FACHBEREICH C - FACHGRUPPE PHYSIK
BERGISCHE UNIVERSITÄT
WUPPERTAL

New approaches in user-centric job monitoring on the LHC Computing Grid

Application of remote debugging and real time data selection techniques

Dissertation
by
Tim dos Santos

July 25, 2011

Diese Dissertation kann wie folgt zitiert werden:

urn:nbn:de:hbz:468-20110727-113510-6
[http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:468-20110727-113510-6]

Contents

I Introduction **9**

1 Context: On High Energy Physics (HEP) **13**
1.1 Current research in HEP 13
1.1.1 The Standard Model 13
1.1.2 Examples for open questions 17
1.2 CERN and the LHC 18
1.2.1 The Large Hadron Collider 18
1.2.2 The ATLAS Experiment 19
1.2.3 Data flow in ATLAS 21
1.2.4 Real-time data reduction: Triggers 22
1.3 Software in HEP 22
1.3.1 High-performance maths and core services: ROOT . . . 23
1.3.2 Event generators and detector simulation tools 24
1.3.3 ATLAS' main physics analysis framework: Athena . . 25

2 Grid Computing **29**
2.1 Overview 29
2.1.1 Definition of the term "Grid Computing" 29
2.1.2 Virtual Organisations 31
2.1.3 Components and services of a Grid 32
2.1.4 Security in the Grid 33
2.2 The WLCG 35
2.2.1 Overview 35
2.2.2 The middleware: gLite 36
2.2.3 Computing model 39
2.2.4 Data storage and distribution 40
2.3 gLite Grid jobs 41
2.3.1 Input- and outputdata 42
2.3.2 Grid job life cycle 43
2.3.3 Job failures 44
2.4 WLCG software 46
2.4.1 Pilot jobs and the pilot factory 46
2.4.2 The user interfaces: pAthena and Ganga 47

3 Conclusion **50**

II Job monitoring **51**

4 Overview **51**
4.1 Site monitoring 51
4.2 User-centric monitoring of Grid jobs 52

5 The Job Execution Monitor **53**
5.1 Concept 54
5.2 Architecture 55
5.2.1 User interface component 56
5.2.2 Worker node component 58
5.2.3 Data transmission 59
5.2.4 Inter-process communication 61
5.3 Acquisition of monitoring data 62
5.3.1 System metrics monitor (“Watchdog”) 62
5.3.2 Script wrappers 62
5.4 User interface 63
5.4.1 Command-line usage 64
5.4.2 Built-in interface 64
5.4.3 Integration into Ganga 65
5.5 Deployment strategy 68
5.6 Shortcomings of this version of the software 68

6 Conclusion **71**

III Tracing the execution of binaries **73**

7 Concept and requirements **73**
7.1 Event notification 74
7.2 Symbol resolving and identifier lookup 74
7.3 Application memory inspection 75
7.4 Publishing of the gathered data 75
7.5 User code prerequisites 75

8 Architecture and implementation **77**
8.1 Event notification 77
8.2 Symbol and value resolving 78
8.3 A victim-thread for safe memory inspection 79
8.3.1 Concept and architecture 80
8.3.2 Usage by the CTracer 80
8.4 Resulting monitoring data 81

9 Usage **83**
9.1 Stand-alone execution for custom binaries 83
9.2 Integration into JEM 85
9.2.1 Configuration and invocation 85
9.2.2 Insertion of CTracer-data into JEMs data stream . . 86
9.2.3 Augmentation of the JEM-Ganga-Integration 86
9.3 Application for HEP Grid jobs 87
9.3.1 Preparation of the user application 88
9.3.2 Activation and configuration in Ganga 88
9.3.3 Results and interpretation in an example run 89
9.4 Performance impact 90

10 Conclusion **92**

IV A real time trigger mechanism **93**

11 Concept and requirements **93**
11.1 Extendible chunk format for monitoring data 93
11.2 Chunk backlog and tagging 94
11.3 Inter-process communication in JEM revised 99

12 Architecture and implementation **103**
12.1 General JEM architecture changes 103
12.2 High-throughput shared ring buffer 104
12.2.1 Working principle 105
12.2.2 Ring buffer operations 109
12.3 Triggers and event handling 111
12.3.1 Trigger architecture 111
12.3.2 Trigger scripting APIs 112
12.3.3 Example trigger scripts 115
12.4 Memory management 116
12.4.1 Management of shared memory 116
12.4.2 Shared identifier cache 116

13 Application in JEM **119**
13.1 Changes in JEM execution 119
13.2 Refactored Ganga-JEM integration 120
13.3 Refactored CTracer 120

14 Testing **123**
14.1 Functional tests 123
14.2 Performance tests 125

15 Conclusion **127**

V Summary 129

16 Use cases and testing 129
16.1 Testing framework . . . 130
16.1.1 Unit tests . . . 130
16.1.2 User tests . . . 131
16.2 Use cases . . . 131
16.2.1 User perspective: Hanging Grid job . . . 132
16.2.2 Admin perspective: Excess dCache mover usage . . . 133

17 Outlook 135
17.1 Open questions . . . 135
17.2 Further development . . . 136

18 Conclusion 140

VI Appendices 141

A Module structure 141

B Example trigger implementations 142

C List of Figures 145

D List of Tables 146

E List of Listings 146

F Acronyms and abbreviations 147

G References 149

Part I
Introduction

There has been one development in science in the last decade that affected almost all fields of knowledge, ranging from humanities, economics and social science to natural sciences. This development is the huge rise in the amounts of data that are created and have to be handled, and in the complexity of the operations and analyses the scientist has to perform on that data. Both issues lead to the conclusion that the single most important aspect of modern science probably is computing.

Computer-assisted analysis and data processing now is an integral component of the scientific process. The need for computing power and data storage capacities rises exponentially, without a conceivable limit. New methods and forms of computing are developed regularly to cope with the increasing requirements of scientists. This process is paralleled with a similar growth of computing in the industry; both areas depend on each other and new developments usually, eventually are shared between them.

In addition to the need for raw computing power and data storage, an equally growing requirement to computing nowadays is communication, or data transfer. The large amounts of data generated and stored have to be distributed all over the world to facilitate the international scientific process.

The newest answer to these challenges chosen by scientists is the Grid, a novel means to couple compute clusters and data centres from all over the world into one logical, giant supercomputer. Handling a computing compound of this size requires extensive monitoring; especially the supervision of one's compute jobs - program executions together with their associated data - is a topic of increasing importance. This is because a computing Grid, from the point of view of the end user, behaves like a batch computer: A compute job, after being submitted, is invisible to the user until it finishes - or until it fails.

With the emergence of new Grid technologies and the launch of a new generation of large-scale scientific experiments, the necessity of job monitoring is evident. The technology presented in this thesis contributes to the developing monitoring infrastructure in Grid computing, and by doing so, aids the Grid users in their daily work. Since the amount of monitoring data created itself soon reaches amounts that can not be handled, it must be controlled and limited to reasonable levels; this meta-monitoring issue is attended here as well.

Acknowledgements

The creation of a work like this one requires the help and support of numerous people: colleagues, friends and family help in "staying on track" and critically question disputable points, give advice and opinions, and - not least - help in maintaining the necessary motivation. With many supporting hands not mentioned, I'd like to name a certain few who especially helped me.

First and foremost, I'd like to thank the two persons making my work for this PhD thesis possible in the first place: At the University of Wuppertal, my doctoral adviser **Professor Dr. Peter Mättig**, for his guidance and support, for being the primary contact concerning CERN, and for supervising my examination; and at the University of Applied Sciences Münster, **Professor Dr. Nikolaus Wulff**, for the continued support after my graduation there and for providing the possibility to stay in Academia and aim at a PhD by establishing the cooperation between the two universities.

At the daily work at the University of Wuppertal, my mentor and contact is **Dr. Torsten Harenberg**. Having supervised the former works - several Diploma- and PhD theses - on the monitoring software already, he has a valuable overview over the history of it, and in his function as the Grid site administrator of the Wuppertal Tier-2 centre, his in-depth knowledge of Grid technology and distributed computing proved a vital input for my work. I'd like to thank him for all this, and for being a good friend at the same time.

In place of all members of our working group in Wuppertal, I'd like to thank **Dr. Klaus Hamacher**, **Dr. Joachim Schultes** and **Dr. Marisa Sandhoff** for being the people to ask all questions concerning physics, and for criticising and questioning aspects of my work, thereby assisting me in its refining process.

I'd like to thank my parents for their love and care, and for paving the way for me whenever there were obstacles.

Finally, and most importantly in her own way, I thank my wife, **Sarah**, for her patience and love in this period of writing that took much of our time, and made me stay in the office longer - especially now, at the end of my work in Wuppertal. Without her continued support, I certainly wouldn't have been able to finish it.

Outline of this work

This thesis has been divided into four main parts. First, the most important fields of knowledge needed to understand the topic and put it into context are introduced. The software developed in Wuppertal is presented next, discussing in-depth the additions that have been implemented for the said software, and drawing a conclusion of what has been done and what needs to follow.

In part one, the context of this work is presented: **High-Energy Physics** (HEP) and the associated software packages and applications. This is to show of what complexity this software is, and how massive the amount of data that has to be processed by it. The computing environment used in HEP, the **Grid**, is presented next, as well as the reasons for the necessity of monitoring applications running in this environment.

Part two ties in with the presented context by discussing the term **Job Monitoring** and by distinguishing it from site monitoring in the context of the Grid. The software solution developed in Wuppertal since 2005, as answer to the aforementioned necessity of monitoring, is then presented and brought into context. By explaining the structure and functionality of this monitoring software, the applicability and usage of it is shown. However, the software has had major shortcomings before this work has been tackled; these are discussed at the end of this part and lead to part three and four, where the additions to the software are shown together with the new concepts used.

In the third part of this work, a novel technology for extending the monitoring of Grid Jobs into formerly "blind spots", the execution of binary modules, is presented. The motivation for the development of this **CTracer** as well as its core concepts are discussed and its architecture explained. Finally, its integration into the existing monitoring framework is shown and examples of basic usage are given.

The second addition, a **real-time trigger system** for data reduction, is presented in the fourth part of this thesis, after the need for it has been shown and the changes to the monitoring software's core necessary for the addition of this system have been discussed. The introduction of this trigger system changed the inner workings of the monitoring software massively - this is reasoned and explained, and leads to the description of how the trigger system is implemented. Example trigger scripts conclude this part.

The final part summarizes the changes and additions to the monitoring software and gives an outlook into further possible development and open questions which can be addressed by following works.

1 Context: On High Energy Physics (HEP)

High energy physics is a discipline of science trying to explain the universe at its smallest scale: the subatomic particles with their properties and behaviours, and the interactions between them. It got its name because to investigate these particles, high amounts of energy have to be spent and focussed in a very small space to give the highest possible energy density. This introductory section gives a short overview over this field of science.

Despite the progress the field of high energy physics has seen in the last decades, the nature of the particles and forces that constitute every existence is not yet fully understood. To find new particles, new relationships between particles, and to create even better models of how the universe works, huge and complex experiments are created by scientists in international collaborations. A description of the experiment dealt with in this work follows in section 1.2.

1.1 Current research in HEP

To illustrate the current research in high energy physics, and to reason the motivation behind this research, the status-quo is briefly discussed here: The Standard Model of Particle Physics. Subsequently, some interesting topics investigated at the moment by international physicist collaborations in HEP are given. For more information on the Standard Model, recommendable further reading is given in[1, 2] .

1.1.1 The Standard Model

To the current understanding, all our existence is built of a few fundamental particles. Several orthogonal classifications of those particles exist with which one can try to bring order in the ever growing list, and which often predict further particles that are searched for in experiments of international scientific collaborations.

The Standard Model of particle physics is, as its name suggests, the most widespread accepted model of particle classification today.

Quantum numbers

Fundamental features and states of particles in the Standard Model are expressed by numerical values called **quantum numbers**, provided with a unit. Each type of particle has an own set of quantum numbers describing its properties and behaviour.

It is important to note that in HEP, different units for concepts like the mass and distances are used than the widespread used SI units. This is because the scales one handles here differ from human-perceivable scales by amounts large enough to make the usual units (like Gram, Meter, Joule, etc.) difficult to work with.

Because mass and energy are equivalent (as shown by Albert Einstein in 1905), the unit for mass used in HEP is a unit of energy: The Electron Volt (eV), defined as the amount of energy an electron perceives when being influenced by an electric potential of one Volt. Larger quantities are shortened by the usual prefixes (kilo, mega, giga): keV, MeV, GeV, and so on[1].

The list of known quantum numbers grew over the last decades, as more and more properties of subatomic particles were discovered. Some of these quantum numbers lead to exclusion- or symmetry-rules that have been used to identify - and search for - missing particles in the Standard Model.

One such property found in 1923 by O. Stern and W. Gerlach, called **spin**, can be used to distinguish two major particle groups: **fermions** and **bosons**. Fermions, the matter constituents, are what all matter in the universe is built of. They have a half-integral spin. The interactions between particles are mediated by bosons, particles with integral spin.

Further properties of particles are a number of **charges**, corresponding to different interactions. If a particle has no appropriate charge, it does not participate in the associated interaction.

For each particle exists a corresponding anti-particle with quantum numbers bearing the opposite sign. There are also particles which are their own anti-particles.

Fermions - the matter constituents

All matter we know consists of **atoms** (gr. $\alpha\tau o\mu o\varsigma$, "not dividable"). Postulated around 400 b.c. by the Greek philosopher Demokrit, atoms were for a long time thought of being fundamental particles without any inner structure. Only in 1910, physicist Ernest Rutherford discovered that the majority of the atoms' mass is concentrated in a massive core, the **nucleus**, of comparatively small size (in the order of $10^{-15}m$). The core is orbited by **electrons** (e^-), defining the atom's diameter[2].

The nucleus carries a positive electric charge, and the electrons in an atom carry a negative electric charge of usually the same amount, making the atom as a whole electrically neutral[3].

Scientists soon discovered that the nucleus is not fundamental, either - it consists of two types of **nucleons**, the positively charged **proton** (p) and the neutral **neutron** (n). The nucleons also can be divided, as they are built of even smaller particles hypothesized in 1964, dubbed **quarks**, and evidenced later by numerous experiments.

There are six quark flavours: **down**, **up**, **strange**, **charm**, **bottom** and **top**. They are listed with their main quantum numbers in tab. 1.

[1] Technically, the eV still is a simplification, as the correct unit for energy interpreted as mass would be *Electron Volt over c squared* - $\frac{eV}{c^2}$. It is common practice to normalize c to one, like used in this work.

[2] roughly $10^5 fm$ (Femtometer or "Fermi" - $1fm = 10^{-15}m$) $= 10^{-10}m = 1$Å (Ångström)

[3] There can be, however, variants of atoms with a non-zero electric balance. These are created if atoms lose or gain electrons or protons and often are not stable.

Nucleons consist of only two quark types: up and down. Protons are a bound form of two up- and one down-quark, and neutrons contain two down- and one up-quark. Particles built from three quarks like this are called **baryons**. Heavier baryons built of other quarks (c, s, b) exist, but are not stable and quickly decay to lighter baryons.

There also exist bound forms of two quarks - always a quark and its anti-quark - called **mesons**. The group of all particles built of quarks (baryons and mesons) together is called **hadrons**. Free, lone quarks seem not to exist, a fact that can be explained by a quantum effect called **confinement**: The quarks are confined in the hadrons. If a quark is moved out of a hadron - for example by hitting it with another, high-energetic particle - a new quark-antiquark pair is created between the hadron and the free quark, filling the spot of the missing quark in the hadron, and turning the free quark into a meson. As the quantum numbers of the quark and the antiquark created add up to zero, the law of the conservation of energy (as well as other conservation laws) is not violated in this process.

There are six **anti-quarks**, and together with the **positron** (e^+), the antimatter twin of the electron, **anti-atoms** can be formed. These antimatter constituents, however, are very short-lived when brought near matter, as matter and antimatter annihilate and turn into energy on contact.

In some particle reactions, one seems to observe discrepancies like missing energy, mass, impulse and so on. In the Standard Model, those discrepancies are explained by defining new particles which, for example, carry the missing energy away from a measured particle combination. Often, later on, those hypothetical particles are detected and thus proven to really exist.

One example of a particle of this family is the **neutrino**. Being a particle without electrical charge, only participating in weak interactions, and with almost no[3] mass, this particle is very hard to detect; only if it interacts with other particles by means of the weak interaction, the resulting new particles can be measured. Experiments have shown, however, that neutrinos exist. They can, for example, explain the missing impulse in beta decays of neutrons into protons, a process taking place in our sun.

	mass ($\frac{MeV}{c^2}$)	electric charge (e)	spin
up	1.5	$\frac{2}{3}$	$\frac{1}{2}$
down	3.5	$-\frac{1}{3}$	$\frac{1}{2}$
charm	1270	$\frac{2}{3}$	$\frac{1}{2}$
strange	104	$-\frac{1}{3}$	$\frac{1}{2}$
top	170900	$\frac{2}{3}$	$\frac{1}{2}$
bottom	4200	$-\frac{1}{3}$	$\frac{1}{2}$

Table 1: The six quark flavours

There are three neutrinos known today, corresponding to three similar electron-like particles only differing in their mass. Those particles, the electron, muon (μ) and tau (τ) together with their neutrinos (ν_e, ν_μ, ν_τ) form the group of the **leptons**, the second group of fermions constituting matter besides the quarks. Interestingly, both quarks and leptons can be divided into three groups, called **generations**, grouped by comparable mass ranges, bearing similar quantum numbers (see fig. 1).

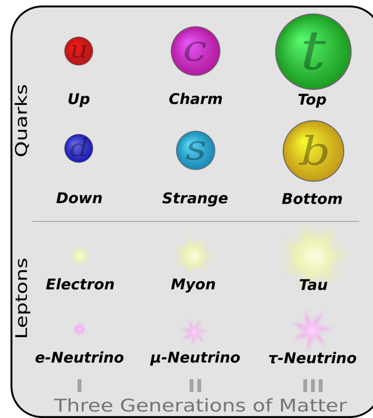


Figure 1: The three generations of known fermions

Bosons - the force carriers

There are four fundamental interactions, two of which are visible in our day-to-day life. The **electromagnetic interaction** that we know as light, magnetism, radio waves, x-ray etc. acts on the **electromagnetic charge** (or short: electric charge). It also makes charged particles induce a force on each other, attracting differently-charged particles and repulsing equally-charged ones, named Coulomb-force. The **gravity**, causing all massive objects to attract each other, is not yet fully explained in the context of the Standard Model (see section 1.1.2).

The two others are not directly visible to us. The **strong interaction**, holding the nucleus together (without it, the nucleus would not be stable, as the positively charged protons experience the Coulomb force, pushing them

interaction	mediating boson	relative strength
strong	gluons	1
electromagnetic	photon (γ)	10^{-2}
weak	Z^0, W^+, W^-	10^{-13}
gravity	(*graviton*)	10^{-39}

Table 2: Bosons mediating the fundamental interactions

apart) corresponds to a number of so-called **colour charges**, and the **weak interaction**, responsible for certain particle decay processes, to the **weak charge**.

Common to all interactions is that they are mediated by particles with an integral spin, called **bosons**, carrying charges from one participating particle to the other. They are shown in tab. 2.

The particles mediating the strong force are called **Gluons**. The electromagnetic interaction is mediated by **photons** (γ). The bosons of the weak interaction are the neutral Z^0 and the charged W^+ and W^-. The **graviton** as the mediating boson of the gravity still is a hypothetical particle that has not been evidenced yet.

1.1.2 Examples for open questions

Although the Standard Model is an excellent model in describing the building blocks of the universe and predicting physical processes like particle interactions, there still are details that yet are not understood, for example:

What constitutes the majority of the universe's mass?
Cosmological studies have shown that all visible matter - stars, galaxies, nebulae - in the universe combined only amounts to roughly 5% of the universe's mass. The largest fraction of the mass is assumed to be **dark matter** and **dark energy**. Scientists try to explain what dark matter and dark energy are, and why there is such a large imbalance between them and the visible, "ordinary" matter.

Where has the Antimatter gone?
The universe is built of matter - this can be seen easily. Theories describing the origin of the universe, however, predict equal amounts of matter and antimatter to be created. The question is why there is no natural antimatter to be found - or even why the universe still does exist at all in its present form, and all matter has not turned into energy by annihilation with antimatter in the universe's infancy.

How can the mass of particles be explained?
Despite being measured quite precisely, the mass of particles cannot be *explained* at the moment. In other words, the Standard Model currently contains no model for the **origin of mass**. One hypothesis in this context is the so-called **Higgs Field**, which would require the existence of an accompanying **Higgs particle**. This heavy boson is assumed to exist somewhere in the mass range of 117 to 153 GeV,[4] and there is hope to spot it in current experiments, like the ones at the Large Hadron Collider.

1.2 CERN and the LHC

The samples one has to examine in HEP are much smaller than the wavelength of light, and so cannot be seen through a microscope. Also, the particles are too short-lived to be detected directly: They are not stable and decay into lighter particles within fractions of a second. It is only possible to detect and measure these decay products. As most of the particles normally don't even exist in the free universe, new means of examination had to be invented. They now are forced into existence by spending very large amounts of energy, for example in accelerating fundamental particles like nuclei and making them collide.

To answer the remaining questions in particle physics by these means, numerous scientific efforts are going on. The experiment this work deals with is located at the European Centre for Nuclear Research, **CERN**.

Located near the Jura mountains next to Geneva, on the Franco-Swiss border, CERN was founded in 1954 by a group of twelve European countries to centralize and facilitate nuclear research at an international scale.

CERN hosts numerous experimental activities, nowadays not limited to nuclear research but also tackling other fields of physics; the main focus has shifted, though, to particle- and high energy physics, part of which the biggest experiments at CERN are: particle accelerators and colliders.

The technology of those experiments has come a long way since its infancy in the 1930s, when R. J. van de Graaff invented the first linear accelerator.[5] Nowadays, the machines accelerate the particles on circular paths, spanning circumferences in the range of kilometres. They use superconducting magnets to create high magnetic fields and require hundreds of engineers and physicists to control and steer them. The currently biggest and most powerful machine is the Large Hadron Collider at CERN.

1.2.1 The Large Hadron Collider

A hundred meters below the surface on average, with 27 km circumference crossing the Franco-Swiss border, the Large Hadron Collider (LHC) is the most advanced particle accelerator and collider at present. It boosts the energy of hadron packages - called **bunches** and each consisting of about $115 \cdot 10^9$ protons[4] - to TeV levels, using the previous generations of accelerators at CERN as preaccelerators, before letting them collide at four spots, inside the four big experiments ALICE, ATLAS, CMS and LHCb.

Each of those experiments was planned and built to answer questions like the ones touched in the previous section, and is run by an international collaboration of physicists and engineers. A number of smaller experiments dealing with single, specialized questions also exists and contributes to the LHCs scientific portfolio.

[4] also heavy ions can be accelerated in the LHC and be brought to collision, for different kinds of experiments

As the remainder of this work deals mostly with the ATLAS experiment, this detector and its computing infrastructure will be described in more detail in the following sections.

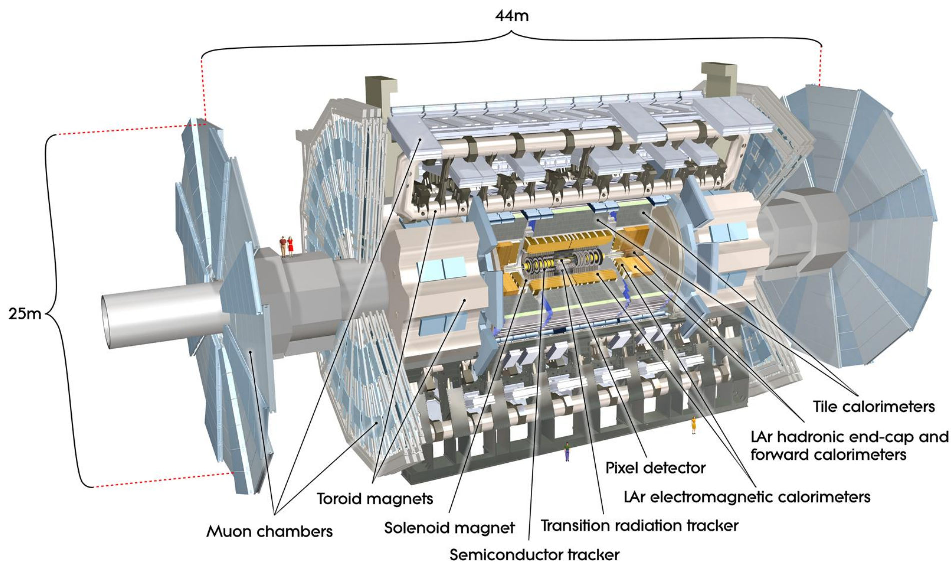


Figure 2: Schematic display of the ATLAS detector[6]

1.2.2 The ATLAS Experiment

ATLAS is a general-purpose particle detector consisting of several subdetectors, each registering and measuring certain aspects of the subatomic particles created in proton-proton-collisions. The subdetectors' results are combined and interpreted to reconstruct the reactions that took place in the **primary vertex**, the spot where the proton bunches intersected and protons were brought to collision.

ATLAS is shown schematically in fig. 2. It is 44 meters long, has a diameter of 25 meters and weights about 7000 tons. Its main parts, ordered from the inside to the outside, are described in the following list.

The Inner Detector
This is the detector that lies closest to the beam pipe - the small, vacuumized tube in which the particle bunches travel and eventually collide. It consists of the first three detector subsystems, and is pervaded by a strong magnetic field created by large solenoid magnets. Because it lies so close to the primary vertex, the systems of the inner detector have to be very resistant to radiation, and the transmission of the detector's data out of ATLAS has been an engineering challenge.

- The **Pixel Detector**, an array of silicon semiconductor sensors arranged in three layers, only a few centimeters away from the interaction point, works like a big digital camera to detect particles crossing on their path outwards from the primary vertex into the outer detectors of ATLAS. Since it lies inside a magnetic field, the curvature of the particle's paths can be determined, leading to the reconstruction of the particle's electric charge and momentum. The Pixel Detector is only able to detect particles that ionise matter by means of the electromagnetic interaction.

- Around the Pixel Detector, the **Silicon Tracker** works at the same principle, but consists of extruded silicon bars instead of small, point-like pixels. It adds further hit locations to the particle tracks, and so contributes to the measuring of the particle's impulse and charge.

- The outmost inner detector, the **Transition Radiation Tracker**, adds further coordinates to charged particle's tracks.

Calorimeters
The calorimeters work by stopping the particles, converting their kinetic energy to other forms of energy which can be measured. ATLAS' calorimeters are arranged around the inner detector, intersected by the inner detector's solenoid magnet coils.

- The **Electromagnetic Calorimeter** stops particles participating in the electromagnetic interaction and determines where their kinetic energy was deposited. By doing so, it measures the kinetic energy the particle has had, and adds coordinates to the particle's track, verifying the extrapolated track given by the inner detector. Since it only detects relatively light electromagnetic interacting particles like e^-, e^+ and γ, and other, high-energetic particles cross them without being stopped, it helps in classifying the particles created in the proton collisions.

- Next in the stack of ATLAS' detectors is the **Hadron Calorimeter**, that detects and measures hadrons that pass the EM Calorimeters. It is filled with strongly interacting material to let these heavier particles produce measurable decay products, and additional layers to stop them. This way, it allows one to classify and investigate the particles at the same general principle as the EM Calorimeter does.

The Muon Chambers
These outmost - and largest - subdetectors finally detect and measure muons created in the collisions which, because of their high mass, are not stopped in the Calorimeters and thus can be separated from other leptons, like electrons. These muon signals are useful in selecting “interesting” candidates out of the high number of collisions.

By combining the information of all the detectors, different particle types can be distinguished and their movement backtracked to the point of their origin. Sophisticated mathematical algorithms are used to infer those, the particles' *vertices*. It is possible this way to reconstruct what happened in the primary collision, which intermediate particles were created, and when - and where - they decayed into what secondary (tertiary, ...) particles.

The data created by this means, its amount and the necessary processing of this data to be able to analyse it, is discussed in the next section.

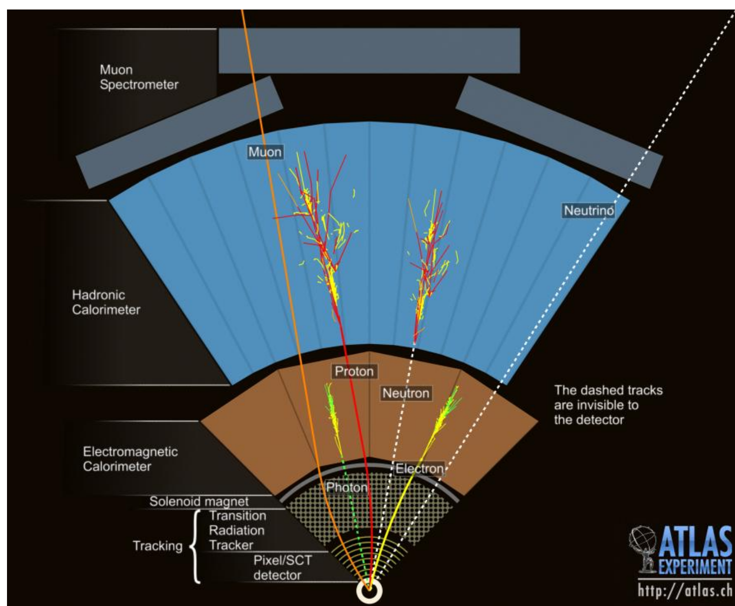


Figure 3: Cross section of ATLAS with a showcase event, showing tracks of different particle types and where they interact with the different subdetectors[7]

1.2.3 Data flow in ATLAS

The multitude of sensors and detectors in ATLAS create large amounts of data for each single event (particle collision). Pixel hits and tracks in the inner detectors, calorimeter data and spatial information, particle classification, vertex data and more is recorded and must be transferred out of ATLAS to be stored and processed, as obviously this is not possible inside the detector itself. For this reason, each detector is connected to the "outside world" by data channels (optical and copper links).

In sum, ATLAS has about 90 million data channels, from which around 90% are used by the pixel detector. On top of those, infrastructure channels like power supply, cooling and steering (combined in the Detector Control System (DCS)) constitute to ATLAS' outside connections. Through these channels, when read out by the Trigger and Data Acquisition System (TDAQ), each event creates about 1.6 Megabyte of data.

As the frequency of events in ATLAS is very high (when working at maximum design parameters, ATLAS observes 40 million bunch crossings per second; at an expected event rate of 20 events per bunch crossing, this sums up to one billion events per second), the amount of data recorded would not be handleable if not reduced. Estimates of around 60 TB raw data per second were calculated. This would fill 100.000 CDs per second, or is equivalent to 50 billion telephone calls at the same time.

1.2.4 Real-time data reduction: Triggers

To deal with this enormous amount of data (it would sum up to 1.8 million PB per year), a three-tier trigger system was created, that selects "interesting" events out of the stream and so reduces the amount of data needed to be stored.

- The **Level 1 Trigger** is implemented in hardware. It consists of massive paralleled pipeline processors (ASIC FPGAs) that need to decide whether to keep an event or not in a time scale of around 2,5 microseconds. The L1 trigger reduces the outgoing data rate to 100.000 events per second.
- The software-based **Level 2 Trigger** has several milliseconds to decide whether to drop or keep each event. It consists of a computing farm of around 1000 CPU cores. It reduces the rate of events to 3000 / s.
- Finally, the **Level 3 Trigger** further refines the event selection using physics parameters and reduces the data rate again 15fold to the final rate of 200 events per second. It is implemented in software, running on a computing farm of 3400 CPU cores.

The trigger system results in an amount of raw data that is to be stored of approximately 320 MB per second / 3.2 PB per year - at a reduction factor of 180.000.

After the raw data stream has been reduced by the trigger system, the detector data is converted to higher specialized data formats more suited to physics analysis, further reducing their size[5]. The software that is used to process and analyse these converted data is described next.

1.3 Software in HEP

The usage of computers in science to solve mathematical problems, analyse data, find patterns in data and to perform similar tasks became widespread since the 1960s. Accordingly, software written for those purposes exists since then, and was since then subsequently improved (often rewritten) and adapted to new requirements. Eventually, large parts of the software have matured and are available as well-established multi-purpose libraries for physicists to utilize in their analysis projects.

[5]The raw data, however, is also stored and archived on long-term mass storage facilities.

Mostly, in ATLAS, those libraries are written in C++ and bootstrapped and configured by Python- or Bash-Scripts (this fact will become important when considering real-time monitoring of such software, see part II - **Job Monitoring**).

Among the tasks those software packages nowadays must cover - as efficiently as possible - are the applications of the Athena software framework (see section 1.3.3 for a more detailed description of those applications):

- the **conversion of data** from raw recorded physics data into easy-to-use derivative formats,
- **reconstruction of the collision events**, combining the data of the different subdetectors to reconstruct the particles crossing the detector, and
- the **analysis of those measured and reconstructed data**, trying to infer the physical processes that happened at the interaction point,

as well as the second big computing topic in HEP: The simulation of physical processes (see section 1.3.2), consisting of

- **"Monte Carlo" production**, named after the centre of gambling because of the random nature of the simulated physical events, and
- the geometrical and functional **detector simulation**.

As all these processes - data conversion, reconstruction, analysis and simulation - are substantially CPU- and Memory-intensive, and the datasets they are dealing with are too large to be stored on the single user's workstation, they usually are performed on a **Grid** (see section 2.1) instead. Some of the software packages used in HEP, locally and on a Grid, will now be described in more detail.

1.3.1 High-performance maths and core services: ROOT

The functionality that HEP-software has to provide, as described in the previous section, commonly involves mathematical tasks like the creation of histograms, the fitting of values to mathematical functions, multidimensional and/or multivariate analysis of data, and the access of large datasets. The framework of choice in the HEP community that provides such tools is **ROOT**.[8]

ROOT is an object-oriented framework based on a high-performance data storage library that provides solutions for the aforementioned use cases interactively - allowing the user to execute C++ statements on a shell, introspective analysis- or Monte Carlo data - or non-interactively on the Grid. Amongst the most-needed tasks physicists use ROOT for are the creation of histograms and plots visualizing analysis data, tweaking the parameters (like binning, data cuts, etc.) in real-time to achieve publication-ready images, and the interactive fitting of curves onto data to infer interesting correlations.

On top of that, HEP-specific classes, definitions and functions have been implemented in ROOT, allowing the user to perform HEP-related work with the least possible preparation needed.

1.3.2 Event generators and detector simulation tools

Before any detector is designed, let alone built and ran, the physics processes that are expected to occur are being simulated. This simulation process has been dubbed Monte Carlo (MC) by the scientists and is the second big computing task in HEP after the actual analysis of "real" data.

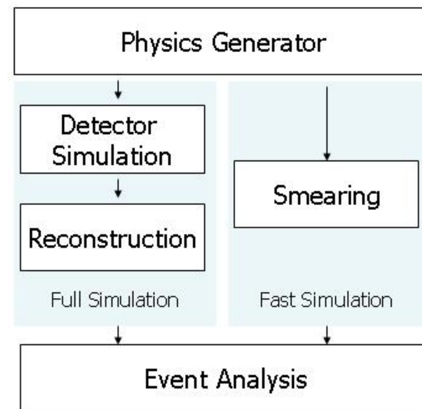


Figure 4: Typical Monte Carlo simulation chain[9]

The Monte Carlo simulation - also called **production** because physics event data is centrally produced by it - is further continued after the experiments have been built, to be able to calibrate them and, most importantly, to be able to understand and interpret the data generated by the detectors. Conversely, new understandings in physics are used to improve the Monte Carlo algorithms. In this, physics research using particle detectors like ATLAS is an iterative process: Physics processes are simulated, detectors and sensors are conceptualized, planned and built according to the simulated expectations, and during the detector's whole lifetime, MC data is produced to calibrate the detectors and to compare the simulated with the measured data, allowing to infer experimental results.

The simulated data eventually is created in the same derivative formats as is the original, measured experimental data. This enables the described iterative process, as both simulation and reconstruction of the data can be verified against each other.

An overview of Monte Carlo event generators used within the ATLAS collaboration is given in [10] . Examples are Pythia,[11] Sherpa[12] and AlpGen.[13]

Furthermore, complex geometrical algorithms are used to model and simulate the detector itself and how the particles interact with it (including its supporting structures and infrastructure like cooling, power supply, detector monitoring, etc). This detector-simulation is tackled by software packages like Geant4[14] and atlfast.[9]

An exemplary Monte Carlo simulation process, including the detector simulation to add detector-specific effects of particles crossing material, is depicted in figure 4.

1.3.3 ATLAS' main physics analysis framework: Athena

Athena is a software framework for physics analysis on top on a broader framework - called Gaudi[15] - that was conceived by the LHCb collaboration and is now used (among others) by ATLAS. It is provided to the users as a library, allowing them to base their custom pieces of code on common "building blocks" that provide infrastructural services like dynamic library loading, component instantiation, configuration and a centralized event loop, as well as advanced reusable services - base implementations of the services discussed in section 1.3.

The modules delivered to the users as part of the Athena distribution, forming a foundation to base their own physics algorithms on, cover tasks like:

Conversion of Data
Physics analysis, most of the time, does not deal with the raw data as it is produced by the experiments (see section 1.2.4). Instead, the data first is converted into leaner, more specialized formats, that are more suitable for analysis purposes. Furthermore, the data is augmented with meta-data during this conversion process, with which the analysis code can make precise assumptions of, for example, how the detectors were calibrated at the time of the event.

The conversion of the data usually is a multi-step process with several intermediate data formats providing different levels of completeness and a different amount of meta-data, suitable for different classes of physics analysis. For ATLAS (and similarly for the other LHC experiments), the conversion steps also are distributed in a hierarchical structure, the ATLAS Data Distribution Model (see 2.2.4), involving the international members of the collaboration and their institutes' computing centres to spread and parallelize the needed work.

Reconstruction of the Collision Events
Using the raw- and converted data, one of the main tasks of HEP software is to reconstruct what happened during the particle collision. As described in section 1.2.3, what is recorded in experiments like ATLAS are detector-specific information like

- where and when did some (secondary, tertiary...) particles cross the Pixel Detector / Silicon Tracker?
- how much transition radiation was measured, and where?
- how much energy was deposited inside the calorimeters, and where?
- where and when did particles cross the Muon Chambers?

Taking these low-level information, the reaction at the collision spot has to be reconstructed. This includes trying to find positions in the detector where

particles were created; these are called **vertices**. Particle hits are combined to **tracks** and their curvature measured, hinting at the particle's electric charge. If many secondary, tertiary, ... particles are created in one direction, forming a "shower" of hits in that direction, they are combined by reconstruction into a **jet**. Also, missing energy in some direction (called missing transversal energy or **missing** E_T) is determined, hinting at undetected particles like neutrinos.

Analysis of Data
The data created by the detectors, converted into derivative formats and fed through the process of reconstruction, is then used by physicists to perform specialized analyses. For this, physicists develop their own algorithms based on ATHENA. Usually, the central task of those algorithms is to examine event after event, building internal data structures that are eventually evaluated, trying to infer results supporting theoretical assumptions.

For this, ATHENA provides the skeleton code that loads the block of data the user algorithm is to process, calls the algorithm's process-method consecutively with each physics event, and then finalizes the algorithm, letting it perform its aggregation and evaluation functionality. The end result of the execution of those analysis-algorithms is often visualized in graphical plots and histograms; the main software environment that is used for this purpose is ROOT (presented in section 1.3.1).

The user algorithms utilizing the ATHENA services are written in C++ and configured by **job option files** written in Python. ATHENA presents itself to the user as runnable Python script with a collection of predelivered shared object files, Python job option files and shell scripts for environment setup. The typical work flow of an ATHENA execution can be divided into four phases, characterized by execution of modules written in different scripting- and programming languages:

1. **Environment Setup**
   By sourcing the setup shell scripts, the ATHENA execution environment is prepared. This includes the definition of binary search paths, the preparation of Python module search paths and the preparation of environment variables ATHENA refers to during execution.

2. **Data Staging**
   The dataset the user algorithm is to run over needs to be loaded to a local storage that can be accessed in real time by ATHENA, usually a local hard disk, as the storage systems the data is deployed to usually don't allow real time access (see section 2.2.4).

3. **Algorithm Execution**
   According to the job options, ATHENA iterates over the physics event data contained in the input dataset, performing the user algorithm's functionality on each single event, and creating output data.

4. **Data Staging**
   The output data has to be packed into an output dataset, the dataset registered and uploaded to permanent storage. The stage-in and the stage-out of the data is executed by highly optimized file transfer services (see section 2.2.2 for examples of services used in HEP).

Preparation of the user algorithms
Before a user algorithm can be used for one of the aforementioned purposes, it obviously has to be written and translated into machine code by a compiler. This, again, depends on the availability of the Athena framework on the user's development machine. A utility named Configuration Management Tool (CMT) is used in this context for source- and revision control, package configuration and compilation management.[16] It automatically resolves the user algorithm's dependencies, checks out the needed Athena-packages from source control and initiates the build process. CMT itself is set up with a collection of shell scripts and binary tools, internally using the Revision Control Systems (RCSs) CVS[17] or SVN[18] for code checkout.

After the code was checked out and the user algorithm based on those packages has been written, CMT again is used to build and package the custom algorithm library that then can be deployed and run - locally, on an institutes batch system or even on a world-wide distributed computing Grid.

2 Grid Computing

Since the ATLAS collaboration is an international union of physicists, the users of the data created by the detector come from all over the world. Consequently, measures have to be taken to allow physicists at their home institutes to access the ATLAS data, and to run their analysis algorithms against those data with the least-possible effort. The solution chosen by the LHC collaborations is called a **Computing Grid**.

2.1 Overview

Since the 1960s, when large-scale computing became widespread in economy and science, there were efforts to allow shared access to computing facilities. At first, this shared access meant taking turns in compute time on a large and expensive batch computer only big organisations and universities could afford. Users had to prepare their algorithms in beforehand, book computing time by submitting their code to the batch computer operators, and fetch the results after they were processed. Usually, this took hours, if not days, with the success of the calculation not being known. The units of computing "work", submitted originally in form of punch-cards, were called **jobs**.

With the emergence of computer networks, the desire arose to be able to perform those basic steps (submission of computing jobs and fetching of the results) remotely. In 1969, when the University of California took efforts to establish a nation-wide computer network in the USA, the computer scientist Len Kleinrock predicted

> "We will probably see the spread of 'computer utilities', which, like present electric and telephone utilities, will service individual homes and offices across the country."[19]

This was eventually elaborated to the concept of a **Computing Grid** we know nowadays, that is described in detail with its components and services in the following sections.

2.1.1 Definition of the term "Grid Computing"

In 1998, Ian Foster and Carl Kesselman, who are considered the fathers of the Grid, defined this term as follows in their work *The Grid: Blueprint for a New Computing Infrastructure*:

> "A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities"[20]

This definition was replaced in 2000 by a more precise one, stating the organisational nature of a Grid - making it more than just the sum of its hard- and software. In *The Anatomy of the Grid*,[21] Foster and Kesselman state:

> "The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs."

This shows the conceptional distinguishment of a Grid from a mere cluster of networked computers, or a computing centre hosting a so-called supercomputer: The administrative and organisational agreement of the sharing of resources. The resources, in this context, consist not only of computing power, but also data storage, data transportation facilities, information systems / catalogues, credential databases for the identification of users across organisational boundaries, etc.

The name "Grid" refers to the analogy of the power Grid delivering electricity to all homes, accessible by an easy-to-use interface (standardized power outlets) to the concept of a Computing Grid "delivering" computing resources to all users via a similarly easy-to-use interface. In that, the complexity of a heterogeneous network of interconnected computing centres and data stores is hidden from the users behind a level of abstraction, allowing one to use the distributed computing resources like a single (batch-) computer.

To separate the Grid from other distributed computing environments like **Cloud Computing** (remotely accessible, paid-for multi-purpose computing resources hosted by single large organisations), **Desktop Grids** (coordinated exploitation of unused compute cycles of desktop computers) and other distributed services like Peer-2-Peer file sharing networks and SAAS[6] frameworks, Ian Foster declared a three-step checklist in his article *What is the Grid? A Three Point Checklist*, reserving the term "Grid" to a distributed computing infrastructure that

> "*coordinates resources that are not subject to centralized control...*
> A Grid integrates and coordinates resources and users that live within different control domains—for example, the user's desktop vs. central computing; different administrative units of the same company; or different companies; and addresses the issues of security, policy, payment, membership, and so forth that arise in these settings. Otherwise, we are dealing with a local management system.
>
> *...using standard, open, general-purpose protocols & interfaces...*
> A Grid is built from multi-purpose protocols and interfaces that address such fundamental issues as authentication, authorization, resource discovery, and resource access. As I discuss further below,

[6]= "Software as a service", web-distributed applications, often browser- (rich client-) based

> it is important that these protocols and interfaces be standard and open. Otherwise, we are dealing with an application-specific system.
>
> *. . . to deliver non-trivial qualities of service.*
> A Grid allows its constituent resources to be used in a coordinated fashion to deliver various qualities of service, relating for example to response time, throughput, availability, and security, and/or co-allocation of multiple resource types to meet complex user demands, so that the utility of the combined system is significantly greater than that of the sum of its parts."[22]

Applying this checklist to the other classes of distributed computing mentioned above yields the reasons not to consider them to be Grids:

- In Cloud Computing, the resources are shared subject to centralized control (the cloud provider, charging money for the said usage), and the protocols and interfaces are usually not open.
- A Desktop-Grid usually is specialized on one specific task (for example the search for extraterrestrial intelligence in the radio-background of outer space in the SETI@HOME-project[23]), a fact that also reflects in the specialization of the software used to form the Desktop-Grid.
- Peer-2-Peer file sharing and SAAS are not considered multi-purpose enough to qualify them as "delivering non-trivial qualities of service".

2.1.2 Virtual Organisations

Considering the organisational properties of a Grid, another term that was coined is the virtual organization (VO), that is described by Ian Foster in *The Anatomy of the Grid* as follows:

> "A set of individuals and/or institutions defined by such sharing rules form what we call a virtual organization.",

referring to the sharing of resources mentioned in the previous quote from that article. A VO is an organizational unit, not confined to geographical boundaries, managing the Grid's resources shared within. One physical Grid, the sum of its computing centres, can be shared by several Virtual Organisations.

In the context of the LHC experiment's collaborations, each experiment (ATLAS, LHCb, CMS and ALICE) has a correspondingly named Virtual Organisation in the World-Wide LHC Computing Grid (WLCG). Each of those VOs consists of the member institutes and the participating users of those institutes, providing a structured organisational representation of the collaboration and making the services of the WLCG accessible to the VO members.

2.1.3 Components and services of a Grid

A Grid usually is referred to mostly with the term “Computing Grid”, whereas it provides more services to the users than just compute cycles for user jobs. The services a typical Grid provides are organised on a per-site basis, where a **Grid site** - or just **site** - denotes a single, geographically confined institute / organisation, providing their computing centre to the VO they are affiliated with. Amongst the available services are:

- **Computing**
  The computation work performed on the Grid is one of the central interests in creating such an infrastructure. Most of the time, the compute cycles for the user jobs are provided by a batch system on a computer cluster whose job queues are transparently disposed to the Grid.

- **Storage**
  Storage on the Grid is comprised of long term data storage for the data the user jobs operate on, and temporary storage, for example for job- and logfile-output.

- **Data Transfer**
  This encompasses all transfer of data between the Grid’s participating computing centres, either for data distribution inside the VO or for management tasks like the transmission of the user job to the computing centre it will run on, and transmission of job results back to the user.

- **Job Brokerage**
  A service that can be considered the core of a Grid infrastructure is the job broker, as the user jobs must be evenly distributed over the available computing centres, while also taking care that as few data as possible has to be transferred.

- **Security Infrastructure**
  To provide a VO-wide consistent security infrastructure that ensures the fulfillment of security goals to the users, central services have to exist and be managed. They are usually considered part of the Grid. They consist of a VO-centric certificate authority (CA) and key store servers, but also of **security policies** that all Grid users must agree to adhere to before being allowed to join the VO.

- **Monitoring and Accounting**
  Often, there is the need to track the usage of Grid services by individuals, identified by their security credentials, for statistical or accounting purposes. Also, the Grid health must be supervised and status information provided to the job brokerage service for it to be able to schedule the user jobs intelligently.

- **Information Services / Catalogues**
  The described services usually are implemented as separate, distributed hosts on the Grid's network, bundled in Grid sites and managed by the VO. To be able for users to contact them to use their services, and for components like the job broker to take decisions, a **catalogue** must exist that allows the discovery of said service hosts. Usually, this catalogue is organized hierarchically and can be queried programmatically, remotely, by other services and the users.

The software implementing those services for a Grid is called a **Grid middleware**. There are several middleware implementations by different groups of interest, associated with different fields of science and economy. Some parallel developments in the field of Grid middlewares have recently converged into a number of established middleware standards used by big commercial- and science-related Grids. The middleware of the WLCG, the Grid used by the HEP community of the LHC collaborations, is called gLite. It is presented in section 2.2.2.

2.1.4 Security in the Grid

In a distributed computing environment - a Grid - utilized by a multitude of users from all over the world at the same time, that handles sensitive information, security is a very important issue. Taking a Grid in the context of HEP as example, one can argue that in modern science, each participant - be it a single scientist, or a larger collaboration - has a strong interest in keeping all discoveries secret until they have been published by the participant. On top of that, accounting data and personal information of the Grid's users, too, can be considered sensitive information. Finally, Grid users rely on being able to trust the correct execution of their calculations and the correctness (and integrity) of their physics results.

Security, in this sense, means all of:[24]

- **Authentication**
  This is the secure, remote determination of the identity of a Grid user[7], enabling the denial of access to any Grid-specific service to unknown or unauthorized users.

- **Authorization**
  Once a user has been authenticated, the authorization encompasses the determination of the set of actions the user is allowed to perform on the Grid (for example, submitting a job to the job broker, transferring data to/from a storage service, querying some information from the information system, etc), leading to a denial of all other actions.

[7]Where "Grid user" does not need to be a human being, but can also be an other computer accessing the service via a machine-to-machine interface

- **Confidentiality**
  Confidentiality means: Data not allowed to be read by a third party must be protected, e.g. by appropriate encryption. It also means in the context of a Grid, that the actions a user performed on the Grid should not be traceable by unauthorized third parties.

- **Data Integrity**
  All data stored on a Grid and transferred between Grid services must stay integer all the time. This, obviously, is of special importance to physics data created by not-repeatable physical experiments. Neither should it be possible that data gets corrupted or lost while being stored, nor should it be possible for (malicious) third parties to alter data during transfers.

- **Non-Repudiation** and **Trackability**
  All actions a user performs on a Grid must be traceable, also in retrospect, and the user must not be able to deny having taken the actions[8].

The solution to those requirements chosen most often is a certificate-based Grid security infrastructure.

Users and Hosts of a Grid whose security requirements are tackled by such a certificate-based infrastructure are identified by a distinguished name (DN). This name consists of the name of the user or host, the associated organisation and the country of origin. For each such entity (user or host), a **digital certificate** - adhering to the **X.509** International Telecommunication Union (ITU-T) standard - to sign requests and messages exist. With this certificate, the entity can be identified unambiguously.

It would be tedious, though, to use exactly this certificate for authentication in each step of a user request (e.g. the submittage of a compute job, a file transfer, the request for information about a compute job, etc.), because by doing so, the user would have to enter his security credentials (password or pass phrase) a multitude of times. For long-running tasks, or for tasks involving several services needing to negotiate, this restraint would completely destroy the usefulness of the Grid.

Instead, the concept of a **proxy certificate** was invented. A proxy certificate is a short-living, password-less certificate signed by the user's main Grid certificate to which the user delegates his permissions. With this proxy, on behalf of the user, his Grid requests are signed. Due to its shorter lifetime (usually in the order of twelve hours), there's also no need to be worried about giving the credential's details out of hand.

[8]Note that the traceability of actions does not contradict with the above-mentioned need for confidentiality; the difference is that no arbitrary third parties are able to trace the performed actions.

2.2 The WLCG

While the previously discussed properties and services of a computing Grid are universal, in this section, the concrete Grid built and run by the LHC collaborations is presented.

2.2.1 Overview

The **World-Wide LHC Computing Grid**, WLCG, was designed in 1999. It is built and driven by the member institutes of the LHC's physicist collaborations and consists of computing centres in 34 states, spread over Europe, Northern America and Asia. To organize the Grid - most importantly, the data distribution over the Grid, see 2.2.4 - the participating institutes have been divided in a hierarchical multi-tier organizational structure.

The place where most of the Grid's data is created, CERN, is called the **Tier 0** centre. In each participating country or region, one main computing centre exists, the **Tier 1**. It is responsible for managing its country's Grid efforts, and it stores a copy of the Tier 0 data. Each Tier 1 also stores one specific variant of reprocessed derivative data - this way, the resources of the Tier 1s are most effectively used.

The participating universities of the ATLAS collaboration, where the real users of the Grid reside, and where the data of the Grid is ultimately needed, are called **Tier 2** and **Tier 3** Grid sites. The Tier 2 sites provide their local computing and storage resources to the Grid and query specific needed data from the Tier 1 as requested by the physicists working on the physics analyses. Sites without own local storage (no housed SE) are Tier 3 sites, contributing computer cores to the Grid. Finally, many smaller universities not housing a computing centre that is integrated into the WLCG exist, providing the Grid's services to their users.

In this WLCG infrastructure, today, roughly 63.000 compute jobs are executed and about 2.5 PB of data is moved between its storage elements[25] each day. Currently, the WLCG consists of over 100.000 computer cores, located at over 170 Grid Sites, and roughly 140 PB of data can be stored persistently.[26]

A specialized software environment has been developed for the WLCG to implement its services. The main platform those software runs on is the free Unix derivate Scientific Linux for CERN (SLC)[9], a flavour of Red Hat Linux[10]. The parts of this software environment that deserve special attention in the context of this work now are discussed: The Grid middleware used by the WLCG, the two most important user interface tools within the WLCG and the integration of the ATLAS-specific HEP software into said user interface tools.

[9] See http://linux.web.cern.ch/linux/

[10] See http://www.redhat.com/

2.2.2 The middleware: gLite

The major Grid middleware used in the WLCG is gLite,[27] a free implementation based on the multi-purpose Grid middleware toolkit Globus, the standard of Grid frameworks in science.[28] It was designed and implemented as part of the European Grid initiative Enabling Grids for E-Science in Europe (EGEE).[29] gLite provides the services discussed in 2.1.3.

The user's view of gLite is a collection of small command line utilities allowing an easy access to Grid functionality. They cover common Grid use-cases like the submission of user jobs, the querying of a job's current status, the fetching of job results, the look-up of data sets to use within an analysis job, etc.

In that, gLite's services are easier to utilize than if using only Globus' API (Application Programmer's Interface) directly. The utilities, though, provide no complete round-up over the Grid's services, but provide separate single-problem solutions each. In other words, the user has to remember one command line tool per task, including its parameters. This problem was solved by the introduction of **job management tools** like Ganga (see 2.4.2).

The WLCG, with the help of gLite, provides the following implementations of the previously defined Grid's main functionality and services:

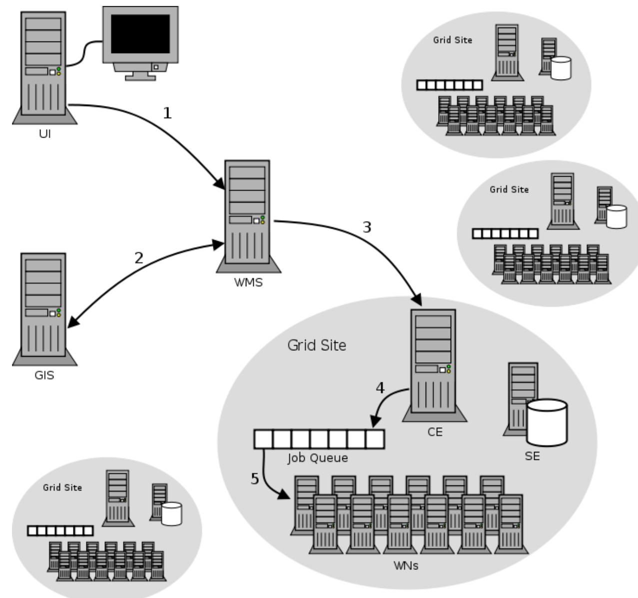


Figure 5: Schematic display of the Grid job brokerage in the WLCG and the involved Grid services. A Grid job is submitted by the user from the UI to the WMS (1). The WMS queries the GIS (2) and picks a suitable Grid site. It sends the job to the site's CE (3) where it is inserted into the job queue (4). Eventually, the job is executed on one or several WNs of the site (5).

- **Grid User Interface**
  The Grid user interface (UI) is a host providing the aforementioned collection of command line utilities in a homogeneous environment. gLite provides standardized UI software packages that can be deployed to computers, turning them into Grid UI machines. These hosts then are used by the Grid users to prepare and submit their jobs.

- **Computing**
  The computation work on the WLCG is performed on worker node (WN) computers organized in compute elements (CEs). The WNs provide their cores for specific amounts of time to individual user jobs, managed by the CE's batch system. There usually is exactly one CE per Grid site, managing up to thousands of computer cores on hundreds of WNs. To the Grid, the collection of cores is exposed by means of **processing queues**, often distinguished by expected job run durations (short, medium and long), preventing short-running jobs with a high submission frequency to have to wait for long-running ones to finish. Behind the homogeneous abstraction of the CE, several implementations of the real computing facilities can exist. Often, a **compute cluster** (a collection of high-performance computers) managed by a **batch-system** is used.

- **Storage**
  Storage is organized in storage elements (SEs) consisting of long-term- and short-term-storage facilities. Since a Grid site running a CE typically also houses a corresponding SE, there is the concept of the close-SE that refers to the site's associated local storage that is reachable via a local network connection instead of having to transfer the data over the internet. SEs can internally use different data storage systems: stacks of hard disk drives, tape systems, RAID storage etc. This heterogeneity is hidden from the Grid by providing a uniform method to access the storage. This, in the WLCG, is given by a software named the Storage Resource Manager (SRM). File transfer services and the user jobs requesting data use the SRM to fetch the data they need from the SE.

- **Data Transfer**
  This means the data transfer from one SE to another or between the SE and the machine used by the user to prepare and submit his job. Specialized software solutions exist in the WLCG to handle such data transfer efficiently, as requested by the single user (or user job), or initiated as a **third-party-transfer** from site to site.

- **Information Services / Catalogues**
  The Monitoring and Discovery Service (MDS) is the main information catalogue of Globus. For gLite, it was extended to the Grid Information Service (GIS). It houses metadata and organizational information about

all other WLCG services, the Grid sites, and the VOs. It can be programmatically queried or browsed by the users.

- **Job Brokerage**
  The job broker in the WLCG is the Workload Management System (WMS). It is designed according to the paradigm that the jobs should be distributed over the Grid - instead of the data: Jobs are preferably scheduled to run on CEs providing local (CLOSE-SE) access to the job's needed data. For this reason, it is important for the data to be distributed across the VO beforehand by the data transfer service (see section 2.2.4 for more information about the data distribution model in the WLCG). The job brokerage process is divided into several steps. After a user submitted his Grid job from his UI machine, the WMS queries the Grid's GIS for information about the available Grid sites, trying to find the CE that is the best match to the job's requirements (available needed data, enough computer cores to handle it, etc). Then, it sends the user job to the CE, where it gets scheduled in the CE's local processing queue. Finally, when the job eventually was executed and finished running, the WMS gets notified by the CE and provides the job's result to the user's UI. This process is depicted in figure 5.

- **Security**
  Security services on the WLCG consist of a CA per contributing country, housed at the country's Tier 1 - site, and the terms of usage every applicant for a VO membership must agree to, which also contain security guidelines.

- **Monitoring and Accounting**
  Different parallel and orthogonal monitoring systems have been developed for the WLCG. They are part of the job brokering mechanisms or stand-alone services gathering monitoring information per-site or per-job. They are discussed in more detail in the sections 4.1 and 4.2.

On top of this basic middleware implementation, additional services have been developed to ease the usage of the Grid. The major additions discussed in this work are

- the concept of **Pilot Jobs** and the **Pilot Factory**, an alternative means of job brokerage, using the PANDA-service (see 2.4.1), and

- the Grid User Interface frameworks GANGA and PATHENA that add another layer of abstraction and ease of use compared to the native GLITE command line tools (2.4.2).

2.2.3 Computing model

The ATLAS computing model was introduced in 2005 to define the requirements of the ATLAS-collaboration in computing. The main requirement was formulated as

> [...] to enable all members of the ATLAS Collaboration speedy access to all reconstructed data for analysis during the data-taking period, and appropriate access to raw data for organised monitoring, calibration and alignment activities.[30]

One central point of the ATLAS Computing Model is the conversion of the raw physics data into more compact format that are easier to work with. The Computing Model defines the following data formats:

- **RAW**
  The raw byte-stream data coming from the detector, after being narrowed to the interesting subset by the triggers (section 1.2.4). The data can be separated into events, but there's no object oriented representation of physical entities in the data. The average size of one event in RAW data is 1.6 MB and the expected production rate is $2 * 10^9$ events per year.

- **SIM**
  Simulated data, generated by the MC production. This is actually a general term, as several types of simulated data exist - each data type corresponding to a step in the processing chain can be (and is) simulated. Overall, a MC production of the size of around one third of the real data is expected in the Computing Model.

- **DRD**
  Derived Reconstruction Data, an intermediate format containing the reconstructed event information (tracks, jets, missing E_T, etc) in object-oriented fashion. Several iterations of DRD data are created during reconstruction, resulting in the ESD data at the end of the reconstruction process.

- **ESD**
  Event Summary Data, the first type of all-purpose compacted event data, with an average size of 1 MB per event. It contains an object oriented representation of the physics event data and is intended to make access to the RAW data unnecessary.

- **AOD**
  Analysis Object Data, a reduced event representation, derived from ESD data. This format is suitable for physical analysis, and is augmented with metadata to ease the analysis process. The average size per event is 100 kB.

- **DPD**
  Derived Physics Data, another representation of event data used for physical analysis. This format contains similar information like the AOD, but conditioned to allow easy processing and histogramming in ROOT.

The data is distributed to the Tier 0, Tier 1 and Tier 2 sites as described in the next section. According to the computing model, user analyses are brokered by Grid means (WMS) to the appropriate site holding the analysis' needed data, instead of brokering them to preselected sites and moving the needed data there. This reduces the need of data replication. Often, each user analysis only needs to look at a fraction of the data contained in a dataset; the derived formats (ESD, AOD, DPD) allow such partially data access.

2.2.4 Data storage and distribution

The data created by the ATLAS-detector and the MC production is spread over the WLCG in a hierarchical distribution pattern involving the three Tiers of Grid sites described in section 2.2.1. This data distribution ensures that the raw data is stored at least twice to reduce the risk of permanent data loss, and that derived data (AOD, DPD) is available at several places in the Grid to make the job brokerage efficient.

After creation in the detector, the recorded physical raw data is stored directly at CERN, the Tier 0. The research facility hosts large tape storage systems for this purpose. To identify each dataset later on, the datasets are consecutively numbered. As the whole detector's status is recorded regularly, as well, means are provided to interpret the measured event data properly. The data then is duplicated by transferring it to Tier 1 centres around the globe. The Tier 1 sites also are responsible for coordinating the conversion of the raw data into ESD, the first derived data format; the ESD-files are stored at the Tier 1's and their replication can be requested by Tier 2 sites according to their physicist members needs. Each Tier 1 originally is responsible for one certain type of ESD; the different types are replicated to sibling Tier 1's with a lower priority.

At the Tier 2 sites, the data is further processed to create AOD- and DPD files. The single user, typically, is located at a Tier 2 site or a connected institute and develops his analysis code against such third-order derivative data formats.

The software needed for physical analysis is deployed in the WLCG in a centrally coordinated fashion, as well. Grid sites have to reserve a fraction of their available disk space for software installations. As new releases of the ATLAS software frameworks becomes available, it is remotely installed onto those volumes. Tools exist to query the GIS for available software releases at different sites.

2.3 gLite Grid jobs

Submission of a Grid job to the WLCG using gLite's shipped command line utilities is a multi-step process. After developing the user algorithm (in the scope of this work, most of the time using ROOT and Athena, see 1.3.3) and compiling it on the user's local machine, a job description has to be created on the Grid UI. This job description is a plain-text file that must be written in the syntax of the Job Description Language (JDL) and is consequently called the JDL-file. It specifies what files the job consists of, what output files are to be generated (so the Grid middleware knows what files to transfer from the UI to the WN and back) and what requirements (e.g. available software installations, expected job run time, available number of cores and amount of memory, etc) exist for the job.

Since the user knows what datasets his job should run over, he can use this information to preselect a subset of Grid sites by querying the availability of these datasets in beforehand - gLite provides command line utilities to do just that. Usually, a specific range of valid Athena-versions exist that is compatible with the user job - most of the time, this is the version the user built his analysis onto. The desired software version can be specified in the JDL-file, as well, narrowing down the list of available sites for this job.

The gLite-utilities use the JDL-file to determine what Grid site to send the job to. The user can at first query the WMS what sites it would consider to broker the job to, and compare this with the list of sites resolved by dataset availability information earlier. Then, the JDL-file can be tweaked, or the job submitted. An example JDL-file is shown in listing 1, and the process of job preparation and submission is depicted in figure 6.

On successful submission, the user gets a unique identifier per job that he can use to query its current status. The same identifier is used after the job completed to fetch the job's output to the user's UI.

One further speciality of WLCG Grid jobs is the possibility to split a job's execution into many jobs submitted in parallel and have the resulting data merged after all created jobs finished running. This is to prevent a single Grid job to have to run over a number of events that is too large for efficient processing. A split Grid job is consequently called a **split job**, and each of the submitted jobs is a **sub job**.

```
1 ############Hello World#################
2 Executable = "/bin/echo";
3 Arguments = "Hello welcome to the Grid ";
4 StdOutput = "hello.out";
5 StdError = "hello.err";
6 OutputSandbox = {"hello.out","hello.err"};
7 VirtualOrganisation = "atlas";
8 ########################################
```

Listing 1: Example JDL-file specifying Grid job requirements

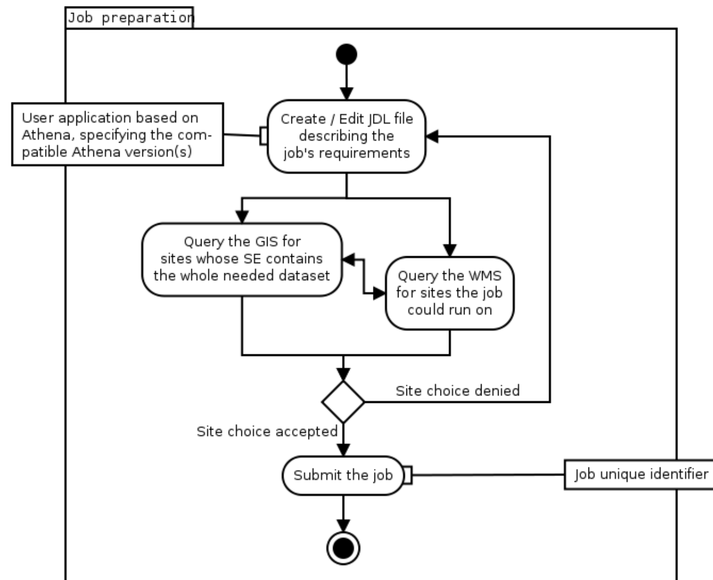


Figure 6: Steps to prepare and submit an ATHENA Grid job on the WLCG. The chosen site may be denied by the user - amongst other reasons - because there are many job failures happening at that specific site, or because the dataset is not completely available.

2.3.1 Input- and outputdata

The associated data of a Grid job can be separated into two classes, the **file based** and the **dataset based** job data, handled differently by the middleware.

File based data is transferred to the WN the job runs on before it starts as part of the so-called **input sandbox**, usually a compressed archive, consisting of all files the job needs to be executed, and is transferred back to the UI after the job finished as part of the **output sandbox**, containing smaller output files, log files and the job's `stdout`- and `stderr`-output saved into files (usually only for successfully terminated jobs - failed jobs mostly write no output-sandbox at all, see 2.3.3).

File based data is restricted by the middleware to a comparatively small size; anything exceeding this size must be made available as dataset based data.

Dataset based data is every data needed or created by the job that is too large for the input- or output-sandboxes. The physics data the job processes is one example of dataset based data, as is condition-data describing the state of the detector at the time the data was created. Usually, the processed data the job outputs is handled as dataset, as well. Datasets are not directly transferred from the UI to the WN and back, but are read from - and stored to - the site's CLOSE SE via the Grid's file transfer service.

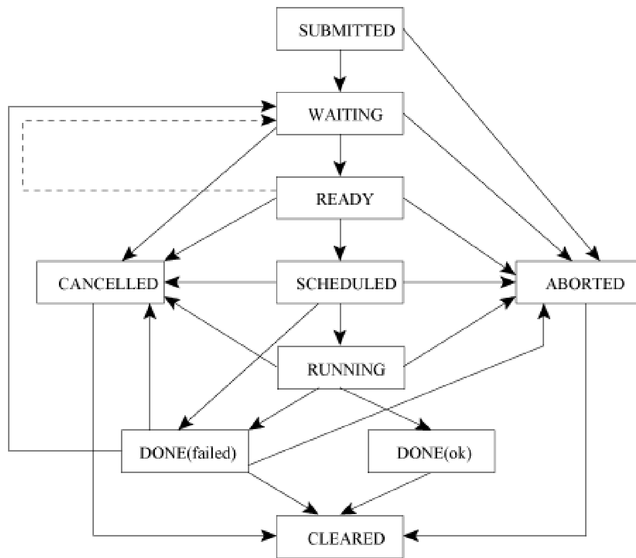


Figure 7:
Grid job states and status transitions in gLite[31]

2.3.2 Grid job life cycle

During its lifetime, each Grid job passes several states. When querying the progress of the job, this status is reported together with bookkeeping information like the timestamps of the latest job status transitions. The states a WLCG Grid job can be in, and the possible transitions between those states, are depicted in fig. 7.

At first, after the job has been submitted to the Grid - this means its job description has been transmitted to the WMS by gLite - it is in status `SUBMITTED`. At this moment, it merely is an entry in a queue at the WMS, and no action has been taken by the WMS in terms of brokerage for this job.

As soon as the WMS begun to look for a suitable Grid site to send the job to, its status is updated to `WAITING`. If a site matching the job's requirements was found, the job is forwarded to this site's CE and updated to `READY`; the files of its input-sandbox are uploaded to the site, as well. It then waits in the site CE's queue of unscheduled jobs. The CE determines what local job queue (e.g. of the site's batch system) to put the job in, schedules it into that queue, and updates its status to `SCHEDULED`. If the local scheduling does not succeed or times out, the WMS possibly re-brokers the job, resetting its status to `WAITING`.

After the job waited in the chosen CE's local queue, it is sent to one (or more) of the site's WNs and executed there. Its status is then updated to `RUNNING`. The user application itself, though, is usually not executed directly on the WN - instead, a small shell script, called the **runner script** in this work, is executed. This script spawns the user application and processes its exit code to determine if the user job ultimately succeeded or not.

If, due to a site-local problem, the job's execution cannot be started at all, the job's status is changed to `DONE (failed)` and the WMS may try to re-

broker this job, making its status return to `WAITING`. If one of the participating services is not able to perform its task for the Grid job (for technical reasons), the job can be aborted and accordingly set to the status `ABORTED`. If the user decides to cancel his Grid job himself, the status of it changes to `CANCELLED`. If, however, the job's execution finished on the WN, the status is set to `DONE (ok)` or `DONE (failed)`, depending on the job's exit code on the WN, as determined by the runner script.

After finishing successfully, the job's results are held by the WMS for some time, allowing the user to fetch them. As soon as that happens, the job's status is set to `CLEARED` and thus it finishes its life cycle. If the job was not successfully executed and ended in the state `DONE (failed)`, `ABORTED` or `CANCELLED`, the results may or may not be available.

2.3.3 Job failures

As can be seen in the Grid job life cycle, there are several states of a job resembling a failure, differing in the status the job was in just before it failed, and in the entity setting the job status to "failed" (`ABORTED`, `CANCELLED`, `DONE (failed)`) - the WMS, the CE on behalf of its local queue (batch system), the user himself, or the runner script running on the worker node detecting a non-zero exit code of the user application.

On top of that, for every failure state, there are numerous possible causes of failure, and even the status `DONE (ok)` need not mean the user job succeeded in generating the output the user did expect - there still could have been logical errors in the job, invisible to the middleware, or failures causing the job to abort its run, and nevertheless returning the exit code 0 (success) to the runner script.

In cases of aborted, cancelled or failed jobs, the job results (output sandbox) may be dropped by the middleware, and as such may be unavailable to the user. This also includes the job's log files. More often than not, a Grid job failure can not be investigated properly in retrospect, because the job's stdout and stderr-output is not available.

Finally, handling Grid job failures in the context of HEP in the WLCG, even if the job's output is available, is exacerbated by the complexity of the involved software frameworks and libraries. As the user application is, most of the time, a physics algorithm run inside - and itself utilizing - large frameworks like Athena and ROOT, the result of application errors is not as simple as a single exception being thrown or a number of error messages logged. Quite to the contrary, the error output usually is buried in large amounts of error traces, stack dumps and consecutive fault output, making the determination of the origin of the failure a tedious and difficult task.

Examples for possible Grid job failures

This is a (by no means exhaustive) list of possible Grid job failures in the WLCG, sorted per job status.

`WAITING`

- The WMS was not able to match the job's requirements to a CE.
- The WMS currently doesn't broker any jobs for technical reasons, and as such, the queue of waiting jobs is not processed.

`READY`

- The job is waiting for the input data to become available, and does not change status indefinitely.
- The CE is not taking any more jobs out of its job queue.

`SCHEDULED`

- The local queue(s) at the CE are not processed due to a malfunction of the local batch system.
- The user credentials (proxy certificate) time out while the job waits in the site local queue.
- The site breaks down and is tagged offline while the user job waits in the local queue.

`RUNNING`

- The user credentials (proxy certificate) time out while the job is running.
- The job execution is aborted due to an exception or error in user code (it also happens that a bug in user code is triggered by a specific data set, so that it is not noticed when testing locally before Grid submittage).
- Input data the job needs cannot be loaded from the CLOSE-SE (it cannot be reached or malfunctioned, or the dataset does not exist there).
- It is not possible to store the job's output data on the CLOSE-SE (it cannot be reached, it malfunctioned, its available space does not suffice).
- Needed local resources on the WN, like free memory or hard disk space, become depleted during the job's execution.
- The job's run time or CPU time exceeds the limit associated with the local queue it was scheduled into by the CE.

These failures - during the state `RUNNING` - are particularly interesting, because without a job centric monitoring software like presented in this work, there is no possibility at all to detect those errors. Before the monitoring software is presented, some further applications that became widely used in the WLCG are introduced.

2.4 WLCG software

In this section, additional software available in the WLCG is presented that eases the usage of the Grid for the purpose of physical analysis with the Athena-framework. In the discussion of Job Monitoring in the main part of this work, this combination of convenience-focussed UI-software and Athena will serve as use case example.

2.4.1 Pilot jobs and the pilot factory

The usual job brokerage process is, as described in section 2.2.2, driven and controlled by the WMS, and works using a “push”-semantic: The user jobs are sent to a CE selected by the WMS according to their requirements and are executed on one or several WNs controlled by that CE, usually after having waited a while in one of the CE’s job queues.

This implies that, after the site has been selected according to the job’s criteria, the job is fixed to this site, and problems such as a high load on the WN could develop during the job’s queue waiting time, and that infrastructural problems like a corrupt or invalid software installation on that site is only detected after the queue waiting time has passed. This happens more often than not, because the WMS can only decide where to send the user job according to the information contained in the GIS, which could be erroneous or out-dated.

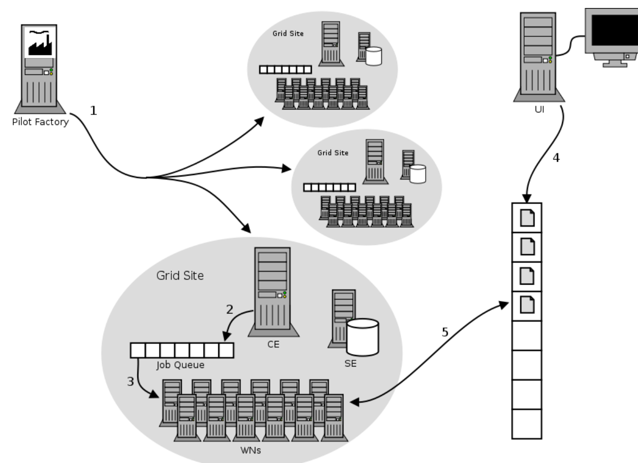


Figure 8: Schematic display of the pilot approach to Grid job brokerage in the WLCG. The Pilot Factory submits a number of Pilot Jobs to the Grid, bypassing the usual WMS service, directly to participating CEs (1). The Pilots get queued (2) and run (3), and wait for suitable user jobs. In the meantime, the user submits his Grid job via Panda, and the job is inserted into the Panda job queue (4). Eventually, one of the Pilots fetches the user job and it is executed on the Pilot’s WN(5).

To address these problems, the PANDA system has been developed for the WLCG, implementing a "pull"-semantic for Grid job brokerage.[32] The central idea in this approach is that the WMS does not dispense user jobs directly to CEs, but rather small **Pilot Jobs** are dispensed by a central service directly to all participating Grid sites. Each Pilot Job consists only of a shell script that checks the overall health of the WN it has been scheduled to run on and determines the actual software- and infrastructure-environment present on that WN (*really* available memory, system load, installed software packages, etc).

The Pilot Job then queries a user job database (containing information about submitted user jobs that wait for execution) for jobs whose requirements match its WN. The selected job is pulled from the database to the Pilot Job's WN and executed there, knowing beforehand that its requirements actually are met. The WN then is called the **Pilot Node** of the job.

When Grid jobs are submitted by the users, they are inserted into the central job database (the PANDA job queue) together with their requirements, instead of being fed to the WMS. The creation and periodic submittage of the small Pilot Jobs to the CEs is done by a specialized Grid service named the **Pilot Factory**. The brokerage process with PANDA is shown in fig. 8.

To minimize the interference of this Pilot Job system with the classic WMS brokerage of jobs, each Pilot Job terminates after a short time when no appropriate user job exists in the job database. Furthermore, the Pilot Factory uses metrics like the number of already waiting jobs on a site to decide how many Pilot Jobs - and in what frequency - to submit to each site.

The main advantage of this approach is that the turnaround time for user jobs is optimized, because jobs don't have to be resubmitted as a cause of an invalid, erroneous or temporarily unavailable environment on their CE. Also, it provides means to monitor site usage and to deploy infrastructural updates with short delays by inserting the updates into the Pilot Factory directly. Finally, the system can be applied without modifying the existing infrastructure.

One disadvantage of the Pilot system can be the evasion of WMS-enforced fair share rules and load balancing efforts by mass-submitting Pilot jobs indifferently to many sites. Also, in Pilot-based systems, it is difficult to infer the real owner of a job, that is, the user who developed the algorithm and submitted it to the Grid, as all Pilot jobs are executed under generic PANDA user accounts.

2.4.2 The user interfaces: pAthena and Ganga

To ease the preparation of user jobs to submit on the WLCG, two differently-approached solutions have been developed. The first solution will briefly be introduced in this section, and the former solution will be described in more detail, as the monitoring software this work is about has been integrated into this one by the author; it is planned, however, to also combine the monitoring software with the other solution, as well (see the outlook in sec. 17).

backend	description	splitting
Local	The job is executed locally on the machine GANGA runs on	no
Batch	A site-local batch system providing access to a compute cluster is used to run the job	no
LCG	The job is submitted to the WLCG using GLITE	yes
Panda	The PANDA system is used to execute the job on a Pilot Node	yes

Table 3: GANGA backend types

pAthena

As described in section 2.3, the preparation and submission of Grid jobs containing an ATHENA-based analysis application is a process consisting of several steps, many of which are the same for every analysis a user wants to perform. Additionally, the process has to be performed manually, by executing the steps shown in fig. 6.

PATHENA - "PANDA and ATHENA" - is a command-line application that was created to address this problem by automating the aforementioned job preparation steps and reducing the user's interaction needed to prepare and submit a Grid job to one single command to execute. As the name suggests, it only works using PANDA as the job submission means, although the job submission process is hidden from the user. PATHENA was created to resemble the local execution of ATHENA on the user's workstation, so one can run his analyses on the Grid almost exactly like he runs his test-runs locally.

The user specifies all information needed for the automatic job preparation as command line parameters when executing PATHENA, e.g. the datasets the job should run over, the compatible ATHENA version, excluded sites, etc. PATHENA then performs the automatic definition of the job requirements and the submittage to PANDA.

Ganga

The job preparation and management tool GANGA follows another approach in making the job submission process easier for the user. It provides an object-oriented abstraction of Grid- and job-concepts on an interactive console, allowing to model the desired job with its input- and output-data, its requirements and restrictions as objects that can be stored into a database. Instead of a unique job identifier, a job is represented by an object, providing properties and methods to the user to perform additional tasks. When submitting, the dataset look-up, site picking and creation of a suitable JDL-file or PANDA call is done transparently, hiding the complexity from the user.

GANGA, unlike the previously described command line tools, doesn't stop at the actual submittage of the job. Instead, it allows one to manage and supervise the running jobs during their execution, as well as the finished or failed jobs, using the same object-oriented view. This way, use cases like the querying of the current job status or the fetching of job results after it finished can be performed via method calls on the job-object in the interactive console.

GANGA provides abstractions of several different job execution **back-ends**. A back-end in this sense is a means of execution of a job, with the submission to the Grid being only one example. A (non-exhaustive) list of back-ends GANGA provides access to is given in table 3.

Similarly, different kinds of **application abstractions** are provided, including an abstraction of ATHENA (GANGA is not only used in the context of HEP, and other fields of science use different application abstractions in GANGA).

Finally, the concept of **job splitting and merging** is provided by GANGA, if a back-end allowing this functionality is used (see table 3). It consists of the automatic splitting of the user job into smaller pieces, each executed as an own job on the Grid to parallelize processing as well as the merging of the separate job results into a global result. Typically, an event-based splitting is fulfilled for HEP jobs, dividing the workload on the basis of the physical events.

```
*** Welcome to Ganga ***
Version: Ganga-5-5-15
Documentation and support: http://cern.ch/ganga
Type help() or help('index') for online help.

This is free software (GPL), and you are welcome to redistribute it
under certain conditions; type license() for details.

In [1]: j = Job()
In [2]: j.application = Athena()
In [3]: j.application.atlas_dbrelease = ''
In [4]: j.application.atlas_release = '15.6.10'
In [5]: j.application.option_file = 'HelloWorldOptions.py'
In [6]: j.application.max_events = 10
In [7]: j.application.atlas_cmtconfig = 'i686-slc5-gcc43-opt'
In [8]: j.backend = LCG()
In [9]: j.backend.requirements = AtlasLCGRequirements()
In [10]: j.backend.requirements.cloud = 'DE'
In [11]: j.submit()
Ganga.GPIDev.Lib.Job        : INFO     submitting job 20
Ganga.GPIDev.Adapters       : INFO     submitting job 20 to LCG backend
Ganga.GPIDev.Lib.Job        : INFO     job 20 status changed to "submitted"
Out [11]: 1

In [12]:
```

Figure 9: Example GANGA session

One strength of this object-oriented approach to job modelling is the possibility to run a physics job over a small number of events locally, and then, after it has been tested and proven working, to duplicate the job object in GANGA, switching its back-end to LCG or PANDA, and resubmitting it with an increased event count and appropriate splitting. This work flow speeds up development and improves the job turnaround for the user. Additionally, GANGA provides an integrated help system and advanced editing capabilities like the smart completion of commands and a context-sensitive command history.

Figure 9 shows an excerpt of a GANGA session preparing and submitting a WLCG Grid job running an ATHENA analysis.

3 Conclusion

As could be seen, Grid jobs in the context of the WLCG are participants in a very complex, globally distributed system, in which errors are not easy to detect and trace (see 2.3.3).

Due to the inherent complexity of the Grid jobs themselves (passing through multiple states, handled by - and relying on - numerous distributed services), and the involvement of many libraries and application layers in HEP software on the Grid (see 1.3), the search for the reasons of Grid job failures is difficult, even if information sources like log files and exception stack traces are available - a not too-common situation in itself.

The processes of detecting, reasoning and preventing of Grid job failures can be supported - or even made possible in the first place - by means of a real-time job monitoring software. In the next part of this work, such a job monitoring solution is presented and its architecture described.

Part II
Job monitoring

4 Overview

The term **Monitoring** in the context of the Grid refers to the gathering or creation of information about Grid services, sites and jobs for the purpose of supervision, accounting and fault detection. It was considered an important part of any Grid venture from the beginning on due to the Grid's distributed and complex nature. It is hardly possible to run a Grid without knowing the states of the Grid sites, the current job load, a list of malfunctioning services, etc.

In this work, a distinction is made between monitoring on behalf of the organisation(s) running the Grid for the above-mentioned purposes, and monitoring on behalf of the single user, the **user-centric** monitoring. The latter type is what most of this work focuses on, so the distinction is explained before describing this user-centric monitoring - and the software solution developed at the University of Wuppertal in particular - in detail.

4.1 Site monitoring

Site monitoring is an umbrella term for efforts to gather status information about Grid sites. Information worth recording includes the general availability status of the site's services, their load and response time; site-wide statistics like the job throughput, the amount of data stored in the SE, the efficiency (job error rate over job count), etc. are aggregated in those monitoring efforts, as well.

The information is made available via the GIS and is used by the WMS to take decisions. It is also used to generate reports that aid the Grid support teams in their supervising and issue escalation activities. Finally, the information is used for administrative and reporting purposes. Typically, large parts of Grid efforts - this holds for the WLCG, too - are funded by public interest via governmental aid programmes. Since these programmes are paid with tax money, reporting is an important issue. The site monitoring is one of the main data sources for those reports.

As the name suggests, this type of monitoring is driven by the sites, or centrally in querying each participating site's status. It is not executed by the single user or the single user's jobs. One can, via web views on the monitoring data that are made available for all interested Grid users, access the data and explore it to infer information about the site where one's jobs are executed; there is, however, no possibility to project the data onto single users or jobs. For this reason, site monitoring is not suitable for job failure analysis by the single user.

Nevertheless, site monitoring can help in identifying site-wide problems and misconfigurations (e.g. broken software installations, causing all jobs using this specific software version to fail, or corrupt datasets that then can be re-replicated to fix the problem). In the WLCG, a site black list is maintained, filled with information based on site monitoring. This list prevents further jobs to suffer from well-known problems, until they are resolved and the offending site is being removed from the list again. However, there still are enough failure scenarios (like presented in section 2.3.3) reasoning the need for a user- or job-centric monitoring solution, as evidenced in figure 10, a summary of the job success rates over a period of high WLCG activity.

Figure 10: Job success rates in the WLCG from September 2009 to June 2010 (period of high job activity on the Grid).[33]

4.2 User-centric monitoring of Grid jobs

If the monitoring effort is done by the user owning the monitored jobs himself, the term **user-centric monitoring** is appropriate. Its main separation from site monitoring is that it is not performed indifferently and periodically over a large number of systems (Grid sites, services on a site). Instead, it is executed only as requested for specific single jobs. Thus, there is a completely distinct scalability requirement, allowing for much more in-depth monitoring efforts (because these efforts are not executed massively in parallel). The benefit in this is that one can get a much deeper and much more detailed insight into the monitored job's execution, allowing one to detect job faults earlier and with more supporting information allowing one to find the job failure reason.

Thus, in user-centric monitoring, as opposed to the site monitoring, the monitoring application is not run centrally - being "automatically" available to query it - but has to be launched somehow by the user himself (or rather by his job), while taking measures to let the monitoring software not interfere with the job's execution, altering its results, and not to create too much of a performance impact on the job's execution. On the other hand, the addition of the user centric monitoring to his jobs should be transparent and easy to configure, and should be exposed to the used job preparation frameworks, so users can add monitoring to their jobs without significant effort.

This approach has been taken by the job monitoring software developed at the University of Wuppertal that is described in detail in the following sections.

5 The Job Execution Monitor

The Job Execution Monitor (JEM) is a Grid job monitoring software developed at the University of Wuppertal since 2005. It is meant to augment the execution of Grid jobs with the provision of real-time progress- and debugging-information to the user. Its development has been performed iteratively by several developers, rethinking and rewriting several parts of the software to form its current state. The major steps in JEMs development are shown in table 4.

The first version of JEM, written as part of his master thesis[34] by Ahmad Hammad, was based on a module for bash script instrumentation written by Dmitri Igdalov as part of his diploma thesis.[35] The instrumentation was added to bash scripts via syntactical analysis and command injection. JEM v1 further featured a system metrics monitor called "Watchdog" that periodically recorded system information, and communicated via R-GMA[11] with the UI while using Unix domain sockets for inter-process communication. To this time, the system was called Job Monitoring System (JMS).

In 2007, the project was renamed to JEM and the bash script wrapper was rewritten using an other approach, because it turned out that syntactical analysis of bash scripts was not feasible in terms of complexity and robustness. The new solution was conceptualized and written by Andreas Baldeau in his diploma thesis.[36]

His work consisted of a modification of the bash itself (a branch of bash's source code was created for this purpose). The ansatz was to ship the modified bash to the WN to replace the existing shell, instrumenting the script execution to gather monitoring information.

During the same year, Dr. Stefan Borovac rewrote JEMs core modules for better maintainability and extensibility, forming what now is called JEMv2 ("version 2"). This new implementation allowed for addition of further monitor types and used named pipes for inter-process communication.

[11] see section 5.2.3 for details about R-GMA.

author	**year version**	**remarks**
Ahmad Hammad	2005, v1	Initial impl. of the JEM
Dmitri Igdalov	2005, v1	Initial impl. of a bash script wrapper
Dr. Stefan Borovac	2007, v2	Concept and impl. of version 2
Andreas Baldeau	2007, v2	Re-write of bash script wrapper
Joachim Clemens	2007, v2	Addition of the python script wrapper
Markus Mechtel	2007, v2	Grid centric expert system
Martin Rau	2008, v2	Integration into Ganga

Table 4: Major JEM version history milestones before this work.

Also in this period, Diploma student Joachim Clemens added a python script monitor to JEM, that now was able to report monitoring events not only inside of bash scripts, but also inside of python scripts (see sec. 5.3.2 for more details).

Further work was done by Markus Mechtel as part of his PhD thesis: Aside supporting Dr Borovac in the refactoring efforts, he developed an expert system using JEM data to infer Grid job failure reasons and became the lead developer of JEM until the author took over the responsibility. Finally, Martin Rau worked for his diploma thesis[37] on the integration of JEM into Ganga (see sec. 2.4.2).

In the following section, the version of the software up to this point - that was available just before the author begun development of the later-on described additions - is presented, and its shortcomings discussed that were tackled in the author's work.

5.1 Concept

The Job Execution Monitor is a monitoring software run in user space, in parallel to a Grid job, on the same machine (its WN). It supervises the Grid job's execution - generating monitoring data about the job application's execution itself - and records system metric information periodically. All monitoring data gathered this way is forwarded in real time to the user, who thus is able to follow his job's progress and to get notified of job failures, problems occurring during the job's run and less-than-optimal job results, while the job is still running.

By getting this information, the user can decide to abort and correct his job without having to wait for its completion, and in cases of job failures, the user has a larger amount of useful information at hands to analyse the failure and to take appropriate measures.

The data is sent to the UI the user submitted his job, and received there by a server component of JEM. This server component provides a simple command-line interface, an integration in the Grid job management application Ganga and a web-based graphical user interface displaying summary information about the job's execution.

JEM was designed to be as small and lightweight as possible, to make the addition of it to the user job feasible (JEMs library has to be transmitted as part of the job's input sandbox to the WN), and to minimize the impact on the job's execution time. Furthermore, the possible future extension of JEM with more script monitors (handling user applications written in different scripting languages like Perl) was foreseen in JEMs architecture.

5.2 Architecture

The JEM is a client-server application, appropriate to run in a distributed computing environment. The part of JEM running on the UI machine acts as the server, listening for incoming monitoring data to record and provide in summarized and visualized form. The part running on the WN, gathering the monitoring data about the user job running there, acts as the client, establishing a connection to the server component on the UI. The communication channel between the client and the server can be chosen from a number of alternatives described in section 5.2.3.

The monitoring software is implemented in the Python programming language. Being an easy to use but powerful scripting language, with the appropriate interpreter software available on all WLCG worker nodes and UIs, Python was a suitable choice of language for JEM (until central parts of the software were rewritten in C for performance reasons by the author of this work, see section 12.4.2).

This general architecture of JEM is depicted in fig. 11, and further detailed in the following sections, each describing one of the major parts of the architecture (UI, WN and the data transfer).

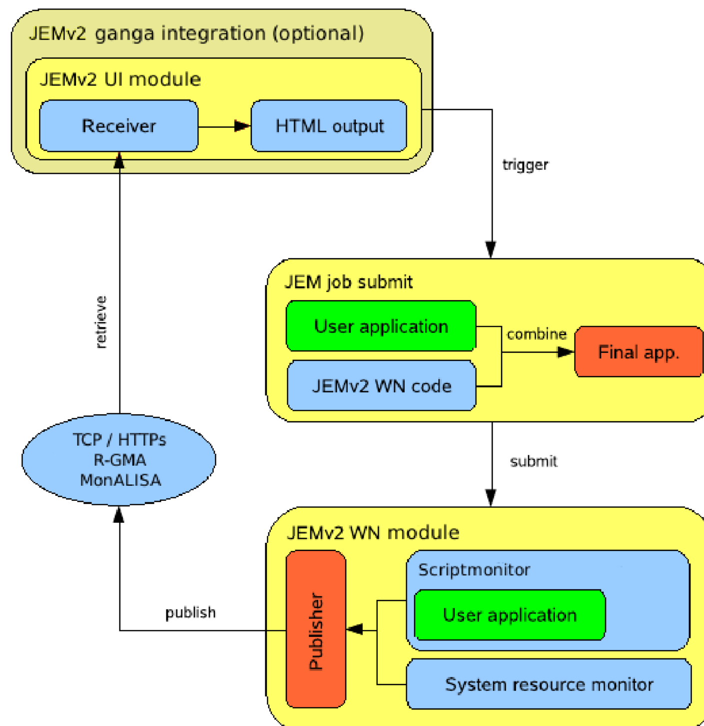


Figure 11: General architecture of JEM.[38] In this picture, only the HTML output mode is shown; further publishing modes are described in the next section.

5.2.1 User interface component

The UI component of JEM is responsible for receiving the monitoring data of a job, for publishing this data into a format chosen by the user, and for adding the monitoring functionality to the user job before it gets submitted to the Grid.

Because the mode of operation of JEM is user-centric job monitoring as described in section 4.2, JEM-specific software has to be made available on the Grid worker node where the user job is executed. However, it can usually not be foreseen on which WN the job is going to be executed at submission time, and it is not easy to deploy a new type of software to all Grid WNs. Thus, it was decided for JEMs architecture that the monitoring software itself gets added to the user job before submission, shipped as part of the job's input sandbox to the WN and extracted there before the user job starts running.

To reach this goal, the user job has to be altered at three points:

- The **executable** of the job, originally the user job's runner script (see sec. 2.3.2) has to be replaced with a script that spawns JEMs WN module on the worker node before starting a new process with the original runner script and its arguments.
- JEMs libraries and the bootstrap-script mentioned in the first point have to be added to the job's **input sandbox**.
- The job's **environment** may have to be extended by JEM-specific configuration values, to change the behaviour of the JEM (logging verbosity, chosen data transfer mode, etc.) on the worker node according to the user's preferences.

Of course, it is entirely possible to modify the job's JDL file manually according to these prerequisites. However, to ease the application of the JEM for the user, the performing of these modifications was considered to be part of the responsibility of JEMs UI component. Thus, scripts have been developed as replacements to the gLite job submission commands that modify the user job's JDL and subsequently call the original gLite commands with the modified job description, adding JEM to the job transparently.

At the same time, these scripts launch JEMs UI component to receive the monitoring data of the submitted job. The type of receiving process that is launched depends on the chosen transfer mode; architecturally, these processes are similar; each one acts as a data receiving server that gets connected to by the WN component of JEM, receives the monitoring data and forwards it to the publishing process.

For users of the Grid job management software Ganga (see sec. 2.4.2), a modification was implemented that fulfils the same role as the described UI-script (modification of the job description, start of data receiver process, publishing of the data) in the context of Ganga.

Figure 12: Example of JEMs HTML output[38] showing plotted system information like CPU usage, Memory consumption and network traffic on the Worker Node.

It integrates JEM monitoring functionality into GANGA's job preparation and submission process. This modification is presented in section 5.4.3.

The data received by the UI component can be published to several, user-configured formats for interpretation during or after the job's execution. The possible output formats in the discussed version of JEM are:

- **Human-readable log file**
  This is a plain text file containing entries describing the occurred monitoring events verbosely. This format can be used to reproduce the job's performance manually and to get detailed information about the job's execution at critical points of time (e.g. just before a job crashed).

- **HTML files**
  The data is formatted via HTML[12] in this mode and can be viewed in any browser by pointing it to the created directory structure on the UI. The recorded monitoring data is visualized in graphical plots when using the HTML publishing mode, so this mode is suitable to get a quick overview over the job's status. An example of the HTML-output is shown in fig. 12.

[12]HTML is a document mark-up language invented by CERN physicist Tim Berners-Lee in 1990 for the exchange of scientific data; it is the base of the World Wide Web and as such globally used nowadays.

- **XML file**
  By exporting the data into an XML encoded format, interoperation with other software is possible. The integration of JEM into GANGA, for example, uses the XML publishing mode as data channel from JEM into the job management software[13].

- **Direct inter-process feed**
  In this publishing mode, JEMs UI component launches a third-party application configurable by the user and feeds all received monitoring data into that processes standard input stream. The third-party software then can perform arbitrary calculations and processings on the data. The usage of JEM-created monitoring data in the Grid centric expert system developed by Markus Mechtel (see table 4) was built on top of this publishing mode, for example.

5.2.2 Worker node component

The main part of JEM is the WN component supervising the user job execution and gathering system information. Like the UI part, it consists of several processes each providing one part of JEMs services, using inter-process communication to aggregate the data in one central process forwarding the data to the UI. This collection of processes is shown in fig. 13.

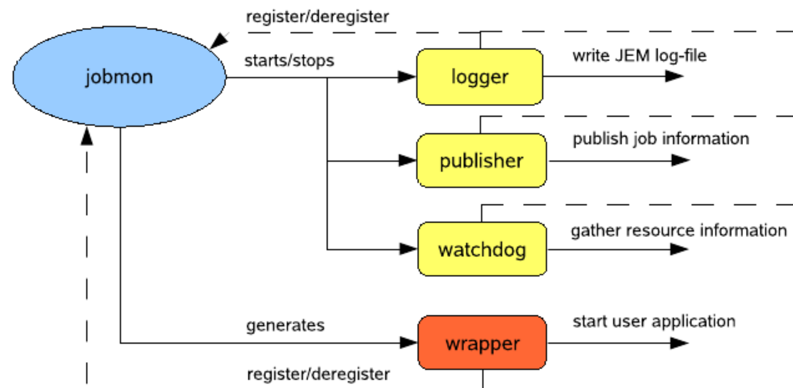


Figure 13: JEMs worker node component processes.[38]

The processes gathering the actual monitoring data (Script Wrappers and "Watchdog", the system monitoring process) are further described in section 5.3. The central process that aggregates the data and feeds it into the data transfer

[13]This holds for the version of JEM described here. Part of the author's modifications was the replacement of the file-based communication between JEM and GANGA with an in-memory communication - see sec. 12.2.

module(s), called JOBMON (for "Job Monitor"), is responsible for the launch and initialization of all other processes of the JEM WN component, as well as for establishing the communication between them. Also, the user configuration is processed by the JOBMON and a logging service started; this service creates a log file each WN process can write into, that is transferred to the UI in the output sandbox of the job, providing debug information in case of errors in JEM itself.

The job execution process on the Grid worker node machine, with added JEM monitoring, differs from the plain, un-monitored execution: Whereas in the latter case, the runner script itself is launched on the WN, in the former case, JEMs WN component is launched with the runner script given to it as command-line argument. The WN component creates the central process (JOBMON) that in turns launches the processes described earlier. After each of those processes initialized and gave feedback to JOBMON - for example, the data transfer module(s) must at first establish the connection to the UI - the script wrapper is requested to start the execution of the user application (the runner script).

If, on the other hand, anything goes wrong in initialization of one of JEMs processes, the user application is launched in unmonitored fashion (just as if JEM wasn't shipped to the WN at all). This is to ensure a correct user job execution in any case, to prevent the addition of JEM to influence job success rates.

The first task the script wrapper performs when starting the user application is to determine of which type this application is: a shell script, a Python script or neither. If there is a suitable script wrapper module for the type of the application, the execution is handed over to it. Otherwise, the application is run unmonitored. This way, JEM is extensible to new script types, as they just have to be inserted into the script type detection, and to provide a common script wrapper interface.

After the user job finished running (successful or by a failure), the JOBMON requests all its child processes to finalize and terminate; in this, all open files (log files, output sandbox data, etc.) and network connections are closed. Finally, it hands the user application's exit code to the Grid middleware, just as the user application would have done if run unmonitored.

5.2.3 Data transmission

As was mentioned earlier, there are several possible ways of transferring the monitoring data from the WN to the UI, from which a user can choose by his preference. The modes differ in real-time capability and readiness for production use. There is, in fact, exactly one suggested real-time transfer mode plus the output sandbox logfile. The available modes of data transfer from the WN to the UI component are:

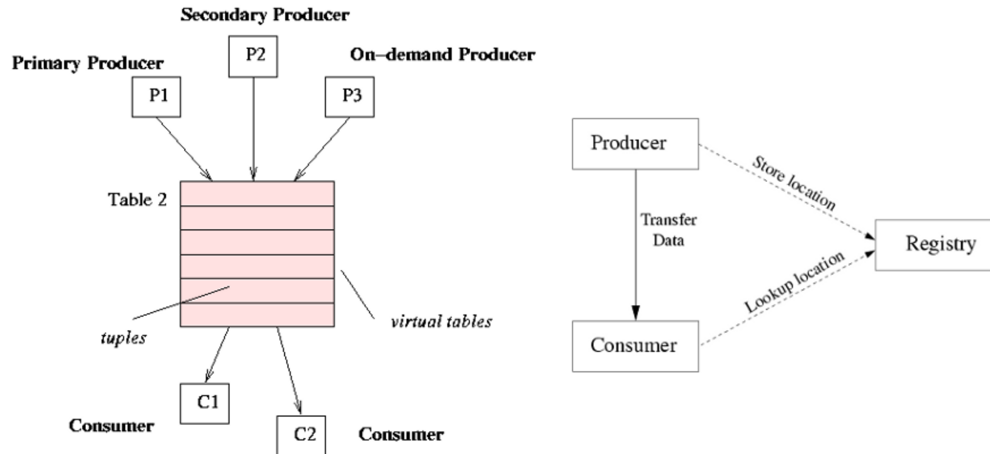


Figure 14: The Relational Grid Monitoring Architecture (R-GMA). On the left side, the virtual database concept is shown. The brokerage process according to the Grid Monitoring Architecture is shown on the right.[39]

- **plain TCP socket**
  The data is sent as an encoded ASCII byte stream over a TCP socket. Being implemented only for debugging purposes, this mode of transmission is not enabled by default.

- **log file**
  Here, the data is written into a log file on the WN and downloaded to the UI as part of the output sandbox (see 2.3) after the job completed. Thus, there is no real-time data transfer performed in this channel.

- **MonALISA**
  MonALISA is a proposed generic monitoring data transmission protocol in the context of the WLCG. As a protocol, it is not limited to one lower-level transport, and can in principle be transferred over any (socket-like) connection; it usually just uses a TCP socket, though. Support for MonALISA in JEM never left the experimental stage, and is not production-ready, so it is not enabled by default.

- **R-GMA**
  The Relational Grid Monitoring Architecture, R-GMA, is a set of a transfer protocol for monitoring data, producer- and consumer-service clients in different programming languages and a distributed message brokerage backend. It is described on the project's web-site as:

  > "R-GMA makes all the information appear like one large Relational Database that may be queried to find the information required. It consists of Producers which publish information

> into R-GMA, and Consumers which subscribe. It may be seen as a relational pub-sub system."[39]

R-GMA is the suggested mode of transfer for this version of JEM, and as such enabled by default. Its general working principle - it conforms to the Grid Monitoring Architecture, a broker-based monitoring data transfer system - is shown in figure 14.

One specific of R-GMA is, as the name suggests, that all data seen as relational tabular data, and transmitted via SQL `insert`- and `select`-queries instead of the usual `send`- and `receive`-calls of other transfer modes. This implies a certain amount of data representation conversion overhead, as all data has to be mapped to and from a virtual database table during transfer.

Despite being the default data transfer mode of JEM in the presented version, the transmission over R-GMA introduces scalability problems. These are discussed in section 5.6.

Depending on which transfer mode is selected in JEMs configuration, the respective receiver module (server) is started on the UI, and the corresponding data sender module is used by JEMs WN component.

5.2.4 Inter-process communication

Apart of the data transfer from WN to UI described in the previous section, the monitoring data has to be transferred from one of JEMs processes to the other, both on WN and on UI side, as the software is split into multiple, in-parallel running processes.

In JEMs first version, this inter-process communication (IPC) was implemented using UNIX domain sockets. These behave like network sockets and can be accessed with the same system calls, but no network data transfer happens when sending data through them; instead, the virtual loop-back network device is used, essentially causing the data to just be queued in a kernel buffer by the sending process and dequeued from the buffer by the other process.

When JEM was refactored to the version 2, the IPC was changed to a system using named pipes. Such named pipes, in principle, work similarly to sockets, with the difference being that named pipes have a file-like representation in the file system, and can be utilized using POSIX system calls (like ordinary files). This makes dealing with IPC very easy, but introduces major problems. They will be discussed in section 5.6.

The protocol for data transmission through the named pipe, designed at this time, is an ASCII-text representation with a key-value-structure. This is human-readable and easy to handle, albeit slow to marshal and de-marshal.

5.3 Acquisition of monitoring data

The main task of JEM is the gathering of monitoring data on the supervised job's WN as described in sec. 5.1. As was discussed before, the two main types of monitoring data are system metrics and script progress events, recorded by the system metrics monitor and the script monitors, respectively.

5.3.1 System metrics monitor ("Watchdog")

The recording of system metrics information in JEM is performed periodically by a process called the "Watchdog"[14]. The data usually is collected every 30 seconds and inserted into the stream of monitoring data sent to the UI by JEM. The user can configure in his JEM preferences whether the watchdog should be run at all, and in which interval.

The system metric data is taken mainly from the `/proc` virtual file system, and includes:

- CPU usage information
- Memory information (free RAM, free swap space)
- Disk information (free disk space in the job's working directory and in the temporary directory)
- Network information (inbound and outbound traffic amounts)

Furthermore, on the first sampling of the system metrics data, general information about the WN the job runs on is gathered, like the hostname, the working directory, etc. This first event is also used by JEM to detect the begin of the user application execution. On exit of the user application, and the consequent shut-down of JEMs WN component, the watchdog creates a terminal event including the application's exit code.

5.3.2 Script wrappers

As mentioned in section 5.2.2, in the described version of JEM, Bash- and Python-scripts can be monitored. Apart of the script execution supervision functionality itself, each script wrapper also has functionality to decide if it is able to supervise a certain application - essentially, this means that the bash script wrapper can identify bash scripts, the Python script wrapper can identify Python scripts, etc. The script wrappers are queried in turns by the JOBMON if they can monitor the given application. If no suitable wrapper can be found, the application is run unmonitored.

[14] this (a bit misleading) name was changed in the transition to JEM version 0.3.

Bash script wrapper
Bash scripts are monitored by shipping a modified Bash interpreter to the job's WN[15] that was compiled both for 32 bit and 64 bit systems. As the OS environment on Grid WNs in the WLCG is homogeneous, this Bash interpreter can be expected to run on all WNs.

The modification of the Bash interpreter is described in detail in Andreas Baldeau's work,[36] so only a short summary is given here (the modified Bash source code is available on CD in the appendix).

By adding callbacks to a small JEM specific library at appropriate spots to the Bash source code, the Bash script wrapper gets notified of function calls in Bash scripts, corresponding function returns, and executed commands. Those information then is fed to JEMs monitoring data stream by the Bash script wrapper. One can configure whether all events should be recorded, or only function calls and returns (to reduce the amount of data generated). For each event, the time of its occurrence is stored, together with the code location in the Bash script. These information suffice to follow the script's execution and to be able to detect faulty progress, as described in section 2.3.3.

An important secondary functionality of the Bash script wrapper, aside the gathering of the monitoring information, is the detection of child script executions. If another Bash script, or a Python script, is executed by the monitored Bash script, a new instance of a Bash- or Python script wrapper is injected to replace the respective script interpreter. This ensures the correct monitoring of child scripts throughout the whole user application run.

Python script wrapper
The Python script wrapper is a drop-in replacement for the Python interpreter, itself written in Python. It exploits Python's built-in trace ability (refer to the documentation of Python's `sys.settrace` builtin[40] for more information) to get notified of function calls and returns, script line executions and exceptions happening in the Python script. Similarly to the Bash script wrapper, the Python script wrapper's verbosity can be configured.

5.4 User interface

When using the term "User interface" in the context of this work, one has to distinguish between two meanings. The UI machine that is used to prepare, submit and possibly monitor the job, together with the corresponding component of JEM, is the meaning used up till now in the previous sections. The second meaning of the term is the way one can interact with the monitoring software. This is discussed in the next sections, as there are several different ways JEM can be used and presents itself to the user.

As detailed in section 5.2.1, the tasks of the UI component are the modification of the Grid job to run JEMs WN component instead of the user

[15] based on Bash 3.2

application, the receipt of the monitoring data and the publishing of this data in the desired format(s). Three possible ways of performing these steps are available in JEM (aside the fully manual modification of the JDL file, launch of the suitable monitoring data server of JEM and conversion of the monitoring data by the user himself).

5.4.1 Command-line usage

Part of JEMs distribution are command-line interface (CLI) tools that replace the usual gLite Grid job submission commands. A usual gLite Grid job submission command, taking a user-prepared JDL file as input, is shown in figure 15. The tool contacts the WMS, passes the job description to it, and receives back the job ID assigned to the new Grid job. This job ID can then be used to query the status of the job afterwards.

The corresponding launch of JEMs submission tool, being almost identical, is shown in figure 16. It passes the call internally to the gLite command after modifying the JDL-file. It does not, however, automatically start a monitoring data receiver process. This can be done by using JEMs built-in interface described in the next section; so, using the CLI command to submit the job makes most sense if no real-time data is to be received (only JEM logs in the output sandbox), as otherwise, one can better use the built-in interface for submittage, as well.

```
$ glite-wms-job-submit my_job.jdl

================= glite-wms-job-submit Success =================

The job has been successfully submitted to the WMProxy
Your job identifier is:

https://lb106.cern.ch:9000/_qXQgXqlXVNNChT5A-M22g

================================================================
```

Figure 15: Example gLite job submit call

```
$ $JEM_PACKAGEPATH/glite-wms-job-submit-jem my_job.jdl
```

Figure 16: The equivalent JEM CLI job submit call

5.4.2 Built-in interface

In the preceding section, it became clear that the replacement CLI commands for submittage that are shipped with JEM are not an all-purpose solution. They don't allow for real-time monitoring of the user job, because no data

receiving process is spawned automatically. To address this, part of JEM is a - still console based, but more complete - built-in interface.

The interface allows for selection, submittage and consecutive management of Grid jobs by JDL files using single-character keyboard shortcut commands. “Management” in this context means the possibility of querying the status of a job, killing the job, getting a job’s output or requesting a summary of all managed jobs. Furthermore, for all submitted jobs, a corresponding data receiving process (JEM UI instance) is spawned. Finally, the interface provides means to spawn a data receiving process for Grid jobs that have been submitted differently. An example session of the built-in interface is shown in figure 17.

```
$ $JEM_PACKAGEPATH/JEM_UI_main.py

This is JEM-interactive in GLITE-WMS-mode.
Use the 'h' or 'H' command to get some help!
JEM>>>s
JEM[JDL-Filename]>>>test_wu.jdl
JEM[Info]: Processing command at Fri Nov 16 15:33:15 2007 ..
JEM[Info]: Job
  https://glite-wms.physik.uni-wuppertal.de:9000/cLjvv__C_crL...
  succesfully submitted

JEM>>>
```

Figure 17: Example JEM UI session

The user is now able, after issuing this simple command and specifying his JDL-file, to monitor this job’s execution using a browser pointed to the created HTML output (created on the same machine in a default directory).

5.4.3 Integration into Ganga

The integration of JEM into the Grid job management software Ganga was implemented for two reasons:

1. Ganga is used by a considerably large fraction of the WLCG users
2. The built-in interface of JEM unnecessarily duplicates well-established functionality of Ganga (job life-cycle management)

By creating a Ganga plug-in integrating JEMs functionality transparently into the job management software, the focus of JEMs further development could be switched to improving the monitoring quality, robustness and completeness.

After the first proof-of-concept implementation by Martin Rau[37] was introduced that allowed one to monitor WLCG jobs from within Ganga with minor configuration effort by the user, but that needed to be installed manually into an existing, local Ganga installation, the JEM-plug-in matured and now is tightly integrated into Ganga and automatically deployed with it.

With this combination (Ganga and deeply integrated JEM), it is very easy to submit monitored jobs to the Grid and to visualize and analyse real-time monitoring data. Aside adding a JEM-specific object instance (`JobExecutionMonitor`-object) to the job representation within Ganga (see figure 18), no additional step is needed to be performed by the user. The data receiver is started automatically in the background as the job is submitted (and also survives Ganga restarts), and the data can be accessed from within Ganga with simple object method calls, as shown in the picture - there is no need to manually browse through log files or to visit a generated web site in another window, like it is the case with the built-in interface.

```
In [1]: j = Job(application=Athena(), backend=LCG())
In [2]: j.application.atlas_dbrelease = ''
In [3]: j.application.atlas_release = '15.6.10'
In [4]: j.application.option_file = 'HelloWorldOptions.py'
In [5]: j.application.max_events = 10
In [6]: j.application.atlas_cmtconfig = 'i686-slc5-gcc43-opt'
In [7]: j.backend.requirements = AtlasLCGRequirements()
In [8]: j.backend.requirements.cloud = 'DE'
In [9]: j.info.monitor = JobExecutionMonitor()
In [10]: j.submit()
Ganga.GPIDev.Lib.Job  : INFO    submitting job 21
Ganga.GPIDev.Adapters : INFO    submitting job 21 to LCG backend
GangaJEM.Lib.JEM.info : INFO    Enabling JEM monitoring for job 21
Ganga.GPIDev.Lib.Job  : INFO    job 21 status changed to "submitted"
Out [10]: 1
...
GangaJEM.Lib.JEM.info : INFO    Begun to receive monitoring data for job 21
In [14]: j.info.monitor.peek(n = 10)
Out [14]: peeking at output (last 10 of stdout, skipping 0)
(10   ) 10:34:36.060 | Py:Athena            INFO including file "post_002de
(9    ) 10:34:36.129 | Py:Athena            INFO including file "AthenaComm
(8    ) 10:34:36.148 | ApplicationMgr       INFO Updating ROOT::Reflex::Plu
(7    ) 10:34:36.302 | ApplicationMgr    SUCCESS
(6    ) 10:34:36.965 | ====================================================
(5    ) 10:34:36.984 |      Welcome to ApplicationMgr $Revision: 1.77 $
(4    ) 10:34:37.052 | running on c-104-22.aglt2.org on Tue Sep 28 10:34:37
(3    ) 10:34:37.133 | ====================================================
(2    ) 10:34:37.183 | ApplicationMgr       INFO Successfully loaded module
(1    ) 10:34:37.210 | ApplicationMgr       INFO Application Manager Config
```

Figure 18: Excerpt of a Ganga session with JEM integration

The monitoring data is fed from JEM to Ganga in this version of the integration by enabling the XML-writing data publisher in JEM, and having the Ganga plug-in periodically read this XML-file to check for recently received data. When the user queries for specific parts of the data by issuing one of the commands to the `JobExecutionMonitor`-object, the XML-file is read and the corresponding data extracted.

Available commands to the `JobExecutionMonitor`-object and the data displayed to the user by them are:

- `getStatus()`
  Displays general information about the job that does not change during the job run, like the hostname of the WN, the Grid job ID, etc.

- `getMetrics()`
  Displays the last sampled system metric data (CPU and RAM usage, disk space, network traffic, etc.)

- `peek()`
  Displays the last few STDOUT or STDERR lines the job has produced.

- `listCommands()`
  The last few commands executed in a Bash- or Python-script (shell commands, function calls, function returns, etc.) by the job are displayed in a short summary.

- `showCommand()`
  Displays more detailed information about an executed command.

- `listExceptions()`
  If exceptions occurred in the user job, they are displayed in a short summary.

- `showException()`
  An exception is displayed in a more detailed fashion, including the code location and local variable info (only available for Python scripts).

- `plotMetrics()`
  The recorded system metrics are visualized in a graphical plot[16] similar to the one shown in figure 19.

The `JobExecutionMonitor`-object also provides configuration options to the user, which can be set on a per-job basis. The major ones are:

- `enabled`
  This can be set to False to disable JEM for this user job.

- `realtime`
  If set to False, only output sandbox log files are created by JEM and no live data transfer is performed.

[16] To view this plot, the UI machine must support X-forwarding to the user's login machine

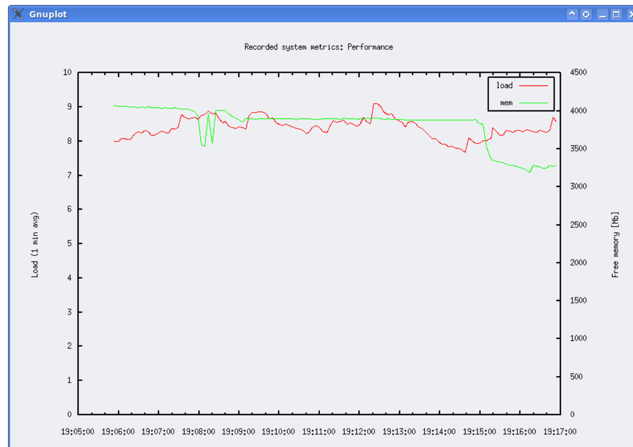


Figure 19:
System metrics plot as presented by JEM in GANGA. Shown is the development of the CPUs load and memory usage of a user job over the job's run time.

5.5 Deployment strategy

For the deployment of JEMs library, one has to distinguish the library as a whole, and the part of it that is submitted alongside the (Grid) jobs.

This part is deployed to the WNs where the job will run automatically, as described in section 5.2.2. As this happens automatically by design, it does not need to be discussed further. The former case is the actually interesting one, as JEM has to be deployed to each UI machine Grid users may use to prepare and submit their compute jobs at.

For this deployment of JEMs library as a whole, one can further discriminate between two cases: The deployment alongside of GANGA, and the separate installation of JEM alone.

The main application area of JEM at the moment is ATHENA analysis jobs on the WLCG. As those jobs are to a significant fraction executed using GANGA, the main focus of the developers lies on deployment of the monitoring software as a plug-in of GANGA. Thus, the release cycle of JEM is synchronized to the one of GANGA, gaining the benefit of automatic deployment in the whole WLCG. Furthermore, as many Grid sites don't maintain own ATLAS software installations, but rely on AFS access to central installations at CERN, the deployment is eased further. JEMs library is deployed to CERNs central software repository and automatically is available on all sites using AFS access.

Apart from that, single (custom) installations of JEM are supported by providing a manual library download from the project's web site.

5.6 Shortcomings of this version of the software

The described monitoring software works sufficiently for small test applications, submitted to the WLCG and run locally on the same machine as JEM. The GANGA-integration fulfils the requirements for exemplary test applications, as well. However, several shortcomings of this version of the Job Execution Monitor prevent the widespread deployment and usage in a Grid-wide distributed scale,

and lower the user acceptance and suitability for remote Grid job failure reasoning. After these shortcomings have been listed, the solutions to them are described in the next sections of this work.

Scalability issues

When trying to scale up the usage of JEM to production level in terms of monitored script complexity, length and the desired verbosity for typical Grid jobs in the WLCG, one observes data loss and increasing data transmission delay leading to a massive data loss at the end of the job's execution (as the backlog of monitoring data that has accumulated until then is too large to be processed and transferred before the job exits). Furthermore, it is unfeasible to execute JEM in parallel for all of a Grid job's sub-jobs (in case of splitted jobs, see section 2.3).

These problems are based on a number of architecture- and implementation-specifics in JEM described in short in the following.

- **inter-process communication**
  The choice of using named pipes for all JEM-internal inter-process communication turned out to be insufficient for our purposes. The involved POSIX wrapping and the involved kernel buffer whose size cannot be tweaked lead to data loss when writing data too fast. Also, its strict first in, first out (FIFO) semantics are too restrictive for advanced data selection and filtering techniques. Furthermore, named pipes are fragile to setup and error-prone if the initialization order is not adhered to, or single communication partners misbehave, and there are situations where the POSIX wrapping fails (some file systems do not allow for virtual file entries). Finally, the design decision to use an ASCII-based string-representation for the transmission protocol introduced a large marshalling-overhead.

- **internal data management**
  Several internal components of JEM buffer monitoring data in thread-safe queue data structures provided by the Python system library. While these data structures provide an easy way to ensure thread-safety in asynchronous data management, they add a source of buffering errors like data loss to the application. As a comparatively small buffer is used by the Queue objects, if data is queued too fast, data is lost. On top of that, the data structures are used throughout the code at locations where thread-safety is not an issue; the reason for usage of the Queue-objects is merely the ease of implementation.

  Additionally, data is discarded by JEMs WN component indifferently if a data rate is detected that is considered too high. No architectural measures have been taken to allow for more "intelligent" data selection later-on.

- **R-GMA transfer**
  It has been determined that data gets lost during R-GMA data transfer,[41] as the system is optimized to the propagation of single events, and not designed for continuous data streams like the one created by JEM. Also, the signalling phase of a R-GMA-transmission session takes relatively long, a period in that early monitoring events cannot be transferred because the data transfer channel is not ready yet. On top of that, the R-GMA software is not maintained as regularly as one could desire.

- **risk of data flood on the UI**
  Despite the aforementioned discarding of data, if JEM would be run on all WNs of a Grid split job, creating a stream of monitoring data to the same UI at the same time, the risk of the sum of the data streams being too much for the UI to handle becomes a critical issue.

  To prevent this scenario, JEMs usage has been restricted to the first sub job of a split job, when Ganga is used for job submission. This automatic safety limitation, though, can be easily evaded by the user, and is not enabled at all when applying JEM to a Grid job manually. Also, it is not desirable to limit job monitoring to only one of a job's sub jobs, and as such, the limitation can only be considered a temporary workaround.

Blind spot outside of Bash- and Python-scripts
JEM gives insight into a job's progress for the periods of execution it spends inside of Bash- and Python-scripts. Everything else, especially the periods in compiled binaries, is still a blind spot for the user. This is unfortunate, as often, these periods in particular are interesting if job failures occur, especially in the main target application of JEM: user analysis algorithms based on the Athena framework, written in C++ and compiled to a shared library loaded by the runner script.

Suboptimal maintainability and extendibility
Several aspects of the concrete implementation of JEM, caused by its prototype character and quick implementation turnaround with multiple developers, make it difficult to maintain and extend the system, and to find and fix programming errors within. The entrance of new developers is also made unnecessarily difficult this way. A non-coherent naming scheme of classes and files, and discussion-worthy naming of some of the frameworks components add to this problem.

The mapping of all monitoring data on a fixed table structure, as necessary for the R-GMA data transmission, also must be critically questioned for future maintainability.

Another problem of the current implementation is the reliance on temporary directories and files for many purposes (not only the POSIX representation of

the named pipes, but also user-specific and system-global temporary directories). This technique is error-prone and highly file system dependent; on top of that, it introduces security problems by making the system vulnerable to so-called symlink attacks.

Finally, several helper components of JEM are pieces of custom code whereas well-proven open source components exist which provide the same services while being feature-rich and customizable. In particular, a logging facility, the configuration management and handling of command line switches are candidates for using established third party libraries.

6 Conclusion

In this chapter, the previous implementation of the job monitoring software was presented. This version was functioning and could be used to analyse Grid job failures in WLCG jobs in real time. Still, it reached its design goals only for comparatively small test jobs and lacked scalability to long-running jobs and large split job sets; it also could not show the job's progress inside of the actual user algorithm, if this was implemented in C/C++.

These issues with the presented version of the job monitoring software were addressed by two novel developments. The first one - a binary tracing module, presented in part III - attempts to fill the blind spot existing in the monitoring coverage of JEM by providing insight into the execution of binary modules if prepared appropriately. The second addition, a scalable and flexible data selection facility designed to allow for adaptive reduction of the amount of transmitted monitoring data, once again includes a refactoring of JEMs architecture to improve its extendibility and maintainability. It is presented in part IV.

Part III
Tracing the execution of binaries

Without the monitoring software presented in the previous part of this work, all of a Grid job's run during the job state `RUNNING` is a blind spot for the user. Submission- and middleware-errors are reported as such, but errors in the job's run itself only give sparse error messages, or nothing at all. JEM, in the presented version, reports what happens during running state, for the periods of time where the job processes Bash- and Python scripts.

The physics data analysis itself in our context most often consists of compiled binaries (C/C++ shared libraries loaded and called by a Python launcher script), as discussed in section 2.2.3.

With an optional library developed for JEM, called the **CTracer**, the blind spot is narrowed further, as the job's actions are reported from inside the compiled user algorithms as well. The CTracer logs function or method calls and corresponding return events in/from C or C++ code and passes this information to JEM, possibly including variable and parameter values and types.

In this part, the CTracer library is presented and its architecture and usage explained. Also, the integration of this technology into the current JEM framework is shown, and its limits and open questions discussed.

7 Concept and requirements

The approach of JEMs existing monitor modules to supervise the executed script is to replace the used script interpreter by a modified one, as shown in section 5.3.2. For gathering monitoring data about a binary application's execution progress, on the other hand, this approach is not feasible, as there is no interpreter for binary applications that could be exchanged. One can not inject trace callbacks into an unchanged application at run time, like it is possible in interpreted languages like Bash- and Python scripts.

One can rather take advantage of the fact that in general, users are not interested in the execution of Grid-wide available software frameworks like the ones discussed in section 1.3.3. Instead, the user's own algorithms are the ones to be monitored. Thus, it's feasible to have the user prepare his application in a special way to allow for binary tracing, to have its progress monitored in a monitoring framework like JEM.

This is the approach that was taken in this work. The user code, when turned into the algorithm library shipped with the Grid job and loaded by the Athena framework, is augmented with trace points that then are exploited by the CTracer to collect the monitoring data. This is explained in detail in the following sections. It is to be noted, though, that this application instrumentation worsens the user application's performance by a considerable

amount; the proposed mode of application, thus, is to resubmit reproducibly failing jobs with increasing verbosity, adding the CTracer as needed.

In the following sections, the principal concept of the gathering of monitoring data (and insertion into JEMs data processing) via the CTracer is outlined step by step, by specifying the requirements for such a binary tracing service. The technical details of how this actually works are then described in section 8.

7.1 Event notification

As suggested in the introductory section above, to follow the progress of a binary program's execution, the program code itself has to be instrumented at prominent points, at which a monitoring event is published; this is because there is no interpreter that can inject such monitoring probe instructions into the job's execution.

The most obvious points in the application's code to instrument are function calls and -returns. This includes method calls/returns in applications built in object oriented languages, as in the compiled binaries, there's no semantic difference between the two concepts - although there may be differences between functions and methods in how exactly the calls and returns are being implemented in machine code by the compiler. In the remainder of this work, the term “function” will be used, and may be exchanged by “method” as appropriate.

Although it also may be possible to instrument arbitrary code locations inside of functions (e.g. each begin of a new scope, each branching instruction, etc.), one has to be aware that for binary applications, a much higher command execution frequency is reached than in interpreted languages like Bash scripts or Python. Just instrumenting calls and returns seems to be the ideal trade-off between monitoring granularity and overhead.

The CTracer must, therefore, be able to detect function calls and returns, and create monitoring events similar to those created by the Bash- and Python-monitors. Ideally, the user code should not need to be changed for this instrumentation to work; more precisely, the user should not need to insert instrumentation calls manually.

The monitoring events created at function calls and returns should, at the least, include the timestamp of when the event occurred and the source code location of the called/returned-from function and its caller. This of course requires the inspected binary to include information needed to refer to the source code location of arbitrary byte code locations (see section 7.5 for information on how this is achieved in the CTracer).

7.2 Symbol resolving and identifier lookup

To bring the instrumentation on par with the corresponding script monitor implementations, and to make the created application progress data usable in

the context of user job monitoring, symbols should be resolved in the scope of the generated call- and return events. This means, call arguments and return values should be determined and added to the monitoring event data, as well as possibly the local (automatic) variables in the scope of the entered/left function.

Also, identifiers should be looked up, if possible, to help the user in identifying the code location (by readable function name and type) of the calling- and the called function, and the context (by readable argument names and types, and local variable names and types).

7.3 Application memory inspection

If desired, a helpful addition to the mentioned monitoring data would be the current values of function call arguments and local variables. This essentially means the CTracer should be able to read memory locations associated with arguments and variables (on the application's stack as well as on its heap) and format that data according to the determined data types, if possible.

This memory inspection, practically turning the CTracer into a remote debugger (albeit a non-interactive one), should be made an optional augmentation of the monitoring process, because it likely increases the instrumentation's overhead in terms of application execution speed significantly.

In addition, measures have to be taken to prevent the monitoring software causing user application crashes due to invalid memory accesses - for example, in the case of uninitialized or "dangling pointers"[17] in local variables, etc. This is particularly important because the CTracer has no possibility of assuring the validity of user pointers. If the user code defines a pointer to nowhere and the monitoring software tries to resolve pointed-to memory locations and read the data from there, efforts have to be taken to secure those accesses.

7.4 Publishing of the gathered data

The created monitoring events (function calls and returns with symbol names, types, and possibly values) then must be fed in JEMs monitoring data stream in the same format as the similar events created by the existing script monitors. In the current version of JEM, this means writing the events into a named pipe in the appropriate format.

7.5 User code prerequisites

As declared a requirement of the CTracer, the user code should not need to be manually instrumented by the user - this means, no changes to the actual source code of the user application is necessary. There is, however, the need to

[17]a "dangling pointer" is a memory reference that became invalid, for example because the memory it points to was released in the meantime

prepare the user application in a special way for the CTracer to work, and at present there is a restriction on the supported binary formats of the user application.

To be able to resolve symbol names and types, and to be able to read symbol values from the associated memory locations, the monitored binary must

- be of the Executable and Linkable Format (ELF) format (32/64bit)
- not be optimized by the compiler
- include debug information.

As will be described in the following section, the CTracer works by exploiting a special functionality of the GCC compiler that adds instrumentation callbacks to each function at its begin and end, that can be implemented by custom code dynamically linked to the run application. This means, only applications created with this compiler can be monitored by JEM at the moment.

To summarize, user code to be monitored with JEM and the CTracer must be compiled with the GCC, with added compile flags similar to the ones shown in figure 20, that add the trace callbacks (`-finstrument-functions`), add debug information to the binary (`-g`) and disable the compiler's optimization techniques (`-O0`).

```
$ gcc -o myApp myApp.c some_more.c -lsomeLib -g -O0 -finstrument-functions
```

Figure 20: Example GCC call to prepare a user application for the CTracer

8 Architecture and implementation

The architecture of the CTracer, implementing the tasks declared in the previous section, is a dynamic library written in C and shipped with JEM together with Bash scripts to augment a user application with it, that has been prepared according to the specified user code requirements. The restriction on a specific compiler for the user code - that is possible because in the WLCG, practically only the GCC is used - defined part of the library's architecture. The finished and working proof-of-concept library is now presented in this section and its inner workings explained.

The CTracer consists of three major parts, providing the aforementioned functionalities: The registration for notification about function call- and return-events, the resolving of symbols and their metadata (names, types) as well as their content (if desired), and finally the insertion of the gathered data in JEMs data stream.

8.1 Event notification

When an application has been compiled with the GCC and with the special trace flag `-finstrument-functions` enabled for the compiler, a library dynamically linked to it is notified of all call- and return-events in the application. For this, the library must implement two special callback functions that automatically get called on function entry and -exit in the application. Those callback functions are given in listing 2, lines 1 & 2.

These functions get passed the address of the just-called function (the "callee") and the address of the calling function (the "caller"). Those addresses point to the locations in memory where the actual byte code of the functions is stored. This is a read-only, executable piece of memory named **code memory**[18] that is shared between all running instances of the application.

Because it can be relied on that the GCC was used to build the application (otherwise, the code instrumentation would not be present), another GCC-specific functionality can be exploited by the CTracer to gain further information that is needed for symbol and value resolving. This built-in extension function, shown in listing 2, line 3, returns the address of the current function's **stack frame**. This is a pointer to the memory location where the application's **call stack** is stored at. With this, information about the current

[18] when an application is executed, its byte code is loaded from disk to the code memory first

```
1 void __cyg_profile_func_enter(void *func, void *callsite);
2 void __cyg_profile_func_exit(void *func, void *callsite);
3 void *__builtin_frame_address(unsigned int level);
```

Listing 2: Trace callback function signatures

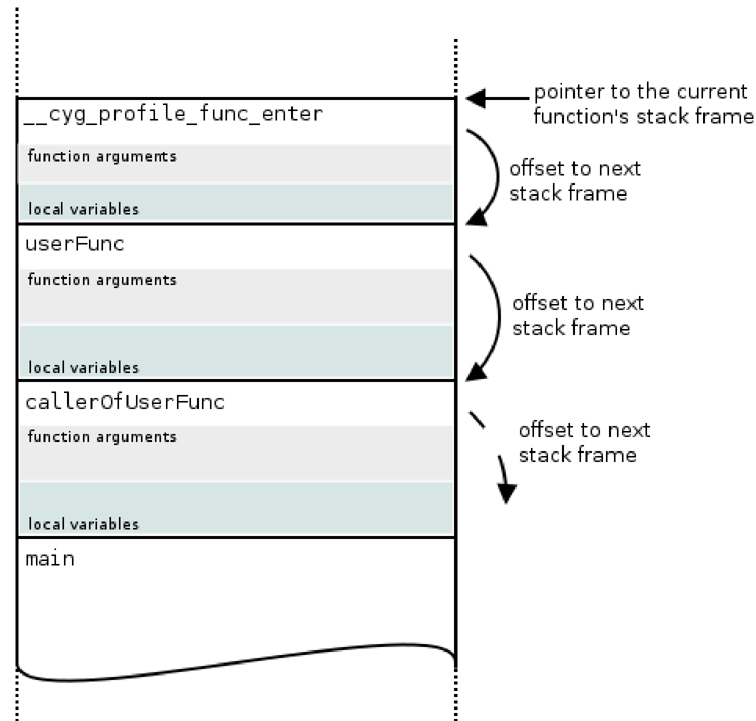


Figure 21: The principle of call stack traversal. The usual call stack when the trace callback has just been called is shown. Note that the callback itself is at the top of the stack, and the to-be-traced function is next.

function's automatic variables (local variables and function arguments) can be gathered. By using the pointer to the stack frame, one also is able to traverse backwards through the call stack, allowing access to the caller's stack frame, the stack frame of the function calling the caller, and so on. This is depicted in figure 21.

In this context, it has to be noted that the trace callback-function itself of course is the topmost function on the call stack. So, `__builtin_frame_address` will return the pointer to the trace callback's stack frame and the described stack frame traversal must be performed at least once to get data about the callee's stack frame. The function accepts, however, the argument of the desired level (the number of traversals to perform before the address is returned); thus, the traversal has not to be performed manually.

8.2 Symbol and value resolving

To determine what symbols exist in the current function's scope (automatic variables), and to get symbol metadata (data types and names) about those symbols and the function itself, the debug data contained in the binary file is used. In all GCC versions in usage nowadays, this debug data is in the

"DWARF" format[42][19]. Since in the homogeneous Grid software environment in the WLCG, GCC 3 and GCC 4 are the available compilers, one can rely on the DWARF format to be used for debug information.

The debug data is loaded from the binary by the CTracer on application start-up and indexed by (function-, variable-, ...) address to allow rapid lookup during the binaries execution. A custom DWARF-like data format had to be developed for this purpose, because the built-in formats of existing DWARF-reader libraries are not optimized for such a real time access. Since all addresses naturally are of the same size (32 or 64 bit), and after the insertion at the application's start, a rapid, random read-access is most important while no entries are added or deleted any more, a red-black tree[43] was considered the optimal data structure to be used here. This red-black tree is later used in other parts of the rewritten JEM framework, as well.

To remove uninteresting call- and return-events from the monitoring data as soon as possible, and to reduce the performance impact of the CTracer to the user application, at this stage the function entries in the red-black tree can be tagged as "to be ignored", and a simple black list is used to tag all functions from system libraries automatically.

If the CTracer was configured by the user accordingly (see section 9.2.3), after the symbols have been looked up and their metadata extracted from the look-up structure, the current values of all symbols are tried to be read from memory (the DWARF data in the file contains data access information like dataset sizes, offsets from the stack frame, indirection information and so on, for this purpose). To protect the user application from crashing by those memory accesses, a special protection system had to be added to the CTracer.

8.3 A victim-thread for safe memory inspection

Invalid memory accesses usually result in the offending application to crash, that means, the application cannot continue running. This is an important safety measure enforced by the operating system, because usually, the application's state after such an invalid access is undefined, and random undesired effects could happen if the application would continue running.

In the case of the CTracer accessing user-defined memory addresses for the purpose of symbol value resolving, though, this protection mechanism is not wanted. Because the CTracer works by dynamically linking it against the user application at run time, the application that would crash on invalid accesses is, in fact, the user application. As user job monitoring should by no means alter the user application's semantics, this would not be acceptable.

On the other hand, there is no other possibility to read an application's private memory than doing so from within its own process. Accessing another

[19]DWARF originally was no acronym; the debug data format was called so to complement the file format name, ELF. Later on, the meaning "Debugging with attributed record formats" was invented, turning DWARF into what is known as a "backronym".

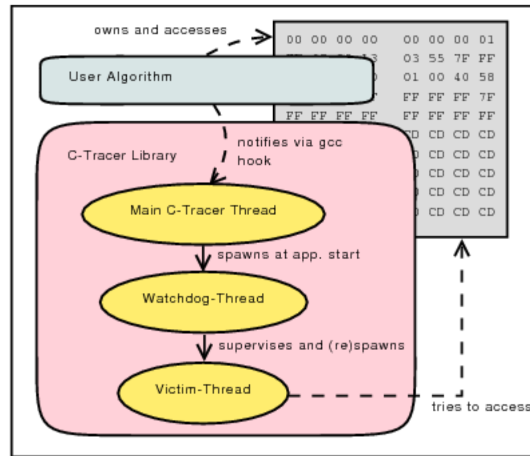


Figure 22: General concept of the CTracer's victim-watchdog system

processes private memory is considered harmful and leads to process abort by the operating system. Consequently, a method had to be found to safely read the memory while preventing the user application's execution to be aborted on failure to do so.

This method, in the CTracer, is a specifically designed, dedicated "victim-watchdog" system.

8.3.1 Concept and architecture

The victim-watchdog system is depicted in figure 22. The main components are a **victim thread**, in which the actual memory access is performed, and a **watchdog thread** supervising the victim thread and restarting it, when needed. As the name suggests, the victim thread is "allowed to" crash on invalid memory accesses, and is restarted if that happens. If the access succeeds, the read value is handed to the main CTracer thread. If the access failed - for example because the user memory has not been initialized yet - the main thread is notified, as well.

This system works as intended on Linux systems because a thread abort as a result of an invalid memory access is performed by the operating system by means of a **synchronous** signal delivered exclusively to that thread. Thus, the thread can handle the signal and leave the main thread of the application unaffected. If no such signal handler would be registered, all threads would receive the signal and the application's run would be aborted.

8.3.2 Usage by the CTracer

The described system for safe memory access is used by the CTracer by launching the watchdog thread at application start, after all DWARF data has been loaded and indexed. The watchdog thread in turns spawns the victim thread, and both threads suspend, waiting for memory locations to read.

On function call- or return-events, in the executed callback function (see sec. 8.1), when a symbol is encountered whose value should be determined - for example, a function argument - the respective metadata (address of the argument on the function's stack frame and the argument's data type specifying how many bytes should be read from that memory location) is passed to the victim thread. The main thread now suspends until it gets either the wanted data from the victim thread, or the notification that the victim thread crashed, by the watchdog thread. In the latter case, the argument's value is denoted by "not available" in the monitoring data handed to JEM.

8.4 Resulting monitoring data

As opposed to the Bash- and Python-monitors, in the CTracer, only call- and return-events are available (for reasons given in sec. 7.1). The data added to JEMs monitoring data stream, then, includes the event type ("call" or "return"), the timestamp at which the event happened, the code location (file, frame and line number) of both the caller and the callee, and a list of variables with their type and name (function arguments, local variables).

If the victim-watchdog system is enabled, the list of variables is augmented with the variables' contents. For structural- or pointer-variables, the contained (or referred-to) symbols are included into the data, as well. A maximal indirection level is configurable; this is needed to prevent data flood and to properly handle circular references. An example of data that can be extracted from a running program is shown in figure 23. The displayable data resembles the visible data in a debugging software. For this reason, the CTracer can be called a non-interactive **remote debugger**.

Type	Name	Value
const HelloAlg*	this	0x09b06220 [= <HelloAlg instance
bool	m_myBool	1
double	m_myDouble	3.14159
int	m_myInt	42
int	m_runCount	0
map<std::basic_string<char, ...	m_myDict	<map<std::basic_string<char, std::
ToolHandle<IHelloTool>	m_myPrivateHelloTool	<ToolHandle<IHelloTool> instance @
ServiceHandle<IToolSvc>	m_pToolSvc	<ServiceHandle<IToolSvc> instance
IMessageSvc*	m_pMessageSvc	0x0b03771c [= <IMessageSvc insta
ISvcLocator*	m_pSvcLocator	0x0ad798b4 [= <ISvcLocator instar
ToolHandle<IHelloTool>	m_myPublicHelloTool	<ToolHandle<IHelloTool> instance @
vector<std::basic_string<char...	m_myStringVec	<vector<std::basic_string<char, std
vector<std::pair<double, dou...	m_myTable	<vector<std::pair<double, double>
vector<std::vector<double, st...	m_myMatrix	<vector<std::vector<double, std::a

Figure 23: Excerpt of example data gathered using the CTracer. One can see the hierarchical structure of referred-to data, that is read by the CTracer up to a configurable indirection level.

9 Usage

To allow for flexible application of the described CTracer library both stand-alone and as part of JEM, it has been designed in a way that allows both kinds of work flows. Being implemented in C and deployed as a shared library, it is configurable by means of a set of environment variables (that can be defined in the shell on Unix systems and is provided to all spawned processes).

Furthermore, the CTracer can be configured to directly publish the gathered monitoring data into log files, or write it into a specified named pipe (for JEM integration). This made, on the one hand, the development and testing of the CTracer an comparatively easy task, as JEM didn't have to be used for all test runs. On the other hand, it is an expandable design that can be built upon for further JEM version iterations.

9.1 Stand-alone execution for custom binaries

The CTracer is applied to an application's binary - that has been prepared as described before - by setting the environment variable `LD_PRELOAD` to the path to the CTracer's shared library (`libctrace.so`) before launching the application. This instructs the dynamic loader (`dl`) to load the shared library into the program memory alongside the binary's image.

The shared library contains implementations for the trace callback functions (see listing 2) the binary is instrumented to call at each function call and return; this way, the CTracer's functionality is added to the user application.

After the to-be-traced application exits, the shell variable should be unset, to avoid unnecessary loading and initialization of the CTracer. The setup and removal of this shell variable is achieved most easily by wrapping the application into a short shell script like given in listing 3. In this script, the CTracer can also be configured by setting further environment variables, allowing to control the CTracer's verbosity, the way the data should be published, and more. The most important configuration variables are given in table 5.

The user has to configure the CTracer to be invoked for some custom code by specifying both the to-be-monitored binary (executable or library) and the binary on whose launch the CTracer is initialized, as shown in the table. The former binary will be called the **trigger-binary** in the remainder of this work, the latter one the **traced binary**. These two binaries may - but do not necessarily need to - be the same. For example, if a shared library (`.so` file) is to be monitored, the binary that is launched by the user (or a script) loading this library is the trigger-application.

This distinguishment is needed to avoid the necessity to load the CTracer shared library against each and every application launched in Bash scripts. Instead, only for the trigger-binaries, the CTracer is loaded and initialized. During this initialization, the traced binary - from which the debug information

option	description
JEM_CTRACE_DISABLE	By setting this to "1", the CTracer is loaded, but takes no action in the callback functions. The overhead, then, reduces to two additional function calls/returns per function called in the traced module.
JEM_CTRACE_APPS	This defines the trigger-application(s) for the CTracer. For applications not in this list, the CTracer gets disabled automatically. If used from within JEM, the CTracer library is not even loaded for them.
JEM_CTRACE_MODULES	This defines the traced binaries. The symbol- and metadata, as described in section 8.2, is loaded from these modules. This implies that for modules *not* in this list, even if instrumented because of the CTRACE_APPS option, no symbol resolving can be performed. Thus, the CTracer ignores the callbacks for those modules.
JEM_CTRACE_FIFO	The path (file system entry) of the named pipe used to publish the data is configured with this variable.
...CONTINUE_ON_PIPE_ERROR	If set to "0", the application's execution is aborted if the CTracer cannot successfully open the IPC named pipe to JEM for writing.
...DONT_RESOLVE_VALUES	If set to "1", the victim-watchdog system is **not** used to resolve symbol values. This speeds up the application's execution by a considerable amount.
...MAX_STRUCT_DEPTH	The maximum number of indirections the CTracer will follow when resolving the values of pointers or struct members.
...EXCLUDE_CALLERS ...EXCLUDE_CALLEES ...EXCLUDE_BOTH	These options can be used to specify a list of header files; the functions defined in those headers will not be traced (when called), the functions called *from* this functions are not traced, or both. This is used to filter out system library-internal calls to speed up the program execution.

Table 5: Configuration options of the CTracer. The prefix "JEM_CTRACE_" is omitted for the later options.

```
export JEM_CTRACE_APPS="./myApp"
export JEM_CTRACE_MODULES="./myApp"
export JEM_CTRACE_CONTINUE_ON_PIPE_ERROR="1"
export JEM_CTRACE_MAX_STRUCT_DEPTH="2"

export LD_PRELOAD=/path/to/JEM/JEMlib/wrapper/ctrace/libctrace32.so
./myApp
unset LD_PRELOAD
# more un-traced commands
```

Listing 3: Wrapper script for manual invocation of the CTracer

is loaded - can be either the trigger-application itself or one of its loaded shared libraries.

To illustrate this concept, consider a shared library written in C++, containing an Athena user algorithm, called `libMyAlgo.so`. This algorithm obviously is the traced binary, because the user wants to see his algorithm's progress using JEM. The Athena framework, however, is launched by the runner script using a Python start-up-script (see section 1.3.3); so, the trigger-application in this case is the Python interpreter.

9.2 Integration into JEM

The CTracer was integrated into JEM with one premise: In the first version of the CTracer, the binaries monitored by it are assumed to be spawned from Bash scripts. Thus, the Bash monitor was extended by code that detects binary launches for a configurable set of binaries that are to be traced, and then launches the CTracer for these programs automatically.

While refactoring JEM in line with the implementation of the new project structure presented in part IV of this work, the integration of the CTracer into the Python monitor was added. This way, binaries launched from Python scripts (e.g. using `os.system('someApp')`) are monitored using the CTracer, as well.

9.2.1 Configuration and invocation

The CTracer's configuration, that works by using environment variables as described above, is prepared automatically by JEMs Bash monitor for the spawned binaries, instead of letting the user set the variables manually in the Bash script (like it is shown in the stand-alone usage case). Most settings for the CTracer were duplicated as JEM configuration options to allow the set-up of the CTracer for its usage inside of JEM.

The CTracer is invoked by the Bash monitor for a given binary by automatically setting the `LD_PRELOAD` shell variable, advising the dynamic loader to add the CTracer's shared library to the binary on execution, and by resetting this shell variable after the binary exited. This automatic invocation

of the CTracer is only performed by the Bash monitor for the binaries listed in the `CTRACE_APPS`-option. This way, the loading of the shared library and thus the initialization overhead is prevented for generic executables inside the Bash script.

9.2.2 Insertion of CTracer-data into JEMs data stream

Having the CTracer insert its monitoring data into JEMs data stream concludes the JEM-CTracer-integration. This is achieved in the discussed version of JEM by giving the name of the named pipe used by this JEM instance to the CTracer. It opens the virtual file and writes the data directly in the expected format.

Since the data provided by the CTracer, as listed in section 8.4, is similar to the data for function call- and return-events created by the Bash- and Python-monitors, the format JEM expects to be sent over the named pipe is well defined. This format, being designed for version 2 of JEM in 2007, is a key-value-formatted ASCII-text representation of the data. With standard POSIX system calls, this data is written by the CTracer, after being formatted in a custom memory buffer allowing for efficient composition of string data.

The main design challenge of this buffer - besides of making it re-usable to avoid the allocation and deallocation of memory for each call- or return-event - is that it is not known in beforehand how long the resulting string will be. Furthermore, during the composition of the string, it is not known if the next part of it needs to be appended at the end of the string, or prepended to it. It would be unperformant to move the existing data backwards to create room for the new data if it needs to be prepended, neither would it be good to have to resize the buffer often.

The approach taken in the customized buffer for this reason is to estimate the maximum length of typical data that needs to be assembled, allocate twice as much memory, and start the data insertion at the middle. By keeping two pointers to the buffer, one at the start of the written data, one at the end, it is possible to append data by just writing behind the data and moving the end-pointer, and to prepend data by moving the start-pointer backwards and writing to this position. After assembly, the data can be retrieved by adding one 0-byte after the end-pointer, and copying the data from the start-pointer onwards. This concept is visualized in figure 24.

9.2.3 Augmentation of the JEM-Ganga-Integration

To ease the usage of the CTracer for the main target user base of JEM, the Ganga integration, efforts were taken to automate as much as possible in this scenario. For example, the user does not have to specify the trigger-application if Athena is used as the job's application - the used Python-interpreter on the WN is filled in by Ganga in that case.

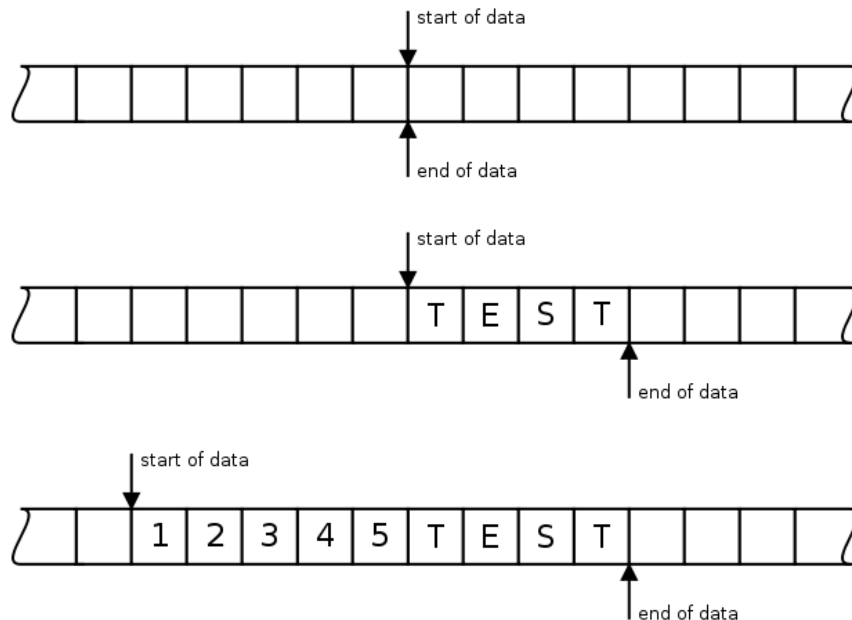


Figure 24: Schematic display of the custom appendable buffer. As is shown, data can be appended ("TEST") and prepended ("12345"). Getting the value of the buffer now would yield "12345TEST".

The configuration options described above are provided on the JEM-object in GANGA, as well, to allow for easy configuration of the CTRACER for the job run.

Finally, the data created by the CTRACER is automatically available from within GANGA, because it gets inserted into JEMs data stream transparently and is processed the same on the UI.

9.3 Application for HEP Grid jobs

As a use-case example for the CTRACER, a typical HEP Grid job is now considered, with its preparation, configuration, submittage and the results that can be gained by adding the CTRACER to the monitoring. The user application, in this example, is based on the ATHENA framework - a typical case in the ATLAS-collaboration.

A basic "Hello World" algorithm featuring next-to-no functionality is taken from an ATHENA tutorial[44] and modified by the user. This example-algorithm, delivered to all users with the default ATHENA distribution, performs a stub iteration over some fictional data samples and creates some output per sample, but no actual calculations are done.

9.3.1 Preparation of the user application

In section 1.3.3, the process of writing a custom, Athena-based user algorithm already has been briefly described. The user runs CMT to check out the base packages he wants to use and modify, writes the custom code in C++, and again uses CMT to have the library built on his PC. When using Ganga, the deployment of the library and the inclusion in a WLCG Grid job is then automated.

However, as a first step, the to-be-written algorithm has to be described with its dependencies in a special `requirements`-file. This file is used by CMT to resolve the needed packages to build the user code, for the build process configuration, etc.

Since the build process must be customized as described in section 7.5 to allow the usage of the CTracer, a user has to modify the `requirements`-file accordingly (there is no manual invocation of the compiler one could add the needed compile flags to). Listing 4 shows a CMT `requirements`-file for the simple “Hello World” Athena algorithm,[44] with the modifications needed for the CTracer to work already being included (lines 12 - 15).

```
1  package HelloWorld
2  author ATLAS Workbook
3  # declare package dependencies
4  use AtlasPolicy AtlasPolicy-01-*
5  use GaudiInterface GaudiInterface-01-* External
6  use AthenaBaseComps AthenaBaseComps-*   Control
7  # declare to-be-built library
8  library HelloWorld *.cxx -s=components *.cxx
9  apply_pattern component_library
10 apply_pattern declare_joboptions files="HelloWorldOptions.py"
11 # JEM CTracer needs debug symbols and trace instrumentation
12 private
13 macro_append cppflags " -finstrument-functions -g -O0"
14 macro cppdebugflags '$(cppdebugflags_s)'
15 macro_remove componentshr_linkopts "-Wl,-s"
```

Listing 4: Example CMT `requirements`-file for a simple “Hello World” Athena library, including CTracer-specific additions for the build process

9.3.2 Activation and configuration in Ganga

When the custom algorithm was successfully built as described, an Athena Grid job can be prepared for submission in Ganga, similar to the example in figure 18 on page 66. All a user has to do to have his custom library be monitored with the CTracer while the job is running on a Grid WN then is to modify the configuration of his job inside Ganga (see figure 25) before submitting.

```
j.info.monitor.ctracer.enable = True
j.info.monitor.ctracer.traceModules='.../libHelloWorld.so' # path omitted
j.info.monitor.ctracer.traceApps='' # app is set automatically for athena
                                    # jobs in Ganga
```

Figure 25: Additional GANGA-job-configuration to enable the CTRACER

As the job now begins running on the worker node, the runner script is executed using JEMs Bash monitor; When ATHENA is launched through the Python monitor (that itself is run through the system's Python interpreter, see sec. 5.3.2), the CTRACER is loaded and attached to the newly spawned process. Function call- and return-events inside the user library now are traced and reported to JEM automatically.

9.3.3 Results and interpretation in an example run

Figure 26 shows the typical sections of an ATHENA job execution, overlayed with its network activity. The sections implemented in Bash- and Python-scripts are marked, as is the time spent in the compiled user library. The visible phases are the environment- and CMT-setup (bash script, marked yellow), ATHENA initialization (python script, marked orange) and data stage-in (green), followed by the actual user algorithm run over the set of physics events (binary execution, marked blue).

As can be seen, the actual analysis is an important part of the job's execution to monitor. Also, its relative size compared to the fixed setup- and shut-down-periods scales with the number of physics events processed, and the usual ATHENA job processes orders of magnitude larger numbers of events compared to the small test jobs analysed for this work. The CTRACER helps in removing the blind spot in this section of the job.

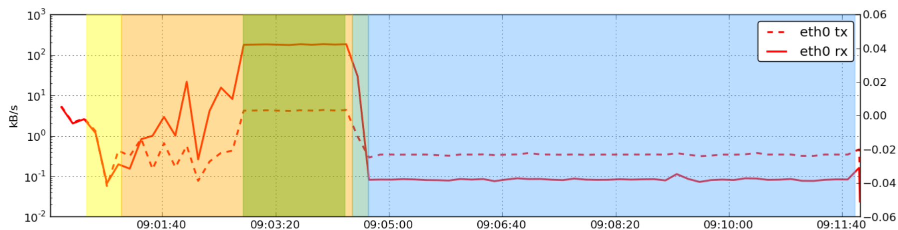


Figure 26: Periods of a typical ATHENA job, shown as overlay over a network traffic measurement plot (tx: sent data, rx: received data).

An excerpt of the events inside the user library as seen by the CTRACER is shown in figure 27. The user can see easily when the processing of a new physics event started, and together with the stdout output, he can correlate which event number is processed at that time. If the job aborts, the user knows the last few event numbers before the crash.

File	Frame	Called File / Exception	Called Frame / Exception Reason	Code / Command
GaudiHandle.h:194	void StatusCode::StatusCode			
ToolHandle.h:121	StatusCode GaudiHandle<IHelloT...			
HelloAlg.cxx:103	StatusCode ToolHandle<IHelloToo...			if (m_myPublicHelloTool.retrie...
HelloAlg.cxx:103	StatusCode HelloAlg::initialize	StatusCode.h:131	bool StatusCode::isFailure	if (m_myPublicHelloTool.retrie...
HelloAlg.cxx:103	bool StatusCode::isFailure			if (m_myPublicHelloTool.retrie...
HelloAlg.cxx:107	StatusCode HelloAlg::initialize	GaudiHandle.h:38	const string (= basic_string<ch...	log << MSG::INFO << m_myP...
HelloAlg.cxx:107	const string (= basic_string<cha...			log << MSG::INFO << m_myP...
HelloAlg.cxx:110	StatusCode HelloAlg::initialize	StatusCode.h:104	void StatusCode::StatusCode	return StatusCode::SUCCESS;
HelloAlg.cxx:110	void StatusCode::StatusCode			return StatusCode::SUCCESS;
???:-2	StatusCode HelloAlg::initialize			
???	???	HelloAlg.cxx:162	StatusCode HelloAlg::beginRun	StatusCode HelloAlg::beginRu...
HelloAlg.cxx:167	StatusCode HelloAlg::beginRun	StatusCode.h:104	void StatusCode::StatusCode	return StatusCode::SUCCESS;
HelloAlg.cxx:167	void StatusCode::StatusCode			return StatusCode::SUCCESS;
???:-2	StatusCode HelloAlg::beginRun			
???	???	HelloAlg.cxx:115	StatusCode HelloAlg::execute	StatusCode HelloAlg::execute(...
HelloAlg.cxx:146	StatusCode HelloAlg::execute	StatusCode.h:104	void StatusCode::StatusCode	return StatusCode::SUCCESS;

Figure 27: CTracer events in a typical Athena job (excerpt).

9.4 Performance impact

To determine the performance impact of applying the CTracer to an algorithm's execution, a test application was run multiple times with an increasing number of function calls performed (1 to 1.000.000), and with increasing CTracer verbosity. The verbosity levels measured were:

- CTracer not applied at all (shared library not loaded)
- Library loaded / CTracer applied, but disabled
- CTracer enabled but set to treat all events as "ignored"
- Enabled using the default event black list (see section 8.2)
- Fully enabled with value resolving[20])

It is to be noted that the data was taken with the newer version of the CTracer that was adapted to JEMs new infrastructure presented in part IV. Since the measurements were performed in standalone mode (see section 9.1), without passing the data to JEM, however, this doesn't affect the results.

As can be taken from the plots (figure 28), applying the CTracer to a running application in mostly disabled mode (being loaded and enabled, but ignoring all events) adds a modest overhead to the job's execution time, comparable to running the job through classic debugging tools like `strace` or the gdb-debugger.[45] If, on the other hand, all events are recorded, the overhead becomes significant: It can be seen that it will not be feasible to record all events happening. Other approaches have to be found (see the corresponding section, section 17.2, in the outlook in part V).

An interesting effect can be observed for the single-call case: In this scenario, not attaching the CTracer-library at all results in slower execution than attaching it, but in disabled state. A possible explanation for this effect

[20] Value resolving is the inspection of application memory, protected by the victim/watchdog system, as described in section 8.3.

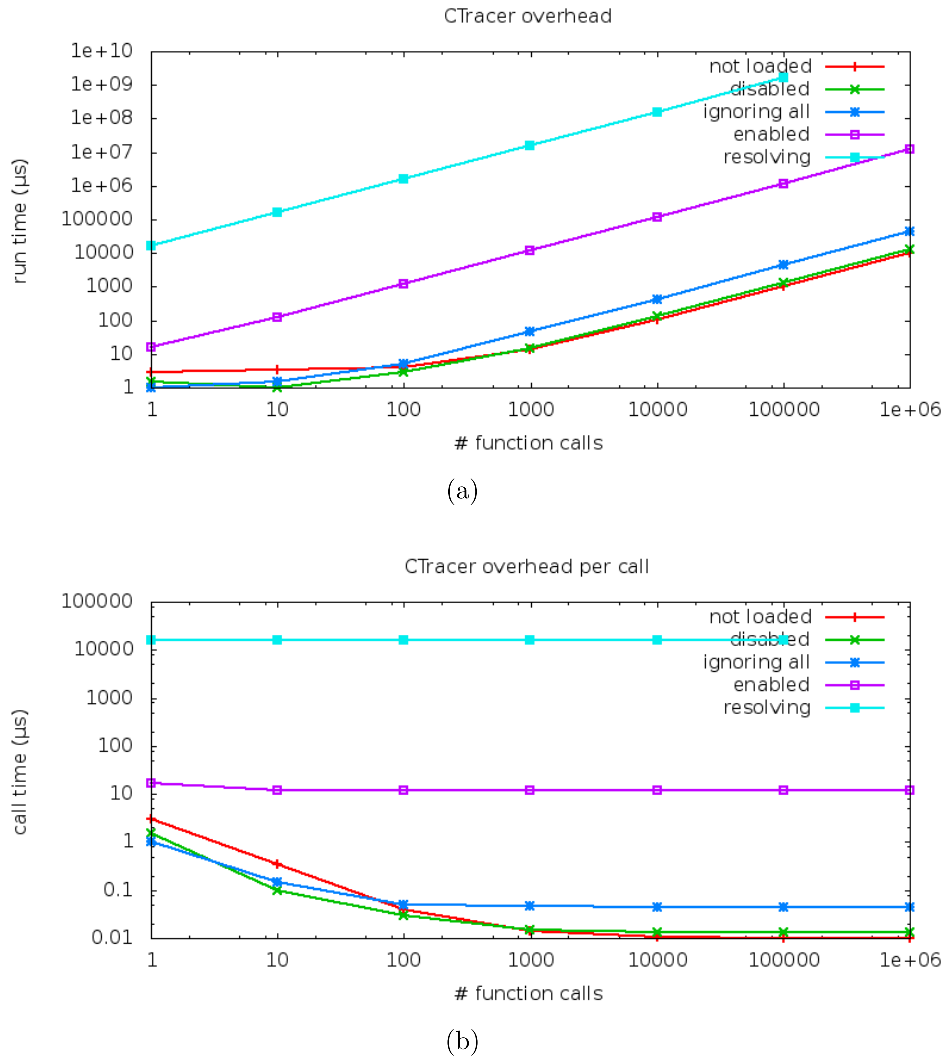


Figure 28: Performance impact of the CTracer. The plot shows the overall (a) and per-call (b) run time of a test application performing 1 to 1.000.000 function calls, for increasing CTracer activation levels. The higher call time for low call numbers can be explained by the one-time setup time of the CTracer, becoming less and less significant with increasing call numbers.

is the search for attached libraries accepting the trace-callback calls by the instrumented user application, performed by the dynamic loader.

Even with the comparatively low performance penalty that is added to the run time if only a small subset of the events is recorded, it is not advisable to add an enabled CTracer to all user analyses regardlessly; this, however, is not the intention of such a debugging tool. Instead, the addition to the job monitoring is to be activated selectively for example on reoccurring, hard to find user code errors happening in the production environment, but not in local user testing, to narrow down the problematic code location.

10 Conclusion

As part of this work, a new user application monitor has been added to JEM: The binary tracing library, CTracer. It can be applied to user algorithms written in C / C++ and reports function call- and return-events to JEM. It removes the last significant "blind spot" from Grid job monitoring at the cost of increased performance degradation. Being a debugging tool, one has to weigh the need for in-depth monitoring data about binary algorithm runs against the need for fast and efficient execution. Whether to apply the CTracer to his algorithm, thus, must be decided by the user on a per-job basis.

Being a proof-of-concept implementation, the first version of the CTracer presented in this part is fully operational for JEM v2 runs. The code quality of the CTracer itself, however, had to be improved to reach production level. In particular, the extraction of the DWARF debug data from the binary files should be performed by an established 3rd party library instead of by custom code.

These issues have been resolved as part of the general JEM-refactoring described in the next section. The CTracer has been updated to use the new internal infrastructure of the software, and was slimmed down to implement only the novel, custom functionality, but to rely on established libraries for the debug data extraction. This refactoring is described briefly at the end of the next part (see section 13.3).

The library, both in the presented and in the refactored state, is suitable to extend the monitoring coverage of JEM onto binary executables and libraries that have been prepared for tracing before. It must be considered, however, that the application of the CTracer is a very in-depth monitoring process and thus leads to considerable performance degradation.

When applied reasonably, for example by iteratively resubmitting jobs with increasing verbosity on reproducible errors, the CTracer can help in narrowing down hard-to-find Grid job failures inside the user algorithm itself.

Part IV
A real time trigger mechanism

One of the biggest problems in the former version of JEM is the lack of scalability. If one tries to apply JEM-monitoring to a sufficiently large and complex user application - and all of the applications in JEMs main target area are that large and complex - monitoring data gets lost, or the whole execution fails due to excess overhead and the resulting job time-out.[46]

These scalability problems can be explained mostly by the correlation between application complexity and the amount of created monitoring data, the increasing delay between creation and processing of the monitoring data in the WN component (because the processing part can not keep up with the data rate), and the limited temporary buffer size where the data is stored before transfer.

To address these issues and to make JEM even more extendible, maintainable and overall useful, a new internal architecture was found to be necessary. This new architecture, inspired by the real time data reduction in ATLAS by means of the **triggers**, is discussed in the following part of this work.

11 Concept and requirements

While the major goal of the refactoring of JEMs inner workings executed by the author was the improvement of its scalability, soon it became clear that these changes also meant an opportunity to introduce a flexible way to react to specific monitoring events with arbitrary predefined behaviours.

One of the most useful behaviours to implement here would be the on-the-fly adaption of JEMs verbosity: **adaptive monitoring**. This way, the amount of monitoring data transferred to the user in the default setting can be kept at a minimum, while on important (or interesting) events, more data gets transferred. The data essentially gets filtered to only process the absolutely needed amount, minimizing performance impacts on the user job's execution.

Also, it would be possible to let the user define his own reaction behaviours to implement custom filtering and flagging of monitoring data. These behaviours are called **triggers** here because they are triggered by specific monitoring events (or combinations of events).

11.1 Extendible chunk format for monitoring data

To enable a real time trigger architecture, as a first step, a new data format had to be designed. It replaces the ASCII-based format used by JEM v2, a format suitable for transfer through named pipes and designed with the mapping to (virtual) relational database tables in mind. This mapping was needed for the transmission via R-GMA (see section 5.2.3), but renders the format not

suitable for the desired flexible data selection techniques. To reach the design goals of small data size overhead and performant processing, a binary format was chosen. The format is suitable for inter-process communication but can also be sent over a network socket as-is.

The main concept of this format for JEMs monitoring data is the **monitoring data chunk**, consisting of a chunk **header** and a number of data **blocks**. Each block, again, is a combination of header and payload. The length of the whole chunk (header plus all blocks) is stored in the chunk header, and the length of each block is stored in its block header.

One is able to iterate through a number of chunks stored consecutively in memory, and through the blocks of a chunk, using these length information. Each chunk has a **chunk type** that corresponds to a monitoring event, and each block has a **block type** specifying its contents.

The chunk format that has been designed is shown in figure 29 and its header fields are described in table 6.

By moving all data into blocks and allowing an arbitrary number of blocks in each chunk, it is easy to extend the data format later-on (just add new chunk- and block types), and have processors skip the blocks not understood (by skipping the number of bytes specified in the unknown block's headers). This way, optional content can be added to the monitoring data at run time. As examples, the ability to add a checksum to all chunks and to add a running number (to detect chunk losses over the network, etc) was implemented[21].

A list of the most common data blocks is given in table 7. The main monitoring chunk types used by JEM are listed in table 8. Chunk types correspond to types of monitoring events. For each chunk type, a list of the typically contained blocks is specified. Most chunks at least contain a similarly-named block that contains the chunk's main information.

Additionally, each chunk may contain optional blocks like a checksum block, identifier-blocks, etc. Some of the blocks, like the identifier-specification or variable-info, can also exist several times in the same chunk.

Because a binary data format is hard to read and interpret for the user, a complementary human-readable format representation was specified. It is an XML based textual representation of monitoring data that contains the same information as the binary chunk format. Figure 30 shows exemplary monitoring data chunks together with their XML representation.

11.2 Chunk backlog and tagging

To enable the flexible data selection described in the introductory section of the present part of this work, a number of central design concepts were established that then must be implemented. While the change of data format and data transfer mode (inter process communication, IPC) might have already solved

[21] If new blocks are added to a chunk, of course its chunk size must be updated in the chunk header.

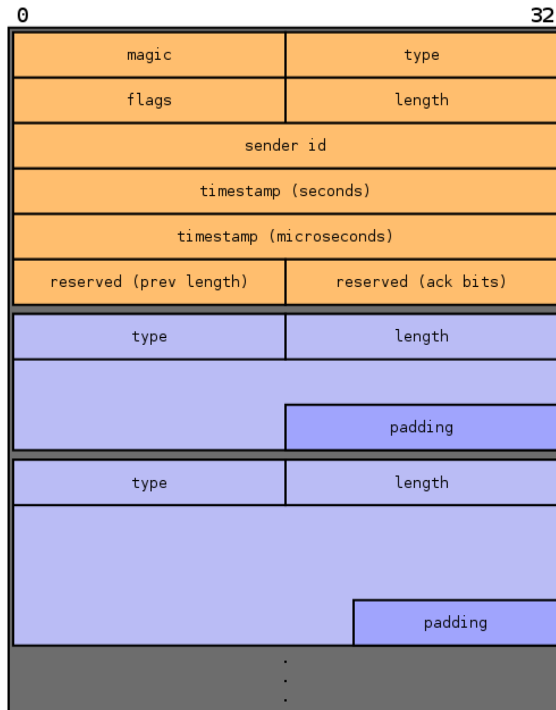


Figure 29: Binary chunk format for monitoring data. Shown is the chunk header and the list of blocks, each with its own block header. The chunks can be stored continuously in memory and iterated over by reading the chunk size in bytes from its header. Each block is padded to 32 bit boundaries (therefore the whole chunk is 32 bit aligned, as well).

the scalability problems, it was decided to proceed one step further and create the trigger system according to the following design concepts.

These design concepts - architecture requirements for the trigger system - comprise at least the following points. As one can easily deduce, the former internal and external data channels[22] of JEM do not suffice to provide, or even allow, all of the declared functionality.

- Several data-creating processes (called "producer" in the remainder of this part) must be able to pass data to the processing component (also called "consumer") at the same time, preferably without affecting (blocking) each other.

- The data format described in the previous section should be used to benefit from its advantages. Most importantly, no data mapping should be necessary when transferring the data to the UI part of the system. In other words, the data format should be applicable both in inter process communication as well as in data transfer over the Grid.

- The consumer should be able to decide, for each monitoring event that was created by some producer, if the event should be discarded or not. Discarded events are not further processed - and as such, not forwarded to the UI. This decision, however, should not only be possible while the event in question is the most recently created event, but also later-on

[22] IPC on the WN and data transfer from WN to UI

field	size	description
`magic marker`	16	These first two bytes are reserved for an internal "magic marker" signature *.
`type`	16	The chunk type (equivalent to monitoring event type) - see tab. 8.
`flags`	16	Space for sixteen chunk flags *.
`length`	16	The non-padded length of the chunk. Readers must pad this value to 32 bit bounds when computing the offset to the next chunk.
`sender id`	32	Each entity that creates monitoring data chunks is called a "sender" in JEM. It is assigned a job-unique four-byte identifier. This can be used for filtering the data by creator or for statistics.
`timestamp`	64	The creation time of the chunk, stored as 4 byte seconds + 4 byte microseconds.
`prev-length`	16	The length of the previous chunk (if known). This can be used to iterate backwards through the monitoring data.
`ack bits`	16	Internal acknowledgement flags for up to sixteen entities *.

Table 6: Chunk header fields. The size is given in bits. Fields marked with * are described in detail in section 12.2.

in retrospect (deferred decision). A FIFO data transfer mode between JEMs processes, consequently, is not suitable any more.

- Because of this deferred decision on whether to keep each monitoring event, the consumer must be able to examine each event multiple times - as often as needed to decide on its processing.

- It should be possible to base the decision whether to process or to discard any given monitoring event on arbitrary criteria. Also, it should be possible to base these criteria on user defined rules. Each rule, implemented as trigger in whatever form, should be able to build internal data structures while the job is running, that it then can use to rate each consecutive monitoring event.

- While events are kept, until they are decided on, they should reside in a single data store. There should be no need to move and/or copy them until they are either discarded or forwarded to the UI. Furthermore, this data store should be big enough to hold all pending events, and its access should be as performant as possible, both for writing and reading.

block type	description / data contents
SENDER_INTRO	The very first time a sender publishes some data, it has to introduce itself and its unique identifier. The information about the sender is contained in this block.
JOB_START	In this block, general information about the job is stored.
TRACE_EVENT_BASE	Trace events are monitoring events about a job's execution, like a function being called, an exception occurring, a command being launched, etc. All those events have in common that they happen at a point in time, correlated to a code location. That data is stored in this block.
CALLEE_INFO	In events relating not only to a single code location, but also to a second one (e.g. the called or returned-from function), this block stores the second location.
COMMAND_LAUNCH	Information about a launched command.
COMMAND_EXIT	Information about a command that exited, and its exit code.
SOURCE_LINE	A line of source code associated with the event. This is used, for example, to publish the code vicinity of an exception event.
EXPRESSION	Information about a script expression executed.
VARIABLE_INFO	The name, type and value of a variable. This can be e.g. a function argument, a local variable or a command line option passed to a launched command.
SYSTEM_INFO	This block stores static information about the computer the job is executed at, like the amount of physical memory, number of cores, etc.
SYSTEM_METRICS	System metrics data (CPU usage, free memory, network traffic, ...).
OUTPUT_LINE	A line of job output (STDOUT or STDERR) with a timestamp.
EXCEPTION_INFO	Information about an exception that occurred in the job (= run time error).
IDENTIFIER	The definition of a 32 bit ID for an identifier. This is used so that identifiers (like function names, file paths, etc.) that appear multiple times in the monitoring data (of a single job run) are sent only once; all further times, only the ID is transferred.
CHECKSUM	A 32 bit checksum of the whole chunk.

Table 7: Monitoring data block types (symbolic names - in the implementation, to each type corresponds an unsigned 16 bit integer value).

chunk type	description	blocks
JOB_STARTED	A chunk of this type is sent once at the very beginning of the JEM run.	JOB_START
JOB_MILESTONE	The job reached a milestone (e.g. next physics event in an ATHENA run).	*None*
JOB_FINISHED	The job finished execution.	*None*
MONITOR_STARTED	A new monitor process was launched (script monitor, system monitor).	SENDER_INTRO
MONITOR_EXITED	A monitor process exited.	*None*
COMMAND_LAUNCH	A command was launched by the user job.	TRACE_EVENT_BASE COMMAND_LAUNCH
COMMAND_EXIT	A command exited.	COMMAND_EXIT
FUNCTION_CALL	A function (method) was called.	TRACE_EVENT_BASE CALLEE_INFO
FUNCTION_RETURN	A function call returned.	TRACE_EVENT_BASE CALLEE_INFO
SCRIPT_LINE	An arbitrary script line was executed.	TRACE_EVENT_BASE EXPRESSION
SYSTEM_METRICS	System information was sampled.	SYSTEM_METRICS
OUTPUT_LINE	A line of output was written.	OUTPUT_LINE
EXCEPTION	An exception (run time error) occurred.	TRACE_EVENT_BASE EXCEPTION_INFO

Table 8: Monitoring data chunk types (symbolic names). The blocks specified in the last column are the *mandatory* blocks for these chunk types; there may be more, optional blocks contained (see tab. 7).

- To reduce the created data traffic, optimization techniques like the usage of identifier IDs described in the previous section should be applied.

These architecture requirements have been used to design the trigger system described in the following sections. Implementing it included major refactorings of JEMs inner workings and the design of an advanced producer/consumer data store. The deferred decision of processing an event or not was called the **tagging** of the event, and the list of not-yet-tagged events was named the monitoring event **backlog**.

The IPC mechanism used in the implementation is **shared memory**, a piece of random access memory (RAM) accessible by all of JEMs processes that contains the mentioned data store. The shared memory together with the data store implementation detailed in section 12.2 form a system fulfilling all requirements declared above, and will be described in-depth after being compared directly with named pipe IPC in the next section.

11.3 Inter-process communication in JEM revised

Before the architecture of the trigger system - its data store and the event processing flow - is described in detail, the differences between the formerly used named pipe IPC and the shared memory IPC are summarized to reason the choice of technology in the refactored system.

Named pipe IPC
Being a FIFO data structure, the named pipe has the following limitations, rendering it inappropriate for the trigger system:

- Only one producer can reasonably write to the pipe at the same time. All others have to wait for this one to complete before being able to write into the pipe. If not adhering to this principle, data corruption may occur. There is the possibility to open the pipe in a line-buffered mode, preventing data corruption; this, however, is equivalent to blocking producers until the previous one finished its write operation.

- The consumer can only read the entries in the pipe sequentially, and can always remove only the oldest entry from the data structure.

- A strict order of access to the pipe is mandatory. Producers and consumers can not disconnect from / connect to the pipe in arbitrary order. If all writing processes disconnect from the pipe, or the consumer connects before the first producer, a run time error occurs ("broken pipe"). This also means the pipe does not survive if all processes disconnect from it.

- The pipe uses an internal data buffer that resides in kernel space, and is not resizeable by the user.

- Access to the pipe is limited to POSIX file operations, introducing a further overhead.

On top of that, the way the pipe is used by the former version of JEM included a data mapping on writing and reading, and a human-readable, ASCII-based data format was used (having a human-readable format is advantageous for the prototyping phase of software development, but naturally performs worse than a well-designed binary format).

Shared memory IPC
In contrast to these limitations, a shared memory has the following advantages:

- Access to the data means direct, rapid and random memory access without mapping, as soon as a producer or consumer connected to the shared memory.

- Consumers and producers can connect and disconnect to/from the data structure in arbitrary order, and the life time of the data structure is controllable by the system. This includes keeping the data structure alive even if all processes intermediately disconnect from it.
- Random access is possible, and every present monitoring event can be removed from the data structure, if an appropriate data format is used.
- The data structure lies in main memory, and the size can be arbitrarily chosen (within platform specific limitations). To each connected process, the usage resembles private memory access.
- The data has to be brought into the dedicated chunk format (see sec. 11.1) once, and can then be written into and read from the shared memory - and further processed - without mapping.

```
        0  1  2  3  4  5  6  7   8  9  a  b  c  d  e  f
0x0000  AA AA 00 10 00 06 78 00  6D AC 28 63 FB AC BE 4C  ......x. m.(c...L
0x0010  4C 94 08 00 00 00 00 00  04 00 1B 00 6D AC 28 63  L....... ....m.(c
0x0020  53 79 73 74 65 6D 4D 6F  6E 69 74 6F 72 2E 33 31  SystemMo nitor.31
0x0030  30 34 39 00 04 00 25 00  F4 E7 EF 5B 63 68 61 72  049...%. ...[char
0x0040  6D 2E 70 68 79 73 69 6B  2E 75 6E 69 2D 77 75 70  m.physik .uni-wup
0x0050  70 65 72 74 61 6C 2E 64  65 00 00 00 10 00 0C 00  pertal.d e.......
0x0060  49 79 00 00 6D AC 28 63  00 02 0F 00 08 00 03 1C  Iy..m.(c ........
0x0070  A6 78 01 F4 E7 EF 5B 00                           .x....[.

<monitor-start timestamp='10:48:59.562252' creator=0x6328ac6d length=0x78
               prev-length=0x0 flags=0x600 ackBits=0x00>
   <identifier id=0x6328ac6d identifier='SystemMonitor.31049' />
   <identifier id=0x5befe7f4 identifier='charm.physik.uni-wuppertal.de' />
   <sender-intro pid=31049 name=0x6328ac6d len=0xc />
   <system-info cpus=1 cores=8 networkdevices=3 physicalmemory=24684060
                hostname=0x5befe7f4 />
</monitor-start>

        0  1  2  3  4  5  6  7   8  9  a  b  c  d  e  f
0x0000  AA AA 02 10 00 06 34 00  5B 9B 30 5B FB AC BE 4C  ......4. [.0[...L
0x0010  38 2D 0B 00 00 00 00 00  00 03 1C 00 00 72 65 73  8-...... .....res
0x0020  75 6C 74 3A 20 2F 74 6D  70 2F 74 6D 70 30 59 31  ult: /tm p/tmp0Y1
0x0030  53 5A 66 0A                                       SZf.

<output-line timestamp='10:48:59.732472' creator=0x5b309b5b length=0x34
             prev-length=0x0 flags=0x600 ackBits=0x00>
   <stdout out='result: /tmp/tmp0Y1SZf' />
</output-line>
```

Figure 30: Example monitoring data chunks and their XML representation.

While taking advantage of these benefits of a shared memory block as means of IPC, one has to be aware that suitable measures have to be taken to **synchronize** access to the memory. Synchronizing access means prevention of data corruption through parallel access to the data. This becomes necessary as several processes access the same block of memory at the same time; the POSIX file system access to the named pipe in the old solution was serialized, assuring synchronization automatically (at the cost of slower access).

By refactoring first the WN components of JEM to use the shared memory IPC, followed by the UI part and the Ganga-integration, it was possible to gain all the benefits of this technology and to implement the sophisticated trigger system, while keeping the time needed to reasonable order.

12 Architecture and implementation

In this section, the implemented trigger system will be detailed and its usage for JEM explained. Necessary changes to the project's component structure and base services are described first, followed by the description of the data store and the event processing.

12.1 General JEM architecture changes

Because the exchange of the old named pipe IPC, ASCII data format and relational table mapping with the new system already meant a rewrite of large parts of JEMs worker node modules, it was decided to use the opportunity to refactor the project as a whole.

The goals of this refactoring, aside the preparation of the trigger system, were (see section 5.6 for a description of the motivation behind those goals):

- Creation of a consistent, non-misleading and non-redundant directory (module) structure and naming scheme for modules. Refer to appendix A for an overview of the modules.
- Elimination of home-grown solutions for problems already solved by commonly available libraries, like logging, command line parsing, etc.
- Consolidation of the project's source code (being worked on by several developers, the code style was not homogeneous at all).
- Better code reuse by creating a truly modular architecture featuring a single invocation point for all purposes of JEM and the possibility of providing (subsets of) JEMs services as a reusable library (see section 13.2).
- Introduction of a user friendly, yet powerful configuration scheme with customizable configuration files, config override via command line options and config export/import to/from Ganga.
- Removal of the necessity to create temporary directories and files at run time. These were needed before to store session data and for the POSIX access to the named pipes.

This refactoring work resulted in several general-purpose, reusable infrastructure modules not specific to JEM. These modules then were used throughout the project in the WN-component, the UI-component and the Ganga-integration. Examples for those general-purpose modules are:

The `ConfigManager`

Application configuration is an inherent complex issue, although it doesn't seem so at the first look. Sophisticated configuration management solutions, like the one implemented for JEM, allow a module-specific configuration.

This is convenient for module authors and allows for a hierarchical config file structure. The settings can be loaded from such a file, and a default file can be automatically generated. Each module-specific configuration option can be given a description and default value for this purpose. The `ConfigManager` allows for the export and import of the settings to/from the environment, the definition of command line arguments corresponding to the settings and external configuration when JEM is used as a library (e.g. by the Ganga-integration), as well. Instead of having to maintain each of these aspects separately (defining command line arguments, parsing a config file, reading the environment, etc.) - an error-prone and redundant approach - everything is centralized to the definition of configuration options in each module. The `ConfigManager` then automatically handles all described aspects.

The `PluginLoader`

Modules whose instantiation and execution are optional and configurable - for example, the event triggers described in this part of the work, the valves (data transfer modules) or the data publishers at the UI - are handled by the `PluginLoader`. This module handles the discovery, loading, instantiating and initialization of those modules. Optional features can so be handled in a uniform way. The `PluginLoader` cooperates with the `ConfigManager`, allowing the specification of modules to be loaded in the configuration, and the setup of those modules as described above.

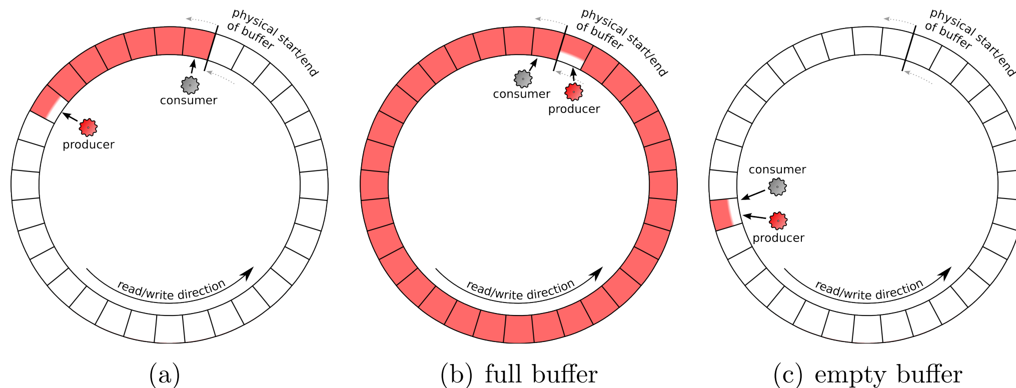
Logging

Although an established third-party facility is used for logging purposes (as as much functionality as possible is to be covered by existing libraries to prevent "reinventing the wheel"), a wrapper around this logging facility was created that integrates the Logger with the `ConfigManager` and allows one to replace the logging library with alternatives like plain console output or externally provided logging services (like Gangas own Logger) at run time.

12.2 High-throughput shared ring buffer

On top of the refactoring of JEMs IPC-mechanism from the named pipe to a block of shared memory (that all processes can access in parallel), the way of access to this buffer was designed to allow the most efficient usage. The access model chosen by the author is a **ring buffer**, allowing for the creation of a backlog system as described in section 11.2.

A ring buffer is, as opposed to a plain FIFO channel, of virtually unlimited size, as the end of the buffer is linked transparently to its beginning. This means a producer can continue to write into the buffer, even if its end is reached, and the data storage location is wrapped to the start of the buffer automatically, without the producer noticing.



(a) (b) full buffer (c) empty buffer

Figure 31: General principle of a ring buffer. The producer writes data while the consumer takes it out of the buffer (a). If the consumer doesn't keep up, the producer has to wait (b) and vice versa (c).

As long as there is space left - meaning as long the producer does not "overtake" the consumer - data can be written. Similarly, as long as there is data left - meaning the consumer does not overtake the producer - data can be read. This general principle of a ring buffer is shown in fig. 31. By adding control structures as described in the next section, access for multiple producers and/or consumers could be enabled.

12.2.1 Working principle

The ring buffer for JEMs IPC was implemented by placing a administrative header structure at the begin of the shared memory. Every process attaching to the shared memory block can read this header and use the contained data pointers for reading from - or writing to - the ringbuffer. The header is shown schematically in figure 32.

The major elements of the header are two relative pointers[23]: the next write-position (the next free position where a new data entry may be written into the ring) and the next read-position (the first readable data entry in the ring). Further, the overall size of the shared memory block must be stored in the header structure so that producers and consumers can calculate when to wrap around to the start of the ring buffer.

A producer wanting to add a new entry to the ring buffer would look up the next write-position, increment it by the size of the entry that is to be written, and copy the entry to the former write-position. If a consumer wants to read the first readable entry (if not accessing entries randomly, this is the first-written one), it looks up the next read-position, reads the entry and increments the read-position by the entry's size.

[23]the pointers must be relative offsets to the start address of the shared memory block, instead of being absolute addresses, because the shared memory block has a different base address for each attached process.

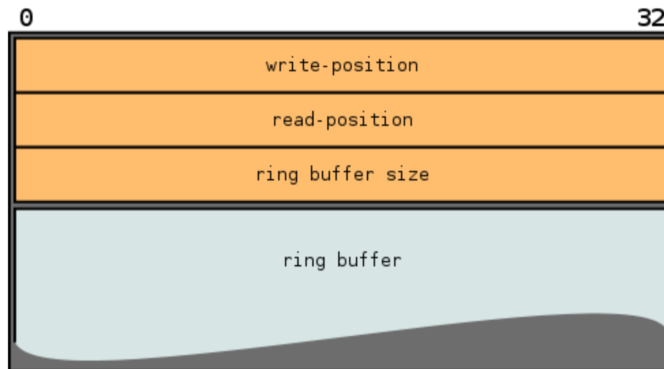


Figure 32:
Ring buffer header with pointers to the next write-position, first unread chunk and overall size of the buffer.

To allow multiple producers and/or consumers, the read- and write access to the ring buffer - and thus, to the header structure - has to be synchronized. A naive, first solution to this synchronization requirement of the ring buffer would be to synchronize the whole access to the header structure and the following data movement. So, each producer wanting to write an entry, and each consumer wanting to take one entry out of the buffer would need to acquire a global synchronization token protecting the header structure before allowed to alter the start- or end-pointer. This, however, would restrict the ring buffer to strict single-process usage, preventing any concurrent usage of the ring buffer and destroying all its benefits (see fig. 33).

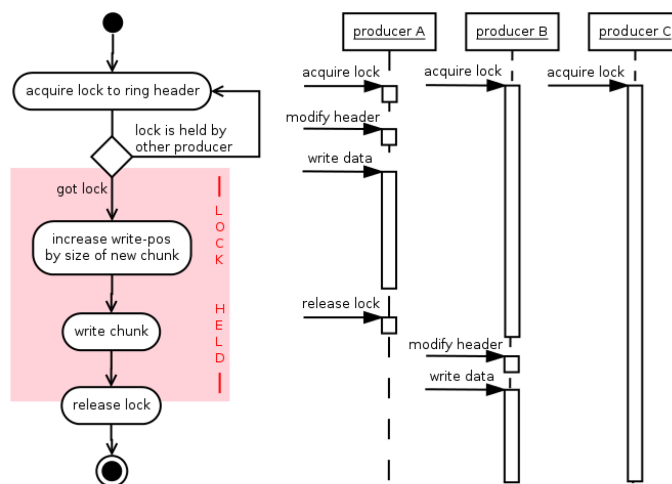


Figure 33:
Sequential writes into a ring buffer. As each producer needs to wait for the complete previous write process to complete, unnecessary delays are induced.

When analysing the access patterns of such a producer-consumer system, one comes to the conclusion that the only task needing synchronization is the alteration of the pointers in the header structure; the read- or write-process itself needs not to be synchronized. Even multiple producer processes could write to their respectively reserved areas of the ring buffer at the same time, if the *reservation* of the said buffer areas is properly synchronized (see fig. 34). Each producer would, in this model, acquire the synchronization token, increment

the write-position pointer, release the token, and then start to write the data to the piece of memory pointed-at by the former write-position value.

Furthermore, if all data written into the ring buffer has a common format featuring a chunk header like the one designed for JEM (see sec. 11.1), part of the synchronization responsibility can be transferred to each chunk. A "**magic marker**" field is added to the chunk header that is used to mark a chunk as "completely written" by its producer after it finished the write process.

By storing this information in the chunk itself at a known position (in the case of JEM, it is written at the very beginning of each chunk's header), and by introducing the convention that each producer must not set the magic marker before all data of this chunk has been written, no synchronization at all is needed for the consumer process: It can read chunk by chunk starting at the read-position until it encounters a chunk not yet marked with the magic marker. As soon as the magic marker is set, the consumer can continue reading that chunk. This mechanism is shown in figure 34(c).

The magic marker field in the chunk header was used in the implementation of JEMs ring buffer for a second purpose: To indicate a **wraparound** - this means, to mark the spot near the buffer's physical end, where the buffer continues again at its physical start. Consumers can this way easily detect the wraparound spot and just continue reading at the ringbuffer's start.

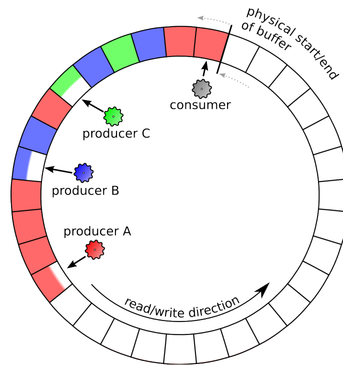
The last conceptual requirement for the ring buffer for JEM, as presented in section 11.2, is the **deferred decision** whether to keep or throw away each chunk. This means the consumer can not only delete the oldest existing data entries by incrementing the read-position in the ring header; instead, each chunk itself may be marked as "discarded" by setting an appropriate flag in its header. This has the additional benefit of being a very fast operation.

Consumers encountering a chunk marked as discarded just skip it, causing it to effectively be invisible. Chunks are deleted finally if discarded and at the start of the ringbuffer's occupied region, by incrementing the read-position in the ring header. Physically, the chunks are not deleted at all; they are just overwritten by a producer if the ring wrapped around. Not-yet-deleted chunks are protected from being overwritten, even if discarded, by the convention that the write-position never may overtake the read-position. The deferred deletion concept is shown in figure 35.

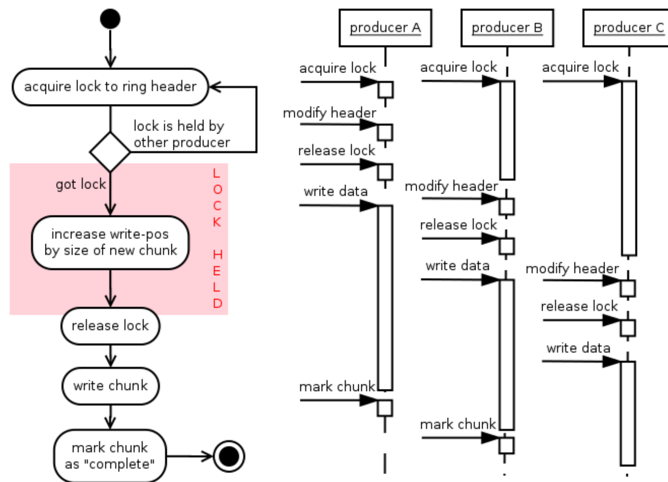
Because reading the chunks always starts at the read-position, chunks may be read multiple times[24] by each consumer. To prevent chunks from being read twice, each consumer may mark chunks as "known" by setting an acknowledge-flag in the chunk's header. Up to 16 independent so-called "ack-bits" are foreseen per chunk.

By reserving a certain amount of chunk header space for acknowledgement flags like this, multiple consumers can mark chunks as "seen" independently. This way, more than one consumer can use the ring buffer at the same time. Of

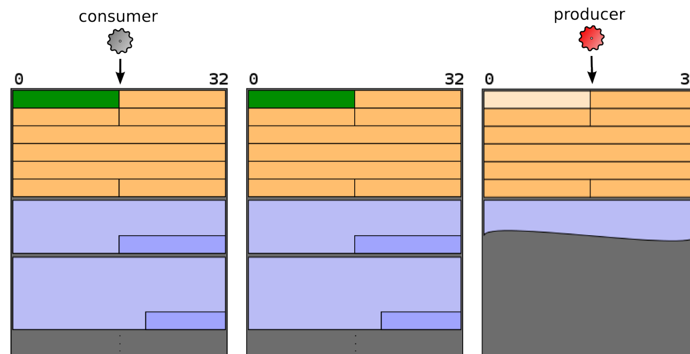
[24]if not discarded - those chunks are skipped as described before



(a) Each producer has its own reserved piece of memory to write to.

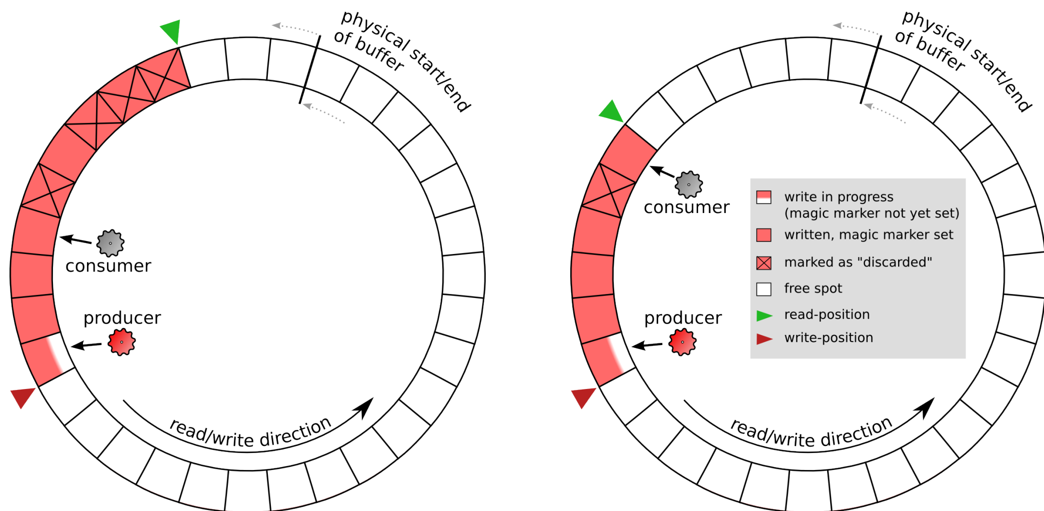


(b) The process of writing requires only a short-time lock when reserving a new piece of memory; the actual write processes are parallelized.



(c) A "magic marker" is added to the chunks and set to a predefined value after the producer finished writing the chunk. The consumer waits for the marker to be set before processing.

Figure 34: Concurrent writes into a ring buffer.



(a) Chunks are deleted by setting a chunk header flag to the value "discarded".

(b) Consumers skip discarded chunks, and update the read-position to the first un-discarded chunk.

Figure 35: Deferred deletion in the ring buffer.

course, as soon as chunks are invalidated (by being marked as "discarded" and then passed by the read position), no consumer is able to read them any more.

Another possibility is for consumers to build internal data structures, remembering the offset of specific non-discarded chunks, to directly (randomly) access them. They have to take care to check if the chunk is still valid, though, in case another consumer discarded the chunk in the meantime.

Aside from being able to mark chunks as discarded, chunks in JEM can also be marked as **approved** by consumers. This is used to select the chunks that are forwarded to the UI component over the Grid. This means, the decision to forward a chunk can be deferred, as well. There have been measures taken to prevent chunks from staying in non-discarded and non-approved status too long - otherwise, the ring buffer could fill up, leading to possible data loss.

12.2.2 Ring buffer operations

The following list of operations that can be performed on the ring buffer contained in shared memory have been defined and exposed as C API for usage by different JEM components - most importantly the data producers and the main consumer (see sec. 13). In addition to this C API, a Python API was created that resembles the former one and provides the same functionality to Python scripts, enriched with an object-oriented model of the monitoring data chunks for easy usage.

`int32 shmem_create(uint32 key, uint32 size, shmemHandle *handle);`
Creates a block of shared memory of `size` bytes and prepares the ring buffer structure in it. Each shared memory block is assigned a unique key that can be specified when creating; it's possible to have several distinct shared memory blocks on one machine this way. The shared memory **handle** stores internal bookkeeping data about the created shared memory and must be passed to all further API calls.

`int32 shmem_connect(uint32 key, uint32 size, shmemHandle *);`
Attaches to the shared memory block and prepares the synchronization facilities. The first client must in prior `create` the buffer; there is a convenience function, `shmem_create_or_connect`, that tries to connect and creates the block on connection failure.

`int32 shmem_disconnect(shmemHandle *);`
Disconnects from the shared memory pointed to by the handle and destroys that handle. This does not, however, destroy the shared memory block.

`int32 shmem_destroy(shmemHandle *, uint8 force);`
Destroys the shared memory block pointed to by the handle to free the resources. If further clients are connected to the shared memory, it is not destroyed, unless the `force` parameter evaluates to `true`.

`chunk *shmem_acquire_chunk(shmemHandle *, uint16 length);`
By calling this function, a new piece of `length` bytes (plus the chunk header size) of ring buffer space is reserved to the caller, increasing the write-position. Producers can use this to perform custom write operations on that piece of memory. After finishing writing the data, the magic marker at the beginning of that memory block must be set.

`uint32 shmem_write_raw(shmemHandle *, char *data, uint16 length);`
Writes data to the ring buffer pointed at by the handle in a synchronous (blocking) call. A piece of memory is reserved using `shmem_acquire_chunk` internally, the data copied, and the magic marker set.

`chunk *shmem_read(shmemHandle *);`
Returns the first available data chunk at the read-position.

`chunkIterator shmem_iterate(shmemHandle *, chunk *start);`
Returns a chunk-iterator pointing at the first available data chunk (if `start` is unspecified) or pointing to the given start-chunk. This iterator can be dereferenced to get a reference to the currently pointed-at chunk, or incremented to make it point at the next chunk. If no further chunk is available, the iterator points to a sentinel value so consumers can detect that.

`chunkIterator shmem_iterate_backwards(shmemHandle *);`
Returns a backwards-iterator pointing at the last available data chunk. To make backwards-iteration possible, the length of the previous chunk is recorded in each chunk's header.

`uint32 shmem_invalidate(shmemHandle *, uint32 count);`
This "deletes" the first `count` chunks in the ring buffer by increasing the read-position in the ring header. This is a low-level operation that does **not** take chunk status (whether it is is discarded) into account.

Further operations can be done by directly editing the chunk header of data chunks written into - or read from - the ring buffer. The marking of chunks as "discarded" and "approved" is one example; the iteration over the chunk's data blocks is another. When describing the trigger APIs in the next section, examples for this direct chunk header access will be shown.

12.3 Triggers and event handling

The presented ring buffer data store now could be used to implement a trigger system like described in section 11.2. At first, its general design will be detailed, and its APIs described. Then, example trigger scripts will be described.

12.3.1 Trigger architecture

JEMs WN component consists of multiple ring buffer producers (the script monitors and the system monitor) as well as one central consumer, the WN **core**. This core houses the main event loop of the WN component. After the job is started on the WN machine, the core is created by JEMs launcher script. It determines the script type of the user script (Bash or Python scripts are supported at the moment), spawns the system monitor and the root script monitor, and then starts the core event loop.

While the producer processes create monitoring data chunks and write them into the ring buffer as described in section 12.2, in each event loop iteration, the core performs the following steps:

- All consecutive chunks at the start of the ring buffer[25] that are marked as "discarded" are deleted by incrementing the read-position pointer.

- All new chunks - chunks that have been produced by the monitor processes since the last main event loop iteration - are fed to the triggers and marked as "seen by WN core".

[25]= at the read-position

- All chunks marked as "approved" are passed to the data publishing process(es) of the WN component (the valves) and then marked as "discarded" automatically. The valves publish the chunks through different channels - over the Grid to the UI, into log files, etc.

Each trigger defines a list of chunk types it is interested in (or "all chunks") as well as a priority between 1 and 100. With each chunk processed by the core event loop, all triggers registered for the chunk's type are called in the order of their priorities. Each trigger now has the possibility to inspect the new chunk and take one of the following actions:

- Mark the chunk as approved, causing it to be forwarded to the data publishing module(s) regardless of the other trigger's decisions
- Mark the chunk as discarded, causing it to be dropped if no other trigger approves it
- None of the above, but inserting (a reference to) the chunk into an own data structure of chunks to decide on later
- Deciding on one or more of the locally stored chunks (if any) based on the new chunk.
- Ignore this chunk completely.

The triggers are created as dynamically loadable modules, from which the user can choose the active ones separately for each job run (although it is intended to define a set of "fall back" default triggers, for example taking care that each chunk is eventually discarded to prevent the ring buffer from overflowing). They can be written either in C or Python, and can access the shared memory API described above as well as JEMs core services like logging and configuration management.

12.3.2 Trigger scripting APIs

New triggers can be created in C - by writing a C-file in a specific directory of the JEM source tree, including the trigger API's header - or in Python by sub-classing a JEM-provided Trigger base class. Implementing the trigger in Python is easier than writing a trigger in C, but introduces some disadvantages:

- The trigger performance will be worse; the trigger will need more time to perform the same steps compared to a similar trigger implemented in C.
- The to-be-inspected chunk must be copied at least once in memory for each trigger to analyse it (this, too, adds to the performance penalty).
- No low-level direct memory access to the chunk is possible.

```c
#include "Trigger.h"

typedef struct {
    Trigger base;
} DiscardAllTrigger;

static int init(DiscardAllTrigger *self, PyObject *args, PyObject *kwds) {
    BUILTIN_TRIGGER_SUPER_CTOR;
    BUILTIN_TRIGGER_ADD_CHUNK_TYPE(CHUNK_TYPE_ALL);
    return 0;
}

static void dealloc(DiscardAllTrigger *self) {
    BUILTIN_TRIGGER_SUPER_DTOR;
}

static void examine(Trigger *self, chunk *current) {
    current->hdr.flags |= FLAG_WN_CORE_DISCARDED_CHUNK;
}

static void finalize(Trigger *self) {}

BUILTIN_TRIGGER_DEFINITION(DiscardAllTrigger, "DiscardAllTrigger",
                           "Trigger just discarding every chunk.")
```

Listing 5: Minimal trigger implementation to show the C trigger API.

In contrast, the implementation in C allows direct memory access and is much faster. The build process for custom triggers is automated in most parts. By placing the C-file of the new trigger into JEMs source tree at a special place, and running a provided build script, a loadable library is created and placed into JEMs run time directory.

To show the trigger API, a minimal trigger implementation in C is given in listing 5. All trigger implementations must first include the JEM-provided trigger header file and declare a data structure for the new trigger. This data structure always at least contains a reference to the trigger base; that structure contains the trigger's priority and chunk type list. Own internal data can just be added to the trigger struct (line 5).

The trigger's functionality is implemented in four functions. Construction and destruction of the trigger object is done in `init` and `dealloc`; since the steps one has to perform to create and destroy the trigger object is always similar, convenience-macros are provided to the programmer. Line 9 shows how to define the chunk types the trigger should react on.

`examine` is the main function called for every to-be-inspected chunk. It gets passed a direct memory reference to the chunk inside the ring buffer, as well as a reference to the trigger data structure (for the case the trigger needs to

```python
from Common.SharedMemory import shm
from Modes.WN.Trigger.Trigger import Trigger

chunktypes, blocktypes, chunk_names, block_names = shm.getChunksDict()

class DiscardAllTrigger(Trigger):
    "Trigger just discarding every chunk."
    def __init__(self):
        Trigger.__init__(self)
        self.chunk_types += [chunktypes["CHUNK_TYPE_ALL"]]

    def examine(self):
        self.discard()

    def finalize(self):
        pass
```

Listing 6: Equivalent trigger implementation in python.

access internal data)[26]. The pointer to the chunk in the ring buffer can be used to read the chunk's data, but also to modify it (since it operates on the original chunk, not on a copy). One has to take care, though, not to write outside of the chunk's bounds to not destroy the adjacent chunks' data.

A dedicated `finalize` function can be implemented with shut-down code; it gets called by the JEM framework just before it exits. It is separated from `dealloc`, and code that e.g. summarizes the gathered data should be put here, as JEM services (like the logging facility) are guaranteed to still be available at this point (in contrast to `dealloc`).

Finally, the trigger must register itself to JEM when it is loaded. A macro is available to include this registration code conveniently (line 23). This defines the trigger's name and a documentation string, and links the static functions defined in the C file to the base trigger code provided by the framework to build a loadable library file.

The corresponding Python-trigger is given in listing 6. The main methods that must be implemented are the same. To just approve or discard the inspected chunk, a call to a method of the base Trigger class is sufficient (line 13). To access the raw chunk, a call to a further method, the accessor `chunk()`, is necessary; this way a **copy** of the chunk is created. One has to be aware of the performance impact of this copying process when implementing triggers in python. This accessor usage is shown in the example trigger scripts in the next section.

[26]This trigger data pointer is of the generic trigger type, but can be casted to the custom trigger's type.

12.3.3 Example trigger scripts

To show what is possible with the presented trigger system, a number of more complex example triggers, which utilize the possibility to create internal data structures and deferring the decision on the backlog of monitoring chunks, are described here. Their source code can be found in appendix B (page 142).

A simple statistics trigger
The statistics trigger is responsible for the creation of statistics about the number and size of the monitoring events processed by JEM. It does so by registering for all chunk types and keeping the number of seen chunks and the sum of the size (in bytes) of the seen chunks as internal data. Once in a second, this data is divided by the exact time passed since the last sample, and logged. On shut-down, the total number of chunks and bytes transferred is logged.

Exception tracker
Exceptions happening in the user application are most interesting if they are not catched in user code, or catched too late. The exception tracker defers the decision whether to publish exception events to the point at which the exception is created and catched (if it is catched at all).

Exceptions raised in user code are always logged. If the exception is raised by system library code, but is catched inside user code within a certain limit of left scopes (configurable by the user), it is discarded. If it is not catched, or catched only after more left scopes than configured, it is logged. This ensures that "expected exceptions" are not logged.

User-defined progress tracker
In many applications, land-mark events could be defined, that imply a natural separation of distinct execution sections. On entering a new section, the monitoring events of the previous section that have not yet been decided on become uninteresting.

For example, in physics analysis code, each analysed event defines such a section: If the processing of one event finishes and the processing of the next event begins, a new section is entered, and all remaining monitoring events of the left section can be discarded. This is done by the progress tracker by looking for typical output strings in event data using regular expressions. If a match is found, all undecided-on events left in the ring are immediately discarded.

Event multiplexing on the UI
Because the UI component of JEM was designed similar to the WN, with one main consumer of ring buffer data, but potentially several publishing channels could be thought of (file based publishing, forwarding into Ganga, etc.) a way must have been found to spread the events to this collection of publishers.

This task is now fulfilled by a number of trigger scripts registering for all event types and feeding one publisher each. The publishers fed this way are (among others):

- a simple dumpfile-writer that creates a dump file similar to the one the file-writer valve writes on the WN,
- a live statistics displayer showing statistic values like number of received chunks per second, etc. to the user,
- the module providing the real time data to the Ganga-integration,
- a stdout-peeker that displays stdout-line-data in real time (similar to the "`tail -f`" shell command).

12.4 Memory management

With the ring buffer being a block of shared memory attached to each producer and consumer process, and having different virtual addresses in each process, the memory management becomes a challenging task. Also, the allocation and deallocation of memory blocks on the system level for each event should be avoided, as it would introduce a severe performance penalty. Thus, a custom memory management had to be created for the ring buffer.

12.4.1 Management of shared memory

As custom replacement for the standard library's memory allocation mechanism, a relatively-addressing allocation manager has been implemented for JEM. It uses a buffer of the shared memory of predefined size and adds the needed bookkeeping data to be able to hand out and free memory blocks into the buffer itself. Processes can, just by calling the usual `malloc`- and `free`-calls, get and release memory blocks in the shared memory without interfering with other processes attached to it.

For rapid look-up of indexed data, a red-black-tree was implemented; this tree, too, relies on relative addressing only and uses the custom memory manager to allocate tree nodes. This way, a tree created and filled by one process can be used to look up values by another process, accessing the tree nodes via different absolute virtual addresses.

12.4.2 Shared identifier cache

The most prominent application of the look-up tree in JEM is the **identifier cache**. Each identifier (function name, symbol name, path and file name, etc.) is added into a monitoring data chunk only exactly once during the run time of a whole, JEM-monitored application.

It gets assigned a unique, fixed-length ID value stored into the shared look-up tree and is referred-to by this ID in the remainder of the application

run. All consecutive events referring to this identifier only contain the ID instead of the whole string. By building a similar look-up tree on the client side and inserting the identifiers when first receiving them, the UI-component uses the same hashed IDs when displaying the monitoring data.

13 Application in JEM

After the new base of JEMs architecture had been implemented, the data and workflows of the WN- and UI-components were revised and implemented using the new backend system. As mentioned in section 12.1, at this opportunity, most of JEMs functionality was critically re-evaluated and, in parts, replaced by established third party libraries.

At first, the precise composition of producers and consumers using the ring buffer infrastructure and the data flow between them was designed for the WN part. Then, similarly, the UI part was refactored. Finally, the CTracer that has been described in-depth in part III of this work was adapted to the new infrastructure.

13.1 Changes in JEM execution

The access pattern foreseen in the IPC of JEMs WN-processes is multiple concurrent producers (all data creating systems like the script monitors and the system monitor) and exactly one consumer that takes data out of the ring buffer and processes it (forwards it to the UI component eventually, using one or more valve modules). This access structure and data flow was described in section 12.3.1.

The UI module, after being refactored, works in a similar fashion. Here, only one producer exists (the real time data receiving module the data is transferred to, over the Grid, by the valve modules), but potentially several (logical) consumers (which write the data to a file, visualize the data, present the data inside of Ganga, etc). A central UI core module has been implemented, similar to the WN core, housing the UI main event loop and acting as the consumer process reading the data from the ring.

The data is processed by this UI core module much like it is on the WN: After being written into a ring buffer instance by the producer (data receiving module), the core module reads each chunk and passes it to a configurable, prioritized list of **data processors** similar to the triggers on the WN.

It is these processors that do the real work on the UI; there's a log file processor writing the data into a file on disk, a `GangaProcessor` passing the data to Ganga, a statistics processor recording statistical information about the monitoring data, etc[27].

Both modules - WN and UI - use the same collection of infrastructure modules that provide services like (remote-) logging, configuration management, command line option parsing, signal handling etc. The invocation of the modules was consolidated, as well, resulting in exactly one invocation point for all use cases of JEM, the main **launcher module**.

[27]By letting them attach to an already existing ring buffer and registering only consumers, completely distinct applications that use the monitoring data could also be implemented.

The different use cases, now called **running modes**, are set up and run by the launcher module, which provides general launch configuration options like daemonizing the execution, attaching a debugger to itself, logging to a remote logging server or to stdout instead of into a log file, printing usage instructions to the console, etc. The launcher module also can be used as library by arbitrary Python programs, providing all of JEMs functionality to them.

Further running modes that have been implemented provide services like packing and deployment, data visualization and analysis, and the creation of JEM usage statistics.

13.2 Refactored Ganga-JEM integration

The new Ganga-JEM integration now is based on the ring buffer IPC instead of on temporary XML files. This allows faster data access (forward- and backward iteration through the monitoring events) and is more robust in general. It is implemented as further JEM running mode and imports JEM as a library inside Ganga. That way, it uses the same code base as the rest of the project and benefits from maintenance updates.

Since JEM is used as library, no configuration by command line parameters is possible. Instead, a corresponding section for the Ganga user preferences file has been defined, whose settings are forwarded to JEM when the library is loaded. Furthermore, as JEM is applied to a job in Ganga (by instantiating the `JobExecutionMonitor`-object, see sec. 5.4.3) the current default per-job-settings of JEM are converted into similarly named Ganga schema attributes (see table 9 on page 121 for some examples). These settings are exported to the job's environment on the WN, and thus are available to JEMs components there.

The interface to the user inside of Gangas interactive console has stayed the same for the largest part. One example of an enhancement of the interface is the addition of an event statistic for any job, providing the information of how many events of any event type occurred so far.

Another major difference to the old implementation is that all of the monitoring data is available in real time and in retrospect (by loading it from a file) to the user. The former implementation only allowed access to the last few events that happened. Finally, a new method in the Ganga-integration allows the convenient launch of a monitoring data visualization tool (see figure 36).

13.3 Refactored CTracer

As hinted at at the end of the previous part, the CTracer library was originally written against the former, "v2"-JEM API. Thus, it had to be refactored as well to benefit from the new ring buffer backend.

Ganga schema attribute	description
`moa.global`	JEMs `[Global]` config section.
`moa.global.mode`	The working mode (WN, UI, ...).
`moa.global.jobid`	The job id of the Grid job.
`moa.logging`	The config section `[Logging]`.
`moa.logging.logfile`	Filename of the log file.
`moa.logging.defaultLevel`	The log level (`INFO`, `WARNING`, ...).
`moa.wn`	The `[WN]` configuration.
`moa.wn.script`	The user script spawned by JEM.
`moa.wn.trigger`	List of triggers to instantiate.
`moa.bashmonitor`	The bash monitor's config section.
`moa.bashmonitor.disable`	Whether to monitor bash scripts at all.
`moa.pythonmonitor`	The python monitor's config section.
`moa.pythonmonitor.disable`	Whether to monitor python scripts at all.
`moa.ctracer`	The `[CTracer]` config section.
`moa.ctracer.disable`	Whether to monitor binaries at all.
`moa.ctracer.traceModules`	List of binaries to trace.

Table 9: Configuration options of JEM in Ganga (excerpt). The Ganga object instance `j.info.monitor` contains the attribute `advanced`, exposing JEMs config. This attribute has been abbreviated to `moa`.

At this opportunity, the CTracer library was narrowed down to the main functionality (following call- and return-events, interpreting the related debug information from the object files, inspecting user memory safely) while relying on established 3rd-party-libraries for the extraction of debug information. The size of the CTracer has decreased considerably while the performance improved.

The CTracer now inserts the inspected events into the ring buffer as just a further ring buffer producer, and does not need to compose ASCII-based text representations of the monitoring data. It is configured by the new configuration management and logs internal progress and status to the logging service.

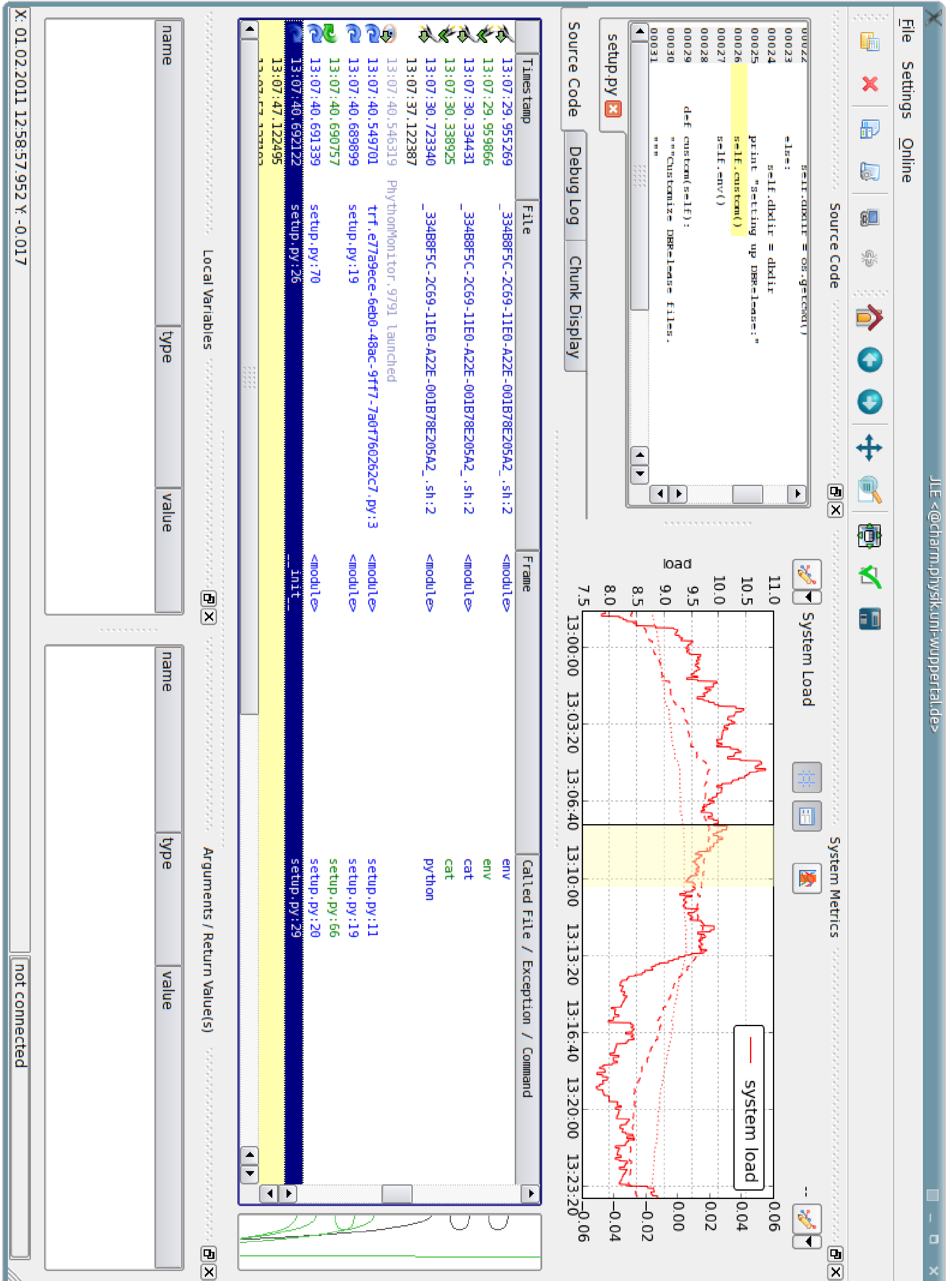


Figure 36: Monitoring data visualization tool - The **JLE** (JEM Log Explorer), showing colour-coded monitoring events, system metric plots and detailed selected event data.

14 Testing

Because the complete backend of JEMs WN and UI-component was exchanged with a newly developed solution, and this way the whole system relies on the reliability of this solution, it has to be tested thoroughly. To address this in a reproducible way, and to use this tests also to create statistics like the maximum throughput using the ring buffer implementation, the tests were split into a set of functional tests and performance measurements.

14.1 Functional tests

The functional tests are a collection of hierarchical organized **unit tests** as well as a number of test applications that can be used to build different test scenarios. The test applications implement producers and consumers for the ring buffer and can be configured in terms of creation/consume speed, average chunk size, discard frequency etc.

Unit tests
This test suite consists of two classes that are part of JEMs unit testing framework, implementing the TestCase interface. If executed, it performs a number of single tests, whose results are displayed or just ignored if no error occurs. If an error is discovered, the execution of this test case is aborted and a summary displayed. If the execution was part of a superior test suite run, the rest of the test case is skipped.

The test case is separated into two distinct parts. The first part tests the low level ring buffer implementation, focussing on potentially problematic details like the wrap-around calculation at the end of the physical buffer and the correct locking behaviour (for example if either the producer or the consumer works much faster than the other). The other part is a test of the ring buffer API, testing the object oriented python access to the ring, the trigger API, and the identifier cache functionality.

More details on the unit test suite can be found in section 16.1.1 on page 130.

Test applications
To stress-test the ring buffer implementation, to test it for robustness and correctness and to analyse its behaviour for different combinations of producers, consumers, producer speeds and data chunk sizes, a set of test applications were implemented. Both C- and python-implementations were created, to compare the performance of the two APIs. The applications implemented are:

- A **producer** (writer) capable of writing data chunks of specified size (or size range; the size of each chunk is chosen randomly within this range then) with a specified delay between two writes. The producer can display statistical information (chunks per second, bytes per second,

failed memory acquirement attempts, etc) or visualize those statistics graphically.

- A **consumer** (reader) capable of reading data chunks with a specified delay. It also can be configured how often the reader discards chunks from the ring buffer. The reader is able to record statistics similarly to the producer. This was used to create the performance test graphs in the next section. Furthermore, the reader can forward all data read from the ring buffer to a network socket to simulate JEMs main mode of operation.
- A memory **dumper** displaying a piece of the ring buffer on the console. This is used for debugging purposes.
- A **test scheduler** application that takes a test case script specifying when to spawn a producer or consumer, and when to quit it again. This is used to automatically execute multi-step test cases like running one consumer and from one to ten producers, adding one producer every few seconds, or running one consumer and one producer while increasing the producer's write rate continuously. An example of such a test case script is given in listing 7.

```
1  # payload size sweep test case
2  #
3  # shmem size         : 16MB
4  # reader             : C              writer             : C
5  # writer delay       : 0ms            reader delay       : 0ms
6  # payload size       : sweeps over 64b, 128b, 256b, 512b, 1k, 2k, 4k, 8k
7  #
8  [ 0,  SPAWN, 0, "./reader -q -p -s 0x1000000 -d testcase001", False]
9  [ 1,  SPAWN, 0, "./writer -q -s 0x1000000 -p 64:64", False]
10 [19, TERM, 1],
11 [21,  SPAWN, 0, "./writer -q -s 0x1000000 -p 128:128", False]
12 [39, TERM, 21],
13 [41,  SPAWN, 0, "./writer -q -s 0x1000000 -p 256:256", False]
14 [59, TERM, 41],
15 [61,  SPAWN, 0, "./writer -q -s 0x1000000 -p 512:512", False]
16 [79, TERM, 61],
17 [81,  SPAWN, 0, "./writer -q -s 0x1000000 -p 1024:1024", False]
18 [99, TERM, 81],
19 [101,  SPAWN, 0, "./writer -q -s 0x1000000 -p 2048:2048", False]
20 [119, TERM, 101],
21 [121,  SPAWN, 0, "./writer -q -s 0x1000000 -p 4096:4096", False]
22 [139, TERM, 121],
23 [141,  SPAWN, 0, "./writer -q -s 0x1000000 -p 8192:8192", False]
24 [159, TERM, 141],
25 [162, TERM, 0],
```

Listing 7: Schedule script for automated ring buffer stress tests (example)

14.2 Performance tests

Using the described stress test applications, a number of performance tests were executed. The questions to answer were "how much data can at maximum be transferred through the ring buffer?", "how does the transfer rates behave if more writers are added?" and "what effect have different delays between writing each piece of data?". All tests were run on a test machine resembling typical Grid worker nodes: A dual Quad-Core Intel Xeon with 2.5 Ghz, 24GB RAM (2GB per core), running CentOS 5.

Maximum throughput for different ring buffer sizes

To determine the maximum data throughput of the ring buffer, one consumer was run taking chunks out of the buffer without delay and immediately discarding them, and one producer writing data of increasing chunk sizes.

In the plots, the number of processed chunks per second and the "blocks", the reader wait times caused by an empty ring buffer, are shown. Furthermore, the transferred bandwidth is shown, and the calculated size of each chunk.

In figure 37 and 38 one can see that a maximum throughput of around 3GB/s can be reached, when appropiately sized chunks are used (the theoretical maximum chunk size in the ring buffer is close to 64kB, as 16 bits are reserved to store the chunk size in the chunk header; this design decision follows the analysis of typical monitoring event sizes not exceeding 16kB).

The number of chunks transferred per second stays constant around 100.000 chunks/s, irrespective of the chunk sizes. This leads to the conclusion that the throughput using the ring buffer in this test scenario is CPU bound. At the larger chunk sizes (32kb and more) and smaller ring buffer size (64kb), the ring buffer quickly gets saturated - as to be expected when the data chunk sizes reach the whole buffer size. Using a larger buffer solves this problem (fig. 38).

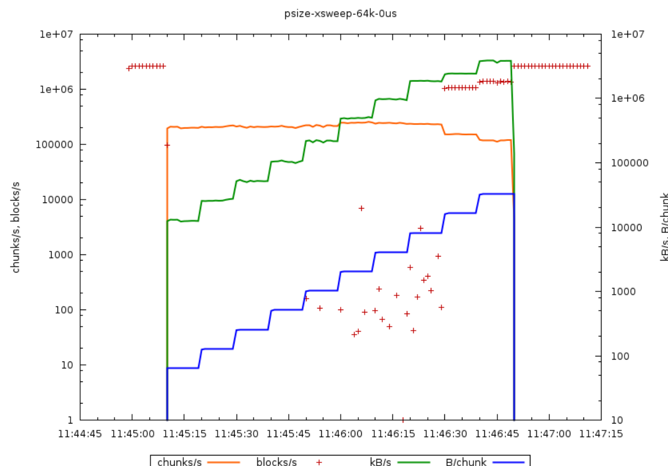


Figure 37: Maximum throughput by chunk size in a 64kB ring buffer. Every ten seconds, the chunk size was doubled, starting at 64 bytes per chunk. One can see the saturation of the ring buffer for chunk size above 16kB; also the block rate rises accordingly. Aside that, the data and chunk rates scale as expected.

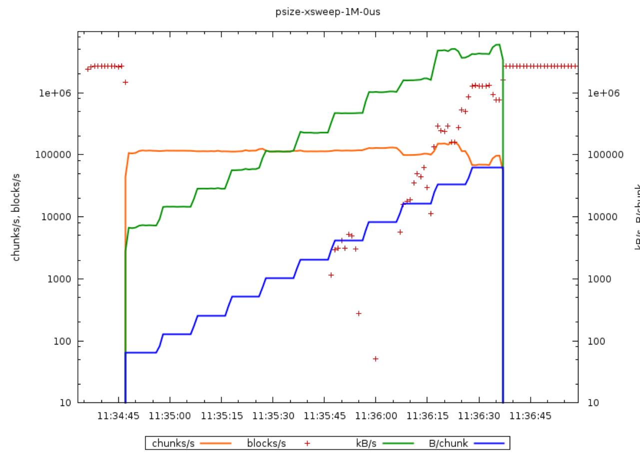


Figure 38:
The same chunk sizes, this time in a one MB ring buffer. The same scaling behaviour can be seen, without saturation.

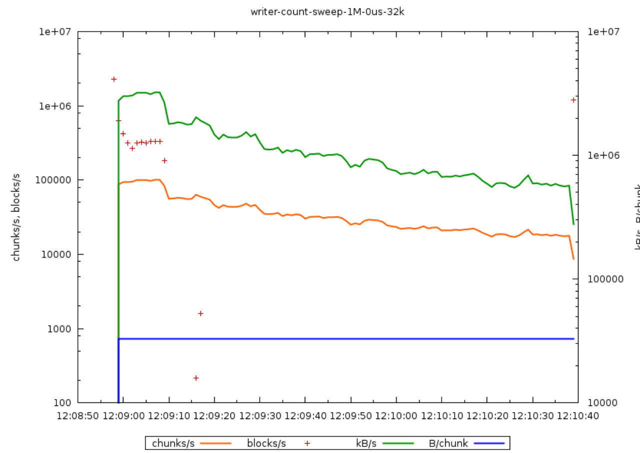


Figure 39:
Increasing number of producers in a 1MB ring, writing 32kB-chunks without delay each. The overall chunk count per second decreases, caused by locking effects.

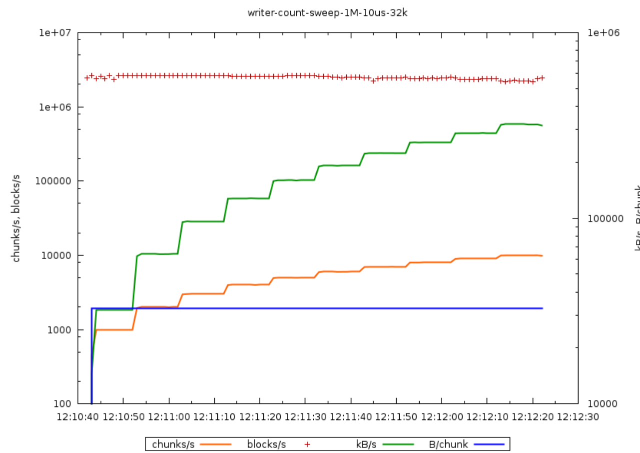


Figure 40:
Increasing number of producers in a 1MB ring, writing 32kB-chunks with 10μs delay each. No locking effects occur, and the chunk- and data rates scale with the number of producers.

Comparison of n:1 producer-consumer systems
In this test, the data chunk size and the ring buffer size were kept constant, while gradually increasing the number of producers (one to ten producers, one additional producer each ten seconds). A maximum number of ten producers was used; this was considered the maximum realistic producer count for JEMs purposes.

In figure 39, significant locking penalties can be seen (indicated by decreasing chunk- and data-rate), as every producer tries to acquire and write chunks as fast as possible (no delay), and the write process itself is a simple memory-copy operation, taking very little time. As write delay is introduced - with 10 μs, this delay is smaller than the expected typical delay in real-world producers, which is caused by the time needed to record the monitoring events themselves - no locking effects occur. Of course, due to the delay, the overall throughput is lower than in the tests before - but as this is a scenario resembling reality (no real-world producer could write as fast as the no-delay-writer in the previous test), assuring that no locking effects occur has highest priority.

15 Conclusion

Implementation of the shared ring buffer infrastructure has proven to be a success already, even without fully exploiting the possibilities introduced by trigger scripting. The more optimal IPC alone, together with the more lightweight data format, decreases the impact of scalability problems in JEM.

First experiments with trigger scripts have been performed as described, and measurements of ring buffer performance taken. By also applying the ring buffer system to JEMs UI- and Ganga-components, further efforts have been taken to understand the system's behaviour and to optimize its implementation.

Especially the CTracer system described in part III of this work benefits largely from the higher possible throughput, the higher robustness and the easier usage of IPC by means of the ring buffer, neglecting the need to perform data mapping and POSIX access to publish data.

Part V
Summary

The described additions to the Job Execution Monitor - the CTracer library allowing to follow a Grid job's execution in compiled user binaries, and the shared ring buffer infrastructure enabling high data throughput and sophisticated data selection techniques - have been implemented, deployed and begun to be tested. The tests performed are described in this final part of this work, as well as use cases of the presented technology.

Adding the CTracer library to JEM has removed the last blind spot from user job monitoring. The whole job execution can now be supervised and problems early detected. The possibility of inspecting the user application's memory during execution - the remote debugging feature of the CTracer - allows to further investigate reoccurring problems down to the data level.

With the ring buffer IPC mechanism, the data rates in JEM can be handled. This is a basic prerequisite to make JEM usable and applicable for production use. The increased data rates by enabling the CTracer, too, are handled by the ring buffer IPC quite well. Basic triggers have been implemented preventing the overflow of the ring buffer. On the UI side, the same technology is used to enable multiple views on the same recorded monitoring data, including a graphical visualization for the user and a seamless Ganga-integration.

In sum, this turns JEM into a mature and stable Grid job monitoring system applicable to the HEP communities requirements. Typical use cases will be described in the next section, and an overview over the way JEM is tested given.

There is, however, still room for improvement and optimization of the system. Also, the novel trigger system, albeit being ready now, has not yet been fully exploited by the development of more-than-trivial trigger scripts. These aspects are discussed in section 17.

16 Use cases and testing

Testing is a quintessential part of the software development cycle. It encompasses the developer tests performed during implementation (the stress test applications described in the previous section being an example), user tests validating the fulfillment of user expectations, functional tests assuring correctness of the implemented algorithms and components and regression tests preventing the recurrence of previously solved problems and programming errors.

In this section, first the available test infrastructure in JEM is described briefly. In the following, some typical use cases of a Grid job monitoring software like JEM in the context of HEP Grid computing are shown.

16.1 Testing framework

Besides the stress tests described in section 14.2, unit tests have been implemented that test JEMs core modules for the fulfillment of algorithmic expectations.

16.1.1 Unit tests

For each module package of JEM, there exists an accompanying test package containing module unit tests. Each unit test module either is a **composite test** running other test modules one after the other, or is a **test case** validating one specific JEM module. The test package structure, thus, resembles the structure of JEMs library packages.

By organizing the tests hierarchically like this, the user has the possibility to test a specific module, a whole part of JEM (for example, the whole worker node component) or the complete software suite at once.

Tests consist of one or several test methods, each testing one aspect of the target module, and resulting in test approvement - usually causing no output at all - or test failure. In case of failures, helpful error output and stack traces are created to aid in finding and fixing the problem. An example of unit test output is given in figure 41.

```
muenchen@endor:~/JEM/test$ python WN/SystemMonitor/TestSystemMonitor.py
...F
======================================================================
FAIL: test10Sample (__main__.TestSystemMonitor)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "WN/SystemMonitor/TestSystemMonitor.py", line 127, in test10Sample
    (chunk_names[chunk.chunktype], chunk_names[ctype]))
Error: Chunk type mismatch! Is: processes-info, expected: monitor-exit

----------------------------------------------------------------------
Ran 4 tests in 1.452s

FAILED (failures=1)

muenchen@endor:~/JEM/test$ python WN/SystemMonitor/TestSystemMonitor.py
....
----------------------------------------------------------------------
Ran 4 tests in 1.487s

OK
```

Figure 41: Example unit test output hinting at a simple implementation problem, and the output of the same test run after the problem was fixed.

The unit tests are designed in a way that they can execute without human intervention. In larger project scenarios with multiple developers, this often is automated (nightly builds with automatically running unit tests) and the

results stored, or automated mail notifications sent. This is called **continuous integration** by software engineers. JEM, as the presented project, however is too small to benefit from such an automated build system.

The unit tests themselves, though, are very helpful in finding implementation errors and, most importantly, to detect regressions caused by later modifications (failures of completed modules not modified that are caused by changes in other modules) that otherwise can be hard to notice.

Finally, by adding unit tests whenever a programming error is fixed, that verifies that the bug fix really solves the problem, one can assure that no regression occurs (in other words, that the found and fixed bug, some time later, does not become a problem again).

16.1.2 User tests

Aside the automated testing by Unit tests, User tests are an important asset in determining the correctness and applicability of any software system from the user's point of view. For JEM, automated gathering of usage statistical data was implemented, and a group of early adopting users applied the job monitoring to their analysis Grid jobs to gain experience with the software.

The statistical data can be used to gain an overview of the current number of JEM users, and the typical job run times and exit codes occurring when JEM was active. For each job, the job-start- and job-finish-events are forwarded to the service that creates the usage statistics. If desired, all information can be anonymized per user. Additionally, data helping in classifying possible JEM problems are transferred, as well. All gathered data is presented on an auto-generated webpage, as displayed in figure 42.

The concept of the user tests was mainly to determine whether the application of JEM to a typical HEP Grid job worsens the user experience in terms of job stability (are there situations where JEM itself crashes or alters the job's semantics?) and job performance (does JEM slow the job down in an unacceptable magnitude?). The results of the internal user tests were satisfying, and JEM deployed to the WLCG.

Large-scale user tests with group-external users are still to be executed, as the deployment of the system just has been performed. This is part of a follow-up project, performed by the author of this work, that aims to integrate JEM also into the pilot-based Grid job brokerage system PanDA (see section 2.4.1).

16.2 Use cases

A number of real-world problems already have been solved with the help of JEM. In this section, two of those use cases are briefly described. They are examples for the possible applications of a monitoring software in a world-wide distributed computing environment, from a user's point of view as well as from a Grid site admin's perspective. Being specific examples, their context has to

Job Execution Monitor

JEM live usage statistics — service uptime: 4 days, 7:55:48

Last jobs

started	finished	job	VO	CE	user	result
24.01.2011 13:32:59.287	24.01.2011 13:32:59.730	BB7773E2-2623-11E0-88AF-002264046A8C	undefined	grid-ce5.physik.uni-wuppertal.de:2119/jobmanager-lcgpbs-dg_long	/C=DE/O=GermanGrid/OU=LMU/CN=Rodney Walker	200
24.01.2011 10:43:08.015	24.01.2011 10:50:53.630	00C223CE-260C-11E0-8D87-0026552413C2	undefined	charm.physik.uni-wuppertal.de	/C=DE/O=GermanGrid/OU=UniWuppertal/CN=Tim Muenchen	0
24.01.2011 10:19:12.669	24.01.2011 10:25:49.509	A96BDCD0-2608-11E0-B25D-0026552413C2	undefined	charm.physik.uni-wuppertal.de	/C=DE/O=GermanGrid/OU=UniWuppertal/CN=Tim Muenchen	0
24.01.2011 09:59:47.380	24.01.2011 10:03:43.539	F29F2962-2605-11E0-8807-0026552413C2	undefined	charm.physik.uni-wuppertal.de	/C=DE/O=GermanGrid/OU=UniWuppertal/CN=Tim Muenchen	9
24.01.2011 09:01:03.220	24.01.2011 09:14:00.220	BE1295E6-25FD-11E0-9C9E-0026552413C2	undefined	charm.physik.uni-wuppertal.de	/C=DE/O=GermanGrid/OU=UniWuppertal/CN=Tim Muenchen	9
24.01.2011 08:53:12.198	24.01.2011 08:54:13.160	A5539568-25FC-11E0-A3C3-0026552413C2	undefined	charm.physik.uni-wuppertal.de	/C=DE/O=GermanGrid/OU=UniWuppertal/CN=Tim Muenchen	2
24.01.2011 08:46:43.707	24.01.2011 08:48:16.690	BDF4AF54-25FB-11E0-93BE-0026552413C2	undefined	charm.physik.uni-wuppertal.de	/C=DE/O=GermanGrid/OU=UniWuppertal/CN=Tim Muenchen	9
23.01.2011 21:28:08.954	23.01.2011 21:35:54.960	F213A7FE-259C-11E0-9F07-0026552413C2	undefined	charm.physik.uni-wuppertal.de	/C=DE/O=GermanGrid/OU=UniWuppertal/CN=Tim Muenchen	0
23.01.2011 21:26:44.606	23.01.2011 21:26:44.900	BFD15B56-259C-11E0-AB94-0026552413C2	undefined	charm.physik.uni-wuppertal.de	/C=DE/O=GermanGrid/OU=UniWuppertal/CN=Tim Muenchen	11
22.01.2011 07:54:38.286	22.01.2011 07:54:47.230	2252071C-2462-11E0-8F3E-0026552413C2	undefined	charm.physik.uni-wuppertal.de	/C=DE/O=GermanGrid/OU=UniWuppertal/CN=Tim Muenchen	1

Job statistics by CE

CE	total	running	succeeded	failed	unknown
ce2.farm.particle.cz:2119/jobmanager-lcgpbs-gridatlas	1	0	1	0	0
grid-ce5.physik.uni-wuppertal.de:2119/jobmanager-lcgpbs-dg_long	17	0	11	6	0
charm.physik.uni-wuppertal.de	33	0	11	22	0
wn004	62	0	41	20	1
all	**113**	**0**	**64**	**48**	**1**

page rendered in 0.50 s

© 2010 Bergische Universität Wuppertal

Figure 42: Example of the auto-generated statistics web page, generated by JEM. Shown are the last some job executions with their exit status, as well as statistical plots showing the fractions of succeeded and failed jobs monitored by JEM.

be briefly outlined for understandability; for a deeper explanation, the cited works have to be taken into account.

16.2.1 User perspective: Hanging Grid job

A problem in Grid job execution that occurs rather frequently is the "hanging" job - a job whose execution simply does not finish at all (until the middleware aborts the job because of its exceeded wall time, of course). Without real time data from the Grid job, a user has no chance to learn at what exact point his job came to a halt.

The figures 43 and 44 show an Athena Grid job executed at Wuppertal that was instrumented with JEM and resubmitted, after it didn't finish running the several times it was submitted before. By looking up the point in time the job didn't continue using the typical amount of resources (e.g. CPU and network) and then using JEMs verbose script monitoring data, it was possible to determine at what semantic step (event to process) the job stopped working, and to debug the user code accordingly.

This class of problems would greatly benefit from a means to selectively increase the monitoring data verbosity created by JEM during the job's run time, as proposed in the next section. As JEM allows to see that the job still is running, just not continuing, the user then could increase the verbosity (or request a full set of just-in-time debugging information like currently running

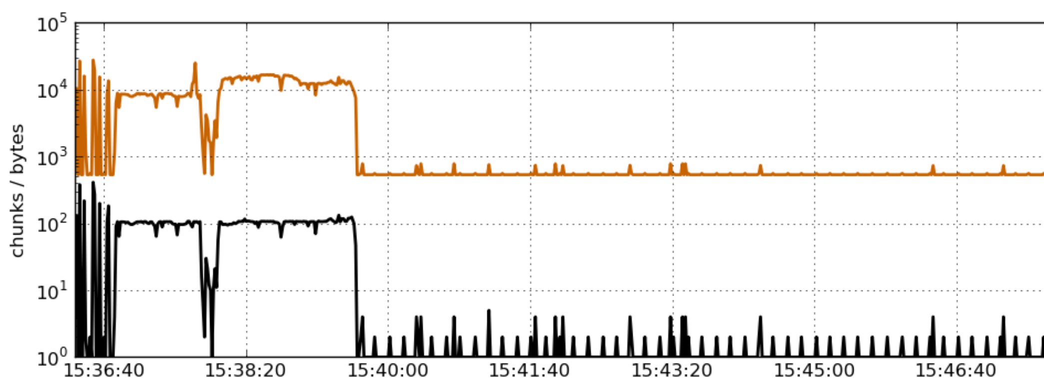


Figure 43: System metrics plot of a Grid job, hinting at it not continuing at one point - the number of monitoring events created drops to a minimum, created by the periodic system monitor samples and keepalive-events.

```
AthenaEventLoopMgr   INFO  ===>>>  start processing event #4659, run #165956 46 events processed so far  <<<===
AthenaEventLoopMgr   INFO  ===>>>  done processing event #4659, run #165956 47 events processed so far  <<<===
JetGlobalEventSetup  INFO  got store cleared event
AthenaEventLoopMgr   INFO  ===>>>  start processing event #4930, run #165956 47 events processed so far  <<<===
AthenaEventLoopMgr   INFO  ===>>>  done processing event #4930, run #165956 48 events processed so far  <<<===
JetGlobalEventSetup  INFO  got store cleared event
AthenaEventLoopMgr   INFO  ===>>>  start processing event #7203, run #165956 48 events processed so far  <<<===
AthenaEventLoopMgr   INFO  ===>>>  done processing event #7203, run #165956 49 events processed so far  <<<===
JetGlobalEventSetup  INFO  got store cleared event
AthenaEventLoopMgr   INFO  ===>>>  start processing event #9197, run #165956 49 events processed so far  <<<===
AthenaEventLoopMgr   INFO  ===>>>  done processing event #9197, run #165956 50 events processed so far  <<<===
JetGlobalEventSetup  INFO  got store cleared event
AthenaEventLoopMgr   INFO  ===>>>  start processing event #9266, run #165956 50 events processed so far  <<<===
AthenaEventLoopMgr   INFO  ===>>>  done processing event #9266, run #165956 51 events processed so far  <<<===
JetGlobalEventSetup  INFO  got store cleared event
AthenaEventLoopMgr   INFO  ===>>>  start processing event #9502, run #165956 51 events processed so far  <<<===
AthenaEventLoopMgr   INFO  ===>>>  done processing event #9502, run #165956 52 events processed so far  <<<===
JetGlobalEventSetup  INFO  got store cleared event
AthenaEventLoopMgr   INFO  ===>>>  start processing event #9167, run #165956 52 events processed so far  <<<===
```

Figure 44: Last events recorded in the job before it stopped progressing.

processes and even memory dumps) to infer not only the exact spot in its execution the job hangs at, but maybe also the reason for it not progressing.

16.2.2 Admin perspective: Excess dCache mover usage

dCache[47] is a multiple-server-based shared file system accessible using a suite of remote file access protocols.

Opening a file via dCache, from the client system's point of view, means opening a network connection to the dCache server. A resource - called **mover** - on the dCache server is assigned to that connection and the file contents delivered. Each dCache installation supports a certain number of concurrent movers; the exact value is configuration dependent.

If all available movers are in use, no further file can be opened on the dCache store, until some client closes an opened file and thus frees one of the

movers on the server. Hence, opening a large number of files and not closing them again can be considered bad practice or even malicious usage, and site admins need to know quickly which user job is holding all movers to be able to abort it, allowing other jobs to continue to run.

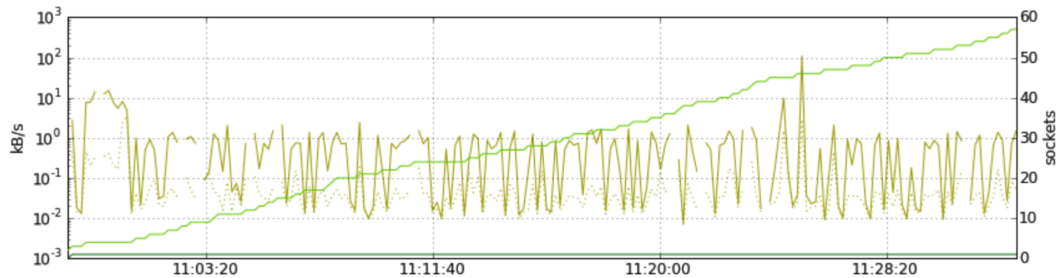


Figure 45: Linearly rising number of open network sockets, hinting at a job opening a large number of files via dCache without closing them.

Using JEM in all jobs enables one to detect jobs using an atypical large number of open network sockets, or ones that continuously open new network sockets without closing any, resulting in a linear rise of used sockets (figure 45). If a suspiciously large number of dCache movers are used at the same time, those jobs are candidates for being responsible for the excess mover usage on the site.

At the ATLAS Grid site installation at the University of Wuppertal, exactly this scenario happened during the author's work: A certain kind of user job, executed by students at Wuppertal, reproducibly caused a rapid dCache mover depletion. Using JEM, the site admins were able to detect such a depletion in time to identify and abort the misbehaving job, protecting the other running jobs and improving the site's job success efficiency.

17 Outlook

Although the monitoring software has reached production-level quality in the scope of this work, and is already available for usage by HEP community members, there is room for improvement and extension.

In this final section of the present work, a number of possible follow-up works are shown, and some open issues discussed that are being worked on by the author and further developers now. To ensure the acceptance of JEM, it is mandatory that such continued maintenance and support is provided.

17.1 Open questions

The larger issues that will be attacked in the near future are:

- **JEM behaviour in split-jobs**
  Grid jobs that don't run on one single worker node, but are splitted at submission into smaller fractions, are called **split jobs** (see section 2.4.2). The fractions - called **sub jobs** - are then executed in parallel on many worker nodes of a single CE, or even on multiple CEs. Since usually the sub jobs do not need to communicate with each other, one can talk about **data parallel execution** in this context. After the execution of all sub jobs finished, their respective results are gathered and merged to form the final split job result.

  Both Ganga and Panda provide this split job functionality to their users. As JEM replaces the Grid job executable by its own launcher script, spawning the original executable in a supervised fashion on each worker node, care has to be taken to prevent massive data flooding of the central servers (JEMs messaging infrastructure, but even more so the receiving UI-machine the split job was submitted from) when not one, but thousands of jobs are spawned at the same time by the same user.

  On top of that, one could argue that it is unnecessary to monitor each and every sub job of a split job, as they all execute the same user code, albeit using different parts of the physics data. It may be sufficient to just monitor a small subset of these jobs in higher verbosity, and to use the trigger system to have JEM run idle with the rest of the sub jobs, only creating monitoring data at all on error- and higher-risk-conditions.

  Finally, the presentation of the monitoring data of split jobs to the user - for example inside of Ganga - has to be rethought of, as it is not sensible to mix the monitoring data of multiple jobs.

- **Assuring full semantic unintrusiveness**
  There are special job execution patterns that, at the moment, still can not be supervised in a one-hundred-percent non-intrusive manner. This means, for a small subset of Grid jobs, the attachment of JEM to the

job alters its behaviour and thus, its outcome. Of course this is not acceptable, so efforts are taken to resolve these issues.

One example of those special execution patterns is the injection of custom memory allocation libraries at run-time into the user job's executable. This is done, for example, by the Athena framework in its default configuration. The custom memory allocators are to provide a better performance in specialized usage schemes and better error-reporting on misuse by the user algorithm. The memory allocators are loaded as shared libraries against the user algorithm (much like JEMs CTracer is); It is essential to provide exactly the same environment to the custom allocators as the bare user algorithm does in such cases.

- **Definition of "interesting" monitoring data**
  As the job monitoring verbosity is increased, it becomes more and more important to filter the data to prevent data flood (as reasoned in section 11). To be able to do so, however, it has to be decided what monitoring data - or subset of that data - is interesting (or useful) enough to be preserved, and what parts of the created data can be discarded without hampering the job problem diagnostics.

  This task can not be done entirely automatically by the software, but has to be predefined as rules in triggers by the developers and/or users, after more experience in large-scale job monitoring using JEM has been gathered.

17.2 Further development

Some further development areas, including ones already tackled by the author and further developers in follow-up projects of this work, are:

- **Integration into the Panda brokerage system**
  In section 2.4.1, the Panda system has been presented. Being an ATLAS-only development, a trend can be seen in ATLAS user analysis Grid jobs that Panda is slowly gaining ground, as more and more users prefer this system over the classic (WMS-) Grid job brokerage. For the centrally-organized Monte Carlo event production, Panda already is the established de-facto standard in ATLAS.

  Concerning the integration of JEM into Panda, the main differences to the WMS-based brokerage are that the user normally does not have control over the exact script that is run on the Grid WN (instead, he just provides his analysis code as library to Athena, which is launched in a standardized fashion by the Panda pilot job), and the different overall job brokerage semantic: pulling jobs from a central database instead of having the users push the jobs via the WMS to the CEs.

Both differences require a changed approach to job monitoring in general, and to the addition of JEM in particular. An adapted integration of JEM has already been prototyped by the author, and is now actively being developed and tested.

- **Implementation of sophisticated trigger scripts**
  The presented example trigger scripts (see section 12.3.3) only scratch the surface of what's possible using the ring buffer infrastructure presented in part IV of this work.

  Applying the trigger API for real-time adaptive monitoring, automatic verbosity control and drop-in extensive remote debugging - the automatic activation of the CTracer at strategic sections of a job's execution, triggered by key events - is a challenging, but very beneficial follow-up to this work. Also, the application of the trigger system for data reduction and split job handling, as hinted at previously in this section, is to be worked on.

- **A new data selection approach in the CTracer**
  As was shown in section 9.4, for reasons of performance degradation, the only sensible way to add the CTracer to a job's execution is to trace and record only a small subset of events (calls, returns) in the user application. A black list based approach, like implemented in the first, experimental versions of the CTracer (see section 8.2), turned out not to be sufficient.

  A number of possible alternatives has to be evaluated. The most obvious optimization, turning the black list into a white list and only recording events specified by the user in beforehand, has to be discussed critically; it would be beneficial not to rely on the user specifying all interesting spots in his code manually.

- **Ring buffer diagnostics**
  The current usage of the shared ring buffer infrastructure - how much of the ring buffer space is still available, how much of the shared identifier cache is used, how much monitoring data of a given type is created at a specific point of time, etc. - are important diagnostic metrics during JEMs run time (see section 12.2).

  In a follow-up project, JEM is extended by a self-diagnostic functionality that records such metrics during the job's execution and inserts this data into the monitoring data stream.

  By gathering such meta-monitoring data (monitoring the monitor), the further development and maintenance of JEM is eased greatly. Several bugs of high severity in JEM itself already have been found and fixed utilizing an early, experimental implementation of this diagnostic functionality.

- **A real time per-job control channel and protocol**
  With the provided infrastructure in terms of messaging and the modular architecture of JEM, a real time communication channel in the opposite direction - from the user to the job - can be thought of. With such a control channel, users would be able to control both the job execution itself, as well as the JEM-monitoring of their job, in real time.

  One has to be cautious, though, to secure the control channel to prevent misusage by other (malicious) users. The control channel has to be encrypted and secure authentication and authorization has to be applied. Existing Grid services could be used for that purpose.

  Examples for possible commands users might want to send to their jobs in real time are:

  - Abort the running job - because the user might have seen in the real time monitoring data provided by JEM that the job is semantically erroneous, while being technically running flawless from the Grid- and WN-perspective;
  - Query the job's current exact status and progress, up to the creation of whole memory dumps on demand;
  - Accessing the job's standard output stream in real time ("remote `tail -f`");
  - Controlling the debug log output of JEM itself (for development purposes);
  - Increasing or decreasing the job monitoring verbosity.

  Some of those use cases are available via the Grid middleware already - for example the cancellation of the job - but would be executed faster (immediately) when performed "from the inside" of the job via JEM. Other use cases are not possible at all without a control channel like the discussed one (e.g. the change of the job's logging verbosity).

  As part of this work, the author already technically implemented a back channel from the user to the job, allowing to send simple commands composed of key, value pairs[28]. What needs to be done now is the full design of a command protocol, the implementation of the command actions in JEM itself (this means implementing what exactly the WN component of JEM has to perform for each received command), and the securing of the command communication using existing Grid security services.

[28] e.g. "`command=set_log_verbosity,logger=ScriptLauncher,verbosity=30`"

- **Integration of JEM statistics into the ATLAS dashboards**
  The ATLAS collaboration has implemented a number of diagnostic overview pages that show Grid site health, job throughput and other such metrics, called "dashboards".[48–50] Grid maintenance personnel ("shifters") use these dashboards to get a quick overview about the status of the Grid (or more specific, the ATLAS VO), to decide whether to escalate problems and to inform sites about possible misconfigurations, etc.

  These dashboards at the moment are being consolidated, technically and concerning their user interface and design. The front-end of all ATLAS dashboards is changed to a browser-based rich-client solution[29] based on HTML and JavaScript. This takes load off of the back-end servers, because for example graphical plots are generated on each client viewing the dashboard pages, instead of centrally on the server for all users. Most of the dashboards have a database backend and a web-service based middleware, mostly using the JSON protocol.[51]

  For JEM, it would be beneficial to also implement such a "web 3.0", JSON-based dashboard view. On the one hand, it will be easy then to match the look-and-feel of the other ATLAS dashboards, making it more probable that JEM is accepted by both the users and the other dashboard developers, integrating JEM into the dashboard landscape and creating visibility to the shifters. On the other hand, by providing the gathered data in the standardized (and ATLAS-approved) JSON format, the other dashboards can then use JEM as an additional data source for their purposes. This scenario, of course, most benefits from an addition of JEM in a minimum-verbosity mode to *all* executed Grid jobs.

In addition, further large-scale extensions of JEM can be thought of, that are not being requested by users at the moment, but may help in a broader deployment of the system to a larger audience. One example for such an extension is the implementation of new script monitors, for example for PERL scripts or for JAVA applications. It is being discussed currently if such new script monitors should be developed and integrated into JEM; technically, the integration of new data producers in JEMs infrastructure can be easily achieved after they have been developed in a stand-alone fashion. The provision of JEMs new central services (logging, configuration management, etc) ease such a development further.

[29]this is dubbed "web 3.0" in the industry

18 Conclusion

The user centric monitoring solution JEM, developed at the University of Wuppertal, has reached production quality in the scope of this work. It is deployed and ready for usage in the Worldwide LHC Computing Grid, and integrated into its middleware environment. Users can apply in-depth monitoring to their Grid jobs without much effort and receive and analyse the monitoring data in real time. JEM already has helped in identifying and solving of typical Grid job failure scenarios, and can help single users as well as operational staff (shift personnel) in improving Grid job efficiency.

The changes and additions described in this work helped in making the software stable, easy to maintain and optimized it to scale well. Refactoring the code base of JEM for modularity, reusability and generality helped in achieving these goals. Numerous further development opportunities have been identified and several follow-up projects already have been launched. Additionally, there are feature requests from other ATLAS infrastructure groups, as well as joint efforts with site monitoring projects.

Part VI
Appendices

A Module structure

This is the module structure of JEM after the major refactoring described in section 12.1. Parts of the application dealt with in this work are marked by adding references to the corresponding sections in parentheses behind the description of the entry. Also, smaller (helper-) modules are omitted in the list.

module	description
`Common`	General purpose- and shared modules (12.1)
`App`	Program entry point for child processes
`Config`	Configuration management and cmdline parsing
`Logging`	Logging facility (file-, console- & remote-logging)
`PluginLoader`	Handler module for optional program components
`SharedMemory`	The shared memory / ringbuffer Python API (12.2)
`SignalHandler`	Signal handling functions
`Utils`	Miscellaneous shared modules and helper functions
`Modes`	The running modes of JEM
`Analysis`	Graphical monitoring data analysis application
`Ganga`	GANGA-integration mode
`Packer`	Mode for automatic deployment of the JEM-libraries
`Spy`	Monitoring data statistics mode (16.1.2)
`UI`	User interface (data receiving and displaying) mode
`Core`	Main UI event loop
`Processors`	Trigger-like event processor modules (13.1)
`Publishers`	Data publishers (presentation of the data to the user)
`Receivers`	Data receivers for real time monitoring data
`WN`	Central monitoring mode, usually run on the Grid WN
`Core`	Main WN event loop (12.3.1)
`ScriptMonitors`	Script execution supervision modules (5.3.2)
`Bash`	Bash script monitor
`CTracer`	Binary tracing module / remote debugger (9.2)
`Python`	Python script monitor
`SystemMonitor`	Periodically sampling system metrics monitor
`Trigger`	Trigger API and implementations (12.3)
`Valves`	Data exporter- and real time transmitter modules
`ModuleSources`	Source code of all non-python parts of JEM
`CTracer`	CTRACER source code
`JEMbash`	Modified bash 3.2 source code
`SharedMemory`	Shared memory / ring buffer core library code
`UICore`	C-Part of the UI core
`uStomp`	Fast and lightweight C-STOMP-library
`WNCore`	C-Part of the WN core
`test`	Unit-test hierarchy (16.1.1)

Table 10: Module structure of JEM.

B Example trigger implementations

These example triggers have been described in section 12.3.3. Refer to section 12.3.2 for a description of the API. The python-implementation is shown here, analogous C versions can be implemented similarly. Internal methods not needed for understanding have been omitted.

Statistics trigger

```
from Common.Logging.Logging import Logger
from Common.SharedMemory import shm
from Modes.WN.Trigger.Trigger import Trigger
import time

chunktypes, blocktypes, chunk_names, block_names = shm.getChunksDict()
logger = Logger("WN.Trigger")

class StatisticsTrigger(Trigger):
    def __init__(self):
        Trigger.__init__(self)
        self.chunk_types += [chunktypes["CHUNK_TYPE_ALL"]]
        self.chunksTotal = 0
        self.chunkCount = 0
        self.bytesTotal = 0
        self.byteCount = 0
        self.lastSample = time.time()

    def examine(self):
        self.chunksTotal += 1
        self.chunkCount += 1

        l = len(self.chunk)
        self.bytesTotal += l
        self.byteCount += l

        dt = time.time() - self.lastSample
        if dt >= 1.0:
            self.lastSample = time.time()
            logger.info("chunks/s: %.1f, bytes/s: %.1f" % \
                ((self.chunkCount / dt), (self.byteCount / dt)))
            self.chunkCount = 0
            self.byteCount = 0

    def finalize(self):
        logger.info("chunks total: %d, bytes total: %d" % \
            (self.chunksTotal, self.bytesTotal))
```

Exception tracker

```
from Common.SharedMemory import shm
from Common.SharedMemory.FilteredIterator import FilteredIterator
from Modes.WN.Trigger.Trigger import Trigger

chunktypes, blocktypes, chunk_names, block_names = shm.getChunksDict()

class ExceptionTracker(Trigger):
    def __init__(self, shmkey):
      Trigger.__init__(self)
      self.chunk_types += [chunktypes["CHUNK_TYPE_EXCEPTION"],
                           chunktypes["CHUNK_TYPE_SCRIPT_LINE_EVENT"]]
      self.__frames = 0    # number of frames a raised exc. passes
      self.__except = None # the exception event we're tracking

    def __is_lib_location(self, path, file):
      # func impl omitted - consists of string comparisons to infer
      #                    whether the given path/filename is a python
      #                    system library file or user file.

    def __is_exc_catched_event(self, chunk):
      # func impl omitted - uses the python interpreter interface to
      #                    infer whether the script line is an "except"
      #                    clause, and what exception types are catched

    def examine(self):
      if self.chunk.chunk_type == chunktypes["CHUNK_TYPE_EXCEPTION"]:
        path, file, line, frame = self.chunk.get_location_tuple()
        if not self.__is_lib_location(path,file):
          if self.__except is None:
            # we didn't see that one before - inner-most exc frame!
            self.__except = self.chunk
            self.__frames = 1
          else:
            # an exception being thrown over n frames creates n
            # exception-chunks (one for each frame left without
            # being catched). we count the consecutive exc events.
            self.__frames += 1
            if self.__frames > 5: # in reality, this is configurable
              # approve the original exception chunk
              self.approve(self.__except) # "approve" impl is in Trigger
              self.__except = None
              self.__frames = 0
      else: # no exception-chunk: must be generic line-event
        if self.__except is None: return # if we're not tracking...
        if self.__is_exc_catched_event(self.chunk): # is it a "catch"?
          self.discard(self.__except) # "discard" impl also is in Trigger
          self.__except = None
          self.__frames = 0

    def finalize(self):
      pass
```

User-defined progress tracker

```
from Common.SharedMemory import shm
from Modes.WN.Trigger.Trigger import Trigger
import re

chunktypes, blocktypes, chunk_names, block_names = shm.getChunksDict()

class ProgressTrackingTrigger(Trigger):
    def __init__(self, shmkey):
      Trigger.__init__(self)
      self.chunk_types += [chunktypes["CHUNK_TYPE_OUTPUT_LINE"]]
      self.__p = re.compile(r"(?:Begin of event )([0-9]+)")

    def examine(self): # the event has an output-line-block for sure
      for pb in self.chunk.payload:
        if isinstance(pb, OutputLineBlock) and self.__p.match(pb.line):
          try: # discard all older chunks
            shm.invalidate(-1) # "-1" means "all"
          except:
            pass # in reality, errors should be handled.

          try: # add mile stone event, so the UI can highlight this spot
            ct = chunktypes["CHUNK_TYPE_MILESTONE_EVENT"]
            chunk = Chunk(creator = 0, chunktype = ct)
            # ...add information about the milestone to the chunk here...
            chunk.writeToShmem(shm, self.__shm_handle)
        except:
            pass # in reality, errors should be handled.

    def finalize(self):
      pass
```

Event multiplexing on the UI

```
from Common.SharedMemory import shm
from Modes.WN.Trigger.Trigger import Trigger
chunktypes, blocktypes, chunk_names, block_names = shm.getChunksDict()

class MultiplexTrigger(Trigger):
    def __init__(self):
      Trigger.__init__(self)
      self.chunk_types += [chunktypes["CHUNK_TYPE_ALL"]]
      self.__processors = []

    def addProcessor(self, p):
      self.__processors.append(p)

    def removeProcessor(self, p):
      try:
        self.__processors.remove(p)
      except: pass # error handling omitted
```

```
18    def examine(self):
19      c = self.chunk # make a copy
20      for p in self.__processors:
21        try:
22          p.process(c)
23        except:
24          pass # error handling omitted
25
26    def finalize(self):
27      for p in self.__processors:
28        p.finalize()
```

C List of Figures

1 The three generations of known fermions 16
2 Schematic display of the ATLAS detector 19
3 Showcase event in ATLAS 21
4 Typical Monte Carlo simulation chain 24
5 gLite WMS job brokerage 36
6 Steps to prepare and submit an Athena Grid job on the WLCG 42
7 Grid job states and status transitions in gLite 43
8 Pilot job brokerage 46
9 Example Ganga session 49
10 Job success rates in the WLCG 52
11 General architecture of JEM 55
12 Example of JEMs HTML output 57
13 JEMs worker node component processes 58
14 The Relational Grid Monitoring Architecture (R-GMA) 60
15 Example gLite job submit call 64
16 The equivalent JEM CLI job submit call 64
17 Example JEM UI session 65
18 Excerpt of a Ganga session with JEM integration 66
19 System metrics plot as presented by JEM in Ganga 68
20 Example gcc call to prepare a user application for the CTracer 76
21 Call stack traversal 78
22 General concept of the CTracer's victim-watchdog system . . 80
23 Excerpt of example data gathered using the CTracer 81
24 Schematic display of a custom appendable buffer 87
25 Additional Ganga-job-configuration to enable the CTracer . 89
26 Athena job periods 89
27 CTracer events in a typical Athena job 90
28 Performance impact of the CTracer 91
29 Binary chunk format for monitoring data 95
30 Example monitoring data chunks 100
31 General principle of a ring buffer 105

32 Ring buffer header . . . 106
33 Sequential writes into a ring buffer . . . 106
34 Concurrent writes into a ring buffer . . . 108
35 Deferred deletion in the ring buffer . . . 109
36 Monitoring data visualization tool . . . 122
37 Maximum throughput by chunk size, 64kB ring . . . 125
38 Maximum throughput by chunk size, 1MB ring . . . 126
39 Increasing number of producers, 1MB ring . . . 126
40 Increasing number of delayed producers, 1MB ring . . . 126
41 Example unit test output . . . 130
42 Auto-generated statistics web page . . . 132
43 System metric plot of a “hanging” user job . . . 133
44 Recorded events in a “hanging” user job . . . 133
45 Linearly rising number of open network sockets . . . 134

D List of Tables

1 The six quark flavours . . . 15
2 Bosons mediating the fundamental interactions . . . 16
3 Ganga backend types . . . 48
4 Major JEM version history milestones before this work. . . . 53
5 Configuration options of the CTracer . . . 84
6 Chunk header fields . . . 96
7 Monitoring data block types . . . 97
8 Monitoring data chunk types . . . 98
9 Configuration options of JEM in Ganga . . . 121
10 Module structure of JEM. . . . 141

E List of Listings

1 Example JDL-file . . . 41
2 Trace callback function signatures . . . 77
3 Wrapper script for manual invocation of the CTracer . . . 85
4 Example CMT `requirements`-file . . . 88
5 C trigger API . . . 113
6 Python trigger API . . . 114
7 Schedule script for automated ring buffer stress tests . . . 124

F Acronyms and abbreviations

AFS	"Andrew's File System", a filesystem that can be shared over wide area networks
ALICE	"**A** **L**arge **I**on **C**ollider **E**xperiment", an experiment bringing heavy ion nuclei to collision
AOD	Analysis Object Data
API	application programmer's interface
ASCII	American Standard Code for Information Interchange
Athena	ATLAS' main physics analysis software framework
ATLAS	"**A** **T**oroidal **L**HC **A**pparatu**s**", a particle physics experiment at the LHC trying - among other things - to find the Higgs Boson
CA	certificate authority
CE	compute element
CERN	Conseil Européen pour la Recherche Nucléaire, the European center for nuclear- and particle physics research near Geneva, Switzerland
CLI	command-line interface
CMS	"**C**ompact **M**uon **S**olenoid", the second all-purpose experiment at the LHC
CMT	a dependency-solving build management utility used in ATLAS
CVS	Concurrent Version System, a utility for revision control
DCS	Detector Control System
DN	distinguished name
DPD	Derived Physics Data
DWARF	Debugging With Attributed Record Formats
EGEE	Enabling Grids for E-Science in Europe
ELF	Executable and Linkable Format
ESD	Event Summary Data
FIFO	first in, first out
Ganga	"Gaudi and Grid Alliance", an object-oriented Grid job management UI
gcc	The GNU C Compiler
gdb	the GNU debugger
GIS	Grid Information Service

gLite	The Globus-based Grid middleware implementation used by the WLCG
Globus	The de-facto standard all-purpose Grid middleware
HEP	High-Energy Physics
HTML	Hypertext Markup Language
IPC	Inter-Process Communication
ITU-T	International Telecommunication Union
JDL	Job Description Language
JEM	Job Execution Monitor
JLE	JEM Log Explorer
JSON	Javascript Object Notation
LFN	logical file name
LHC	Large Hadron Collider
LHCb	The "LHC b-Physics Experiment", trying to reason the disequilibrium between matter and antimatter in the universe
MC	Monte Carlo Production - the simulation of physics events via software
MDS	Monitoring and Discovery Service
OS	operating system
Panda	**P**roduction **An**d **D**istributed **A**nalysis, the Pilot Factory based job brokerage system on the WLCG
pAthena	"Panda and Athena", a command-line tool designed to automate Grid job preparation and submittage in the WLCG
POSIX	Portable Operating System Interface for Unix
R-GMA	Relational Grid Monitoring Architecture
RAID	Redundant Array of Independent Disks
RAM	random access memory
RCS	Revision Control System
ROOT	A high-performance maths and data management framework
SAAS	"Software as a Service", (remote) collections of reusable software components
SE	storage element
SI	Système international d'unités
SLC	Scientific Linux for CERN
SRM	Storage Resource Manager

STOMP	Simple text over messaging protocol
SVN	Short for “Subversion”, a utility for revision control
TDAQ	Trigger and Data Acquisition System
UI	user interface
VO	virtual organization
WLCG	World-Wide LHC Computing Grid
WMS	Workload Management System
WN	worker node
XML	Extensible Markup Language

G References

[1] M. E. Peskin, , D. V. Schroeder. An Introduction to quantum field theory. Reading, USA: Addison-Wesley (1995) 842 p.

[2] Francis Halzen, Alan D. Martin. *Quarks and Leptons: An Introductory Course in Modern Particle Physics*. John Wiley and Sons Inc., 1984.

[3] G. Karagiorgi et al. Leptonic CP violation studies at MiniBooNE in the $(3+2)$ sterile neutrino oscillation hypothesis. *Phys. Rev. D*, 75(1):013011, Jan 2007.

[4] K. Nakamura et al. (Particle Data Group). 2010 Review of Particle Physics. 2010.

[5] R.J. Van de Graaff. A 1500000 Volt Electrostatic Generator. *Phys. Rev.*, 38, 1931.

[6] Joao Pequenao. Computer generated image of the whole ATLAS detector. Mar 2008.

[7] Joao Pequenao. Event Cross Section in a computer generated image of the ATLAS detector. Mar 2008.

[8] Rene Brun, Fons Rademakers. ROOT - An Object Oriented Data Analysis Framework. In *Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389*, 1997. See also `http://root.cern.ch/`.

[9] D. Froidevaux E. Richter-Was, L. Poggioli. ATLFAST 2.0 a fast simulation package for ATLAS. ATL-PHYS-98-131, 1998.

[10] M.A.Dobbs et al. Les Houches guidebook to Monte Carlo generators for hadron collider physics. hep-ph/0403045, 2004.

[11] T. Sjöstrand et al. High-Energy-Physics Event Generation with PYTHIA 6.1. Computer Phys. Commun. 135 (2001) 238 (LU TP 00-30, hep-ph/0010017), 2000.

[12] T. Gleisberg et al. Event generation with SHERPA 1.1. JHEP02 (2009) 007, 2009.

[13] M.L. Mangano et al. ALPGEN, a generator for hard multiparton processes in hadronic collisions. JHEP 0307:001, 2003.

[14] J. Allison et al. Geant4 Developments and Applications. In *IEEE Transactions on Nuclear Science*, 2006. 53 No. 1, p270-278.

[15] G. Barrand et al. GAUDI - A software architecture and framework for building LHCb data processing applications. In *Proceedings of CHEP 2000*, 2000.

[16] CMT, Configuration Management Tool.
`https://www.cmtsite.org`.
Accessed October 2010.

[17] CVS - Open Source Version Control.
`www.nongnu.org/cvs`.
Accessed October 2010.

[18] Apache Subversion.
`http://subversion.apache.org`.
Accessed October 2010.

[19] Len Kleinrock. UCLA to be first station in nationwide computer network. Press release, University of California, Los Angeles, 1969.

[20] Ian Foster, Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[21] Ian Foster, Carl Kesselman. The Anatomy of the Grid. 2000.

[22] Ian Foster. What is the Grid? A Three Point Checklist. 2002.

[23] David P. Anderson et al. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45 no. 11:56–61, Nov 2002.

[24] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[25] GridView: Monitoring and Visualization Tool for LCG.
`http://gridview.cern.ch/GRIDVIEW/dt_index.php`.
Accessed on 23.08.2010, query period: July 2010.

[26] The WLCG Board. WLCG Regional Centre Resource Tables. presented at April 2010 RRB (v 12 April 2010), 2010.

[27] E. Laure et al. Programming the Grid with gLite. In *Computational Methods in Science and Technology*, page 2006, 2006.

[28] Ian Foster, Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[29] Enabling Grids for E-Science. http://www.eu-egee.org/.

[30] D. Adams et al. The ATLAS Computing Model. 2005. ATL-SOFT-2004-007, CERN-LHCC-2004-037/G-085, V1.2.

[31] Kouřil D. et al. Distributed Tracking, Storage, and Re-use of Job State Information on the Grid. *Computing in High Energy and Nuclear Physics (CHEP04)*, 2004.

[32] The PanDA Production and Distributed Analysis System.
`https://twiki.cern.ch/twiki/bin/view/Atlas/PanDA`.
Accessed August 2010.

[33] GridView: A Monitoring and Visualization Tool for LCG.
`http://gridview.cern.ch`.
Accessed February 2011.

[34] Ahmad Hammad. Entwicklung eines Überwachungssystems für verteilte Prozesse im LHC-Computing Grid. 2005.

[35] Dmitri Igdalov. Entwicklung eines Systems zur Analyse und Überwachung der Verarbeitung von Rechenanforderungen im LHC Computing-Grid. 2005.

[36] Andreas Baldeau. Skriptüberwachung im Job-Execution-Monitor für das LHC Computing Grid. 2007.

[37] Martin Rau. Erweiterung der Benutzerschnittstelle für LHC Grid Jobs um die Monitoring-Funktionalität. 2008.

[38] Stefan Borovac. A users guide to JEMv2. 2007.

[39] Relational Grid Monitoring Architecture.
`https://www.r-gma.org`.
Accessed September 2010.

[40] The Python 2.3 Library Reference.
http://docs.python.org/release/2.3.5/lib/lib.html.
Accessed September 2010.

[41] Zafar Abbass. Data Transfer Analysis of a Monitoring Software for User Jobs in a World Wide Distributed Computing Grid. 2007.

[42] Free Standards Group. DWARF Debugging Information Format Version 3. 2005.

[43] R. Bayer. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972. 10.1007/BF00289509.

[44] Running Athena Hello World Example.
ATLAS Wiki on https://twiki.cern.ch
page WorkBookRunAthenaHelloWorld.
Accessed October 2010.

[45] the GNU debugger.
http://www.gnu.org/software/gdb.
Accessed February 2011.

[46] Ali I. Jehangiri. Performance Analysis of Monitoring Software for User Jobs in a World Wide Distributed Computing Grid. 2007.

[47] Patrick Fuhrmann. dCache: the commodity cache. In *Twelfth NASA Goddard and Twenty First IEEE Conference on Mass Storage Systems and Technologies*, 2004.

[48] ATLAS prodsys dashboard.
http://dashb-atlas-prodsys.cern.ch/dashboard/request.py/overview.
Accessed February 2011.

[49] ATLAS DDM dashboard.
http://dashb-atlas-data.cern.ch/dashboard/request.py/site.
Accessed February 2011.

[50] ATLAS analysis dashboard.
http://dashb-atlas-job.cern.ch/dashboard/request.py/jobsummary.
Accessed February 2011.

[51] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON).
http://www.ietf.org/rfc/rfc4627.txt?number=4627, 2006.