The authentication module was developed after some CMS users highlighted privacy concerns regarding users being able to view and follow the tasks submitted by others. Another reason was to personalise the available content shown to the user. When the user logs in to the application, the information will be presented automatically by the application and this information is focused on the user only and not to all the existing CMS users.

The authentication module is optional and not used by default. Hence, everyone is an administrator by default. When the module is enabled, the Grid Certificate must be loaded in the user's browser. If the client's Grid Certificate is loaded on the browser, we check if the user's Distinguished Name (DN) matches any entries from the table 'ADMIN_USERS'. If it matches, the user is an administrator and we execute the following query that fetches the full list of the users on the system.

```
userQuery = 'select distinct users."GridName" from users, task
    where users."UserId" = task."UserId" and task."TaskCreatedTimeStamp" >
    sysdate – 31 and task."TaskTypeId" in (select "TaskTypeId" from task_type
    where "Type" in (\'analysis\', \'JobRobot\')) order by users."GridName"'
```

Listing 5.2: Fetching the full list of the users on the system.

If there is no match between the user's DN and an entry from the table 'ADMIN_USERS', authentication is being used and the user is not an administrator. We execute the following query so that the user will only see his own jobs.

```
userQuery = 'select distinct users."GridName" from users, task
where users."GridCertificateSubject" = :clientDNstring and users."UserId" =
    task."UserId" and task."TaskCreatedTimeStamp" > sysdate – 31 and
    task."TaskTypeId" in (select "TaskTypeId" from task_type where "Type" in
    (\'analysis\', \'JobRobot\'))'
```

Listing 5.3: Fetching only the user's jobs.

5.3.4 Advanced Graphical Plots

Graphical plots were developed to present to the physicist user a more usable and attractive interface and to visually represent the data contained in an analysis operation. The Graphtool python library was used to create the plots. The sequence of actions for the generation of a graphical plot is illustrated in Figure 5.11.

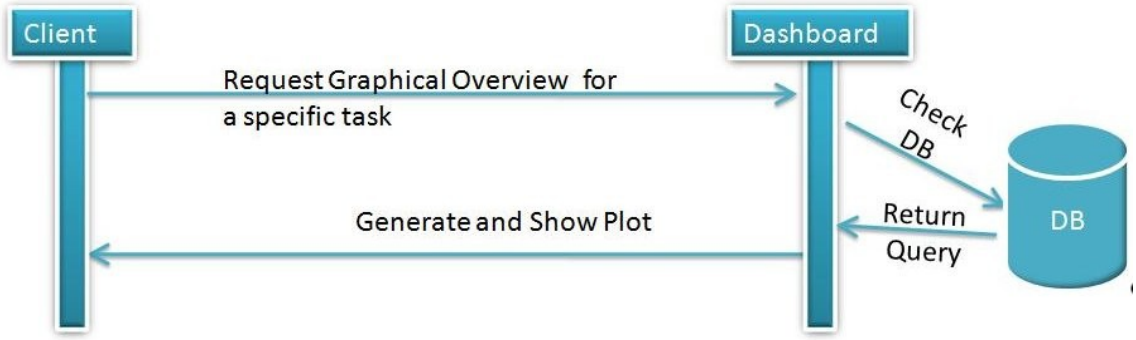


Figure 5.11: Sequence of Actions for the Advanced Plot Generation.

The python code for the generation of a simple graphical overview plot can be seen in Appendix A.5. The library has been patched and extended to support custom colouring of the legends by using the 'color_override' option. The patches are available in the Appendix A.2. The application offers a wide-variety of graphical plots and these plots will be presented in the next section.

5.3.5 User Interface and Monitoring Features

CMS Dashboard Task Monitoring provides monitoring functionality regardless of the job submission method or the middleware flavour and it works transparently across various Grid infrastructures which is the reason why it is so heavily used by many analysis users [131][155]. It is easy to understand how it works and how to navigate throughout the tool. It is clean and intuitive in layout and it contains no unnecessary information as illustrated in Figure 5.12.

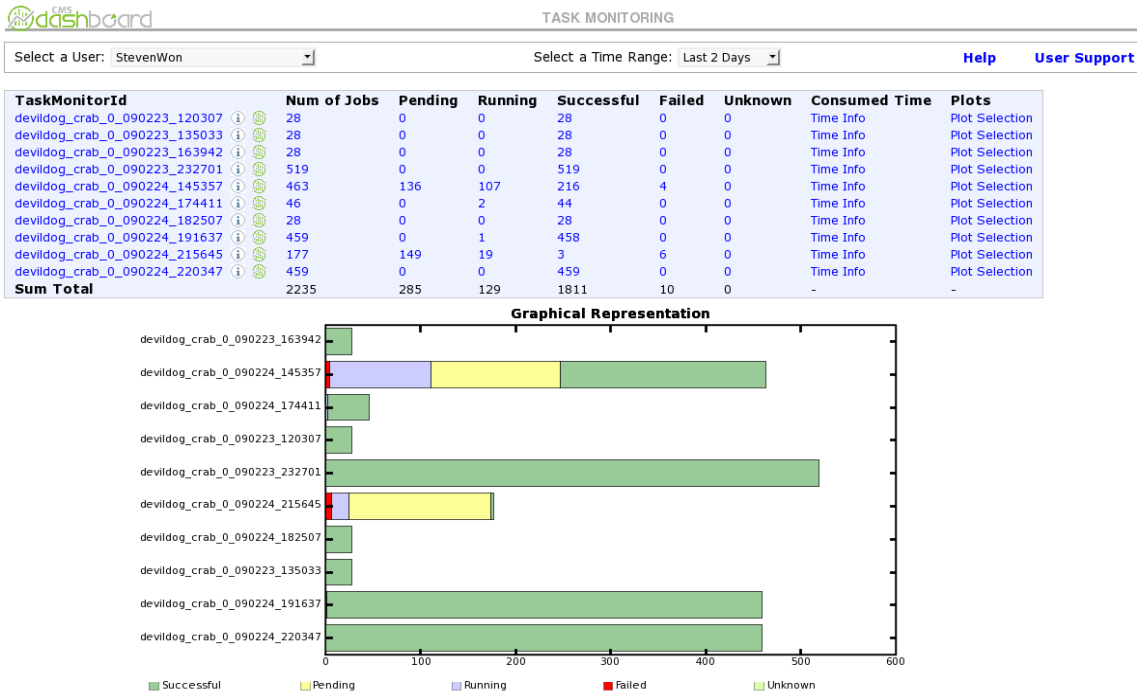


Figure 5.12: The User Interface.

A snapshot of the user interface can be seen in Figure 5.12. The user interface is divided into three parts. On the first, upper, part of the interface, the user can choose his/her identity from the “Select a User” field, select the time window to define the tasks submitted during a given time range. The user should get a list with all of his/her tasks submitted over the chosen time range on the second part of the interface. The graphical representation of the table will be available on the third part of the interface. The “Help” and “User Support” buttons, available on the upper right part of the interface, provide a quick access to the user's manual and the meanings of the error exit codes. The user manual is available in Appendix A.4. The user can also retrieve the result of this table in the XML format by using the following command:

```
$ curl -H 'Accept: text/xml' 'http://dashb-cms-sam.cern.ch/dashboard/request.py/taskstablexml?
        &typeofrequest=A&timerange=TIMERANGE&usergridname=USERNAME' > /tmp/action.xml
```

Listing 5.4: Retrieving the results in the XML format.

Where the USERNAME is the user's username and the TIMERANGE can be lastDay, last2Days, last3Days, lastWeek, last2Weeks and lastMonth.

The XML output will be a bit hard to read because there is no newline break. The output can be reformatted by using the 'xmllint' command:

```
$ xmllint --format /tmp/action.xml
```

Listing 5.5: Reformatting the XML output.

Clicking on the information link next to the name of the task provides meta-information such as input dataset, version of the software used by the task and of the submission tool and the task creation time. Clicking on the number of jobs corresponding to a given status provides a detailed information of all the jobs of a selected category as presented in Figure 5.13.

SchedulerJobId	Id in Task	Status	Appl Exit Code	Grid End Status	Retries	Site	Submitted	Started	Finished
https://lb001.cnaf.infn.it:9000/BKppgBs4ZbTA5MQ8x_ErBA	1	Success	0	Done	1	T2_DE_RWTH	2009-05-07 12:14:35	2009-05-07 12:15:46	2009-05-07 12:55:07
https://lb001.cnaf.infn.it:9000/lP-sXaZ9gUCtuIoZVmaUhw	2	Failed	60307	Done	1	T2_ES_CIEMAT	2009-05-07 12:14:35	2009-05-07 12:16:39	2009-05-07 13:43:17
https://lb006.cnaf.infn.it:9000/0soRAPyMFeC0sW9Mi0saVg	3	Success	0	Done	4	T2_ES_IFCA	2009-05-07 14:59:06	2009-05-07 15:00:57	2009-05-07 16:25:13
https://lb001.cnaf.infn.it:9000/KiOPRkSqWNMg56svJnf1_Q	4	Success	0	Done	1	T2_ES_CIEMAT	2009-05-07 12:14:35	2009-05-07 12:16:38	2009-05-07 13:05:43
https://lb006.cnaf.infn.it:9000/c8ekYBXGagaaPnniTah7AA	5	Success	0	Done	4	T2_ES_IFCA	2009-05-07 14:59:06	2009-05-07 15:02:33	2009-05-07 16:12:40
https://lb001.cnaf.infn.it:9000/mq2oNvZSmSQLK01bKsX2AQ	6	Success	0	Done	1	T2_DE_RWTH	2009-05-07 12:14:35	2009-05-07 12:15:34	2009-05-07 13:09:01
https://lb001.cnaf.infn.it:9000/qdvv4gGq7xWIwELK8j46Qg	7	Success	0	Done	1	T2_ES_CIEMAT	2009-05-07 12:14:35	2009-05-07 12:16:38	2009-05-07 13:14:31
https://lb001.cnaf.infn.it:9000/wiwc2I16pNQNWVmUS1HSmg	8	Success	0	Done	1	T2_DE_RWTH	2009-05-07 12:14:35	2009-05-07 12:15:57	2009-05-07 12:54:33
https://lb001.cnaf.infn.it:9000/LH3Qn42kxmSP5g3IJefsww	9	Success	0	Done	1	T2_ES_CIEMAT	2009-05-07 12:14:35	2009-05-07 12:17:37	2009-05-07 13:20:43
https://lb001.cnaf.infn.it:9000/1aUfBAtf8QpaPDIYWzr3wA	10	Success	0	Done	1	T2_US_Nebraska	2009-05-07 12:14:35	2009-05-07 12:16:36	2009-05-07 13:08:04

Figure 5.13: Detailed Job Information.

The user can also retrieve the result of this table in the XML format by using the following command:

```
$ curl -H 'Accept: text/xml' 'http://dashb-cms-sam.cern.ch/dashboard/request.py/taskjobsxml?
    &timerange=TIMERANGEs&what=all&taskmonid=TASKNAME' > /tmp/action.xml
```

Listing 5.6: Retrieving the jobs of a task in the XML output.

Where the TASKNAME is the name of the task, the TIMERANGE can be one of the options mentioned previously and 'what' can be: 'all' for all the jobs, 'f' for the failed ones, 'r' for the running ones, 'p' for the pending ones, 's' for the successful ones and 'u' for the unknown jobs. The XML output will be a bit hard to read and it can be reformatted by using the 'xmllint' command provided in Listing 5.5.

Clicking on any name on the 'Site' column opens the Site Status Board for the CMS Sites [156], providing a 24-hour status availability of the selected site allowing to identify any problematic site and blacklist it from resubmissions as illustrated in Figure 5.14.

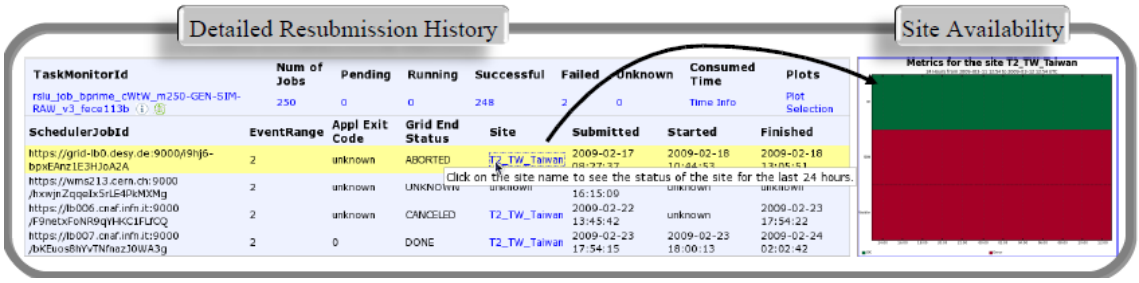


Figure 5.14: Site Availability for the CMS Sites.

Also, clicking on the *'Retries'* column provides a detailed re-submission history of every single job which can be very useful for debugging purposes. An example can be seen in Figure 5.15; the job produced an output to the Storage Element (SE) but the staging out finished with an error (exit code: 60307), thus, all following resubmissions had no chance to succeed, since the file was already created on the SE (exit code: 60303). Before any further resubmission, the output file generated by the previous attempt should be removed from the SE.

TaskMonitorId	Num of Jobs	Pending	Running	Successful	Failed	Unknown	Consumed Time	Plots
dimatteo_crab_Tan10_60_250_famos_u6w2z0	500	0	5	257	235	3	Time Info	Plot Selection

SchedulerJobId	Id in Task	Appl Exit Code	Grid End Status	Site	Submitted	Started	Finished
https://lb006.cnaf.infn.it:9000/qEhEc7GPI7veIFfqLoiBPw	1	60307	Unknown	T3_UK_London_QMUL	2009-05-06 09:23:43	2009-05-06 09:27:01	2009-05-06 18:57:12
https://wms212.cern.ch:9000/yKNo9UB8G_mSiBvt_bnzVA	1	60303	Done	T2_UK_SGrid_Bristol	2009-05-07 03:32:35	2009-05-07 03:38:15	2009-05-07 06:09:06

Figure 5.15: Detailed Resubmission Information

Currently, the strongest point of the application is the failure diagnostics for the Application failures. It is extremely useful to get not only the exit-code of the failed job, which sometimes can be misleading, but a detailed reason of failure as well, i.e. 'Could not save output file A on the storage element B'. The ideal goal would be to reach to a point where a user shouldn't have to open the log file and search for what went wrong with the job. The user could get everything from the monitoring tool. An example can be seen in Figure 5.16.

SchedulerJobId	Id in Task	Appl Exit Code	Appl Exit Reason	Grid End Status	Site	Submitted	Started	Finished
https://lb008.cnaf.infn.it:9000 /WX4c1wqOnp45-3M6OBlxdA	218	60307	'Copy succ...	Unknown	T2_IT_Pisa	2009-09-19 23:58:54	2009-09-20 00:31:11	2009-09-20 01:30:48
https://lb001.cnaf.infn.it:9000 /mX0paBwOMTT_UPK9pFtPXw	218	60303	'file tota...	Unknown	T2_IT_Pisa	2009-09-20 02:04:00	2009-09-20 02:16:13	2009-09-20 02:28:51

file total_events_218.root already exist

Figure 5.16: Detailed Reason of Failure.

The application offers a wide variety of graphical plots that will visually assist the user to understand the status of the task. These plots show the distribution by site of successful, failed, running and pending jobs as well as for the processed events (Figure 5.17a) and they can help identify any problematic site and blacklist it from further resubmissions (Figure 5.17b). They also demonstrate the terminated jobs in terms of success or failure and over the time range that the task has been running (Figure 5.17c). In the case of failure, the distribution by reason is demonstrated, whether it be Grid-Aborted or Application-Failed jobs (Figure 5.17d).

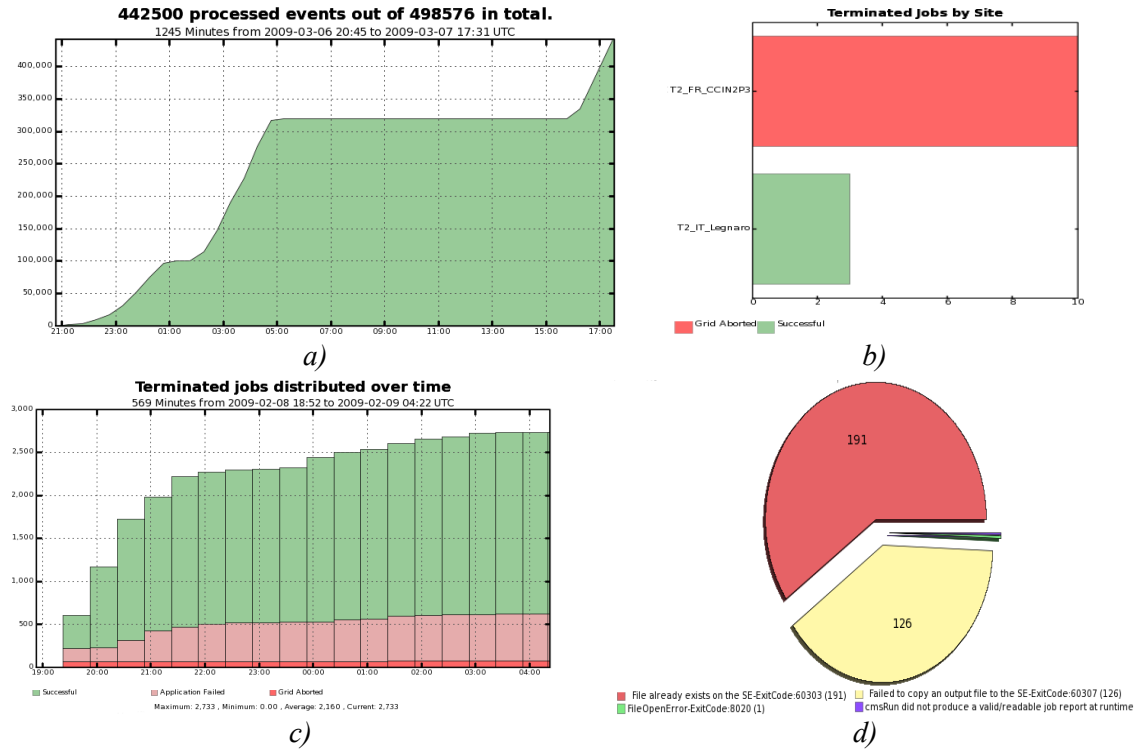


Figure 5.17: Graphical Plots: a) Processed Events over Time, b) Terminated Jobs by Site, c) Terminated Jobs over Time, d) Reason of Failure.

Various kinds of consumed time plots are available such as the distribution of CPU and Wall Clock time spent for successful and failed jobs and the average efficiency distributed by site as illustrated in Figure 5.18. These plots will help the user to see how the CPU time per event and efficiency can vary depending on the site that the jobs are running on. The user gets information regarding the time that has been consumed for a specific task or a given job.

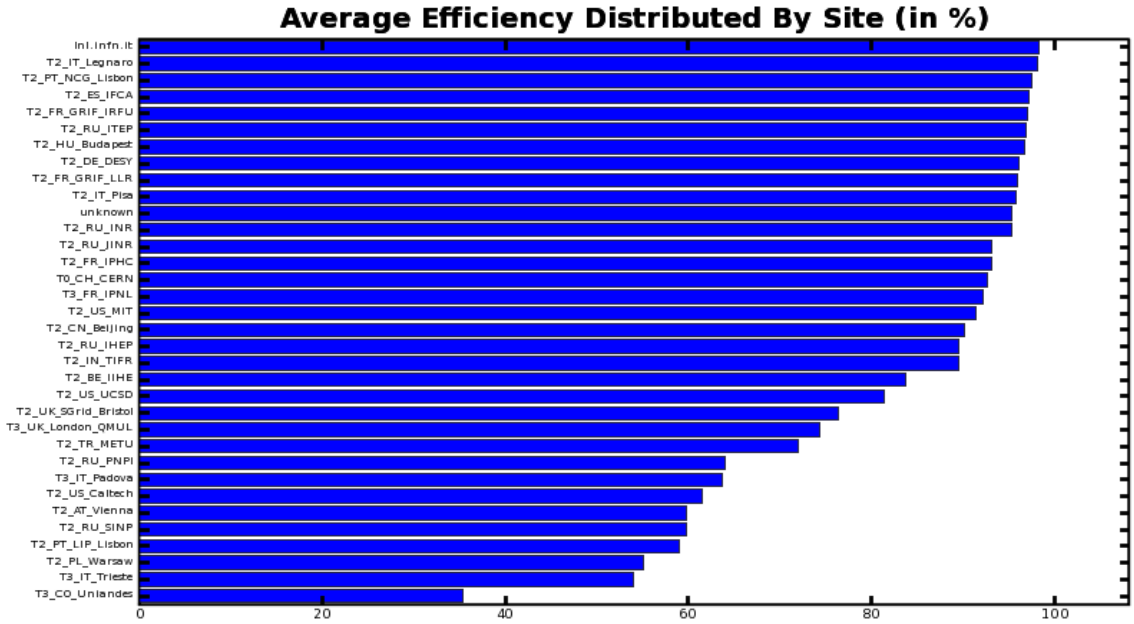


Figure 5.18: Efficiency Distributed by Site.

For any given task (Figure 5.19), the following information is available: the average efficiency of the task, the total and the average CMSSW CPU and job wrapper Wall Clock time usage and the average CPU time spent per event. The average efficiency per task is calculated by the following formula:

$$Efficiency\ per\ task = \sum (WC\ Time / CPU\ Time)$$

TaskMonitorId	Num of Jobs	Pending	Running	Successful	Failed	Unknown	Consumed Time	Plots
grace_crab_0_090208_201828	575	0	0	406	169	0	Time Info	Plot Selection

Time Plots

Average CPU Time Per Event Distributed by Site	Average Efficiency Distributed by Site	Total CPU Time Distributed by Site	Total Wall Clock Time Distributed by Site	Distribution of CPU time	Distribution of Wall Clock time	CPU Time Distributed by Site over Time	Wall Clock Time Distributed by Site over Time

Consumed Time in (HH:MM:SS) format

Total CMSSW CPU Time	Total job wrapper Wall Clock Time	Average CPU Time Per Event (in sec.)	Average Efficiency	Average CMSSW CPU Time per Job	Average job wrapper Wall Clock Time per Job
1 day, 18:20:00	12 days, 11:53:15	0.0340613841924	23.96%	0:04:45	0:33:45

Figure 5.19: Consumed Time information for a selected task.

At the job-level the user gets information about the efficiency of every single job separately as illustrated in Figure 5.20. The processing efficiency per job is calculated by the following formula:

$$Efficiency\ per\ job = WC\ Time / CPU\ Time$$

SchedulerJobId	Id in Task	Retries	Appl Exit Code	Grid End Status	Events	Site	Submitted	Started	Finished	Efficiency
https://lb001.cnaf.infn.it:9000/J46_5SQdnCF3hgPFfyMPxQ	1	1	0	Done	1000	T2_UK_London_Brunel	2009-06-03 12:20:33	2009-06-03 12:22:14	2009-06-03 12:25:28	35.25%

Figure 5.20: Job-level processing efficiency.

A selection of snapshots of the application can be seen in Figure 5.21.

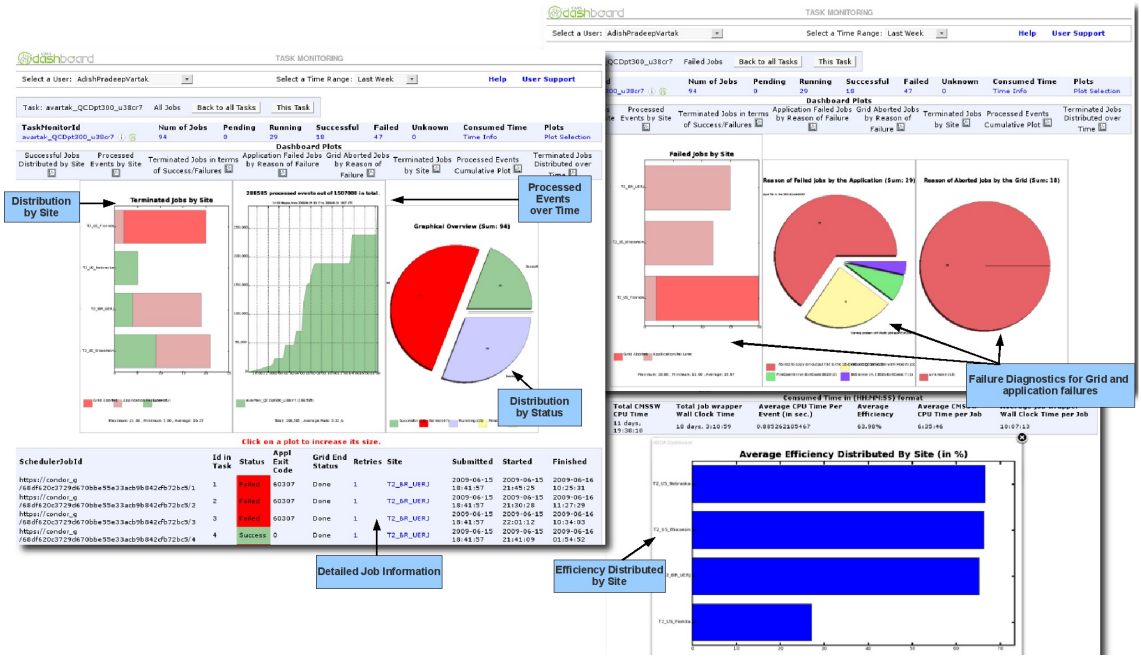


Figure 5.21: A selection of snapshots of the application.

5.4 Experience of the CMS User Community with Task Monitoring

In the CMS Community, the CMS Remote Analysis Builder (CRAB) is used for the job submission. CRAB is a Python programme simplifying the process of creation and submission of CMS analysis jobs to the Grid environment. CRAB can be used in two ways; as a standalone application and with a server.

The standalone mode is suited for small tasks and it submits the jobs directly to the scheduler and these jobs are under the user's responsibility. In the server mode, suited for larger tasks, the jobs are prepared locally and then passed on to a dedicated CRAB Server which then interacts with the scheduler on behalf of the user and performs additional services such as automatic resubmissions and output retrieval.

Rather often, CMS Dashboard Task Monitoring discovers previously undetected problems with the CRAB Server or the Workload Management Systems (WMS). The Dashboard reports a job as 'finished' when the job finishes on the worker node but the job status updates by the Grid services can introduce some latency and they are quite often delayed due to a component of the CRAB Server or due to problems of the WMS or of the Logging and Bookkeeping system (LB). Thus, when the users see a big delay in status updates in CRAB compared to the status shown in CMS Dashboard Task Monitoring, they report the problem and after investigation either the CRAB Server is fixed or the faulty WMS is blacklisted.

A user support campaign has been performed to bring awareness to the CMS User Community for the CMS Dashboard Task Monitoring application, collect feedback, assemble further requirements and identify weaknesses with the application. Two hundred analysis users were contacted via e-mail. A very positive feedback response has been received; the results of our user survey are available in Appendix A.3 along with their feature requests.

According to our web statistics [131][155], more than one hundred distinct analysis users are using CMS Dashboard Task Monitoring for their everyday work as illustrated in Figure 5.22. The Dashboard Applications Usage Statistics programme was developed by the author to count the daily total number of distinct users using a selected number of CMS Dashboard applications. In order to count the distinct daily users, the daily access_log file of the apache http web server was used.

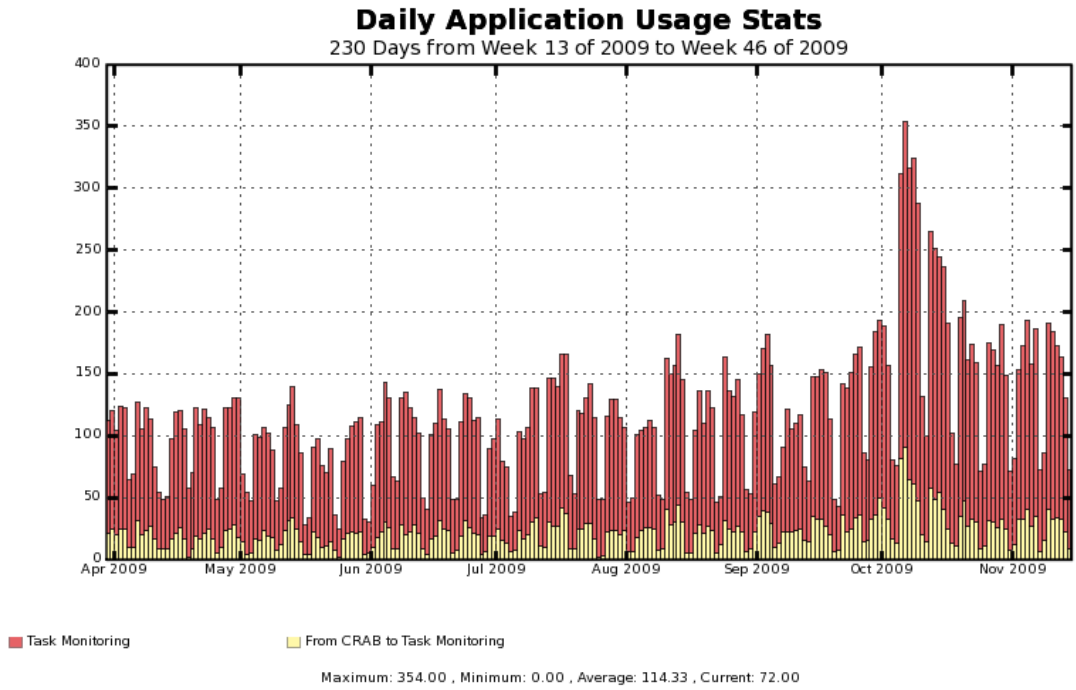


Figure 5.22: Daily Usage Statistics.

The following bash script commands (Listing 5.7) were used in a python programme to determine the date of the month and the total number of distinct daily users using some selected applications according to the total number of unique visitor IPs.

```
# Command to get the date of the month:
getDate = "zgrep +0 /var/log/httpd/access_log.1.gz | awk '{print $4}'| uniq | head -n 1| cut -c 2-13"
# Commands for the usage of the following applications:
TaskMon = "zcat /var/log/httpd/access_log.1.gz | grep taskmonitoring | awk '{print $1}' | sort | uniq |wc -l"
TaskMonCRAB = "zcat /var/log/httpd/access_log.1.gz | grep taskmon.html | awk '{print $1}' | sort | uniq |wc -l"
```

Listing 5.7: Unix bash script to determine the total number of distinct daily users.

The "TaskMon" bash command counts the total number of distinct users using the application and the "TaskMonCRAB" command counts the total number of distinct CRAB users accessing the application directly from the CRAB status output. The following unix cron command (Listing 5.8) was scheduled to run the programme daily at 06:00am for the updating of the statistics.

```
0 6 * * * python /usr/share/dashboard-stats/dashb_stats.py 2>&1 >> /var/log/script_output.log
```

Listing 5.8: Unix Cron job scheduled to update the statistics daily.

The Graphtool library was used to create the graphical plot. The daily statistics graphical plot is available in [155].

5.5 Summary

While the existing monitoring tools are coupled to a specific middleware, CMS Dashboard Task Monitoring provides monitoring functionality regardless of the job submission method or the middleware platform offering a complete and detailed view of the user's tasks including failure diagnostics, processing efficiency and resubmission history.

The monitoring tool has become very critical among the CMS users. According to our web statistics [131][155], more than one hundred distinct analysis users are using it for their everyday work. Close collaboration with several CMS users resulted in the tool being focused on their exact monitoring needs.

CHAPTER 6.

CMS DASHBOARD JOB SUMMARY

The CMS Dashboard Job Summary was the first job monitoring application to be developed, based on a vision more than experience, therefore emphasis was put on flexibility. The application provides a job-centric view aimed at understanding and debugging what happens in real-time.

This chapter discusses the development of the CMS Dashboard Job Summary application that was performed by the author and is published in [59] and [60].

6.1 Introduction

The CMS Virtual Organisation (VO) uses various fully distributed job submission methods and execution backends. The CMS jobs are processed on several middleware platforms such as the gLite, the ARC and the OSG. Up to 200,000 CMS jobs are submitted daily to the Worldwide LHC Computing Grid (WLCG) infrastructure and this number is steadily growing. These factors increase the complexity of the monitoring of the user analysis activities within the CMS VO.

Distributed analysis on the WLCG infrastructure is currently one of the main challenges of the LHC computing. Reliable monitoring is an aspect of particular importance; it is a vital factor for the overall improvement of the quality of the CMS VO infrastructure. Transparent access to the LHC data has to be provided for more than five thousand scientists all over the world. Users who run analysis jobs on the Grid do not necessarily have expertise in Grid computing. Currently, 100-150 distinct CMS users submit their analysis jobs to the WLCG daily. The significant streamlining of operations and the simplification of end-users' interaction with the Grid are required for effective organisation of the LHC user analysis. Simple, user-friendly, reliable

monitoring of the analysis jobs is one of the key components of the operations of the distributed analysis.

The goal of the CMS Dashboard Job Summary is to follow the job processing of the CMS experiment on the distributed infrastructure. The entry point of the application is the number of the jobs submitted or terminated in a chosen time period categorised by their activity such as the analysis, the production and the job robot (testing) jobs.

The CMS Dashboard Job Summary, also known as the “interactive interface”, allows to explore further down on the available information, expanding the set of jobs by various relevant properties such as the execution site, the grid gateway, the user, various completion statuses, the grid workload management host, the activity type and the dataset used, until all details stored in the Dashboard database about a chosen (set of) job(s) can be accessed. The interface reports success/failure rates according to the grid/site/application, and information on used wall clock and cpu time consumed by the jobs.

Information related to the job processing can be aggregated and presented per user, per site or Computing Element (CE), per resource broker, per application and per input collection.

The application offers very flexible access to recent monitoring data and shows the job processing at runtime. The interactive UI contains the distribution of active jobs and jobs terminated during a selected time window by their status. Jobs can be sorted by various attributes, for example, the type of activity (such as the production, analysis and test), site or CE where they are being processed, job submission tool, input dataset, software version and many others. The information is presented in a bar plot and in a table. A user can navigate to a page with very detailed information about a particular job, for example, the exit code and exit reason, important time stamps of processing the job and the number of processed events.

The CMS Dashboard Job Summary was the very first monitoring application developed in the Dashboard project. The motivation for this development, started at the

summer of 2005, was to show whether the Grid is operational, because at that period of time people were rather pessimistic about the Grid, and to show what is the status of the job processing in real time, detect any problems or inefficiencies, not necessarily with the site, but for example with a particular dataset, or particular instance of RB, or particular application version.

This is the reason why the application provides such a wide flexibility to the users; a user can sort information by any of the job/task attributes. The application does not offer long term statistics, since there is no pre-cooked information on the database. The application is using raw database data and the database was tuned for better performance with this high level of flexibility.

6.2 Design

The following sections discuss the requirements that shaped the design of the CMS Dashboard Job Summary application.

6.2.1 Objectives

The main objectives for re-developing CMS Dashboard Job Summary is to provide a more stable, maintainable release aimed at various CMS User Communities such as the VO Management Team, the coordinators and participants of various CMS computing projects such as the Analysis Support Team and CMS Site Administrators.

The main beneficiaries of this activity were the users who have come to rely on the functionality that the application provides. The increased stability and performance was a benefit to them and to new users.

6.2.2 Use Cases

A use case analysis was carried out based upon the feedback received by the CMS physicist community. The main use cases are described in Appendix B.1 and illustrated in Figure 6.1.

With the major use cases established it is possible to extract the key requirements that

the application has to fulfil. The following points represent the baseline requirements divided into principal areas.

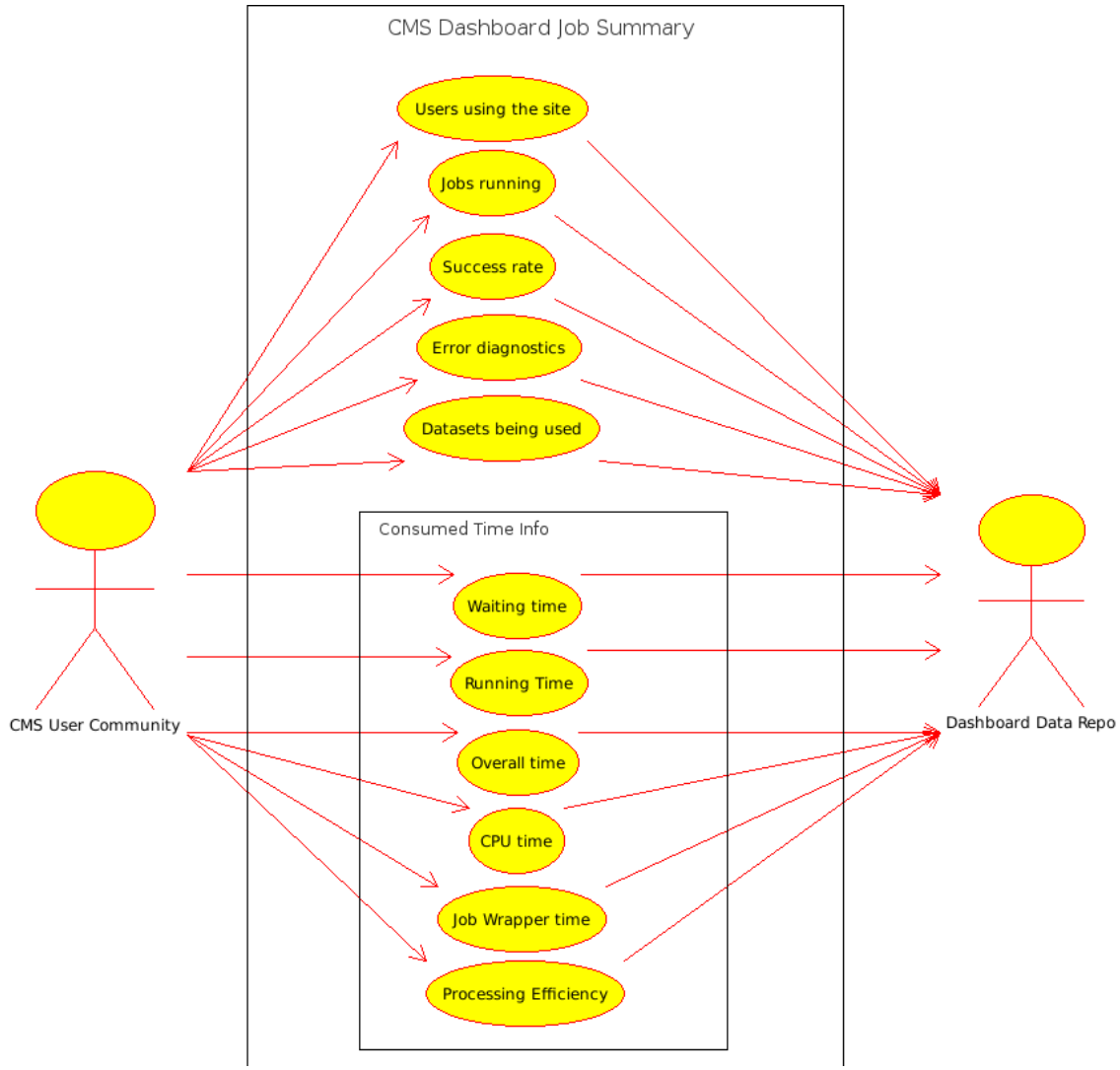


Figure 6.1: The main use cases that the application is expected to implement in conjunction with the CMS User Community Actors and the Dashboard Actor.

6.2.3 Requirements

Assumptions

1. Users have a grid certificate.
2. Users are members of the CMS VO.

User Interface

1. Users control the application via a web interface using a browser.
2. Easy to understand how it works and how to navigate throughout the tool.

3. Compatible with all the recent browsers and operating systems.
4. All of the Grids and the job submission systems that CMS uses will be supported.
5. The user will access a very detailed information of the job processing including every single resubmission that he/she might have performed for each job individually.
6. The application will be connected to the CMS Dashboard Task Monitoring for task-centric information.
7. The application will offer consumed time information and plots such as Waiting Time, Running Time, Overall Time, CPU Time, Job Wrapper Time and Processing Efficiency.
8. The user will be able to search for a specific job by entering its Grid Job ID which is a unique identifier.
9. Update in 'real-time' from the worker nodes where the jobs are running.
10. The user will be able to bookmark his/her favourite tasks for later use or to share them among his/her colleagues.
11. Offer a selection of advanced graphical plots that will visually assist the user.
12. The application will offer success rate calculation.
13. The user will be able to retrieve the results in the XML format as well as the standard HTML, XSL format.
14. The application will be built on top of the CMS Dashboard Job Monitoring Data Repository.
15. Exceptions should be caught by the application and informative error messages will be provided to the users.
16. Verbose logging should be available to identify any problems.
17. Quick access to the application's manual, help, the FAQ and the meanings of the error exit codes should be provided.

Developer's Requirements

1. Variable level of logging will be built in from the start.
2. Logging will write to stdout and to a file to ease debugging.
3. Low coupling between the components is required.
4. Minimum version of Python that is supported is determined by that installed on

lxplus.cern.ch (currently 2.3).

6.2.4 Architecture

The CMS Dashboard Job Summary application is part of the Experiment Dashboard system which is widely used by the four LHC experiments. The architecture does not differ from the one of the CMS Dashboard Task Monitoring covered in depth in Section 5.2.4.

Job status is reported to Dashboard from several information sources. The main ones are the CMS Job Submission systems such as CRAB and ProdAgent. The status changes of the jobs can be triggered by reports sent from the user interface of the Job Submission Systems, when the job status is checked, or reports from the jobs running on the Worker Node (WN). The jobs running on the WN are instrumented to report when they start running and when they finish. The exit status of the job is also reported from the WN. As soon as the job is terminated at the WN, it is turned into “terminated” status in the Dashboard.

6.3 Implementation

Python was chosen as the main development language for the CMS Dashboard Job Summary for the reasons outlined in Section 5.3. Apache 2.0.52 (as of November 2009) was chosen to provide the client interface as it has a history of being flexible, secure and performant. PHP was chosen as the implementation language for the interactive plot, due to its power and the availability of third party libraries. Javascript and AJAX were used to connect the web interface with the database. Finally, the patched version of the Graphtool python library was used for the creation of the consumed time and failure diagnostics plots.

The relation between the Action and the View python classes and their generated output files is illustrated in Figure 6.2. All the Action classes access the database to collect the data and if a calculation in the results is needed, they forward the data to the appropriate View class for the calculation and then the data is returned to the user in the appropriate output format. The output format generated from the Generic Histogram View classes is in XSL containing an image plot and a table with the requested results.

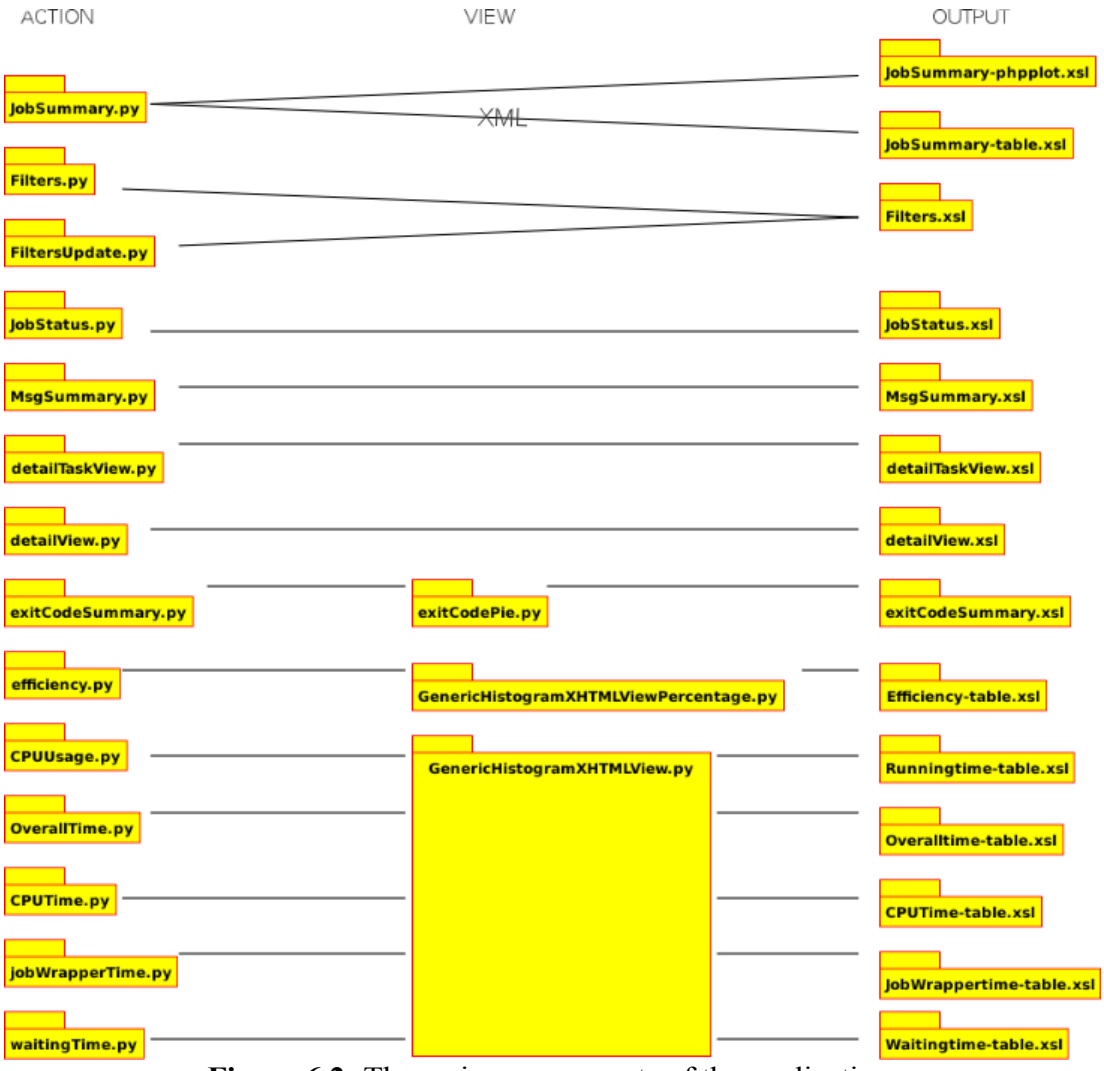


Figure 6.2: The major components of the application.

6.3.1 Filters

The filter classes contain the menu data and all of the sorting parameters. When the user enters the application for the first time of a session, the Filters python class calls the jobFilters function of the Data Access Object (DAO).

The jobFilters function contains the database queries to get the menu data for all the available parameters of the menu. The DAO then executes the queries and the python class puts the data in a shared area, the ActionContext as defined in Section 5.2.4, to be picked up by the Filters.xsl output file. The flowchart of the Filters request is illustrated in Figure 6.3.

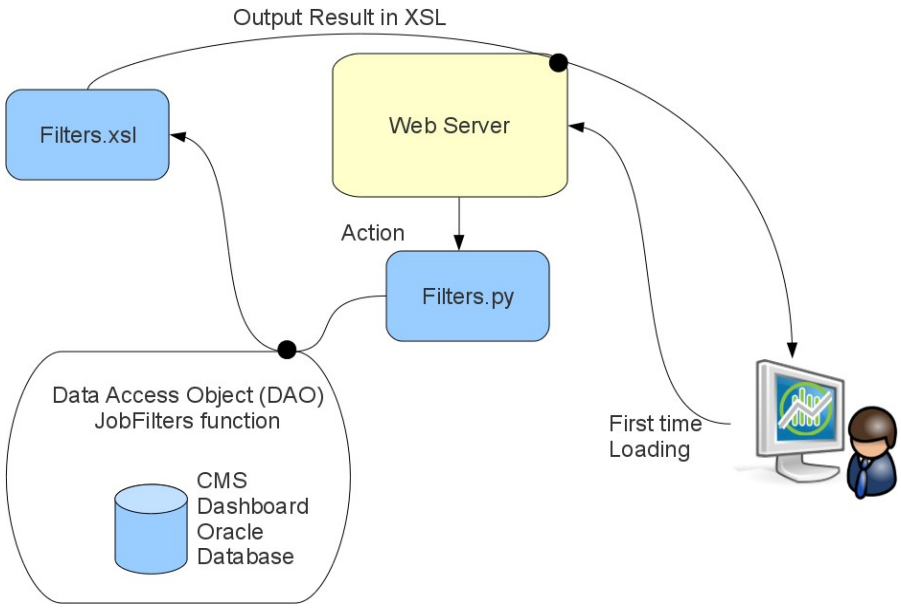


Figure 6.3: Filters Request Flowchart.

The available filter parameters can be seen in Figure 6.4. The DAO JobFilters function, executes 10 queries to get the results for the drop-down menu. The user can also select to view only a selected job status by clicking on any of the check boxes.

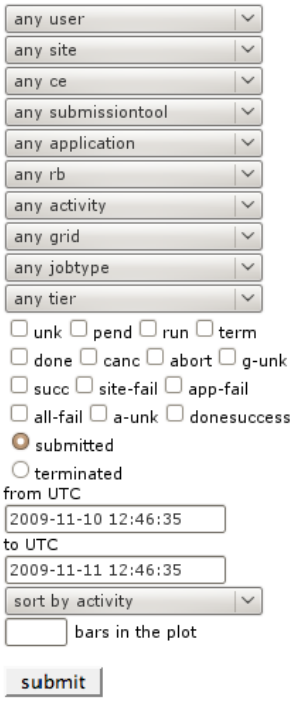


Figure 6.4: All the available parameters of the application.

The application also offers 18 different sorting parameters. These parameters are contained in a single python dictionary as illustrated in Listing 6.1.

```
menus['sortbys'] = [{'sortby':'user'}, {'sortby':'site'},{'sortby':'submissiontool'},
{'sortby':'submissionui'},{'sortby':'dataset'},{'sortby':'application'}, {'sortby':'rb'},
{'sortby':'ce'},{'sortby':'activity'}, {'sortby':'grid'},  {'sortby':'submissiontype'},
{'sortby':'task'}, {'sortby':'jobtype'}, {'sortby':'subtoolver'},{'sortby':'tier'},
{'sortby':'genactivity'},        {'sortby':'outputse'}, {'sortby':'appexitcode'}]
```

Listing 6.1: Sorting Parameters.

6.3.2 CMS Dashboard Database Schema

The CMS Dashboard Job Summary application is built on top of the CMS Dashboard Job Processing Data Repository. To ensure a clear design and maintainability of the application, the actual monitoring queries are decoupled from the internal implementation of the data storage. The application comes with a Data Access Object (DAO) implementation that represents the data access interface. Access to the database is done using a connection pool to reduce the overhead of creating new connections and therefore, the load on the server is reduced and the performance is increased.

Figure 6.5 illustrates the entity relationship diagram between the most important tables of the database used by the CMS Dashboard Job Summary application. The Job table is the most important table and it contains information regarding the job itself such as the number of events to be analysed, the task that it belongs to, the site that the job is running at and various submission timestamps. The Primary Key (P) is the JobId and there are 5 Foreign Keys (F) connecting the Job table with the Site, the Task, the Resource Broker (RB), the Short Computing Element (CE) and the Scheduler table.

The Task table contains task-specific information such as the task creation timestamp, the name of the task, the submission method used, the user that has submitted this task, the input collection and the target Computing Element (CE). The Primary Key is the TaskId and there are 8 Foreign Keys connecting the table with the User, the Task_Type, the Application, the Input_Collection, the Scheduler, the Submission_Tool, the Submission_IU and the Submission_Tool_Ver table.

The Site table contains site-specific information such as the site name, the country that the site belongs to, the Computing Elements of the site and the nodes of the site. The Primary Key is the SiteId and the Foreign Key is the SchedulerId connecting the table with the Scheduler table.

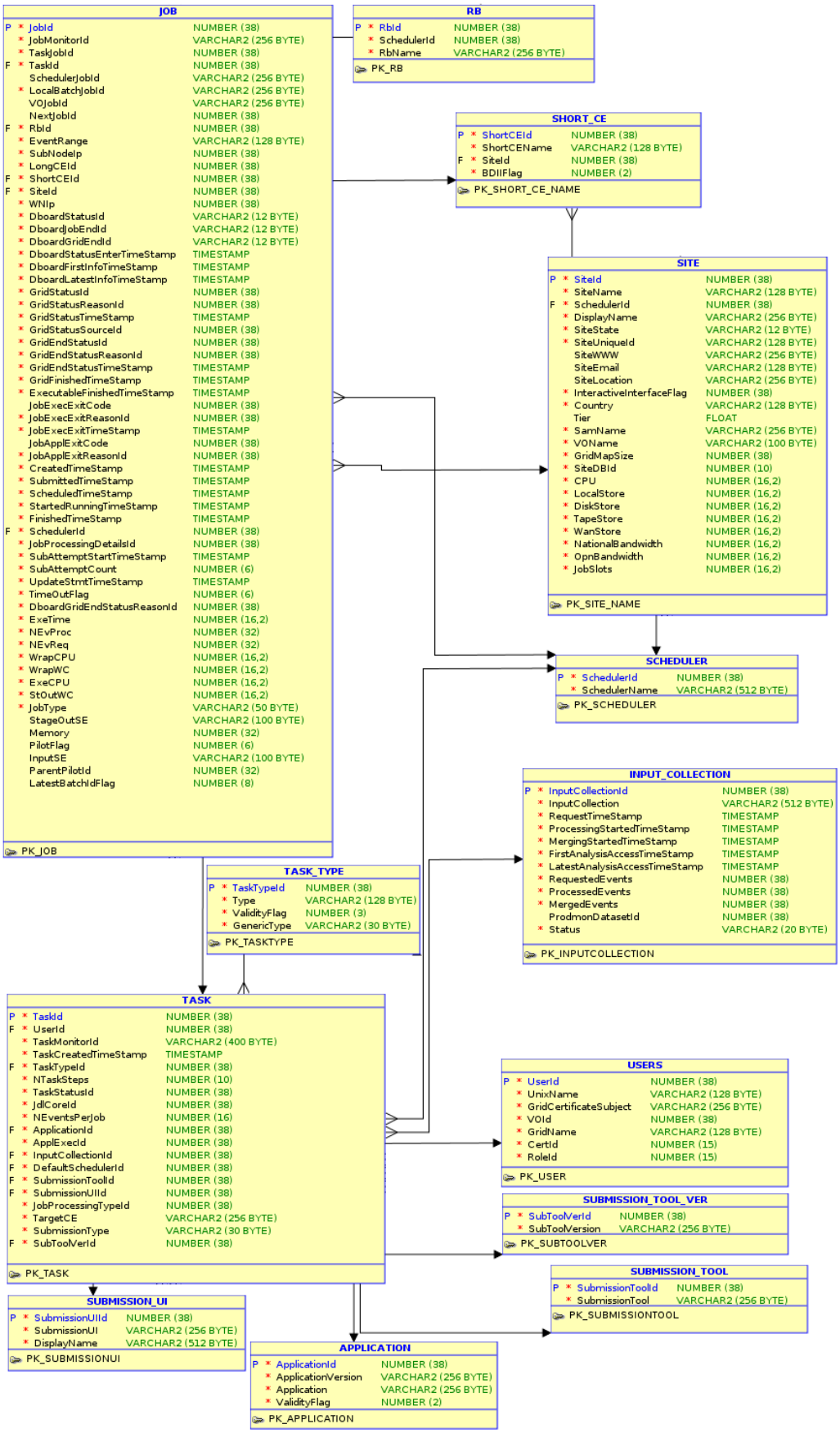


Figure 6.5: The Entity Relationship Diagram.

6.3.3 SQL Queries

The most important SQL database queries of the application can be seen in Appendix B.2.

6.3.4 User Interface

The User Interface of the CMS Dashboard Job Summary is divided in two parts. The graphical plot, the filters with their sorting parameters, the consumed time information buttons and the search field to search for a specific job can be seen in the upper part of the User Interface as illustrated in Figure 6.6.

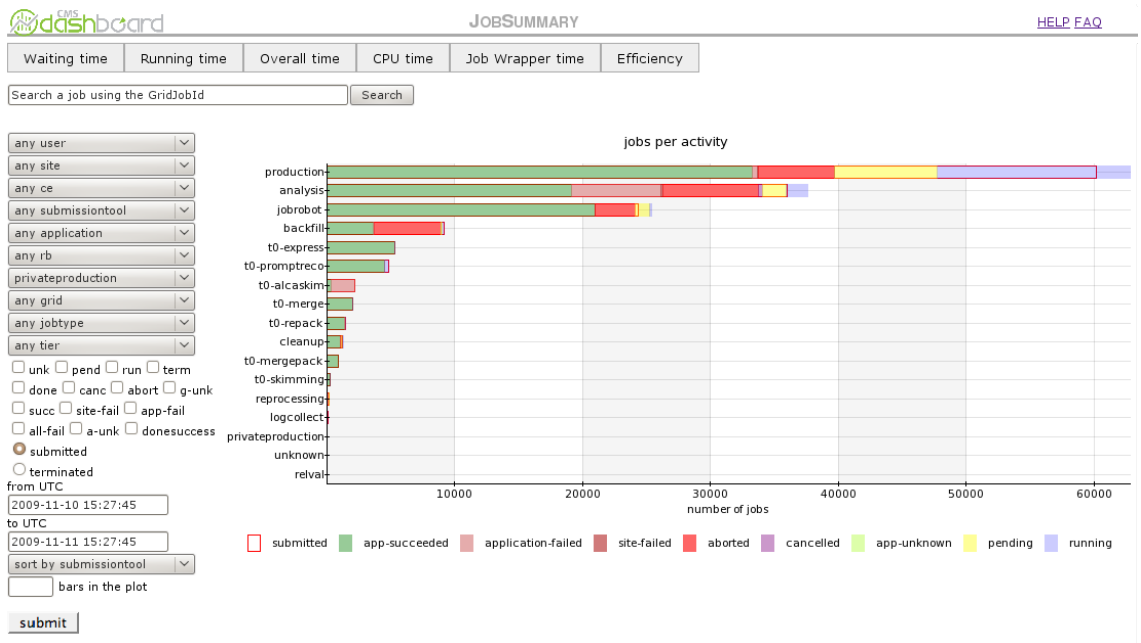


Figure 6.6: The upper part of the User Interface.

By clicking on any category on the plot, a “sort-by” menu appears allowing the user to explore further on the available information as illustrated in Figure 6.7.

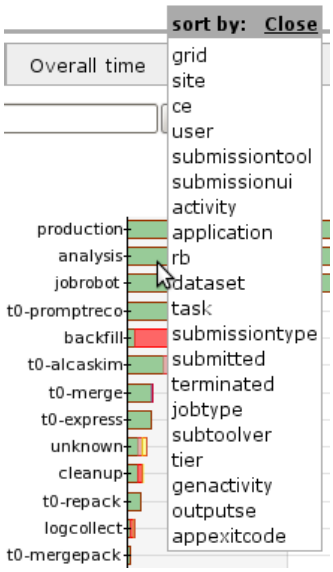
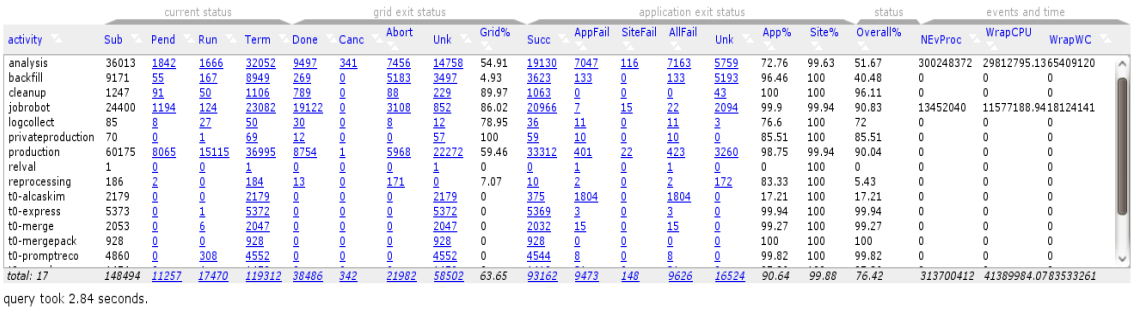


Figure 6.7: Exploring further down on the available information.

The table with all the available numerical data can be seen in the lower part of the User Interface as illustrated in Figure 6.8. The table is categorised by the current status, the grid exit status, the application exit status, the overall status and the number of events processed and the CPU and Job Wrapper time.



	current status				grid exit status					application exit status					status			events and time		
activity	Sub	Pend	Run	Term	Done	Canc	Abort	Unk	Grid%	Succ	AppFail	SiteFail	AllFail	Unk	App%	Site%	Overall%	NEvProc	WrapCPU	WrapWC
analysis	36013	1842	1666	32052	9497	341	7456	14758	54.91	19130	7047	116	7163	5759	72.76	99.63	51.67	300248372	29812795.1	365409120
backfill	9171	55	167	8949	269	0	5183	3497	4.93	3623	133	0	133	5193	96.46	100	40.48	0	0	0
cleanup	1247	91	50	1106	789	0	88	229	89.97	1063	0	0	0	43	100	100	96.11	0	0	0
jobrobot	24400	1194	124	23082	19122	0	3108	852	86.02	20966	7	15	22	2094	99.9	99.94	90.83	13452040	11577188.9	418124141
logcollect	85	8	27	50	30	0	8	12	78.95	36	11	0	11	3	76.6	100	72	0	0	0
privateproduction	70	0	1	69	12	0	0	57	100	59	10	0	10	0	85.51	100	85.51	0	0	0
production	60175	8065	15115	36995	8754	1	5968	22272	59.46	33312	401	22	423	3260	98.75	99.94	90.04	0	0	0
relval	1	0	0	1	0	0	0	1	0	0	1	0	1	0	0	100	0	0	0	0
reprocessing	186	2	0	184	13	0	171	0	7.07	10	2	0	2	172	83.33	100	5.43	0	0	0
t0-alcaskim	2179	0	0	2179	0	0	0	2179	0	375	1804	0	1804	0	17.21	100	17.21	0	0	0
t0-express	5373	0	1	5372	0	0	0	5372	0	5369	3	0	3	0	99.94	100	99.94	0	0	0
t0-merge	2053	0	6	2047	0	0	0	2047	0	2032	15	0	15	0	99.27	100	99.27	0	0	0
t0-mergepack	928	0	0	928	0	0	0	928	0	928	0	0	0	0	100	100	100	0	0	0
t0-promptreco	4860	0	308	4552	0	0	0	4552	0	4544	8	0	8	0	99.82	100	99.82	0	0	0
total: 17	*148494*	*11257*	*17470*	*119312*	*38486*	*342*	*21982*	*58502*	*63.65*	*93162*	*9473*	*148*	*9626*	*16524*	*90.64*	*99.88*	*76.42*	*313700412*	*41389984.0*	*783533261*

query took 2.84 seconds.

Figure 6.8: The lower part of the User Interface.

Bars are sorted by the number of jobs in a given category. Since labels of every category can be rather long, it is difficult to find a given item in the table. The items in the table by default are sorted in the alphabetic order but by clicking on the table header of any selected column, the user can sort the items in the table by a value in a corresponding column.

The table also offers success rate calculation as illustrated in Figure 6.9. The formula to calculate the success rate follows:

- Grid Success Rate (Grid%) = Done / (Done + Abort)
- Application Success Rate (App%) = Success / (Success + Fail)
- Overall Success Rate (Overall%) = (Success- (Success & Abort)) / (Terminated-(GridUnknown & AppUnknown))
- Site Success Rate (Site%) = 1 - ((SiteFailed + GridAborted) / (Terminated-(GridUnknown & AppUnknown))

where

- Done = reported as "Grid success" by the Grid information services.
- Abort = reported as "Grid aborted" jobs by the Grid information services.
- Success = application ran successfully.
- Fail = application failed.
- Terminated = reported as terminated (success or failure) by any of the information sources (grid or application).

Grid%	App%	Site%	Overall%
93.71	96.08	99.97	78.96
100	100	100	100
90.4	40.77	100	40.67
100	100	100	100
100	100	100	100
64.57	92.53	100	76.37
97.3	52.17	100	51.43
8.99	93.79	100	11.22
99.54	5.36	100	5.33
83.12	93.15	100	88.31
81.82	90.91	100	83.87
7.49	82.61	99.9	69.84
100	100	100	100
78.05	59.79	91.67	56.86
61.21	*73.08*	*99.65*	*59.59*

Figure 6.9: Success Rate Calculation.

The user can also retrieve the result of the table in the XML format by using the following command:

```
$ curl -H 'Accept: text/xml' http://dashb-cms-job.cern.ch/dashboard/request.py/jobsummary-plot-or-table > /tmp/action.xml
```

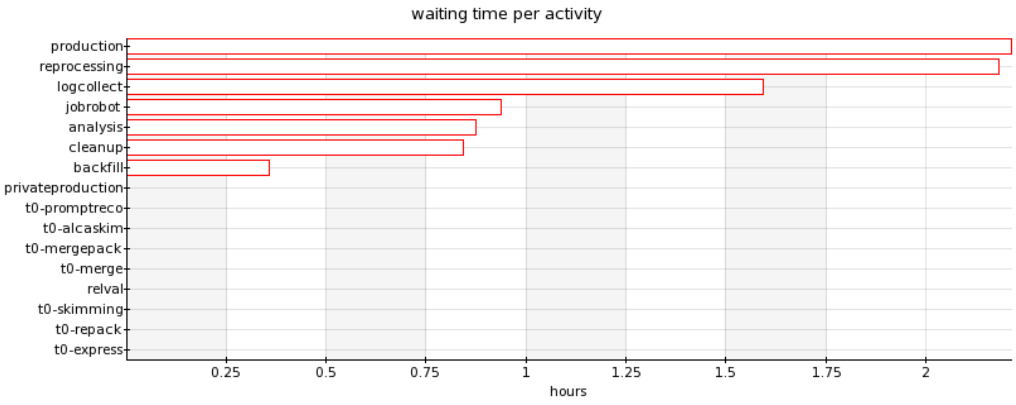
Listing 6.2: Retrieving the result in the XML format.

The XML output will be a bit hard to read because there is no newline break. The output file can be reformatted by using the 'xmllint' command:

```
$ xmllint --format /tmp/action.xml
```

Listing 6.3: Reformatting the XML output.

By clicking on any consumed time button, a new window appears with a graphical plot and a table. The Waiting Time information can be seen in Figure 6.10. This functionality offers a per job average waiting time and it is calculated by subtracting the “Started_Running time” with the “Submission time” timestamps.

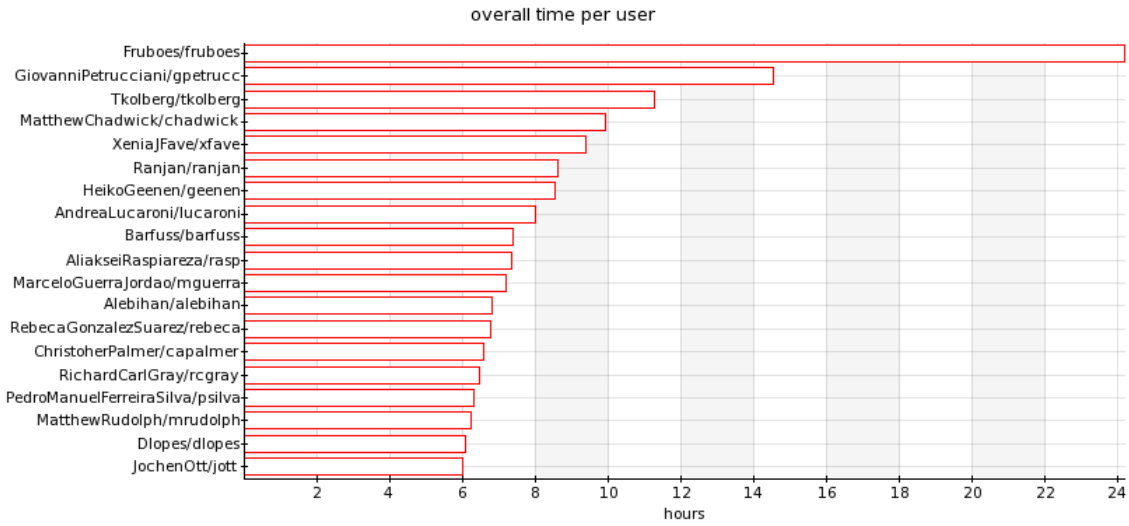


activity	
production	02:12:55
reprocessing	02:10:52
logcollect	01:35:38
jobrobot	00:56:15
analysis	00:52:33
cleanup	00:50:38
backfill	00:21:25
privateproduction	00:00:00
t0-promptreco	00:00:00
t0-alcaskim	00:00:00
t0-mergepack	00:00:00
t0-merge	00:00:00
relval	00:00:00
t0-skimming	00:00:00
t0-repack	00:00:00
t0-express	00:00:00
average	00:33:46

Figure 6.10: Waiting Time Per Activity.

The Overall Time information can be seen in Figure 6.11. This functionality offers per job average overall time and it is calculated by subtracting the “Finished time” with the “Submission time” timestamps. The timestamps are reported by the jobs themselves

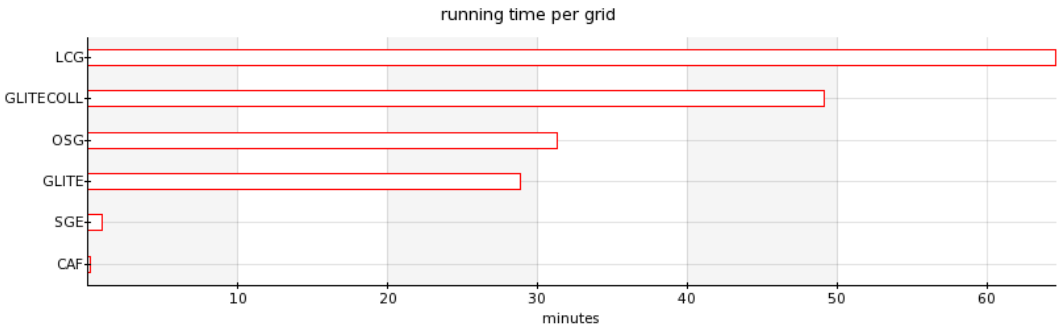
and in case of a job resubmission, only the latest attempt is considered.



user	
Fruboes/fruboes	1d 00:12:02
GiovanniPetrucciani/gpetrucc	14:33:19
Tkolberg/tkolberg	11:16:03
MatthewChadwick/chadwick	09:56:47
XeniaJFave/xfave	09:23:54
Ranjan/ranjan	08:36:21
HeikoGeenen/geenen	08:32:45
AndreaLucaroni/lucaroni	08:00:59
Barfuss/barfuss	07:23:44
AliakseiRaspiareza/rasp	07:21:19
MarceloGuerraJordao/mguerra	07:12:00
Alebihan/alebihan	06:48:04
RebecaGonzalezSuarez/rebeca	06:46:21

Figure 6.11: Overall Time Per User for the Analysis Activity.

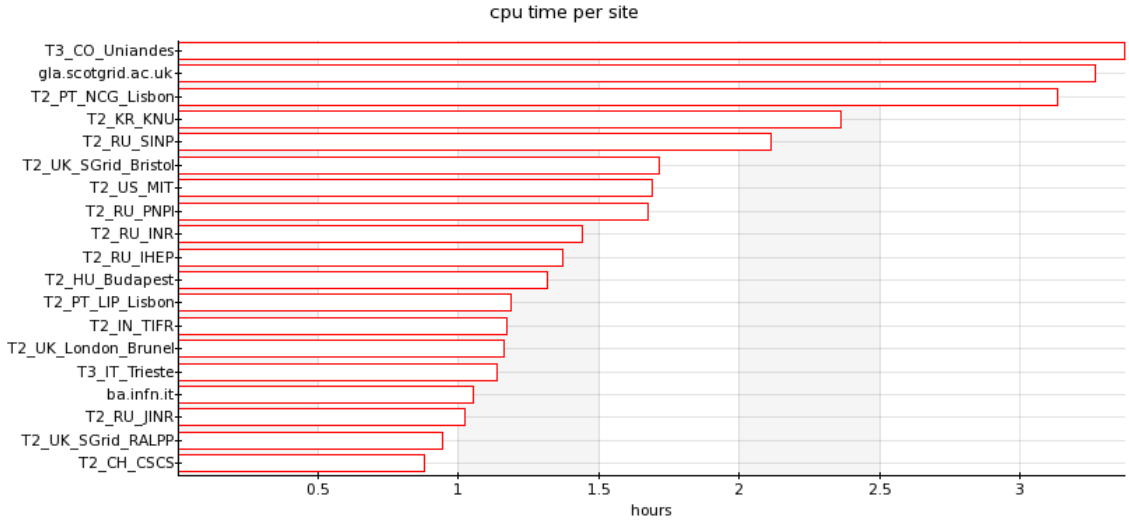
The Running Time information can be seen in Figure 6.12. This functionality offers per job average running time and it is calculated by subtracting the “Finished time” with the “Started_Running time” timestamps. The timestamps are reported by the jobs themselves and in case of a job resubmission, only the latest attempt is considered.



grid	
LCG	01:04:36
GLITECOLL	00:49:09
OSG	00:31:18
GLITE	00:28:52
SGE	00:00:58
CAF	00:00:15
average	00:29:11

Figure 6.12: Running Time Per Grid for the Analysis Activity.

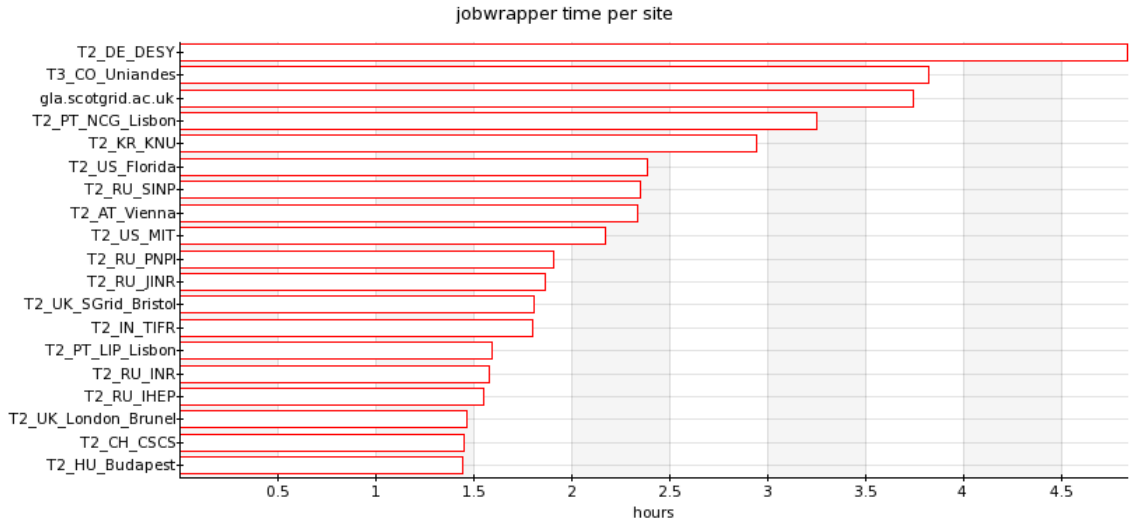
The CPU Time information can be seen in Figure 6.13. This functionality offers per job average CPU time and it is calculated by the sum of the "CPUTime" field ordered by a category, such as the site and the user. Currently, only jobs submitted using CRAB report the "CPUTime" value.



site	
T3_CO_Uniandes	03:22:35
gla.scotgrid.ac.uk	03:16:16
T2_PT_NCG_Lisbon	03:08:01
T2_KR_KNU	02:21:51
T2_RU_SINP	02:06:54
T2_UK_SGrid_Bristol	01:42:49
T2_US_MIT	01:41:24
T2_RU_PNPI	01:40:30
T2_RU_INR	01:26:32

Figure 6.13: CPU Time Per Site for the Analysis Activity.

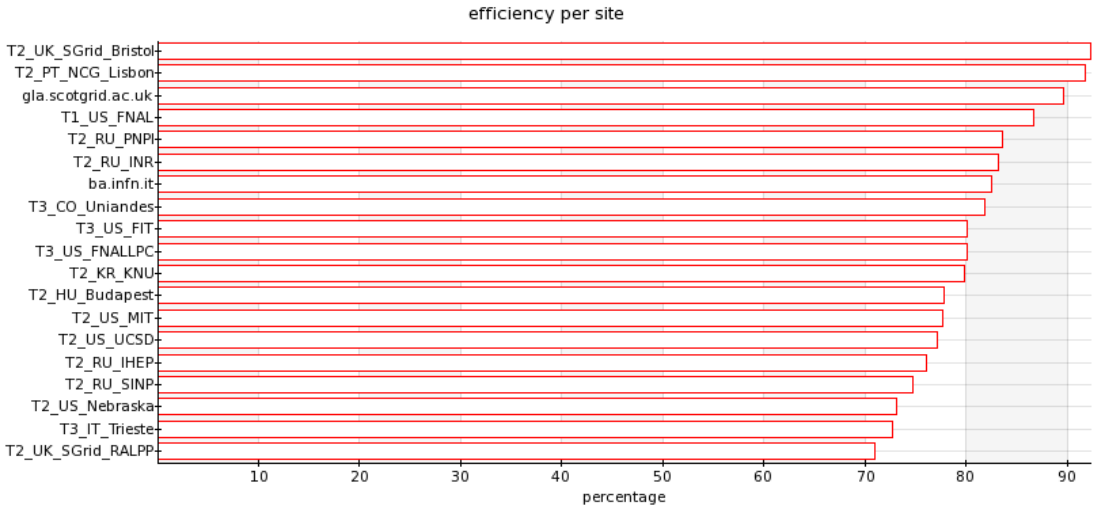
The Job Wrapper Time information can be seen in Figure 6.14. This functionality offers average per job Wall Clock time as reported by the job wrapper and it is calculated by the sum of the "WCTime" field ordered by a category, such as the site and the user. Currently, only jobs submitted using CRAB report the "WCTime" value.



site	
T2_DE_DESY	04:50:29
T3_CO_Uniandes	03:49:35
gla.scotgrid.ac.uk	03:44:50
T2_PT_NCG_Lisbon	03:15:07
T2_KR_KNU	02:56:51
T2_US_Florida	02:23:10
T2_RU_SINP	02:21:16
T2_AT_Vienna	02:20:13
T2_US_MIT	02:10:34

Figure 6.14: Job Wrapper Time Per Site for the Analysis Activity.

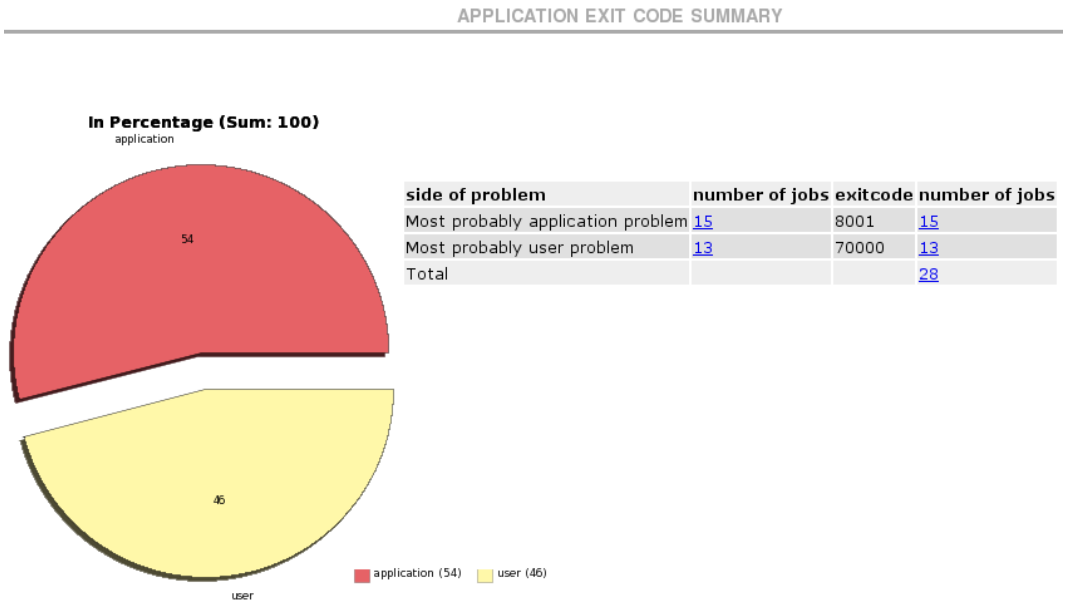
The Processing Efficiency information can be seen in Figure 6.15. This functionality offers average per job processing efficiency as reported by the job wrapper and it is calculated by dividing the "CPUTime" with the "WCTime" ordered by a category, such as the site and the user. Currently, only jobs submitted using CRAB report the "CPUTime" and "WCTime" values.



site	
T2_UK_SGrid_Bristol	92.41%
T2_PT_NCG_Lisbon	91.85%
gla.scotgrid.ac.uk	89.69%
T1_US_FNAL	86.73%
T2_RU_PNPI	83.62%
T2_RU_INR	83.22%
ba.infn.it	82.59%
T3_CO_Uniandes	81.89%
T3_US_FIT	80.16%
T3_US_FNALLPC	80.1%
T2_KR_KNU	79.8%

Figure 6.15: Processing Efficiency Per Site (in %) for the Analysis Activity.

The Exit Code Summary can be seen in Figure 6.16. This page reports error diagnostics by providing a table with numerical values and a graphical plot showing the distribution of user, application and site failures.



side of problem	number of jobs	exitcode	number of jobs
Most probably application problem	15	8001	15
Most probably user problem	13	70000	13
Total			28

Figure 6:16: The Exit Code Summary.

6.4 Experience of the CMS User Community with Job Summary

According to our web statistics [131][155], more than seventy distinct users are using Job Summary for their everyday work as illustrated in Figure 6.17. The Dashboard Applications Usage Statistics programme was developed by the author to count the daily total number of distinct users using a selected number of CMS Dashboard applications.

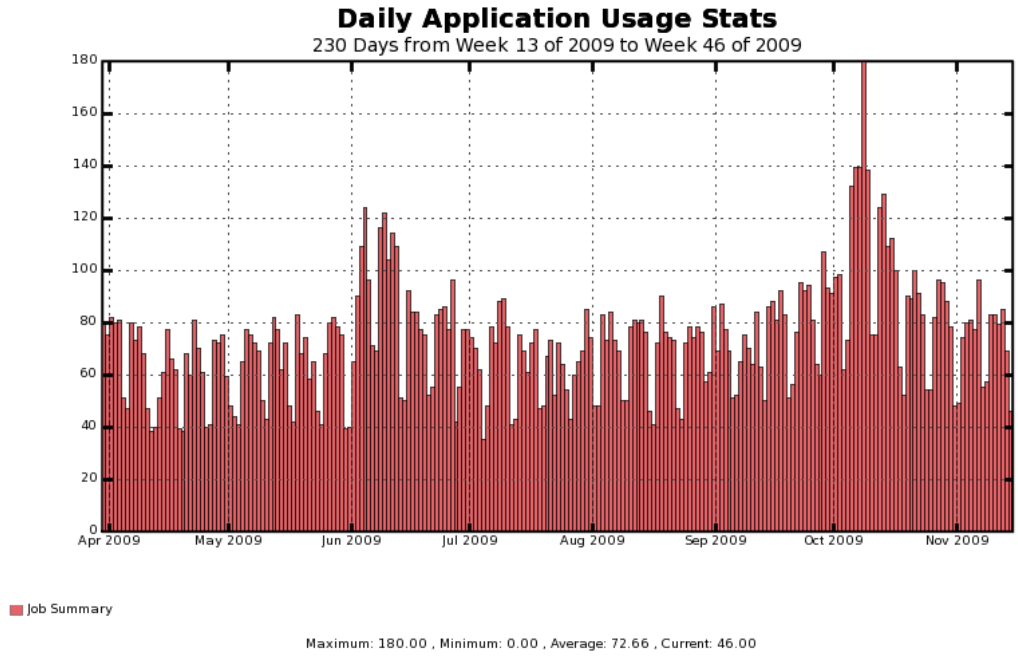


Figure 6.17: Daily Usage Statistics.

In order to count the distinct daily users, the daily access_log file of the apache http web server was used. The following bash script commands were used in a python programme to determine the date of the month and the total number of distinct daily users using some selected applications according to the total number of unique visitor IPs.

```
# Command to get the date of the month:
getDate = "zgrep +0 /var/log/httpd/access_log.1.gz | awk '{print $4}'| uniq | head -n 1| cut -c 2-13"
# Job Summary usage:
JobSum = "zcat /var/log/httpd/access_log.1.gz | grep jobsummary | awk '{print $1}' | sort | uniq |wc -l"
```

Listing 6.4: Unix bash script to determine the total number of distinct daily users.

The “JobSum” bash command counts the total number of distinct users using the application. The following cron command was scheduled to run the programme daily at 06:00am for the updating of the statistics.

```
0 6 * * * python /usr/share/dashboard-stats/dashb_stats.py 2>&1 >> /var/log/script_output.log
```

Listing 6.5: Unix Cron job scheduled to update the statistics daily.

The Graphtool library was used to create the graphical plot of the programme. The daily statistics plot is available in [155].

6.5 Summary

Currently a big variety of monitoring tools on the CMS Virtual Organisation provide job monitoring functionality. Most of them are middleware-specific and are used in the scope of a single middleware. CMS Dashboard Job Summary provides monitoring functionality regardless of the job submission method or the middleware platform offering a complete and detailed view of the Grid.

The CMS Dashboard Job Summary was the first monitoring application developed in the Dashboard project. The motivation for this development, started at the summer of 2005, was to show whether the Grid is operational, because at that period of time people were rather pessimistic about the Grid, and to show what is the status of the job processing in real-time, detect any problems or inefficiencies, not necessarily with the site, but for example with a particular dataset, or particular instance of RB, or particular application version. This is the reason why the application provides such a wide flexibility to the users; a user can sort information by any of the job / task attributes recorded in the CMS Dashboard database.

The application offers an appropriate visualisation of the job processing data, providing navigation from a global to a detailed view by taking into account the requirements of the different categories of the users.

CHAPTER 7.

CONCLUSION

The design of a parallel and distributed computing system is a very complicated task. It requires a detailed understanding of the design issues and of the theoretical and practical aspects of their solutions. A framework capable of analysing the simulation data produced by the commercial Legion Studio pedestrian simulation software has been developed. The programme has been implemented as a multi-threaded and as a prototype distributed system written in C++ with calls to the MPI library. Benchmarking the system on a dual-core PC and on a commodity cluster of high performance PCs demonstrated the system's increase in performance compared to the original single-threaded analyser. We presented a performance increase for the multi-threaded version ranging between 35% to 65.5% compared to the original single-threaded Legion Analyser on a dual core 2GHz system. The performance of the distributed prototype version of the programme scales well as the number of the processors is increased; with one Slave processor the prototype system is able to analyse 56500 simulated pedestrians in 20.17 seconds, whereas with six Slave processors the prototype system analyses 56500 simulated pedestrians in just 3.8 seconds.

Distributed Computing covers the area formerly known as Meta-computing and is the pre-cursor to the Grid. The Grid is typically used to solve problems that would traditionally have run on a single High Performance Computer, but due to memory, storage and/or computational demands it is forced to execute across multiple resources.

The mission of the Worldwide LHC Computing Grid (WLCG) project is to build and maintain a data storage and analysis infrastructure for the entire High Energy Physics (HEP) community that will use the LHC. The WLCG combines the computing resources of more than 170 computing centres in 34 countries, aiming to harness the

power of more than 100,000 CPUs to process, analyse and store data produced from the LHC. These data must be available to all the participating scientists, regardless of their physical location in order to sift through data, looking for new particles that can provid/e clues to the origins of our universe. The WLCG anticipates running between 500,000 to 1,000,000 tasks per day and this number will increase as time goes on and as computing resources and new technologies become ever more available across the world.

The distributed analysis on the WLCG infrastructure is currently one of the main challenges of the LHC computing. Reliable monitoring is an aspect of particular importance; it is a vital factor for the overall improvement of the quality of the WLCG infrastructure. Transparent access to the LHC data has to be provided for more than five thousand scientists all over the world. Users who run analysis jobs on the Grid do not necessarily have expertise in Grid computing.

The CMS Virtual Organisation (VO) uses various fully distributed job submission methods and execution backends. The CMS jobs are processed on several middleware platforms such as the gLite, the ARC and the OSG. Up to 200,000 CMS jobs are submitted daily to the Worldwide LHC Computing Grid (WLCG) infrastructure and this number is steadily growing. These mentioned factors increase the complexity of the monitoring of the user analysis activities within the CMS VO. Currently, 100-150 distinct CMS users submit their analysis jobs to the WLCG daily. Simple, user-friendly and reliable monitoring of the analysis jobs is one of the key components of the operations of the distributed analysis.

There has been a substantial progress in the development of applications for monitoring the user analysis activities during the year of 2009. This work has been very critical, since it contributes to the overall success of the LHC offline computing. The behaviour of the analysis jobs is particularly difficult to predict, as it is a chaotic activity carried out by users who do not have to be necessarily experienced in using the Grid and locating problems themselves.

The scientists must be able to monitor the execution status, application and grid-level

messages of their tasks that may run at any site on the distributed WLCG infrastructure. The existing CMS monitoring systems provide this type of information but they are coupled to a specific middleware and are not focused on the user's perspective. The CMS Dashboard Task Monitoring application addresses this gap by collecting and exposing a user-centric set of information to the user regarding submitted tasks. It provides a clear and precise view of the status of the task including job distribution by sites and over time, reason of failure and advanced graphical plots giving a more usable and attractive interface to the analysis and the production user. The development was user-driven with physicists invited to test the prototype in order to assemble further requirements and identify weaknesses with the application.

The CMS Dashboard Task Monitoring has become the most popular monitoring tool among the CMS community; more than a hundred distinct analysis users are using it for their everyday work. Close collaboration with several CMS users resulted in the tool being focused on their exact monitoring needs.

The goal of the second monitoring application developed by the author, the CMS Dashboard Job Summary, is to follow the job processing of the CMS experiment on the distributed infrastructure. The entry point of the application is the number of the jobs submitted or terminated in a chosen time period categorised by their activity such as the analysis, the production and the job robot (testing) jobs. The CMS Dashboard Job Summary application allows the possibility to explore further on the available information, expanding the set of jobs by various relevant properties such as the execution site, the grid gateway, the user, the completion status, the grid workload management host, the activity type and the used dataset, until all details stored in the Dashboard database regarding a chosen (set of) job(s) can be accessed. The application offers success and failure rates according to the grid/site/application, information on used wall clock and cpu time consumed by the jobs and the average processing efficiency of the jobs.

The CMS Dashboard Job Summary application provides monitoring functionality regardless of the job submission method or the middleware platform offering a complete and detailed view of the Grid. The application provides a wide flexibility to the users; a

user can sort information by any of the job / task attributes recorded in the CMS Dashboard database. It offers an appropriate visualisation of the job processing data, providing navigation from a global to a detailed view and taking into account the requirements of the different categories of the users.

Overall the CMS Dashboard Task Monitoring and Job Summary applications have provided a robust, reliable and useful monitoring service to the CMS community over the last two years as a result of a close collaboration with several CMS users.

ACRONYMS

Abbreviation	Full Notation
ALICE	A Large Ion Collider Experiment
AC	Attribute Certificate
ARC	Advanced Resource Connector
ATLAS	A Toroidal LHC Apparatus
BDII	Berkeley Database Information Index
BLAH	Batch Local ASCII Helper
CE	Computing Element
ClassAd	Classified Advertisement
CLI	Command Line Interface
CMS	Compact Muon Solenoid
CRAB	CMS Remote Analysis Builder
CREAM	Computing Resource Execution And Management
CSV	Comma Separated Values
DAO	Data Access Object
DBS	Dataset Bookkeeping Service
DMS	Data Management System
DSM	Distributed Shared Memory
DN	Distinguished Name
DNS	Domain Name System
EDC	Electronic Digital Computer
EDG	European DataGrid
EDVAC	Electronic Discrete Variable Automatic Computer
EGEE	Enabling Grids for E-SciencE
ENIAC	Electrical Numerical Integrator and Computer
FTP	File Transfer Protocol
FTS	File Transfer Service
GACL	Grid Access Control List

Abbreviation	Full Notation
GASS	Global Access to Secondary Storage
GGF	Global Grid Forum
GIIS	Grid Index Information Service
GLUE	Grid Laboratory Uniform Environment
GRAM	Globus Resource Allocation and Management
GRIS	Grid Resource Information Service
GSI	Globus Security Infrastructure
GT	Globus Toolkit
GWT	Google Web Toolkit
HEP	High Energy Physics
HPC	High Performance Computing
HTC	High-Throughput Computing
HTML	Hypertext Mark-up Language
I-WAY	Information Wide Area Year
IC	Integrated Circuit
ICANN	Internet Corporation for Assigned Names and Numbers
ICRTM	Imperial College Real Time Monitor
IETF	Internet Engineering Task Force
IIS	Integrated Information Services
IP	Intellectual Property
IP	Internet Protocol
IPs	Information Providers
IS	Information System
ISM	Information Super Market
JSON	JavaScript Object Notation
LB	Logging and Bookkeeping
LFN	Logical File Names
LHC	Large Hadron Collider
LHCb	LHC-beauty
LHCf	LHC-forward
LIS	Language Independent Specifications
MC	Monte-Carlo

MDS	Monitoring and Discovery Service
Abbreviation	**Full Notation**
MIMD	Multiple Instruction Stream, Multiple Data Stream
MISD	Multiple Instruction Stream, Single Data Stream
MPI	Message Passing Interface
MSG	Messaging System for the Grid
MVC	Model-View-Controller
OASIS	Organisation for the Advancement of Structured Information Standards
OGF	Open Grid Forum
OGSA	Open Grid Services Architecture
OGSI	Open Grid Services Infrastructure
OpenMP	Open Multi Processing
OS	Operating System
OSG	Open Science Grid
PFN	Physical File Names
PhEDEx	Physics Experiment Data Export
PKI	Public Key Infrastructure
ProdAgent	Production Agent
PVM	Parallel Virtual Machine
QoS	Quality of Service
R-GMA	Relational Grid Monitoring Architecture
RB	Resource Broker
RDBMS	Relational Database Management System
RLS	Replica Location System
SAM	Service Availability Monitoring
SE	Storage Element
SIMD	Single Instruction Stream, Multiple Data Stream
SISD	Single Instruction Stream, Single Data Stream
SOA	Service Orientated Architecture
SOAP	Simple Object Access Protocol
SRM	Storage Resource Management
TOTEM	Total Elastic and Diffractive Cross Section Measurement

UI	User Interface
Abbreviation	**Full Notation**
VDT	Virtual Data Toolkit
VO	Virtual Organisation
VOMS	Virtual Organisation Membership Service
W3C	World Wide Web Consortium
WLCG	Worldwide LHC Computing Grid
WMS	Workload Management System
WN	Worker Node
WSDL	Web Services Description Language
WSRF	Web Services Resource Framework
WWW	World Wide Web
XSL	Extensible Stylesheet Language
XML	eXtensible Markup Language

APPENDIX A. TASK MONITORING

A.1 Use Cases

Use Case	Submitted Tasks
Description	The User should be able to get a list with all of his submitted tasks within a specified time period.
Actors	Physicist, Dashboard Data Repository.
Assumptions	1. The User has a grid certificate. 2. The User is a member of the CMS VO. 3. The User has submitted jobs to the Grid within one month.
Steps	1. The User chooses his/her identity in the "Select a User" field. 2. The User selects the time window to define the tasks submitted during a given time range. 3. The Results are obtained from the Dashboard Data Repository. 4. The User should get at the screen the list of all of his/her tasks submitted over the chosen time range.

Use Case	Task Meta-Information
Description	The User should be able to get a task's meta-information such as the task's creation time, the submission tool, the target Computing Element (CE) and the Input Collection used.
Actors	Physicist, Dashboard Data Repository.
Assumptions	1. The User has a grid certificate. 2. The User is a member of the CMS VO. 3. The User has submitted jobs to the Grid within one month.
Steps	1. The User chooses his/her identity in the "Select a User" field. 2. The User selects the time window to define the tasks submitted during a given time range. 3. The Results are obtained from the Dashboard Data Repository. 4. The User should get at the screen the list of all of his/her tasks submitted over the chosen time range. 5. The User selects a task and clicks on the 'i' icon to view the task's meta-information. 6. The Results are obtained from the Dashboard Data Repository and presented on the screen.

Use Case	Detailed Jobs Information
Description	The User should be able to view a detailed jobs information for a selected task.
Actors	Physicist, Dashboard Data Repository.
Assumptions	1. The User has a grid certificate. 2. The User is a member of the CMS VO. 3. The User has submitted jobs to the Grid within one month.
Steps	1. The User chooses his/her identity in the "Select a User" field. 2. The User selects the time window to define the tasks submitted during a given time range. 3. The Results are obtained from the Dashboard Data Repository. 4. The User should get at the screen the list of all of his/her tasks submitted over the chosen time range. 5. The User selects a task and clicks on the number of jobs corresponding to a given status. 6. The Results are obtained from the Dashboard Data Repository. 7. The application provides a detailed information of all the jobs of a selected category.

Use Case	Resubmission History
Description	The User should be able to view a detailed resubmission history of a selected job.
Actors	Physicist, Dashboard Data Repository.
Assumptions	1. The User has a grid certificate. 2. The User is a member of the CMS VO. 3. The User has submitted jobs to the Grid within one month.
Steps	1. The User chooses his/her identity in the "Select a User" field. 2. The User selects the time window to define the tasks submitted during a given time range. 3. The Results are obtained from the Dashboard Data Repository. 4. The User should get at the screen the list of all of his/her tasks submitted over the chosen time range. 5. The User selects a task and clicks on the number of jobs corresponding to a given status. 6. The Results are obtained from the Dashboard Data Repository and presented on the screen. 7. The application provides a detailed information of all the jobs of a selected category. 8. The User selects a specific job and clicks on the 'Resubmissions' 9. The Results are obtained from the Dashboard Data Repository and presented on the screen.

Use Case	Error Diagnostics
Description	The User should be able to access advanced error diagnostics to understand the status of his/her task.
Actors	Physicist, Dashboard Data Repository.
Assumptions	1. The User has a grid certificate. 2. The User is a member of the CMS VO. 3. The User has submitted jobs to the Grid within one month.
Steps	1. The User chooses his/her identity in the "Select a User" field. 2. The User selects the time window to define the tasks submitted during a given time range. 3. The Results are obtained from the Dashboard Data Repository. 4. The User should get at the screen the list of all of his/her tasks submitted over the chosen time range. 5. The User selects a task and clicks on the failed jobs. 6. The Results are obtained from the Dashboard Data Repository and presented on the screen. 7. The application provides a detailed information of all the failed jobs of the task including any error diagnostics, reasons of failure and exit code numbers.

Use Case	Consumed Time Information
Description	The User should be able to view the consumed time information for a specific task.
Actors	Physicist, Dashboard Data Repository.
Assumptions	1. The User has a grid certificate. 2. The User is a member of the CMS VO. 3. The User has submitted jobs to the Grid within one month.
Steps	1. The User chooses his/her identity in the "Select a User" field. 2. The User selects the time window to define the tasks submitted during a given time range. 3. The Results are obtained from the Dashboard Data Repository. 4. The User should get at the screen the list of all of his/her tasks submitted over the chosen time range. 5. The User selects a task and clicks on the consumed time information. 6. The Results are obtained from the Dashboard Data Repository and presented on the screen.

Use Case	Graphical Plots
Description	The User should be able to access a wide-variety of advanced graphical plots to visually assist him/her.
Actors	Physicist, Dashboard Data Repository.
Assumptions	1. The User has a grid certificate. 2. The User is a member of the CMS VO. 3. The User has submitted jobs to the Grid within one month.
Steps	1. The User chooses his/her identity in the "Select a User" field. 2. The User selects the time window to define the tasks submitted during a given time range. 3. The Results are obtained from the Dashboard Data Repository. 4. The User should get at the screen the list of all of his/her tasks submitted over the chosen time range. 5. The User clicks on the 'Graphical Plots' menu and selects a required plot. 6. The plot is generated and presented on the screen.

A.2 Graphtool Patches

Patched File: graph.py

Revision 1.14

Mon Jan 5 13:41:02 2009 UTC

Changes since 1.13: +15 -3 lines

Description: Applied patch from Edward Karavakis from the ARDA-Dashboard team.

Available at: http://cmssw.cvs.cern.ch/cgi-bin/cmssw.cgi/COMP/WEBTOOLS/Tools/GraphTool/src/graphtool/graphs/graph.py?revision=1.14&view=markup

Diff to the previous version

#	revision 1.13, Wed Oct 8 17:24:37 2008 UTC **Line 384**	revision 1.14, Mon Jan 5 13:41:02 2009 UTC **Line 384**
384	match an application's specific color scheme.	match an application's specific color scheme.
385	"""	"""
386	size_labels = len(labels)	size_labels = len(labels)
387		self.color_override = self.metadata.get('color_override', {})
388		try:
389		if self.color_override == {}:
390		raise Exception('going to the default')
391		colours = self.color_override
392		size_colors = len (colours)
393		retval = []
394		for label in labels:
395		mycolour = colours[label]
396		retval.append(mycolour)
397		except:
398	hex_colors = self.hex_colors	hex_colors = self.hex_colors
399	size_colors = len(hex_colors)	size_colors = len(hex_colors)
400	retval = [hex_colors[i % size_colors] for i in range(size_labels)]	retval = [hex_colors[i % size_colors] for i in range(size_labels)]
401		
402	retval.reverse()	retval.reverse()
403	return retval	return retval
404		

Patched File: common_graphs.py

Revision 1.16

Mon Jan 5 13:41:02 2009 UTC

Changes since 1.15: +24 -17 lines

Description: Applied patch from Edward Karavakis from the ARDA-Dashboard team.

Available at:
http://cmssw.cvs.cern.ch/cgi-bin/cmssw.cgi/COMP/WEBTOOLS/Tools/GraphTool/src/graphtool/graphs/common_graphs.py?revision=1.16&view=markup

Diff to the previous version.

	revision 1.15, Wed Oct 8 17:24 2008 UTC	revision 1.16, Mon Jan 5 13:41 2009 UTC
#	**Line 1313**	**Line 1313**
1313	texts = []	texts = []
1314	slices = []	slices = []
1315	autotexts = []	autotexts = []
1316		color_override = self.color_override
1317		results = self.parsed_data
1318	for frac, label, expl in zip(x,labels, explode):	for frac, label, expl in zip(x,labels, explode):
1319	x, y = center	x, y = center
1320	theta2 = theta1 + frac	theta2 = theta1 + frac
1321	thetam = 2*math.pi*0.5*(theta1+theta2)	thetam = 2*math.pi*0.5*(theta1+theta2)
1322	x += expl*math.cos(thetam)	x += expl*math.cos(thetam)
1323	y += expl*math.sin(thetam)	y += expl*math.sin(thetam)
1324		if color_override == {}:
1325	w = Wedge((x,y), radius, 360.*theta1, 360.*theta2,	w = Wedge((x,y), radius, 360.*theta1, 360.*theta2,
1326	facecolor=colors[i %len(colors)])	facecolor=colors[i%len(colors)])
1327		else:
1328		mycolour = color_override[label]
1329		w = Wedge((x,y), radius, 360.*theta1, 360.*theta2,
1330		facecolor=mycolour)
1331	slices.append(w)	slices.append(w)
1332	self.ax.add_patch(w)	self.ax.add_patch(w)
1333	w.set_label(label)	w.set_label(label)
#	**Line 1355**	**Line 1361**
1361	halign = 'center'	halign = 'center'
1362	else:	else:
1363	halign = 'left'	halign = 'left'
1364		if float(results[label]) / self.amt_sum > self.min_amount:
1365	t = self.ax.text(xt, yt, label,	t = self.ax.text(xt, yt, label,
1366	size=self.prefs['subtitle_size'],	size=self.prefs['subtitle_size'],
1367		horizontalalignment=halign,

```
        horizontalalignment=halign,
#       Line 1407                                   Line 1413
1413
1414        results = self.results                      results = self.results
1415        parsed_data = self.parsed_data              parsed_data = self.parsed_data
1416                                                    self.color_override =
                                                    self.metadata.get('color_override', {})
1417
            column_units = getattr( self,               column_units = getattr( self, 'column_units',
1418    'column_units',                             self.metadata.get('column_units','') )
        self.metadata.get('column_units','') )
1419        column_units = column_units.strip()         column_units = column_units.strip()
#       Line 1447                                   Line 1454
1454        for label in local_labels:                  for label in local_labels:
1455          orig_label = label[:label.rfind(' ')]       orig_label = label[:label.rfind(' ')]
1456          val = float(results[orig_label])            val = float(results[orig_label])
            if val / self.amt_sum >
        self.min_amount:
1457          my_labels.append( orig_label )              my_labels.append( orig_label )
            else:
              my_labels.append( "" )
1458
1459        def my_display( x ):                        def my_display( x ):
1460          if x > 100*self.min_amount:                 if x > 100*self.min_amount:
#       Line 1462                                   Line 1466
1466        explode = [.1 for i in amt]                 explode = [.1 for i in amt]
1467
1468        self.colors.reverse()                       self.colors.reverse()
1469                                                    if self.color_override == {}:
1470        self.wedges, text_labels, percent =         self.wedges, text_labels, percent = self.pie( amt,
        self.pie( amt, explode=explode,             explode=explode,
1471            labels=my_labels, shadow=True,              labels=my_labels, shadow=True,
        colors=self.colors, autopct=my_display )    colors=self.colors, autopct=my_display )
1472                                                    else:
1473                                                        self.wedges, text_labels, percent = self.pie( amt,
                                                    explode=explode,
                                                                                labels=my_labels,
1474                                                shadow=True, colors=self.color_override.values(),
                                                    autopct=my_display )
1475
1476      def get_coords( self ):                     def get_coords( self ):
1477        try:                                        try:
```

A.3 CMS Survey

Dashboard User List

FEEDBACK - 50 out of 201 replied

JasminKiefer

- Positive feedback: Liked the page layout and the clearness of information presented - no unneeded info.

Sbologne

- Already using it.

Alkaloge

- A nice surprise, as he said, to see this monitoring application live and working!

ThomasEDanielson

- It's easy to navigate and provides some useful information regarding the jobs that failed. Likes it a lot.

ChristophPaus

- He liked it, looks good. He liked the visual presentation. He will probably use it from time to time though he does get along reasonably well with crab -status etc.

DanieleBenedetti

- He said it seems to be really cool. He will play with it and in case he has any feedback he will let me now.

GavrilAdrianGiurgiu

- Thinks that the monitoring tool is great. The user is now investigating why most of the jobs are failing.

JavierFernandezMenendez

- It looks perfect. It even updates in "real time".

DanielBloch

- This is extremely nice and useful.

JeremyAndrea

- It will be indeed very useful. Will have a look and let me know if there is any feedback or any feature requests.

JoshBendavid

- Does not work because he is using a Custom executable not cmssw and configured for local condor submission.

Schiefer

- Very helpful tool to monitor the progress of his grid activities.

XinShi

- It looks great. Pleased to see the plotting section with the different plots about the jobs. Will investigate more in the near future.

Yuanchao

- He tried it and found it is quite useful that he doesn't have to run crab -status every single time.

CarstenHof

- Awesome! That's a huge improvement! Congratulations to the team!

SandroFonsecaDeSouza

- Task Monitoring is working well but he thinks that maybe the delay in the results of Jobs status between Task Monitoring and CRAB should be investigated.

RebecaGonzalezSuarez

- Found it very useful. Nothing more to say, it just works fine.

Trommers OR TanjaRommerskirchen

- Looks helpful. Once she runs into more complicated cases (failures and etc) she will give us feedback.

NikolaosRompotis

- He didn't know that there was a task monitoring tool for analysis. He finds it very useful.

SilviaMaselli

- She finds it very useful. She will let me know if she finds any anomalies.

LotteWilke

- Thinks this tool is nice, The user did not know about it before. The user thinks it is particularly nice to be able to see how many events were processed.

MalinaAureliaKirn

- That's a really excellent monitor, it has low latency and excellent plots with clear labels. She is surprised that it even supports the condor scheduler.

YuriGotra

- It's a useful tool. There was an issue with a killed task; the CRAB developers have been notified and it is now fixed.

Bdahmes

- This is a wonderful tool. Clicking through the page, all the information the user wants is present.

PratimaJindal

- It is really helpful.

Vandreev

- Very positive on it. It is very useful tool.

AndrewYork

- It looks very good. It is easy to understand and intuitive in layout. Contains all the information he would like to know.

RobertaVolpe

- Sometimes she noticed that the task monitoring is more updated than crab report

SupreetPalSingh

- This is a really nice way to monitor the jobs submitted in GRID. Keep up the good work.

PedroManuelFerreiraSilva

- Many thanks for drawing his attention to this new version of the Task Monitoring. He finds it much more complete and user friendly.

Ceggel

- She only remembered the old version as it was last summer. Compared to that experience the new version is an immense progress. It's so much faster. The layout is very well done, making it easy to find and access the information you're looking for. It's just great!

Meridian

- Quite useful and browsable, it really gives you the possibility to understand what has happened.

Demattia

- Never used the application before. Seems very useful, especially the possibility to have the failures shown by site. This will make it easier to spot problematic sites and blacklist them. Also finds the graphical representation very good.

VardanKhachatryan

- There is interesting and useful information in this site

litvin

- He really likes the application, he gets statistics faster than crab -status. He really appreciates the tool.

IvanReid

- Looks useful

LucaMartini

- He finds the application very useful. It is also more organised than before. The possibility to watch each single job to check its status from a browser is great. Task Monitoring is faster than crab -status: Task Monitoring says a job ends many minutes before he can get it because crab still says job is running.

AlekoKhukhunaishvili

- It's much better and convenient than everything else he used before.

ThomasPeiffer

- This seems to be a very nice tool. No suggestions for improvement so far.

DilsonDeJesusDamiao

- He was using Task Monitoring. He likes the tool because he can see his jobs 'online', once the crab -status takes some time to return the real situation of the job.

ChristosLazaridis

- He had no idea this existed. It is very useful indeed!

GiuseppeCodispoti

- It looks pretty nice and quite fast!!!! He will use it regularly.

OliverGutsche

- Looks nice, some problems with crab on the US analysis sites, crab was notified some time ago but it's not fixed. The issue will be fixed in the next version of crab.

LetiziaLusito

- The new version is very useful. Easy to understand. She is now using Task Monitoring more intensively.

Cardaci

- Really nice! Time range should be adjustable and to be able to select an interval

FreyaBlekman

- She killed a large part of these jobs but it wasn't shown up on dashboard. CRAB Bug #47309 - Fixed.

EfeYazgan

- Very user-friendly and very well-designed. Finds whatever the user needs without any problem.

FlorianBechtel

- Very helpful improvements indeed.

Slehti

- The user had a quick look, and it looked extremely useful. So far the user has been using crab -status, but this graphical gives him all tasks at the same time.

AdamEverett

- The tool is quite nice and very helpful.

A.4 User Manual

Usage

Choose your identity in the "Select a User" field, select the time window to define the tasks submitted during a given time range, you should get at the screen the list of all your tasks submitted over the time range you have chosen.

Adjusting the Timerange: Shows the Tasks created during the selected time range. For example: If a task was created one week ago and it is still running, you have to select the Last Week option (or a bigger time range value) to be able to view it. If you select any smaller value than Last Week, the task will not appear. The page automatically reloads and updates its records every 5 minutes. If you are using CRAB server, please be aware that only jobs which had been already submitted to the GRID or CAF are available in the task monitoring.

Navigation

Please avoid using the browser's back and forward buttons. Use the buttons provided by the application.

Graphical Plots

1. Click on the plot to zoom in.
2. Click and Drag the plot to move and re-arrange its position.
3. Click again on the plot to zoom out.

Retrieve the data in XML

For retrieving your tasks in the XML format you should use the following comand:

```
$ curl -H 'Accept: text/xml' 'http://dashb-cms-
sam.cern.ch/dashboard/request.py/taskstablexml?
&typeofrequest=A&timerange=TIMERANGE&usergridname=USERNAME' >
/tmp/action.xml
```

where USERNAME is your username and TIMERANGE can be one of the following:

```
lastDay, last2Days, last3Days, lastWeek, last2Weeks, lastMonth
```

For retrieving the detailed list of jobs for a specific task in the XML format you should use the following comand:

```
$ curl -H 'Accept: text/xml'
'http://dashboard02.cern.ch/dashboard/request.py/taskjobsxml?
&timerange=TIMERANGEs&what=all&taskmonid=TASKNAME' > /tmp/action.xml
```

where TASKNAME is the name of the task, TIMERANGE can be one of the above options and 'what' can be one of the following options:
'`all`' for all the jobs, '`f`' for the failed ones, '`r`' for the running ones, '`p`' for the pending ones, '`s`' for the successful ones and '`u`' for the unknown jobs.

The XML output of the dashboard is a bit hard to read because there is no newline. You can use xmllint to reformat the output:

```
$ xmllint --format /tmp/action.xml
```

A.5 Graphical Overview Plot

The following code is from the GraphicalOverviewPyPlot python class that creates a simple graphical overview plot.

```
"""
Implementation of GraphicalOverviewPyPlot
"""
import os, time
from mod_python import util
from dashboard.common import log as logging
from dashboard.common import xml
from dashboard.common.Config import Config
from dashboard.http.View import View
from graphtool.graphs.graph import Grapher
from graphtool.graphs.common_graphs import PieGraph
from dashboard.common.InternalException import InternalException
from dashboard.http.actions.job.argument_filtering import filter_job_arguments

class GraphicalOverviewPyPlot(View):
    """
    @author: ekaravak - edward.karavakis@cern.ch
    @version: $Id: GraphicalOverviewPyPlot.py,v 1.1.2.7 2009/01/29 19:56:33 ekaravak
     """
      _logger =
logging.getLogger("dashboard.http.views.job.task.GraphicalOverviewPyPlot")
      def __init__(self, attributes):
            super(GraphicalOverviewPyPlot, self).__init__(attributes)
      def generate(self, actionCtx, request):
            # get the summaries
            summaries = actionCtx.get("summaries")
            parameters = filter_job_arguments(request.args)
            data = {'Pending': summaries[0][0]['PENDING'], 'Running':summaries[0][0]
['RUNNING'],
                  'Successful': summaries[0][0]['SUCCESS'], 'Failed': summaries[0][0]
['FAILED'],
                  'Unknown': summaries[0][0]['TERMINATED']}
            metadata = {'title': 'Graphical Overview', 'color_override':{'Pending':'#FEFE98',
'Running':'#CCCCFE', 'Successful':'#98CB98', 'Failed':'#FF0000',  'Unknown': '#DDFEAA'},
'title_size':10, 'text_size':8}
            pieJobs = PieGraph()
            file = request
            # Return the plot to the request
            self._logger.debug('Returning the plot to the request')
            pieJobs(data, file, metadata)
```

A.6 SQL Queries

In this section, the most important SQL queries of the application will be presented. The first SQL query fetches the list of all the available users that have submitted jobs during the period of a month.

```
select distinct users."GridName" from users, task where users."UserId" =
        task."UserId" and task."TaskCreatedTimeStamp" > sysdate – 31  and
        task."TaskTypeId" in (select "TaskTypeId" from task_type where "Type"
        in ('analysis', 'JobRobot', 'AnaStep09')) order by users."GridName"
```

The second SQL query fetches all the submitted tasks of the user during a selected period of time.

```
SELECT "TaskId" as taskid, "TaskMonitorId" as taskmonid, "InputCollection"
        as inputcollection, "TaskCreatedTimeStamp",
        MAX(decode(status,'P', jobsInState, 0)) AS pending,
        MAX(decode(status,'R', jobsInState, 0)) AS running,
        MAX(decode(status, 'S', jobsInState, 0)) AS success,
        MAX(decode(status, 'F', jobsInState, 0)) AS failed,
        MAX(decode(status,'U', jobsInState, 0)) AS terminated,
        sum(jobsInState) as numofjobs FROM (
       SELECT "TaskId", "TaskMonitorId", "InputCollection",
       "TaskCreatedTimeStamp", status, COUNT(status) AS jobsInState
       FROM (
      SELECT JS."TaskId", TK."TaskMonitorId", "InputCollection",
      "TaskCreatedTimeStamp", JS.status FROM (
              SELECT  "TaskId", "TaskMonitorId", "InputCollection",
              "TaskCreatedTimeStamp" FROM task T, input_collection
               WHERE T."TaskCreatedTimeStamp" > :startDate AND
              T."TaskTypeId" in (select "TaskTypeId" from task_type where
              "Type" in ('analysis', 'JobRobot', 'AnaStep09'))
             AND T."UserId" IN (SELECT "UserId" FROM users WHERE
```

```
                "GridName" =                    :gridName)
              AND "INPUT_COLLECTION"."InputCollectionId" =
               T."InputCollectionId"
               ) TK JOIN ( SELECT  "TaskId", "EventRange", "JobId",
               "DboardFirstInfoTimeStamp",
        job_status("DboardJobEndId","DboardStatusId","DboardGridEndId")
        AS status, ROW_NUMBER() OVER (PARTITION BY "TaskId",
        "EventRange" ORDER BY "DboardFirstInfoTimeStamp" DESC) AS n
          FROM job WHERE job."NextJobId" is null AND job."TaskId" IN (
          SELECT  "TaskId" FROM task T
          WHERE T."TaskCreatedTimeStamp" > :startDate AND
        T."TaskTypeId" in (select          "TaskTypeId" from task_type
        where "Type" in ('analysis', 'JobRobot', 'AnaStep09')) AND T."UserId" IN
(SELECT "UserId" FROM users WHERE "GridName" = :gridName)
          )) JS ON (JS."TaskId" = TK."TaskId")  WHERE JS.n <= 1) GROUP BY
        "TaskId", "TaskMonitorId", "InputCollection", `
        "TaskCreatedTimeStamp", status) GROUP BY "TaskId",
        "TaskMonitorId", "InputCollection", "TaskCreatedTimeStamp" ORDER
        BY "TaskCreatedTimeStamp"
```

The third query fetches all the jobs of a selected task.

```
SELECT "TaskJobId", "EventRange", "Site", "started", "finished",
        "submitted", "resubmissions",     "SchedulerJobId", status, "GridEndId",
        "GridEndReason", "JobExecExitCode", "AppGenericStatusReasonValue"
FROM (
  SELECT  "TaskJobId", "EventRange", site."VOName" as "Site",
        job_status("DboardJobEndId","DboardStatusId","DboardGridEndId"
        ) AS status, "SubmittedTimeStamp" as "submitted",
        "StartedRunningTimeStamp" as "started",
        "FinishedTimeStamp" as "finished", job_resubmission("TaskJobId") as
        "resubmissions", "SchedulerJobId", ROW_NUMBER()       OVER
        (PARTITION BY "TaskId", "EventRange"                ORDER BY
```

```
        "DboardFirstInfoTimeStamp" DESC) AS n,
        "DboardGridEndId", "DboardGridEndId" as "GridEndId",
        "JobExecExitCode", "AppGenericStatusReasonValue",
        generic_status_reason."GenericStatusReasonValue" as "GridEndReason"
  FROM job, long_ce, short_ce, site, generic_status_reason, grid_status_reason,
        app_generic_status_reason
  WHERE job."NextJobId" is null AND job."TaskId" =
        (  select "TaskId" from task where "TaskMonitorId" = :taskMonId)  AND
        job."LongCEId" = long_ce."LongCEId" and short_ce."ShortCEId" =
        long_ce."ShortCEId"  AND grid_status_reason."GridStatusReasonId" =
        job."GridStatusReasonId"  AND
        grid_status_reason."GenericStatusReasonId" =
        generic_status_reason."GenericStatusReasonId" AND
        app_generic_status_reason."AppGenericErrorCode" =
        nvl(job."JobExecExitCode",-1) and site."SiteId" = job."SiteId" order by
TO_NUMBER("EventRange")
```

The fourth SQL query fetches task meta-information such as the task creation time, the version of the application used, the number of events per job and the input collection data.

```
select task."TaskId", task."TaskMonitorId", task."TaskCreatedTimeStamp",
        task_type."Type" as "TaskType", submission_tool_ver."SubToolVersion",
        application."Application", application."ApplicationVersion",
        task."NEventsPerJob", appl_exec."Executable",
        input_collection."InputCollection",
        submission_tool."SubmissionTool", submission_ui."DisplayName" as
        "SubmissionUI",     "SubmissionType", "TargetCE",
        scheduler."SchedulerName" as "SchedulerName"  from task, task_type,
        task_status, submission_tool_ver, application, appl_exec, input_collection,
        submission_tool, submission_ui, scheduler
          where task."TaskMonitorId" = :taskMonId
           and task_type."TaskTypeId" = task."TaskTypeId"
```

```
        and task."DefaultSchedulerId" = scheduler."SchedulerId"
        and task_status."TaskStatusId" = task."TaskStatusId"
        and application."ApplicationId" = task."ApplicationId"
        and appl_exec."ApplExecId" = task."ApplExecId"
        and input_collection."InputCollectionId" = task."InputCollectionId"
        and submission_tool."SubmissionToolId" = task."SubmissionToolId"
        and submission_ui."SubmissionUIId" = task."SubmissionUIId"
        and submission_tool_ver."SubToolVerId" = task."SubToolVerId"
```

The fifth SQL query fetches all the resubmission history for a selected job.

```
select "JobExecExitCode" as "JobExitCode",
        app_generic_status_reason."AppGenericStatusReasonValue" as
        "JobExitReason", "DboardGridEndId" as "GridEndId",
        "GenericStatusReasonValue" as "GridEndReason",
        "VOName" as "Site","AppStatusReason", "SubmittedTimeStamp" as
        "submitted", "StartedRunningTimeStamp" as "started",
        "FinishedTimeStamp" as "finished", "EventRange", "SchedulerJobId"
        from (select "JobExecExitCode", "DboardGridEndId",
        "GenericStatusReasonValue", "VOName", "SubmittedTimeStamp",
        "StartedRunningTimeStamp",      "FinishedTimeStamp", "EventRange",
        "SchedulerJobId", replace("AppStatusReason",\'\'\'\') as
        "AppStatusReason" from job, long_ce, short_ce, site,
        generic_status_reason, grid_status_reason, app_status_reason
        where "TaskJobId" = :taskJobId  and job."LongCEId" =
        long_ce."LongCEId" and short_ce."ShortCEId" =
        long_ce."ShortCEId" and site."SiteId" = short_ce."SiteId"
        and app_status_reason."AppStatusReasonId" =
        job."JobExecExitReasonId" and
        grid_status_reason."GridStatusReasonId" = job."GridStatusReasonId"
        and grid_status_reason."GenericStatusReasonId" =
        generic_status_reason."GenericStatusReasonId") all_jobs
```

```
        left join app_generic_status_reason on
        app_generic_status_reason."AppGenericErrorCode" =
        nvl(all_jobs."JobExecExitCode", -1)  order by "submitted"
```

The final SQL query presented fetches consumed time information for a specific task. The consumed time information includes the Total CPU Time, Total Wall Clock Time, the Average CPU Time Per Event, the Average Efficiency of a task, the Average CPU Time Per Job and the Average Wall Clock Time Per Job.

```
select total_cpu, total_wc, efficiency, cpu_per_event, (total_cpu/total_jobs) as
        avgcpu, (total_wc/total_jobs) as avgwc from
        (select sum("WrapCPU") as total_cpu, sum("WrapWC") as total_wc,
        ROUND(avg("WrapCPU"/"WrapWC")*100,2) as efficiency,
        COALESCE(avg(("WrapCPU")/NULLIF("NEvProc",0)),0) as
        cpu_per_event,  count("EventRange") as total_jobs from
        task, job where task."TaskMonitorId" = :taskMonId AND
        task."TaskId" = job."TaskId"
       AND "WrapWC" > 0 AND "WrapCPU" > 0 )
```

APPENDIX B. JOB SUMMARY

B.1 Use Cases

Use Case	Users using a site
Description	The CMS Site Administrators need to monitor the usage of their site and who is using it.
Actors	Physicist, Dashboard Data Repository.
Assumptions	The CMS Site Administrator of a specific site needs to monitoring who is using the site.
Steps	1. The CMS Site Administrator enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository. 3. The CMS Site Admin selects an activity from the menu such as the analysis or the production activity. 4. The CMS Site Admin selects 'sort by site' from the menu. 5. The Results are obtained from the Dashboard Data Repository. 6. The CMS Site Admin selects the required site and selects 'sort by user'. 7. The Results are obtained from the Dashboard Data Repository and presented on the screen.

Use Case	Jobs Running
Description	The CMS Site Administrators need to monitor the total jobs running on their site or a CMS User wants to know the total number of jobs running on a specific site or on the WLCG infrastructure.
Actors	Physicist, Dashboard Data Repository.
Steps	1. The User enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository and presented on the screen. 3. The User can now sort by various attributes to get the total number of the jobs running on a specific site, user, storage element, activity and so on.

Use Case	Success Rate
Description	The CMS Site Administrators need to monitor the success rate of the jobs running on their site or a CMS User wants to know the success rate of the jobs running on a specific site, storage element, activity or on the WLCG infrastructure. The Grid, Application, Overall and Site Success Rates are available.
Actors	Physicist, Dashboard Data Repository.
Steps	1. The User enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository and the Success Rate is presented on the screen. 3. The User can now sort by various attributes to get the Grid, Application, Overall and Site Success Rate of the jobs running on a specific site, user, storage element, activity and so on.

Use Case	Error Diagnostics
Description	The CMS User wants quick access to advanced error diagnostics to understand the status of his/her jobs or task.
Actors	Physicist, Dashboard Data Repository.
Steps	1. The User enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository. 3. The User clicks on an error category from the numerical results on the table. 4. The Results are obtained from the Dashboard Data Repository and the error diagnostics are presented on the screen. 5. The User can now sort by various attributes to get the Grid Aborted and Application failed jobs running on a specific site, user, storage element, activity and so on.

Use Case	Datasets being used.
Description	The CMS User wants to view the datasets being used on the CMS VO.
Actors	Physicist, Dashboard Data Repository.
Steps	1. The User enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository. 3. The Users selects 'sort by dataset' from the menu. 4. The Results are obtained from the Dashboard Data Repository and presented on the screen. 5. The User can now sort by various attributes to get the datasets running on a specific site, by a user, on a storage element, by an activity and so on.

Use Case	Waiting Time
Description	The CMS Site Administrator needs to know the total waiting time of the jobs running on their site or a CMS User needs to know the total waiting time of his/her submitted jobs.
Actors	Physicist, Dashboard Data Repository.
Steps	1. The User enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository. 3. The Users clicks on the 'Waiting Time' button. 4. The Results are obtained from the Dashboard Data Repository and presented on the screen. 5. The User can now sort by various attributes to get the total waiting time of the jobs running on a specific site, by a user, on a storage element, by an activity and so on.

Use Case	Running Time
Description	The CMS Site Administrator needs to know the total running time of the jobs running on their site or a CMS User needs to know the total running time of his/her submitted jobs.
Actors	Physicist, Dashboard Data Repository.
Steps	1. The User enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository. 3. The Users clicks on the 'Running Time' button. 4. The Results are obtained from the Dashboard Data Repository and presented on the screen. 5. The User can now sort by various attributes to get the total running time of the jobs running on a specific site, by a user, on a storage element, by an activity and so on.

Use Case	Overall Time
Description	The CMS Site Administrator needs to know the overall time of the jobs running on their site or a CMS User needs to know the overall time of his/her submitted jobs.
Actors	Physicist, Dashboard Data Repository.
Steps	1. The User enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository. 3. The Users clicks on the 'Overall Time' button. 4. The Results are obtained from the Dashboard Data Repository and presented on the screen. 5. The User can now sort by various attributes to get the overall time of the jobs running on a specific site, by a user, on a storage element, by an activity and so on.

Use Case	CPU Time
Description	The CMS Site Administrator needs to know the total CPU time of the jobs running on their site or a CMS User needs to know the total CPU time of his/her submitted jobs.
Actors	Physicist, Dashboard Data Repository.
Steps	1. The User enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository. 3. The Users clicks on the 'CPU Time' button. 4. The Results are obtained from the Dashboard Data Repository and presented on the screen. 5. The User can now sort by various attributes to get the overall CPU time of the jobs running on a specific site, by a user, on a storage element, by an activity and so on.

Use Case	Job Wrapper Time
Description	The CMS Site Administrator needs to know the total job wrapper time of the jobs running on their site or a CMS User needs to know the total job wrapper time of his/her submitted jobs.
Actors	Physicist, Dashboard Data Repository.
Steps	1. The User enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository. 3. The Users clicks on the 'Job Wrapper Time' button. 4. The Results are obtained from the Dashboard Data Repository and presented on the screen. 5. The User can now sort by various attributes to get the overall job wrapper time of the jobs running on a specific site, by a user, on a storage element, by an activity and so on.

Use Case	Processing Efficiency
Description	The CMS Site Administrator needs to know the percentage of the average processing efficiency of the jobs running on their site or a CMS User needs to know the percentage of the average processing efficiency of his/her submitted jobs.
Actors	Physicist, Dashboard Data Repository.
Steps	1. The User enters the Job Summary application. 2. The Results are obtained from the Dashboard Data Repository. 3. The Users clicks on the 'Processing Efficiency' button. 4. The Results are obtained from the Dashboard Data Repository and presented on the screen. 5. The User can now sort by various attributes to get the average processing efficiency of the jobs running on a specific site, by a user, on a storage element, by an activity and so on.

B.2 SQL Queries

In this section, the most important SQL queries of the application will be presented. The first set of SQL queries are responsible for fetching the list with the values of the filters ordered by the name of the value for each category.

```
select distinct "GridName" as "user" from users order by "user"
select distinct "VOName" as "site" from site where "InteractiveInterfaceFlag" = 0 order
    by "site"
select distinct "ShortCEName" as "ce" from short_ce order by "ce"
select distinct "SubmissionTool" as "submissiontool" from submission_tool order by
    "submissiontool"
select distinct "ApplicationVersion" as "application" from application order by
"application" :
select distinct "RbName" as "rb" from rb order by "rb"
select distinct "Type" as "activity" from task_type order by "activity"
select distinct "SchedulerName" as "grid" from scheduler order by "grid"
select distinct "JobType" as "jobtype" from job_type order by "jobtype"
select distinct "Tier" as "tier" from site order by "tier"
```

The SQL queries for the consumed time information are variable and constantly changing according to the selected set of the filters. The following query calculates the overall time per site.

```
select "VOName" as "name", 24*60*60*avg(delay) as "value", 24*60*60*min(delay)
    as "dmin", 24*60*60*max(delay) as "dmax", 24*60*60*sum(delay) as "total"
    from ( select ( to_date(to_char("FinishedTimeStamp",'YYYY-MM-DD
    HH24:MI:SS'),'YYYY-MM-DD HH24:MI:SS') -
    to_date(to_char("DboardFirstInfoTimeStamp",'YYYY-MM-DD
    HH24:MI:SS'),'YYYY- MM-DD HH24:MI:SS') ) as delay,
    site."VOName" as "VOName"
    from job, task, site ,task_type where ("DboardFirstInfoTimeStamp" <=
    :bv_date2) and ("DboardFirstInfoTimeStamp" >= :bv_date1) and
```

```
( TASK."TaskTypeId" = task_type."TaskTypeId" and task_type."Type" =
:bv_activity) and ("FinishedTimeStamp" >= "DboardFirstInfoTimeStamp") and
("FinishedTimeStamp" != '01-Jan-70 12.00.00 AM') and
("DboardFirstInfoTimeStamp" != '01-Jan-70 12.00.00 AM')
    and  job."SiteId" = site."SiteId"
    and (job."TaskId" = task."TaskId")
    and ("DboardStatusId" in ('T'))
  and job."TimeOutFlag"='0'  )  group by "VOName" order by "value" desc
```

The SQL query for the exit code summary calculation is variable according to the selected set of filters. The following query calculates the exit code summary values for a specific site.

```
with temp as (select "exitcode", count("exitcode") as "num", "URLToDoc" as "url",
    "Comment" as "comment", "AppGenericStatusReasonValue" as "value",
    "SiteUserFlag" as "flag"
    from APP_GENERIC_STATUS_REASON app,(
    select Job."DboardStatusId", Job."JobExecExitCode" as "exitcode",
        Job."DboardJobEndId", Task."UserId", site."SiteId",
        Job."DboardFirstInfoTimeStamp",
        site."SchedulerId", Task."ApplicationId", Job."RbId", task_type."Type",
        task_type."GenericType",  Task."InputCollectionId",Task."TaskTypeId",
            Task."SubmissionToolId", task."TaskId" as "TaskId" ,
            submission_tool_ver."SubToolVersion"  from  job,task,site,
            task_type , submission_tool_ver where (task."TaskTypeId" =
            task_type."TaskTypeId")  and (job."SiteId" = site."SiteId") and
            (job."TaskId" = task."TaskId") and
            ("DboardFirstInfoTimeStamp" <= :bv_date2) and
            ("DboardFirstInfoTimeStamp" >=  :bv_date1) and
            (("DboardJobEndId"='F' and "DboardStatusId"='T')) and
            (task_type."Type" = :bv_activity) and (site."VOName" = :bv_site)
            and (task."SubToolVerId" =
            submission_tool_ver."SubToolVerId")) ex
```

```
                    where (app."AppGenericErrorCode"=ex."exitcode"  )
                    group by "exitcode", "URLToDoc", "Comment",
                    "AppGenericStatusReasonValue",
                    "SiteUserFlag"
                    order by "SiteUserFlag" desc)
select * from ((select temp."flag", sum("num") as "sum_n" from  temp group by
temp."flag") sum_n left join temp on temp."flag"=sum_n."flag" )order by sum_n."flag"
```

The following SQL query fetches the data for the plot and the table. The SQL query is not constant and it changes according to the selected set of filters.

```
with subjobs as (
   select Job."DboardStatusId",
   Job."DboardGridEndId", Job."DboardJobEndId", Task."UserId", Site."VOName",
   Job."DboardFirstInfoTimeStamp", Task."DefaultSchedulerId" as "SchedulerId",
   Task."ApplicationId", Task."InputCollectionId", task."TaskTypeId",
   Task."SubmissionToolId", Job."JobExecExitCode", "SiteUserFlag",
   task."TaskId" as "TaskId", Job."RbId", Job."ShortCEId", coalesce("NEvProc",0) as
"NEvProc", Task."SubmissionType",
   coalesce("WrapCPU", 0) as "WrapCPU", coalesce("WrapWC", 0) as "WrapWC",
   job."JobType", submission_tool_ver."SubToolVersion" as "SubToolVersion",
submission_ui."DisplayName" as "DisplayName",
   site."Tier" as "Tier", task_type."GenericType", task_type."Type", Job."StageOutSE"
   from job
   left outer join app_generic_status_reason on JOB."JobExecExitCode" =
APP_GENERIC_STATUS_REASON."AppGenericErrorCode"
   left outer join task on job."TaskId" = task."TaskId"
   left outer join site on job."SiteId"=site."SiteId"
   left outer join submission_tool_ver on
   task."SubToolVerId"=submission_tool_ver."SubToolVerId"
```

```
left outer join submission_ui on
task."SubmissionUIId"=submission_ui."SubmissionUIId"
left outer join task_type on task_type."TaskTypeId" = task."TaskTypeId"
where ("DboardFirstInfoTimeStamp" <= :bv_date2)  and ("DboardFirstInfoTimeStamp"
>= :bv_date1)) select distinct( task_type."Type") as "name" ,
"pending", "running", "unknown", "terminated",
"done", "cancelled", "aborted", "app-succeeded",
"applic-failed", "site-failed", "user-failed", "unk-failed",
"app-unknown", "site-calc-failed","unsuccess", "allunk","events", "cpu", "wc"
from
(
select T123.fid,
"pending", "running", "unknown", "terminated", "done", "cancelled", "aborted",
"app-succeeded",
"applic-failed", "site-failed", "site-calc-failed","user-failed", "unk-failed",
"app-unknown", coalesce(T4."unsuccess", 0) as "unsuccess",
coalesce(T4."allunk", 0) as "allunk","events", "cpu", "wc"
from
(
select T12.fid, "events", "cpu", "wc",
"pending", "running", "unknown", "terminated", coalesce("done", 0) as "done",
coalesce("cancelled", 0) as "cancelled", coalesce("aborted", 0) as "aborted",
coalesce("app-succeeded", 0) as "app-succeeded",
coalesce("applic-failed", 0) as "applic-failed",
coalesce("site-failed", 0) as "site-failed",
coalesce("user-failed", 0) as "user-failed",
coalesce("unk-failed", 0) as "unk-failed",
coalesce("site-calc-failed", 0) as "site-calc-failed",
```

```
coalesce("app-unknown",0) as "app-unknown"
from
(
select T1.fid, "pending", "running", "unknown", "terminated", "done",
"cancelled", "aborted", coalesce(T2."events",0) as "events",
coalesce(T2."cpu",0) as "cpu", coalesce(T2."wc",0) as "wc"
from
(
select fid,
max(decode("DboardStatusId", 'P', count, 0)) as "pending",
max(decode("DboardStatusId", 'R', count, 0)) as "running",
max(decode("DboardStatusId", 'U', count, 0)) as "unknown",
max(decode("DboardStatusId", 'T', count, 0)) as "terminated"
from (select count("DboardStatusId") as count, "TaskTypeId"as fid,
"DboardStatusId" from subjobs
group by "TaskTypeId", "DboardStatusId")
group by fid
) T1
left outer join
(
select fid, sum("events") as "events", sum("cpu") as "cpu", sum("wc") as "wc",
max(decode("DboardGridEndId", 'D', count, 0)) as "done",
max(decode("DboardGridEndId", 'C', count, 0)) as "cancelled",
max(decode("DboardGridEndId", 'A', count, 0)) as "aborted"
from (select count("DboardGridEndId") as count, "TaskTypeId"as fid,
sum("NEvProc") as "events", sum("WrapCPU") as "cpu", sum("WrapWC") as "wc" ,
"DboardGridEndId"  from subjobs where subjobs."DboardStatusId" = 'T'
group by "TaskTypeId", "DboardGridEndId")
```

```
  group by fid
  ) T2
  on T1.fid=T2.fid
  ) T12
  left outer join
  (
  select all_jobs.fid as fid, "app-succeeded", "applic-failed", "site-failed", "user-failed", "unk-failed", "app-unknown", "site-calc-failed"
  from (
  select fid,
  max(decode("DboardJobEndId", 'S', count, 0)) as "app-succeeded",
  max(decode ("DboardJobEndId", 'F', decode("SiteUserFlag", 'application', count, 0))) as "applic-failed",
  max(decode ("DboardJobEndId", 'F', decode("SiteUserFlag", 'site', count, 0))) as "site-failed",
  max(decode ("DboardJobEndId", 'F', decode("SiteUserFlag", 'user', count, 0))) as "user-failed",
  max(decode ("DboardJobEndId", 'F', decode("SiteUserFlag", 'unknown', count, 0))) as "unk-failed",
  max(decode("DboardJobEndId", 'U', count, 0)) as "app-unknown"
  from (select count("DboardJobEndId") as count, "TaskTypeId"as fid,
  sum("NEvProc") as "events", sum("WrapCPU") as "cpu", sum("WrapWC") as "wc", "DboardJobEndId", "SiteUserFlag"  from subjobs where subjobs."DboardStatusId" = 'T'
  group by "TaskTypeId", "DboardJobEndId", "SiteUserFlag" )
  group by fid) all_jobs
  left outer join (select fid, max(decode ("DboardJobEndId", 'F', decode("SiteUserFlag", 'site', count, 0))) as "site-calc-failed"
  from (select count("DboardJobEndId") as count, "TaskTypeId" as fid, "DboardJobEndId", "SiteUserFlag"
  from subjobs where subjobs."DboardStatusId"='T' and subjobs."DboardGridEndId" <> 'A'
```

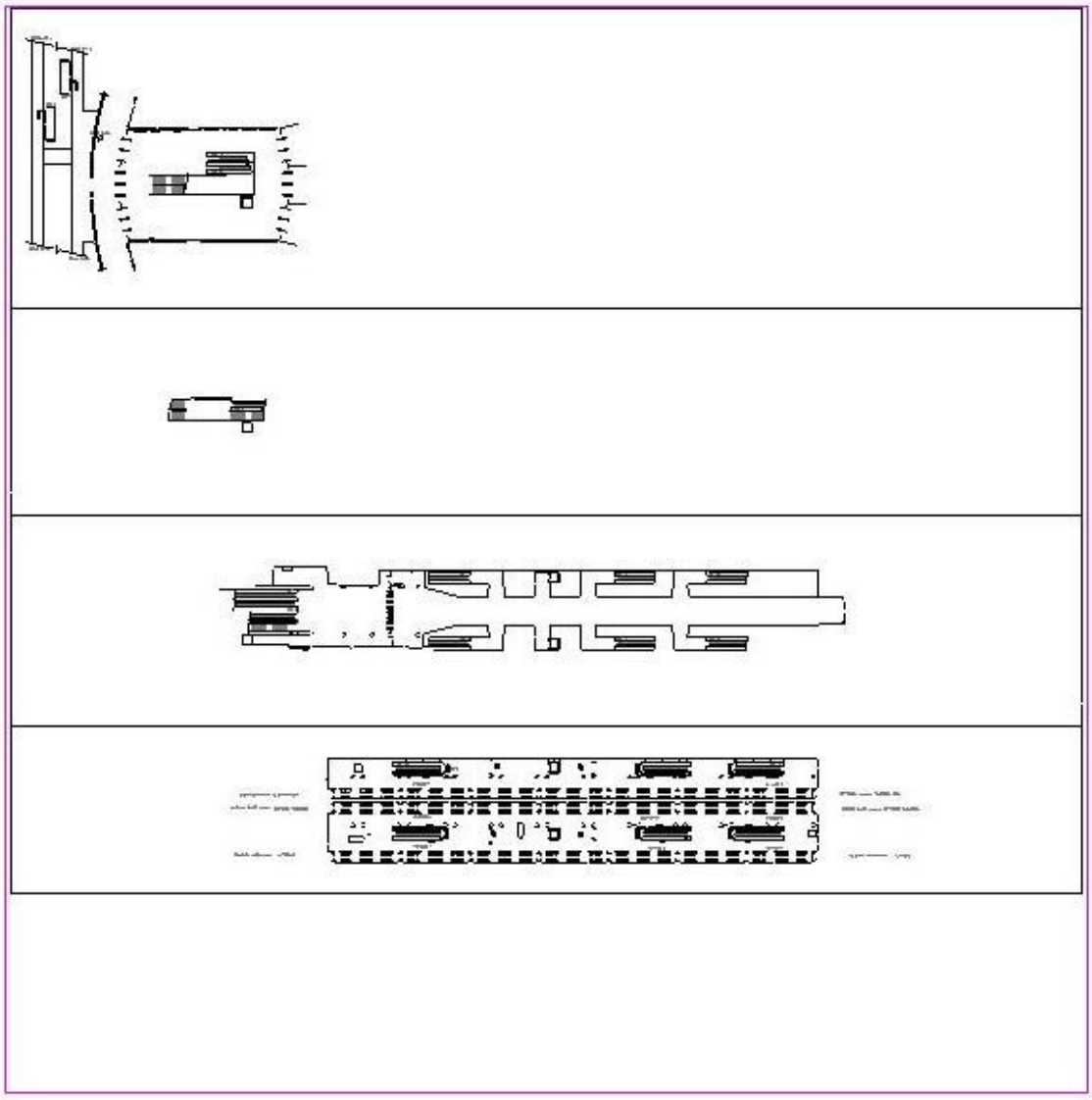
```
group by "TaskTypeId", "DboardJobEndId", "SiteUserFlag") group by fid) calc_jobs
on all_jobs.fid = calc_jobs.fid
) T3
on T12.fid = T3.fid
) T123
left outer join
(
select unk.fid, "unsuccess", "allunk"
from ((select count("DboardJobEndId") as "unsuccess", "TaskTypeId" as fid,
"DboardJobEndId" from subjobs
    where subjobs."DboardJobEndId" = 'S'
    and (subjobs."DboardGridEndId" = 'A'
    or subjobs."DboardGridEndId" = 'C') group by "TaskTypeId", "DboardJobEndId") suc
    left outer join (select count("DboardJobEndId") as "allunk", "TaskTypeId" as fid  from
subjobs
    where subjobs."DboardJobEndId" = 'U'
    and subjobs."DboardStatusId" = 'U'
    group by "TaskTypeId" ) unk
    on suc.fid = unk.fid)
) T4
on T123.fid = T4.fid
) S
join task_type on task_type."TaskTypeId" = S.fid order by
"pending"+"running"+"unknown"+"terminated" desc
```

APPENDIX C. LEGION ANALYSER

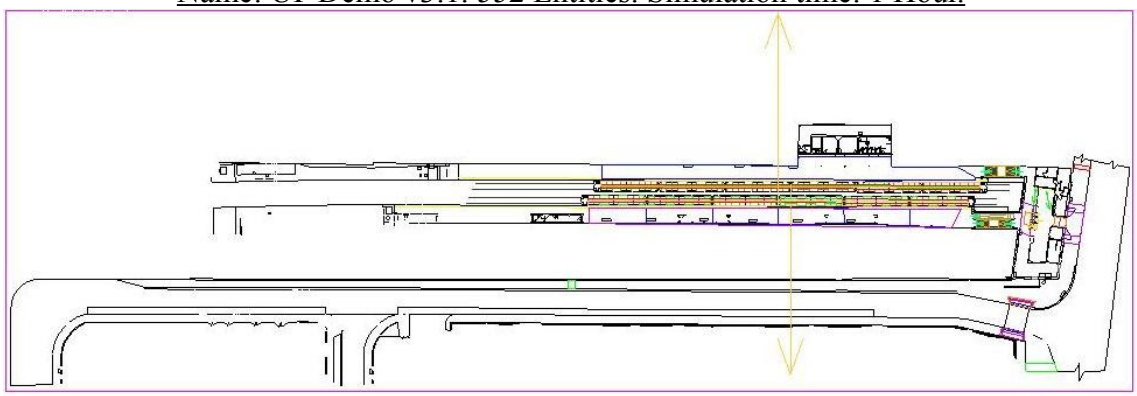
C.1 Simulated Models for the Benchmarking of the Multi-threaded Analyser

Small-sized Models

Name: PM Peak. 350 Entities. Simulation time: 3 Hours.

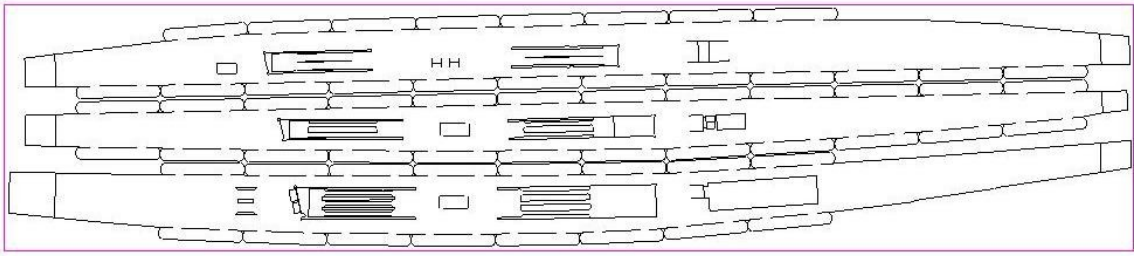


Name: UP Demo v3:1. 552 Entities. Simulation time: 1 Hour.



Medium-sized Models

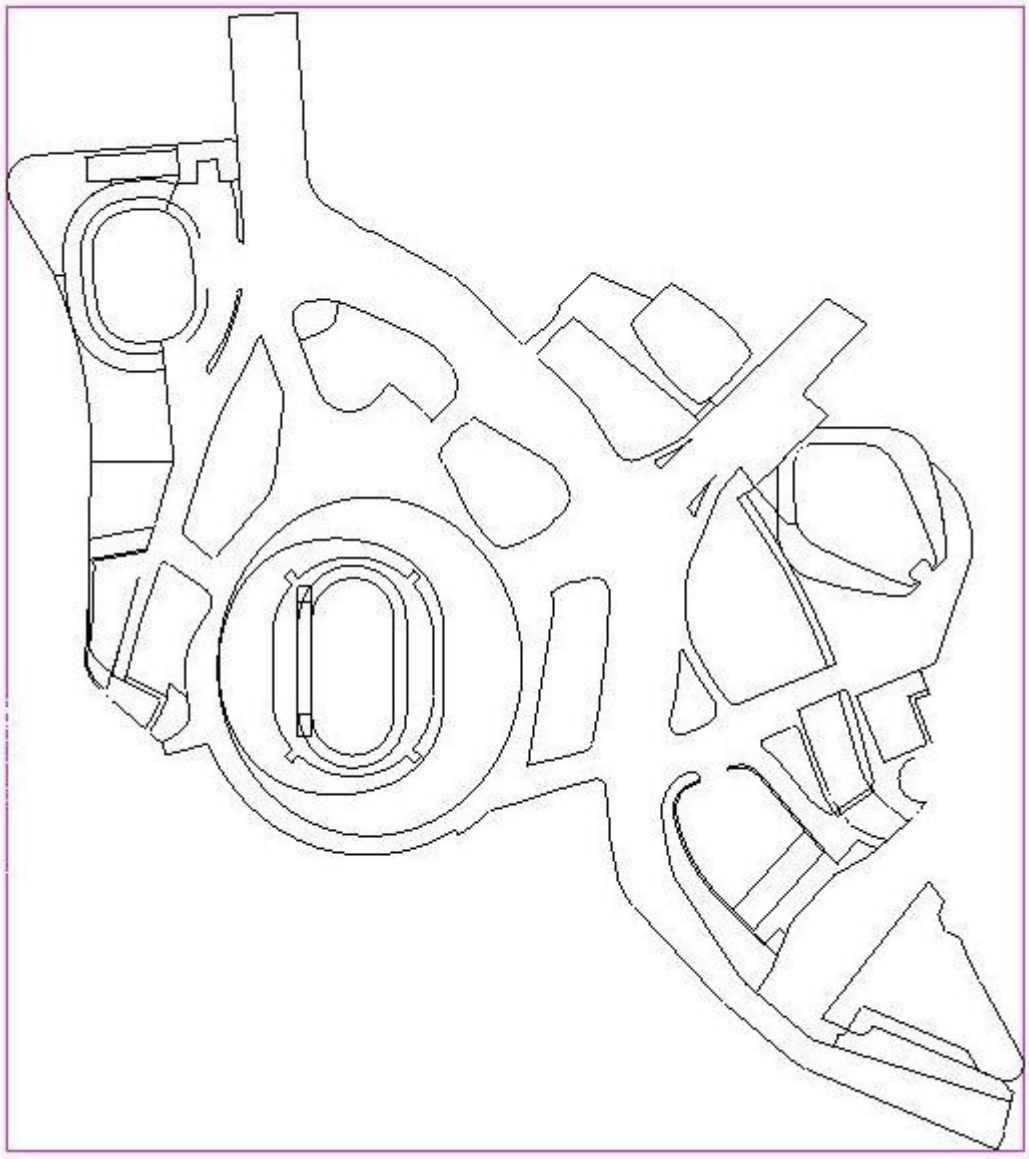
Name: Gatwick Airport Station Re-development. 1200 entities. Sim time: 1 Hour.



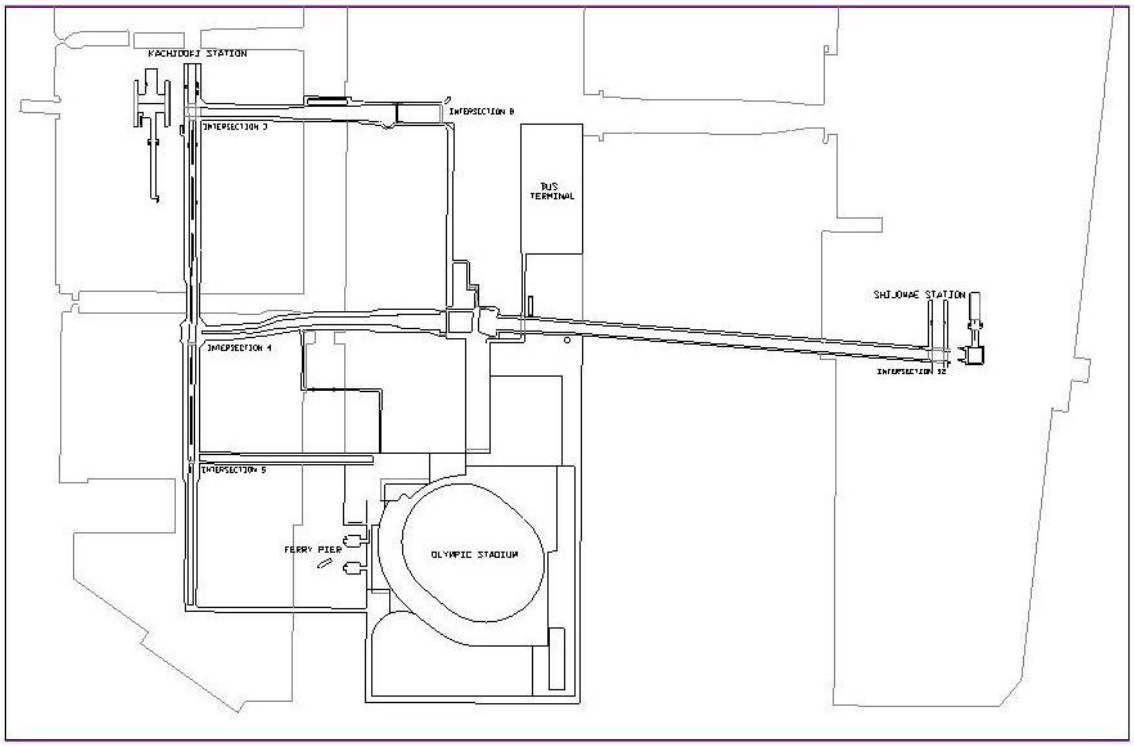
Name: New WTC Model. 2500 entities. Simulation time: 1 Hour and 30 Mins

Large-sized Models

Name: London Olympic Park 2012. 51000 entities. Simulation time: 14 Mins.

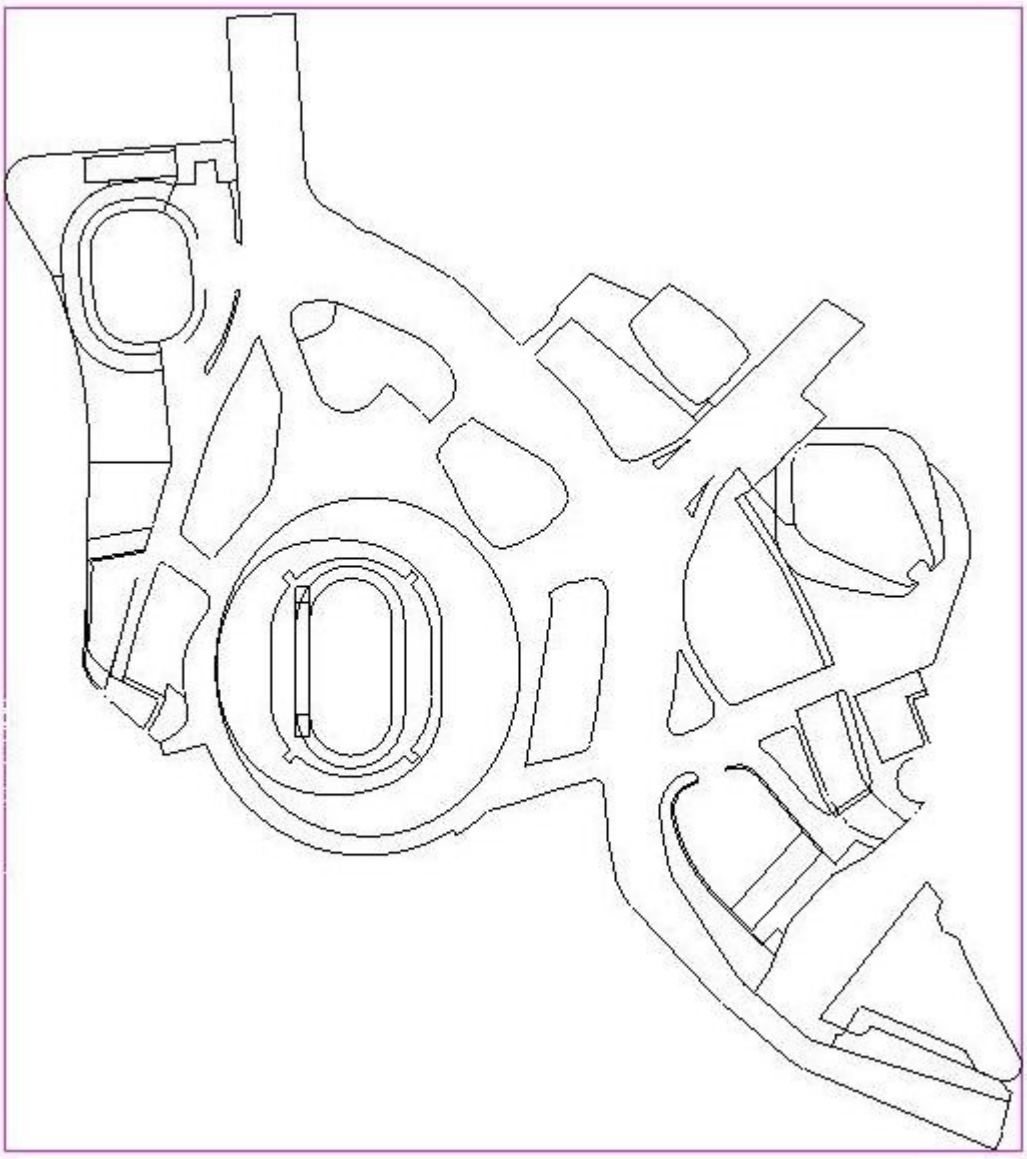


Name: HOS Case3. 52000 entities. Simulation time: 19 Mins.



C.2 Simulated Model for the Benchmarking of the Distributed Analyser

Name: London Olympic Park 2012. 56500 entities.



C.3 Work Division for Six Slave Nodes

The following code illustrates the division of the work for six Slave nodes.

```
//// Split the jobs according to the size of totalnodes
int start, workEnd;
int node1End, node2End, node3End, node4End, node5End;
switch (mynode)
        {
        case 1: // 1st worker node
         start          =       1;
         workEnd        =       mapSize * mynode / (totalnodes-1);
         node1End = workEnd;
         advance(iter, workEnd);
          break;
        case 2: // 2nd worker node
         node1End       =       mapSize * (mynode-1) / (totalnodes-1);
         start          =       node1End+1;
         workEnd        =       mapSize * mynode / (totalnodes-1);
         break;
        case 3: // 3rd worker node
         node2End       =       mapSize * (mynode-1) / (totalnodes-1);
         start          =       node2End+1;
         workEnd        =       mapSize * mynode / (totalnodes-1);
         break;
        case 4:  // 4th worker node
         node3End       =       mapSize * (mynode-1) / (totalnodes-1);
         start          =       node3End+1;
         workEnd        =       mapSize * mynode / (totalnodes-1);
         break;
        case 5: // 5th worker node
         node4End       =       mapSize * (mynode-1) / (totalnodes-1);
         start          =       node4End+1;
         workEnd        =       mapSize * mynode / (totalnodes-1);
         break;
        case 6: // 6th worker node
         node5End       =       mapSize * (mynode-1) / (totalnodes-1);
         start          =       node5End+1;
         workEnd        =       mapSize * mynode / (totalnodes-1);
         break;
        default: // for Root (id=0) - just some debugging msg..
         TRACE ("Hello from root");
         break;
        }
```

C.4 Sender Code

Each Slave node calculates a map in a separate thread and then sends the results back to the Master node as illustrated in the following code listing.

```
// IF we have 6 enabled maps & 1 master + 6 cluster nodes then every node will do
//          calculations for just one map otherwise work will be divided by totalnodes size.
if (mynode != 0) // workers - sender code
{
 MapList::iterator          iter(advance(m_mapList.begin(),start));
 advance(iter, start);      // Beginning of the allocated work for earch worker
 MapList::iterator          end(  m_mapList.begin() );          // Actually it's the beginning...
 advance(end, workEnd); // But now it's the end of the allocated work for each worker
while( iter != end )
          {
                    const COdbSpaceCentricMap* pSpaceMap = dynamic_cast<const
                                                  COdbSpaceCentricMap*>( (*iter)->GetMap() );
                    // Only do calculations for enabled maps
                    if( pSpaceMap->IsEnabled() )
                    {
                     CReSpaceMapManagerItem* pSpaceMapItem =
                                        dynamic_cast<CReSpaceMapManagerItem*>(*iter);
                     ASSERT( pSpaceMapItem );
                     // Execute the thread
                     m_threadPool.schedule( SpaceMapTask( pSpaceMapItem, entities ) );
                    }
                    ++iter;
          }
// Join the thread pool as to wait for all the maps to be finished computing
if( !m_threadPool.empty() )
          {
           m_threadPool.wait();
          }
// Call the serialisation & MPI comm function
m_cellStorageManager->SerialiseMe();
}
```

C.5 Receiver Code

The Master node collects the results, unpacks them and calls the drawing function to draw the results on the screen as illustrated in the following code listing.

```
else // root - Receiver code
{
// Get the data, unpack them (if serialised), draw the results (call the draw function)
int wSlave;
// Use a loop to get all the results from all the nodes (equal to totalnodes)
// then unpack them and call the drawing function
wSlave = totalnodes - 1; // wSlave is equal to the total no of nodes minuss the root node
if (world.rank()==0)
{
        gather(world,legion_mapcalc,0);
        }
}
```

BIBLIOGRAPHY

[1] "Charles Babbage". The MacTutor History of Mathematics archive. School of Mathematics and Statistics, University of St Andrews, Scotland. 1998. http://www-history.mcs.st-and.ac.uk/Mathematicians/Babbage.html

[2] B. Randell (ed.). The Origins of Digital Computers, Selected Papers, 3rd ed. *Springer-Verlag*. 1982.

[3] The Alan Turing Internet Scrapbook, Computable Numbers and the Turing Machine, 1936. http://www.turing.org.uk/turing/scrapbook/machine.html

[4] K. Zuse. The Computer – My Life. *Berlin/Heidelberg: Springer-Verlag*. ISBN 0-387-56453-5. 1993.

[5] M. V. Wilkes. Automatic Digital Computers. *New York: John Wiley & Sons*. pp. 305 pages. QA76.W5 1956.

[6] N. Macrae. John von Neumann: The Scientific Genius Who Pioneered the Modern Computer, Game Theory, Nuclear Deterrence, and Much More. *Pantheon Press*. ISBN 0679413081. 1992.

[7] H. Goldstine and A. Goldstine. The Electronic Numerical Integrator and Computer (ENIAC), 1946. Reprinted in The Origins of Digital Computers: Selected Papers, *Springer-Verlag*, New York, 1982, pp. 359-373.

[8] W. Shockley. Electrons and Holes in Semiconductors, with Applications to Transistor Electronics, *Krieger*. ISBN 0-88275-382-7. 1956.

[9] IEEE Global History Network, Robert Noyce. http://www.ieeeghn.org/wiki/index.php/Robert_Noyce

[10] A. Osborne. An Introduction to Microcomputers. Volume 1: Basic Concepts (2nd ed.). *Berkely, California: Osborne-McGraw Hill*. ISBN 0-931988-34-9. 1980.

[11] P. Mack. The Microcomputer Revolution. 2005. http://www.clemson.edu/caah/history/FacultyPages/PamMack/lec122/micro.htm.

[12] F. Mims. The Altair story; early days at MITS. *Creative Computing (Creative Computing) 10 (11)*: p. 17. 1984. http://www.atarimagazines.com/creative/v10n11/ 17_The_Altair_story_early_d.php

[13] Moore's Law – Wikipedia: The Free Encyclopedia. http://en.wikipedia.org/wiki/Moore%27s_law

[14] Excerpts from A Conversation with Gordon Moore: Moore's Law. Intel. 2005. ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf

[15] Lev B. Levitin and Tommaso Toffoli, Thermodynamic Cost of Reversible Computing, *Physical Review Letters*, Volume 99, Issue 11, 2007.

[16] Top 500 Supercomputing Sites. http://top500.org

[17] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/

[18] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *Computational Science and Engineering*, 5(1):46–55, 1998.

[19] William Gropp, et al, High-Performance, Portable Implementation of the MPI

Message Passing Interface Standard, *Parallel Computing*, Vol. 22, 6, 1996.

[20] Internet Corporation for Assigned Names and Numbers (ICANN). http://www.icann.org

[21] The Internet Engineering Task Force (IETF). http://www.ietf.org

[22] R. Fielding, J. Getty, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. Hypertext Transfer Protocol -- HTTP/1.1, RFC 2616, 1999. http://www.ietf.org/rfc/rfc2616.txt

[23] T. Berner-Lee and R. Cailliau. WorldWideWeb: Proposal for a HyperText Project.. 1990. http://www.w3.org/Proposal.html

[24] World Wide Web Consortium (W3C) http://www.w3.org

[25] Apple Computer, Inc. HyperCard Script Language Guide: The HyperTalk Language. *Reading, MA: Addison-Wesley Publishing Company*. p.181. 1988.

[26] Ten Years Public Domain for the Original Web Software. http://tenyears-www.web.cern.ch/tenyears-www/Welcome.html

[27] World Wide Web Consortium, Web Services Activity. http://www.w3.org/2002/ws/

[28] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web Services Web: An introduction to SOAP, WSDL, UDDI. *IEEE Internet Computing*, 6(2):86–93, March-April 2002.

[29] Organisation for the Advancement of Structured Information Standards. http://www.oasis-open.org/home/index.php

[30] H. Voormann, Wikipedia – The Free Encyclopedia. http://upload.wikimedia.org/wikipedia/commons/4/4a/Webservices.png

[31] The Globus Toolkit 4 Programmer's Tutorial, Chapter 1.2: A Short Introduction to Web Services. http://gdp.globus.org/gt4-tutorial/multiplehtml/ch01s02.html

[32] I. Foster and C. Kesselman. Computational Grids, The Grid: Blueprint for a New Computing Infrastructure. *Morgan-Kaufman*, 1998.

[33] L. Smarr and C.E. Catlett. Metacomputing. Commun. *ACM*, 35(6):44–52, 1992.

[34] T. DeFanti, I. Foster, M. E. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY: Wide Area Visual Supercomputing. *International Journal of Supercomputing Applications*, 10(2), 1996.

[35] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *Journal of Computer Science and Technology*, 21(4):513–520, 2006.

[36] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[37] WestGrid Group. Western Canada Research Grid: WestGrid. http: //www. westgrid.ca

[38] Laura Pearlman, Carl Kesselman, et al. Distributed Hybrid Earthquake Engineering Experiments: Experiences with a Ground-Shaking Grid Application. In *HPDC*, pages 14-23,2004.

[39] A. Chervenak, I. Foster, C. Kesselman, et. al. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications,* 23: 187-200,2001.

[40] The AccessGrid Poject. http: //www-fp.mcs.anl.gov/fl/accessgrid

[41] J. Taylor. Defining e-Science. http://www.nesc.ac.uk/nesc/define.html

[42] LHC – The Large Hadron Collider. http://lhc.web.cern.ch/lhc/

[43] e-Science Core Programme Report. http://www.rcuk.ac.uk/escience/news/cpreport.htm

[44] The ATLAS Experiment. http://atlas.web.cern.ch/Atlas/Collaboration/

[45] The CMS Experiment. http://cms.web.cern.ch/cms/index.html

[46] The ALICE Experiment. http://aliceinfo.cern.ch/Collaboration/index.html

[47] The LHCb Experiment. http://lhcb.web.cern.ch/lhcb/

[48] The TOTEM Experiment. http://public.web.cern.ch/Public/en/LHC/TOTEM-en.html

[49] The LHCf Experiment. http://public.web.cern.ch/public/en/LHC/LHCf-en.html

[50] Worldwide LHC Computing Grid. http://lcg.web.cern.ch/LCG/

[51] F. Gagliardi, B. Jones, F. Grey, M. Bgin, and M. Heikkurinen. Building an infrastructure for scientific Grid computing: status and goals of the EGEE project. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1833):1729–1742, 2005.

[52] R. Pordes et al. The Open Science Grid, *Journal of Physics Conference Series*, 78, 2007.

[53] gLite Middleware. http://cern.ch/glite

[54] M. Ellert, M. Grnager, A. Konstantinov, B. Knya, J. Lindemann, I. Livenson, J.L. Nielsen, M. Niinimki, O. Smirnova, and A. Wnnenh. Advanced Resource Connector Middleware for Lightweight Computational Grids. *Future Generation*

Computer Systems, 23:219–240, 2007.

[55] Alain Roy et. al. Building and testing a production quality grid software distribution for the Open Science Grid. *Journal of Physics: Conference Series 180,* 2009.

[56] M. Campanella and L. Perini. The analysis model and the optimisation of geographical distribution of computing resources:a strong connection. http://monarc.web.cern.ch/MONARC/docs/monarc docs/1998-01.html

[57] The Four-Tiered Model as Proposed by the MONARC Project. http://images.iop.org/objects/physicsweb/world/21/11/34/PWlar2_11-08.jpg

[58] E. Karavakis and A. Khan. A Multi-threaded and Distributed Framework for Pedestrian Simulation Analysis. *7th International Conference of Computational Methods in Sciences and Engineering (ICCMSE), Rhodes, Greece,* To be published in *American Institute of Physics,* 2010.

[59] A. Fanfani, A.Khan, E. Karavakis et al. Distributed Analysis in CMS. To be published in *Journal of Grid Computing,* 2010.

[60] J. Andreeva, E. Karavakis et al. Experiment Dashboard for Monitoring of the Computing Activities of the LHC Experiments. To be published in *Journal of Grid Computing,* 2010.

[61] E. Karavakis, J. Andreeva, A. Khan, G. Maier and B. Gaidioz. CMS Dashboard Task Monitoring: A User-Centric Monitoring View. *17th International Conference on Computing in High Energy and Nuclear Physics (CHEP), Prague, Czech Republic,* To be published in *IOP Publishing,* 2010.

[62] J. Andreeva, E. Karavakis et al. Job Monitoring on the WLCG Scope: Current Status and New Strategy. *17th International Conference on Computing in High Energy and Nuclear Physics (CHEP), Prague, Czech Republic,* To be published in

IOP Publishing, 2010.

[63] E. Karavakis, J. Andreeva, G. Maier and A. Khan. CMS Dashboard for Monitoring of the User Analysis Activities. 7th International Conference of Computational Methods in Sciences and Engineering (ICCMSE) Symposium: Computing in Experimental High Energy Physics, Rhodes, Greece, To be published in *American Institute of Physics,* 2010.

[64] C. Gordon Bell and Allen Newell. Computer Structures: Readings and Examples, *McGraw-Hill Book Company*, New York. 1971.

[65] Bill Lewis: Threads Primer: A Guide to Multithreaded Programming, P*rentice Hall*. 1995.

[66] Steve Kleiman, Devang Shah, Bart Smaalders: Programming With Threads, *SunSoft Press*. 1996

[67] M. Flynn. Some Computer Organizations and Their Effectiveness, *IEEE Trans. Comput.*, Vol. C-21, pp. 948, 1972.

[68] M. Flynn. Parallel Architectures, *ACM Computing Surveys* 28(1):67-70, 1996.

[69] Ajay D. Kshemkalyani and Mukesh Singhal. Distributed Computing: Principles, Algorithms, and Systems. *Cambridge University Press*, 2008.

[70] R. Cleaveland and S. Smolka. Strategic Directions in Concurrency Research. *ACM Computing Surveys 28 (4): 607.* 1996.

[71] Free On-Line Dictionary of Computing: Granularity. http://foldoc.org/granularity

[72] G. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS Joint Computer Conferences,* Vol. 30, pages 483-485, *Thompson Books,* 1967.

[73] J. Gustafson. Re-evaluating Amdahl's Law. *Communications of the ACM* 31(5):532-533, 1988.

[74] Karp, Alan H., and Horace P. Flatt. Measuring Parallel Processor Performance. *Communications of the ACM* 33(5):539-543, 1990.

[75] R. Cypher and E. Leu. The Semantics of Blocking and Non-blocking Send and Receive Primitives. *Proceedings of the 8th International Symposium on Parallel Processing*, 729–735, 1994.

[76] Sayantan Sur, et al. High-performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An In-depth Performance Analysis, *Proceedings of the 2006 ACM/IEEE conference on Supercomputing,* 2006.

[77] George Karniadakis, Robert Kirby II. Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation. *Cambridge University Press,* 2003.

[78] Computer Cluster Architectures, Ainkaboot Limited. http://ainkaboot.co.uk/cluster-architecture.php

[79] Blaise Barney. POSIX Threads Programming Tutorial. Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/pthreads/

[80] PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/

[81] Michael Quinn. Parallel Programming in C with MPI and OpenMP. *McGraw-Hill Science/Engineering/Math,* 2003.

[82] Ayon Basumallik, Seung-Jai Min, Rudolf Eigenmann. Programming Distributed Memory Sytems Using OpenMP. *Parallel and Distributed Processing Symposium, International*, pp. 207, 2007 *IEEE International Parallel and Distributed Processing Symposium*, 2007.

[83] Jean Bacon. Concurrent Systems - Operating Systems, Database and Distributed Systems: An Integrated Approach. *Addison-Wesley*, 2003.

[84] E. W. Dijkstra. The structure of the 'THE'-Multiprogramming System. *Communications of the ACM* 11(5):341 – 346, 1968.

[85] Weijia Jia, Wanlei Zhou. Distributed Network Systems: From Concepts to Implementations. *Springer*, 2004.

[86] Yibei Ling, Tracy Mullen and Xiaola Lin. Analysis of Optimal Thread Pool Size. *ACM SIGOPS Operating System Review* Vol. 34, No. 2, 2000, pp. 42-55.

[87] Noah Gift. Practical Threaded Programming with Python: Threading Usage Patterns. http://www.ibm.com/developerworks/aix/library/au-threadingpython/

[88] A. J. C. van Gemund. The Importance of Synchronization Structure in Parallel Program Optimisation. *In ICS '97: Proceedings of the 11th international conference on Supercomputing*, pp. 164–171. *ACM*, 1997.

[89] Robert M. Fuhrer , Bill Lin , Steven M. Nowick. Algorithms for the Optimal State Assignment of Asynchronous State Machines. *In 1995 Conference on Advanced Research in VLSI,* 1995.

[90] Glossary of CCAT Terms – Indiana University. http://www.extreme.indiana.edu/ccat/glossary.html

[91] Maozhen Li, Mark Barker. The Grid: Core Technologies. *Wiley*, 2005.

[92] I. Foster. What is the grid? A three point Checklist. *GRIDToday (Now: HPC in the Cloud)*, July 2002. http://www.mcs.anl.gov/~itf/Articles/WhatIsTheGrid.pdf

[93] M. Baker, R. Buyya and D. Laforenza. The Grid: International Efforts in Global Computing. *In Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*,

Italy, 2000.

[94] William E. Moen. Realizing the Information Future: The Internet and Beyond. *National Academy Press*, Washington, DC, 1994

[95] Open Grid Forum (OGF). http://www.ogf.org

[96] Global Grid Forum (GGF). http://www.gridforum.org

[97] M. Hatch. Enterprise Grid Alliance and Global Grid Forum Complete Merger to Form Open Grid Forum. 2006
http://www.nesc.ac.uk/news/press_release/OGF_Merger.pdf

[98] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, 2002.
http://www.globus.org/research/papers/ogsa.pdf

[99] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, et al. Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum draft recommendation, 2003.
http://www.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf

[100] Globus Alliance. http://www.globus.org

[101] IBM. http://www.ibm.com

[102] Hewlett-Packard. http://www.hp.com

[103] Karl Czajkowski, Donald F Ferguson, Ian Foster et al. The WS-Resource Framework Version 1.0. http://www.globus.org/wsrf/specs/ws-wsrf.pdf

[104] Karl Czajkowski, Donald F Ferguson, Ian Foster et al. From OGSI to WS-Resource Framework: Refactoring and Evolution. Version 1.1

http://globus.org/wsrf/specs/ogsi_to_wsrf_1.0.pdf

[105] T. DeFanti, I. Foster, M. E. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY: Wide Area Visual Supercomputing. *International Journal of Supercomputing Applications*, 10(2), 1996.

[106] The Globus Toolkit 4 Programmer's Tutorial, Chapter 1.4: The Globus Toolkit 4. http://gdp.globus.org/gt4-tutorial/multiplehtml/ch01s04.html

[107] GLUE Specification v. 2.0. http://www.ogf.org/documents/GFD.147.pdf

[108] Globus Toolkit. News About Globus. http://www.globus.org/news.html#161

[109] Tom Howe. Crux for GT Developers. http://confluence.globus.org/display/whi/Crux+for+GT+Developers

[110] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[111] Condor Version 7.4.2 Manual. http://www.cs.wisc.edu/condor/manual/v7.4/5_Grid_Computing.html

[112] P. Kunst. European DataGrid project: Status and Plans. *Nuclear Instruments and Methods in Physics Research A*, (502):376–381, 2003.

[113] LHC Computing Grid LCG-2 Middleware Overview. http://www.grid.org.tr/servisler/dokumanlar/LCG-mw.pdf

[114] R-GMA. http://www.r-gma.org

[115] R-GMA Documentation. http://www.r-gma.org/fivemins.html

[116] R. Alfieri, R. Cecchini, V. Ciaschini, L. dellrsquo, Agnello, A. Frohner, et al. VOMS, an Authorization System for Virtual Organizations. *In Proceedings of the 1st European Across Grids Conference*, 2003.

[117] C. Aiftimiei et al. Job Submission and Management through Web Services: The Experience with the CREAM Service. *Journal of Physics: Conference Series 119,* 2008.

[118] C. Aiftimiei et al. Using CREAM and CEMON for Job Submission and Management in the gLite Middleware. *17th International Conference on Computing in High Energy and Nuclear Physics*, Prague, Czech Republic, To be published in *IOP Publishing,* 2010.

[119] G.A. Stewart, D. Cameron, G.A. Cowan, and G. McCance. Storage and Data Management in EGEE. I*n ACSW '07: Proceedings of the fifth Australasian symposium on ACSW frontiers*, pages 69–77, Australia, 2007.

[120] C. Grandi, D. Stickland, L. Taylor et al. The CMS Computing Model, CERN-LHCC-2004-035/G-083, 2004.

[121] A. Afaq et al. The CMS Dataset Bookkeeping Service, J*ournal of Physics Conference Series*, 119, 072001, 2008.

[122] Barry Blumenfeld, David Dykstra, Lee Lueking, Eric Wicklund. CMS Conditions Data Access using FroNTier. *International Conference on Computing in High Energy and Nuclear Physics (CHEP'07)*, 2007.

[123] Squid Proxy, http://www.squid-cache.org

[124] R. Egeland et al. Data Transfer Infrastructure for CMS Data Taking, *Proceedings of Science*, PoS (*ACAT08*)033, 2008.

[125] D. Evans et al. The CMS Monte Carlo Production System: Development and

Design, *Nuclear Physics Proceedings Suppl.* 177-178, 285-286, 2008.

[126] D. Spiga et al. The CMS Remote Analysis Builder (CRAB), *14th Int. Conf. On High Performance Computing*, 2007.

[127] P. Andreetto et al. The gLite Workload Management System. J*ournal of Physics Conference Series*, 119, 2008.

[128] A.Tsaregorodsev et al. Dirac: A Community Grid Solution, *CHEP07 Conference Proceedings, Victoria*, Canada, 2007.

[129] P. Nilsson. PanDA System in ATLAS Experiment, *ACAT'08 Conference*, Italy, 2008.

[130] P. Saiz et al. AliEn - ALICE Environment on the GRID, *Nuclear Instruments and Methods in Physics Research*, A502 (2003) 437-440, 2003.

[131] Experiment Dashboard Web Statistics web page. http://lxarda18.cern.ch/awstats/awstats.pl?config=lxarda18.cern.ch

[132] Imperial College Real Time Monitoring. http://gridportal.hep.ph.ic.ac.uk/rtm/

[133] D. Collados et al. Evolution of SAM in an Enhanced Model for Monitoring WLCG Services. *17th International Conference on Computing in High Energy and Nuclear Physics,* Prague, Czech Republic*,* To be published in *IOP Publishing,* 2010.

[134] LB. http://egee.cesnet.cz/cs/JRA1/LB/

[135] J. Moscicki et al. Ganga: A Tool for computational-task Management and Easy Access to Grid Resources, *Computer Physics Communication, Volume 180, Issue 11, November 2009, Pages 2303-2316,* arXiv:0902.2685v2, 2009. http://arxiv.org/pdf/0902.2685v2

[136] I. Legrand, H. Newman, C. Cirstoiu et al. MonALISA: an Agent Based, Dynamic Service System to Monitor, Control and Optimize Grid Based Applications. *Proceedings of Computing for High Energy Physics*, Switzerland, 2004.

[137] James Casey, Daniel Rodrigues, Ulrich Schwickerath, Ricardo Silva. Monitoring the Efficiency of User Jobs, *17th International Conference on Computing in High Energy and Nuclear Physics*, Prague, Czech Republic, To be published in *IOP Publishing,* 2010.

[138] Google Web Toolkit. http://code.google.com/webtoolkit/

[139] A Shoshani, A Sim, J Gu. Storage Resource Managers: Middleware Components for Grid Storage. *NASA Conference Publication,* 2002.

[140] Apache Web Server. http://apache.org

[141] Apache ActiveMQ. http://activemq.apache.org

[142] Janusz Martyniak, David Colling et al. A Real Time Monitoring of Grid Job Executions. *17th International Conference on Computing in High Energy and Nuclear Physics (CHEP), Prague, Czech Republic,* To be published in *IOP Publishing,* 2010.

[143] Legion Studio Software Suite. http://www.legion.com

[144] Legion Studio Case Studies. http://legion.com/case-studies

[145] J. L. Berrou, J. Beecham, P. Quaglia, M. A. Kagarlis, A. Gerodimos. Calibration and Validation of the Legion Simulation Model using Empirical Data. *Pedestrian and Evacuation Dynamics*, *Springer Berlin Heidelberg*, pp. 167-181, 2005.

[146] J. Fruin. Pedestrian and Planning Design. *Metropolitan Association of Urban Designers and Environmental Planners*. 1971.

[147] Transportation Research Board. Highway Capacity Manual, Special Report 204 TRB, Washington D.C, US, 1985.

[148] F. Rademakers, R. Brun. ROOT: An Object-Oriented Data Analysis Framework. *Proceedings AIHENP'96 Workshop, Nucl. Inst. Meth. In Phys. Res.* A389 pp. 81-86, Lausanne, 1997. See also: http://root.cern.ch

[149] L. Dagum. Technical Report - OpenMP: A proposed industry standard API for Shared Memory Programming, 1997. http://www.openmp.org/mp-documents/paper/paper.ps

[150] P. Kambadur, D. Gregor, A. Lumsdaine, A. Dharurkar. Modernizing the C++ interface to MPI. *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, LNCS, pp. 266-274, Germany, *Springer,* 2006.

[151] J. Andreeva et al. Experiment Dashboard: the monitoring system for the LHC experiments. I*n GMW'07: Proceedings of the 2007 workshop on Grid monitoring*, *ACM*, 2007.

[152] P. Saiz et al. Grid Reliability. *In CHEP'07: Proceedings of the 2007 International Conference on Computing in High Energy and Nuclear Physics*, *Journal of Physics: Conference Series 119*, 2007.

[153] Graphtool Library. http://t2.unl.edu/documentation/graphtool/graphtool-overview

[154] A McNab, S. Kaushal. The GridSite Proxy Delegation Service. *Grid Security Workshop*, Oxford, 2004. http://www.gridpp.ac.uk/papers/AHM2006610.pdf

[155] Dashboard Application Usage Statistics. http://lxarda18.cern.ch/usage.html

[156] Dashboard Site Status for the CMS Sites. http://dashb-ssb.cern.ch/ssb.html