Bergische Universität Wuppertal
Fachbereich C
Mathematik und Naturwissenschaften
Fachgruppe Physik

Applying Live Job Monitoring Techniques to Monte Carlo Validation

Dissertation
by
Frank Volkmer

November 19, 2016

Die Dissertation kann wie folgt zitiert werden:

urn:nbn:de:hbz:468-20161123-102559-1
[http://nbn-resolving.de/urn/resolver.pl?urn=urn%3Anbn%3Ade%3Ahbz%3A468-20161123-102559-1]

Für:

Julia und Lara

Danksagung

An dieser Stelle möchte ich die Gelegenheit nutzen, mich bei all denen ganz herzlich zu bedanken, die mich während meiner gesamten Promotion im Allgemeinen und bei der Erstellung dieser Arbeit im Speziellen in vielfacher Art und Weise unterstützt haben.

Herrn Professor Dr. Peter Mättig danke ich für die Bereistellung des Themas, die Betreuung der Arbeit und der nötigen Geduld diese lange Promotion zu begleiten.

Auch Herrn Prof. Dr. Nikolaus Wulff danke ich für die Herstellung des Kontakts nach Wuppertal, ohne den meine Arbeit dort niemals möglich gewesen wäre und außerdem für die aufmunternden Worte und das Vertrauen in einer schwierigen Zeit.

Julia Wolthaus, Saskia Springmann, Julia Fischer, Gunar Ernis, Benedikt Bergenthal, Dr. Lukas Püllen, Dr. Tim dos Santos, Dr. Torsten Harenberg, Dr. Marisa Sandhoff, Dr. Frank Ellinghaus und Dr. Klaus Hamacher danke ich für die Unterstützung bei der Ausmerzung der vielen kleinen Fehler die sich bei der Erstellung dieser Arbeit eingeschlichen haben.

Den Bürokollegen danke ich für die Ausdauer und das Verständnis mich auszuhalten, wenn es nötig war.

Ein besonderer Dank gilt meiner Julia für die Geduld, den Beistand und die Motivation ohne die ich es niemals bis zum Ende geschafft hätte: Danke Schatz.

Outline of this Thesis

Monte Carlo (MC) computation, a method of repeated random sampling to obtain numerical results, is a necessary part of high energy particle physics experiments and is done in large quantities to gain statistical relevant data. Special software to simulate particle collisions is used and provides theoretical data to compare against experimental results.

These software packages are regularly improved and therefore need to be validated against known versions to ensure proper functionality. In this thesis, an extension of the Job Execution Monitor (JEM) framework is presented, which enables the afore mentioned validation efforts with an automated and standardized approach.

This thesis is structured in five parts. The first part briefly introduces particle physics and the Grid computing of the ATLAS experiment – the computational environment this thesis has been conducted in. In the second part the JEM is introduced, a user job centric monitoring approach, that is able to gather a wide variety of measurements of a monitored job instance. In Part III a GNU Debugger (GDB) based backtrace monitor prototype is presented, which allows JEM to peek into a running jobs memory and to gather data about the jobs internals.

Part IV then discusses in depth the various JEM subsystems necessary to aid the Monte Carlo validation efforts. These are a dynamically configurable ActivationService, which allows selective and automated instrumentation of Grid jobs with arbitrary JEM modules to generate quality histograms of production jobs, an automated histogram merging and comparison system to produce measures, which gives a quality value for each comparison and a web service allowing comfortable control of the validation functionality and to view the results on special web pages. The final part then closes with a summary and an outlook for further possible development efforts of JEM.

Contents

Part I.

Primer: Experimenting in High Energy Particle Physics

Introduction

To understand the smallest constituents of matter, high energy particle phenomena need to be studied. Research in high energy particle physics is done, by building large particle accelerators, where particles collide in experiments designed to measure the remnants of these collisions. Particle physics explores the constituents of matter and their interactions at the smallest scales.

The standard model of particle physics is the basic theory describing the building blocks of matter and their interaction by three different forces. It is briefly introduced in Chapter 1. The LHC and the ATLAS Experiment are introduced in Chapter 2.

The computational environment, the Grid, where the experiments data is analysed, is then described in Chapter 3 including its functionality and its evolution over the last years. Also, the ATLAS specific software environment is explained, especially the MC mass production chain.

1. The Standard Model

The Standard Model of particle physics (SM) [1] is a successful theory, that has grown over the past century to explain the existence and interaction of subatomic particles. Particles are generally grouped in two different classes, fermions as the matter of which the observable universe is formed and bosons as particles carrying the fundamental forces.

1.1. Particles

The SM describes two types of fundamental matter particles: leptons and quarks. These form the building blocks of the known matter. Figure 1.1 shows an overview of the elementary particles with the typical values for their mass as well as electrical charge and spin. The particles of the standard model are grouped into three generations. Leptons and quarks of a generation are displayed in one column. Particles in consecutive generations are essentially the same, however, their mass increases with higher generations.

Fermions have half integer spin and bosons have full integer spin. Leptons are either charged particles like the electron e, the muon μ or the tau τ or their neutrinos, ν_e, ν_μ, ν_τ, which are electrically neutral. Quarks either have a charge of $+2/3$ for the up u, charm c and top t quark or $-1/3$ for the down d, strange s and bottom b quark. All leptons and quarks have associated anti-particles, which have identical mass and spin but opposite electrical charge. The force carrying boson, shown in the figure all have a full integer spin 1, except the Higgs boson, which has a spin of 0. They are described in further detail in the following Section 1.2.

All matter particles are fermions and further on, stable matter is constructed from u and d quarks and the charged lepton e of the first family. The quarks form protons and neutrons in the combinations uud and udd, resulting in a total electrical charge of $+1$ for the p and 0 for the n, respectively. These two hadrons are the constituents of nuclei, which together with the electrons in their shells form atoms.

1.2. Forces / Interactions

In the SM matter interacts by exchanging quanta. These quanta, the vector bososns, mediate the forces. The force particles have full integer spin; they carry energy and momentum and, if applicable, strong, weak and/or electric charge from one particle to the other partner. In some cases these bosons can interact among each other. Figure 1.2 gives an overview of the existing particles of the SM and the possible interactions with each other. The details of this figure are explained in the following.

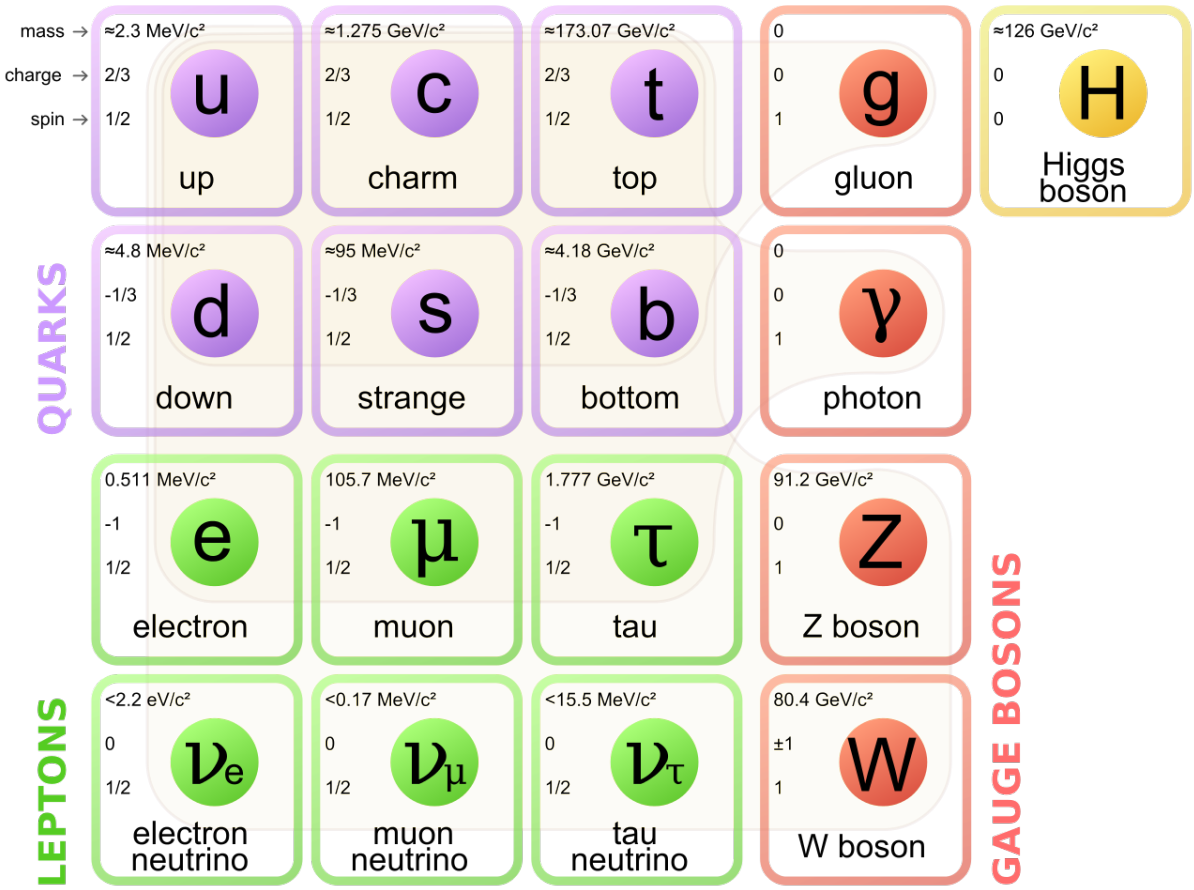


Figure 1.1.: Overview of all particles described in the standard model [2].

1.2.1. The electromagnetic force

The electromagnetic force is the most observable force as it mediates effects well known to our natural senses. It is this force, that forms atoms out of nuclei and electrons and lets them emit photons upon energy changes in these electrons and that lets these atoms form the molecules that allow us as life forms to exist. Electricity and magnetism are also direct effects of this force.

The mediating force carrier of the electromagnetic force is the mass- and chargeless photon γ. Due to its masslessness a photon's reach is infinite. Photons always travel at the speed of light c, have a full integer spin 1 and are therefore bosons.

Photons only interact with electrically charged particles, the e, μ and τ leptons. They also couple to the $W^{\pm}$ bosons and quarks as these also have electrical charge. Photons do not interact with each other.

1.2.2. The weak force

The $W^{\pm}$ and Z^0 bosons are transmitting the weak force. Contrary to all other known force carrying bosons, these bosons are massive, which leads to a very short lifetime or distance over which interactions

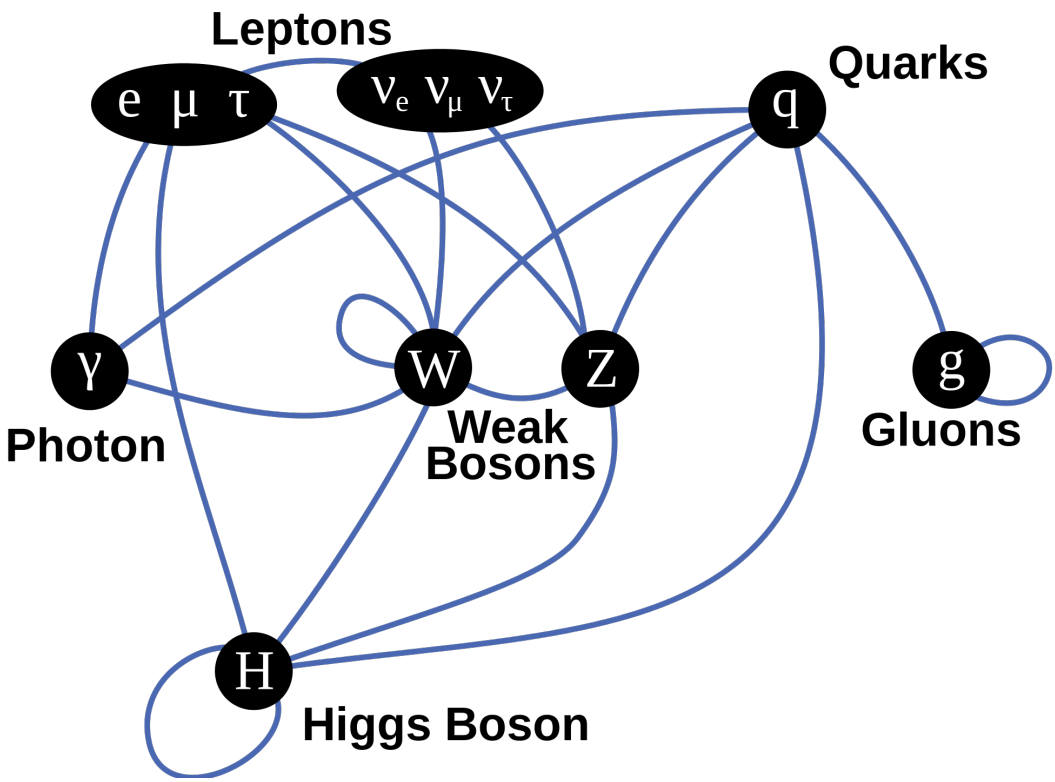


Figure 1.2.: Summary of all known interactions between elementary particles [3].

can occur. All fermions interact weakly. Weak interaction, more precisely the $W^{\pm}$ boson allows them to change their flavour, meaning, that fermions can change into other fermions[1]. The electric and color neutral neutrinos are the only elementary particles that exclusively interact with other particles through the weak force.

The weak force is for example responsible for the radioactive β^- decay, where a neutron is changed into a proton and an electron and an electron anti-neutrino are emitted from the nucleus: $n \rightarrow p^+ + e^- + \bar{\nu_e}$. What happens there is, that one d quark in the neutron changes its flavour to u quark, while emitting a W^- boson, which transports one negative elementary charge, as this needs to be conserved. The W^- then decays into an electron e and an electron anti-neutrino $\bar{\nu_e}$.

1.2.3. The strong force

The strong force is mediated by massless bosons called gluons g. Gluons only interact with quarks and among themselves. Gluons transmit a so called color charge. This name is derived from the three different types of strong charge called colors quarks can have: red, blue and green (and also their anti-colors for anti-quarks: anti-red, anti-blue and anti-green). These names were chosen in analogy to the three base colors, which together form white, uncolored light. Gluons themselves also carry a color and an anti-color charge, which leads to the fact, that gluons also interact with themselves.

[1] Leptons to leptons and quarks to quarks.

Bound states of quarks are called hadrons. Hadrons must be color-neutral or white to an outside observer. Color confinement leads to a constant attractive force between quarks. Once this distance is too big, and therefore enough energy exists in the bond, a new quark / anti-quark pair is created to keep the whole system color neutral. A result of color confinement is, that only three-quark hadrons, called baryons and two-quark $q\bar{q}$ hadrons, called mesons are allowed. Mesons have to consist of a quark and an anti-quark, which contain a color and an anti-color to remain neutral. Baryons can consist of a wide variety of permutations of quarks as long as they are color neutral. The only quark not forming hadrons is the top t, as it is so heavy, that it decays too quickly before it can be bound by the strong force. Quarks can never be observed alone and are always in a bound state.

Of the known hadrons only the proton is stable as a free particle. Neutrons are stable, once bound in atomic nuclei. All other baryons and all mesons are unstable and decay after some time. The strong force also allows the protons and neutrons to be bound together as nuclei.

1.3. Mass

The SM does not predict the mass for the fundamental particles. Observation shows however, that all of the above mentioned particles, except the photon and the gluon, have mass, which is known for some time now and in most cases measured quite precisely.

The Brout-Englert-Higgs-mechanism [4] extends the basic SM. It introduces a field, the so called Higgs field and a particle with spin 0, the Higgs boson H. All massive particles couple to the Higgs, this coupling or interaction in turn explains their mass.

One of the most prominent discoveries so far at the LHC experiments was the discovery of a particle in July 2012 by the CMS and ATLAS experiments [5]. As the data of the experiments got refined and more and more characteristics of the new particle were understood, it became general understanding that the long missing Higgs boson had been found.

1.4. Open questions

The SM successfully describes the interaction of subatomic particles. It is, however, incomplete, as open questions remain, that are not covered:

- For yet unknown reasons, gravity as a force is 10^{32} times weaker than the weak force and this is not fully understood. Also there exists no quantum field theory to describe the effects observed by gravity.
- The oscillation of neutrinos has been observed, meaning that neutrinos change their flavour, for example from ν_e to ν_μ. This can only be explained if neutrinos have mass.
- From cosmological studies it is known, that ordinary matter, from stars and galaxies to interstellar gas clouds and black holes, provides about 5% of the universes total mass. Another 27% is made of so called dark matter. This is a form of matter, that, apparently, does not interact with ordinary matter, i.e. any particle described in this chapter. It does however, have measurable gravitational effects, like gravity lensing. The last 68% is made of so called dark energy, leading to accelerated expansion of the universe.

2. LHC and Atlas

Research in high energy particle physics requires high energy particles and, therefore, particle accelerators, where electrically charged particles, which may be ionized atoms or leptons, are accelerated, using electric fields and are then collided in experiments. With increasing speed, the kinetic energy of the accelerated particles increases. Once getting closer to the speed of light c, the particles energy increases further, due to special relativity

This chapter gives an overview over the LHC and its experiments, most importantly the ATLAS Experiment. The computational environment, the Grid, where the physics analyis is happening, is introduced in Chapter 3.

2.1. The Large Hadron Collider

Located at CERN, the Large Hadron Collider (LHC) [6] is currently the largest hadron collider of the world. It has a circumference of 27 km, a design centre-of-mass energy of $\sqrt{s}$ = 14 TeV and a luminosity of $10^{34} cm^{-2} s^{-1}$. The LHC is in operation since 2009 and has been built in the previously Large Electron-Positron Collider (LEP) tunnel using the already existing infrastructure. It accelerates protons and/or lead ions in two beams with opposite directions.

At four beam crossing points large experiments have been installed: ALICE, ATLAS, CMS and LHCb. While ATLAS and CMS are multi purpose detectors, LHCb is specially designed to measure b-physics and parameters of CP-violation. ALICE is designed to study lead-lead collisions and their remnants.

The LHC relies on the existing infrastructure at CERN and uses the older accelerators as pre-accelerators. The protons originate from one single bottle of compressed hydrogen gas. After release, the gas atoms are ionized and the protons are then accelerated to higher energy levels with each accelerator they pass. The LHC uses the beam output of the Super Proton Synchrotron (SPS), which has a beam energy of 450 GeV per proton. Figure 2.1 shows a schematic of the accelerator complex at CERN with the LHC and its pre acceleration chain.

The beams are bent on their circular route around the LHC by 1,232 super conducting dipole magnets. 392 quadrupole magnets are used to focus the beams. The magnets reach a field strength of up to 8 Tesla once the injected particles reach their maximal energy. When completely filled, the LHC contains 2,808 bunches with 1.15×10^{11} protons per bunch. This results in bunch crossings every 25 ns or a collision rate of 40 MHz. In 2012, at the end of Run 1, the LHC, reached an energy of 4 TeV per beam.

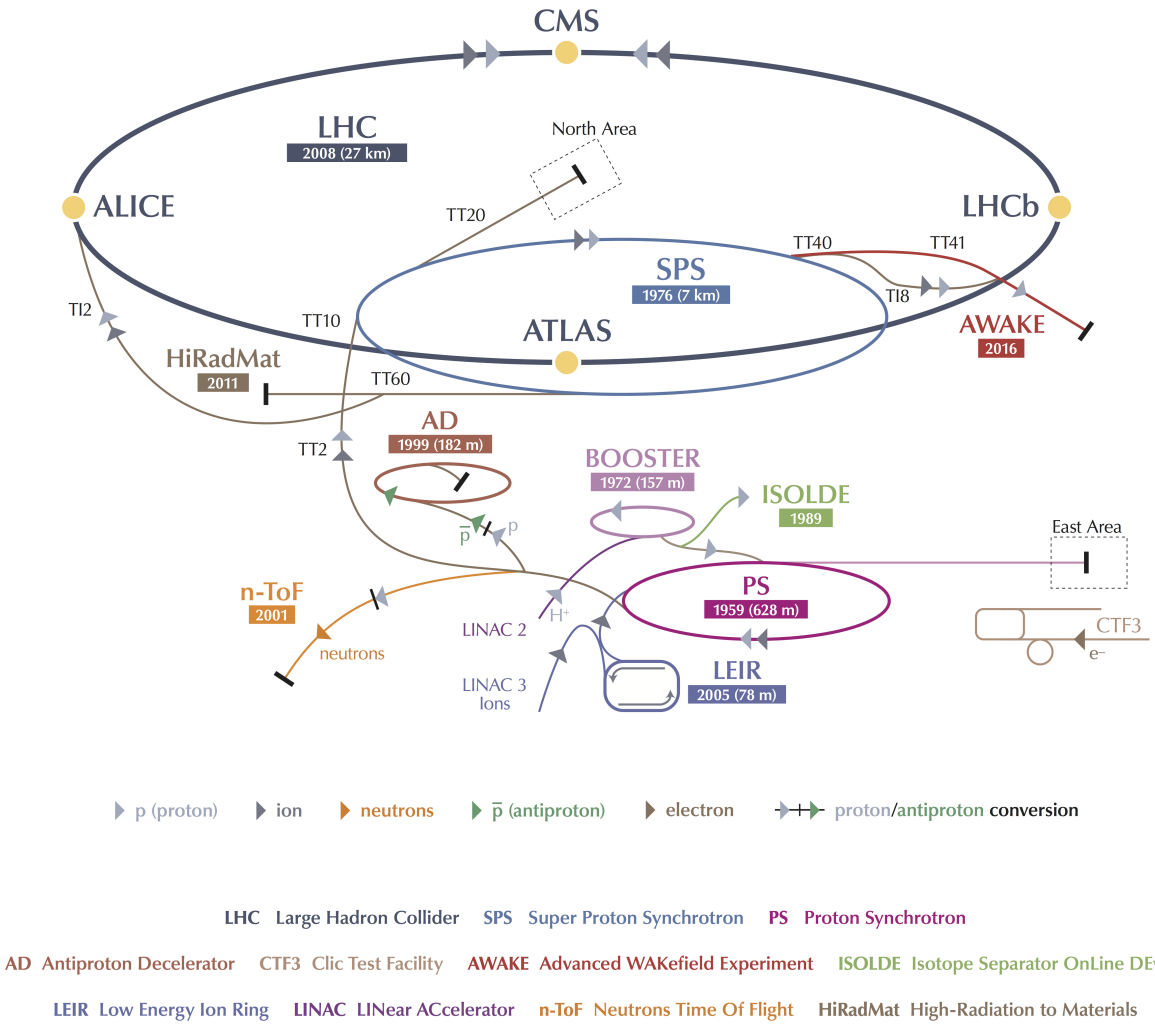


Figure 2.1.: Overview of LHC and its pre accelerators at CERN site [7].

2.2. The ATLAS Experiment

The ATLAS Experiment [9] is the largest multipurpose high energy particle physics detector ever built. It has a length of 44 m, a height of 25 m and a mass of about 7,000 t. A general layout of the detector is shown in Figure 2.2. ATLAS comprises two different types of magnet fields generated by super conducting coils: a solenoid and three toroids, two of these toroids for the endcaps and one for the barrel. Inside the central solenoid, the tracking detectors are installed. The magnetic field of the solenoid has a strength of 2 T, while the toroidal fields have a strength between 0.5 T and 1 T. The detector is homogeneous in azimuth around the beam pipe, as well as forward and backward symmetric.

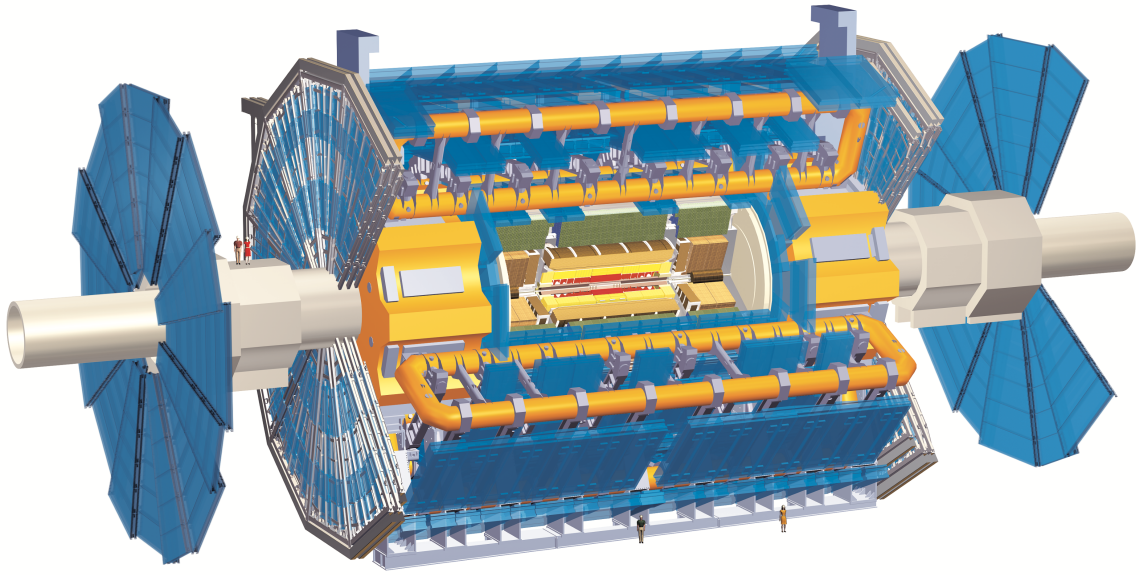


Figure 2.2.: ATLAS schema, showing the onion like layers of detectors and the magnet system around the central interaction point [8].

2.2.1. Detectors

ATLAS is a barrel shaped detector. The different subdetectors are arranged in onion like layers around the central interaction point. The innermost layers are built by the tracking detectors, followed by the calorimeters and outermost the muon detectors.

Tracking in general is done in the innermost subdetectors. Closest to the beam pipe is the silicon based pixel detector. With its 82,000,000 readout channels it provides the bulk of all output data. Built around the pixel detector is the silicon strip tracker, followed by the transition radiation tracker. Additional to the barrel layout, each tracking detector has also several disks in the end caps, which provide forward tracking. The tracking system provides tracking points for all charged particles. From a curvature, due to bending in the magnetic field, the charge and momentum can be measured. Particles can be associated to the primary vertex, meaning that these particles were generated by particle collision in the beam pipe. Or they can be classified as secondary verticies, which are decay products originating from particles of the primary vertex.

Outside the tracking detectors lie the calorimeters, consisting of the electromagnetic and the hadronic calorimeter systems. Both systems measure the energy of incoming particles by absorbing them in

high density materials. The emanating particle showers are sampled and the original particle energies can be inferred by summing up the energy of the shower. The inner layer is the electromagnetic liquid argon sampling calorimeter, which uses lead as an absorber material and provides a very high granularity. It consists of two half barrels with a small gap at the interaction point, orthogonal to the beam axis. At the endcaps there are two coaxial wheels. The electromagnetic calorimeter measures mostly particles that interact via the electromagnetic force like photons, electrons and muons[1]. The large amount of argon requires an active cryostat to cool the calorimeter and to keep the argon liquid.

Further outside lies the system of hadronic calorimeters, which is used to sample all particles, that have passed the electromagnetic calorimeter. These remaining particles are mostly hadrons and interact with the absorber material via the strong force. It is also a sampling calorimeter with scintillating tiles as active material and uses steel as absorber material.

The outermost detector of ATLAS is the muon system, which provides the majority of the volume of the ATLAS Experiment. Here, the muons are deflected by a toroidal magnetic field, which is generated by eight large, superconducting air-cored magnet coils. The muon system detects the momentum of muons, which are not stopped in the calorimeter systems. The detected particles are most likely muons, as these particles passed the hadronic calorimeter, and are therefore most likely not hadrons.

The described subdetectors allow ATLAS to measure almost all stable decay products, except neutrinos, as these do not interact with the detector matter. They manifest themselves only in missing transversal momentum, once one sums up all momentum vectors of all detected particles.

Figure 2.3 shows a partial cross section schematic of the ATLAS experiment and where the particles are registered. All curved lines in this figure show, that these particles are charged and their flight path is therefore bent by the magnetic fields. Muons are registered in the tracking system, the calorimeters and the outermost muon spectrometer while neutrinos are never registered. Hadrons are registered in the hadronic calorimeter and leave tracks in tracking system, should they be charged. Leptons are seen in the tracking systems and the electromagnetic calorimeter.

2.2.2. Trigger

With roughly 90,000,000 data readout channels, ATLAS generates a huge amount of raw data. It sums up to about 1.8 million PB per year, which is too much to be handled. This data is preprocessed by a three-level trigger system, which filters the raw event rate from 40 MHz[2] down to 400 Hz of interesting physics data. The small number of events, that are potentially "interesting", are filtered out, the rest is discarded.

The first trigger level (L1) is realized as a parallel cluster of special ASIC FPGA machines. These decide, in only 2.5 μs, based on a limited amount of data from the detector, mainly from the muon chambers and the calorimeters, whether the current event might be important. Only 100,000 events per second pass this first stage. The passed events get marked and the resulting data from the rest of the experiment is buffered and fed into the second level trigger (L2). Here, the complete event data is worked on by a computer farm of about 1,000 machines with common hardware. Further events are discarded, reducing the event rate to 3 kHz. In the last trigger level (L3) the event data is analysed

[1] However, muons are usually not stopped in the electromagnetic calorimeter.
[2] One bunch crossing every 25 ns.

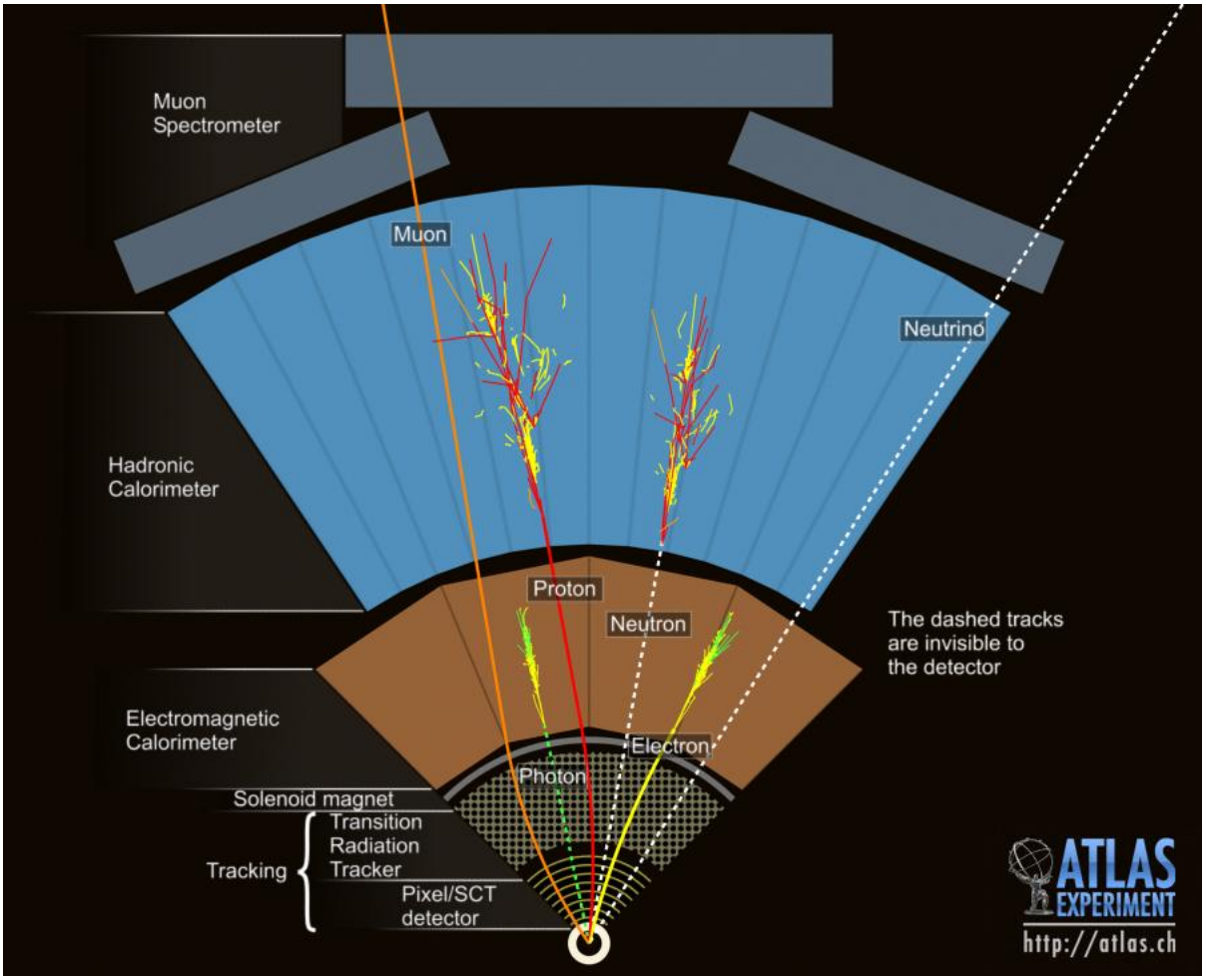


Figure 2.3.: Partial cross section of ATLAS, showing the reach and detection of various different particles [10].

based on physical properties like particle momentum and particle type. In a bigger computing farm of about 3,400 machines the event rate is reduced to its final 400 Hz.

The last instance are high level event builders, which operate in software alone and run in the Tier-0 data centre at CERN. They transform the raw data into a special data format suitable for physics analysis, while also further reducing overall data size. The originally recorded data is stored in its raw format on a Tier-0 based tape library for long term storage. With a final event rate of about 200 Hz the event format data stream still is approximately $320 MB/s$, which results in $3.2 PB/$year.

3. The LHC computing environment

In this chapter the computing environment of the LHC experiments is introduced. It is a necessary infrastructure for the analysis of the acquired physics data. The huge amount of data gathered by high energy particle physics experiments requires enormous computing power and storage capacity as the experimental data need to be analysed. Additionally the data has to be stored in an accessible way, so that it can be properly processed. Describing high energy elementary particle interactions and the detector simulation require an enormous amount of computing power and storage space to store the generated data.

3.1. Grid Computing

The computing environment at the LHC requires, as summarized by Ian Bird:

> "[...](a) to be able to manage very large data volumes at very high data rates; (b) to provide the requisite raw computing capacity – both CPU and storage; (c) to allow thousands of users to access the data; and (d) to provide long-term data archiving in a robust way. In addition, the environment must be able to manage organized data processing for real and simulated data, as well as for so-called chaotic user analysis." [11]

In addition to this, not all physicists of the large collaborations are regularly present at CERN. Therefore, a distributed computing infrastructure was conceived – the *Grid*, which is introduced in the following.

3.1.1. Definition

There are many definitions of Grid computing. However, Ian Foster, one of the original inventor of the Grid computing concept, classified a Grid as a computing system, that ...

- "... coordinates resources that are not subject to centralized control ..."
- "... using standard, open, general-purpose protocols and interfaces ..."
- "... to deliver nontrivial qualities of service." [12]

A Grid connects and coordinates resources and users, that are not necessarily in the same administrative domain. At the same time it provides authentication, authorization, security, monitoring resource access, resource brokering and accountability. A resource can be storage capacity or computing power. The protocols to do this should be as open and as general as possible to accommodate the widest possible range of users and resource sharing, while providing the afore mentioned functionalities. As

a result, a Grid should deliver the aggregated resources to a user in such a way, that an improvement in service quality is reached, either in terms of accessibility, computing throughput, security or response time. The overall utility of a Grid should be greater than the sum of its parts.

Like the power grid abstracts the consumption of power from its generation and transportation, a computing Grid should abstract the Grid user from the computing resources. It should not matter to a Grid user, where its computing job is worked on, but it should rather matter, that this is done in a reliable way.

3.1.2. Grid Services and Concepts

For any Grid, some central services and concepts are almost always necessary and these are summarized here shortly [13]. The software framework providing the following sets of services is called a Grid middleware. Common examples for Grid middleware software stacks are the Globus Toolkit [14], gLite [15] and UNICORE [16].

Computing Site A Computing Site in the Grid is usually a physically coherent collection of computers, providing most of the following services in the Grid. All computers on this site share the same network infrastructure and are often associated, in the case of the Worldwide LHC Computing Grid (WLCG), with a university or research facility.

Computing Element A Computing Element (CE) is a system providing computing resources. To the Grid, a CE announces several batch queues, in which jobs can be run. These queues are distinguished by the maximum runtime of jobs they accept or the amount of resources they provide for each job, for example the amount of RAM or the number of CPU cores. Each individual computer, able to handle one or more Grid jobs, is called a Worker Node (WN).

The individual queues are then managed by the local batch system, which is responsible for distributing the jobs to the local machines, depending on their load and physical properties.

Storage Element A Storage Element (SE) is a system providing long and short term data storage, which is accessible from within the Grid.

In high energy physics, as a general rule, each CE also has a long term SE, unless the computing site is quite small. If no SE is available locally, special solutions have to be developed to use another CE's SE, should the network costs be feasible. The data of a SE is usually accessible to the computers in the CE via local network, allowing fast access. Due to this, jobs, that require access to large files are often brokered to computing sites, where the data is locally available. A common SE software solution is dCache [17], which allows data access via Network File System (NFS) [18] and Web-based Distributed Authoring and Versioning (WebDAV) [19] and others.

Computing Job Computing is usually done in packets, which are called jobs. Each job has a description of the resources needed and also has a maximum lifetime. At CEs computing jobs are distributed to queues, which are handled by the computing sites local batch system.

Grid jobs themselves usually only communicate via so called file sandboxes: the input sandbox and the output sandbox, which both are limited in size. Job sandboxes are generally stored in short term

storage and do automatically expire after a short period of time. It it therefore necessary not to wait too long to inspect a jobs log files or results.

Upon job description the initial files for the input sandbox have to be specified. These files usually include the binary to run, some scripts to fetch data from the SE and some configuration data. The files for the output sandbox also need to be specified at the job definition time and tell the Grid middleware, which files need to be transferred back to the job submitter. With the limited size of the output sandbox, the job has to manage the proper storage of larger result files on a SE. Common files for the output sandbox are log files and small result files.

The duration of a job is limited by the maximum lifetime value of the queue it is scheduled to. After expiration, the computing job will be terminated by the computing resource provider. This is necessary to ensure, that faulty jobs do not run in infinite loops and block resources unnecessarily.

Virtual Organizations Important for any Grid, that shares resources between different entities, is the concept of a Virtual Organization (VO). A VO is a dynamic organizational grouping of resource users and / or resource providers, that agree on a common set of rules and conditions about resource sharing. In one Grid several VOs can share the resources according to established sharing agreements.

Security Security in the Grid is important and is handled inside a VO by employing a cryptographic certificate based infrastructure. Here, a user has to create a proxy certificate, that is submitted with the job into the Grid. It allows each entity, which has to handle the job, to verify, authenticate and authorize the jobs actions and to broker it appropriately. This prevents unauthorized use of Grid resources.

Job brokerage One of the most central services is job brokering, where a newly added job is registered at one of the broker services. These then distribute the overall computing load of the Grid by sending them to the most optimal computing resource.

Job brokers need to consider the expected time and computing cost of the job, free resources in the Grid and estimated data access patterns. In case of large datasets, that should be worked on, it is often useful to send the job to a physical computing centre, where a copy of the data is stored, rather than copy the dataset to the job, once it is running. Additionally, the central job broker needs to be aware of the kinds of resources available at the different computing sites.

As the job broker usually distributes the computing load in the Grid by directing computing jobs to free resources, this approach follows a "push" semantic.

Data distribution Generally, computing intensive tasks also require huge amounts of data, that need to be stored intelligently. Data distribution services in a Grid keep central catalogues of all available datasets, their metadata, location and cardinality[1].

Should the various storage elements of the Grid not be balanced properly, according to the internal metric of the Grid, data distribution services need to restore balance by copying or deleting data in certain locations. This often happens automatically.

[1] number of copies

Data Transfer Occasionally it might be necessary to transfer data from one SE to another. This is managed by the File Transfer Service (FTS) [20].

For these transports, transfer protocols, like GridFtp [14] exist, that notify central file catalogues about the transfers.

Accounting and Monitoring As several different VOs can use the same Grid, there normally is a necessity for accounting of provided and used resources. For this, central accounting services are responsible. Usually, the job brokers report to the accounting services, which jobs have been run where and what amount of resources they used. These central information databases can also be used to monitor the current status of the computing Grid, which is also necessary to detect problems early on.

3.2. Worldwide LHC Computing Grid

In 2005 the Worldwide LHC Computing Grid (WLCG) has been created for the LHC and its experiments, as originally described in a technical design report [21]. Together with the technical design reports of the various experiments at the LHC, the amount of data to be gathered and processed was predicted early on. As described in Section 3.2.1 these estimated data amount were used to predict the computing capacity needed to sufficiently analyse the LHC's experiments result. The WLCG was designed with the Grid Computing concepts described in Section 3.1.2.

3.2.1. MONARC

The simulation project, in which the general Grid topology for the WLCG has been simulated is called MONARC. From the predicted initial data rate of the detectors, transformation times for reconstruction were estimated, which are described for ATLAS in Section 3.4.

Several simulation tools were considered, but in the end it was decided to create an own, specially suited tool to properly capture all relevant Grid computing concepts. The simulation tool is Java based and was first tested to simulate the a server farm at the CERN computing centre. It abstracts several key components of the Grid:

Data model In the data model the client-server mechanism of centrally managed large scale and distributed data stores is simulated. This includes response time, based on file size, hardware load of several participating components and different policies for data storage management.

Multitasking Data Processing Model Here, the CPU, memory and I/O of a worker node are simulated for concurrently running jobs on the same machine. State changes in the usage of resources was simulated based on events, that the individual simulated jobs fire. Each time a simulation event occurs, the simulation is briefly interrupted and the resource consumption is recalculated for all jobs on the machine. These events can for example be the starting or ending of a job or a change in activity. At the beginning of each job, its CPU and memory consumption and its I/O behavior is planed. Prioritized resource usage is also possible. This model allows a realistic simulation of work load on several machines and the resulting job runtime depending on machine load.

Network Model The model allows both modeling of a sites local network connections as well as Wide Area Network (WAN) access. Similar to the previous data processing model the load on the available network infrastructure is simulated over time based on interrupts, which are released every time a state change happens to a party that uses the network. For the sites' internal network, no topology assumptions are made, but rather the general load on the backbone is calculated.

Arrival Patterns A flexible process was created, that allows stochastic scheduling of job arrival patterns on computing sites. To simulate different user groups for the Grid, these arrival patterns can contain simple Java code snippets, that simulate different job behaviours for different job types.

The final Phase 2 report describes the simulations as follows:

> "MONARC has successfully met its major milestones, and has fulfilled its basic goals, including:
>
> - identifying first round baseline Computing Models that could provide viable (and cost effective) solutions to meet the simulation, reconstruction and analysis needs of the LHC experiments.
> - providing a powerful (CPU and time efficient) simulation toolset that will enable further studies and optimisation of the Models,
> - providing guidelines for the configuration and services of Regional Centres, and
> - providing an effective forum where representatives of actual and candidate Regional Centres may meet and develop common strategies for LHC Computing." [22]

As a result of the conducted studies it was decided, that a tiered structure of computing centres was the best solution to satisfy the stated goal:

> "The primary motivation for this organisation is to maximise the intellectual contribution of physicists all over the world, without requiring their physical presence at the CERN. An architecture based on regional centres allows an organisation of computing tasks which permits physicists to analyse data effectively no matter where they are located." [22]

Three levels of Tiers were considered: the Tier-0 is located at the CERN computing centre, where the online data acquisition from the experiments happens and a primary copy of the data is archived to tape. Also the first stage of basic event reconstruction happens centrally at the Tier-0 as well as fractions of MC production and small amounts of analysis.

The next level are the Tier-1 computing centres, which, combined, roughly provide the same computing and storage resources as the Tier-0. They act as national or supra-national facilities, that provide data to the Tier-2 computing centres, where the bulk of user analysis is supposed to happen. The computing and storage resources were planned to be equally divided between the Tier-0, the Tier-1 centres and the Tier-2 centres in a ratio of 1:1:1. A topology of the tiered structure is shown in Figure 3.1.

The Tier-0 and each Tier-1 are connected with a dedicated 10 GB/s fibre optical link. Each Tier-1 is also connected with two other Tier-1 centres with dedicated 2 GB/s data links[2], giving them a star and

[2] Actual network interconnect bandwidth has increased from the planned values.

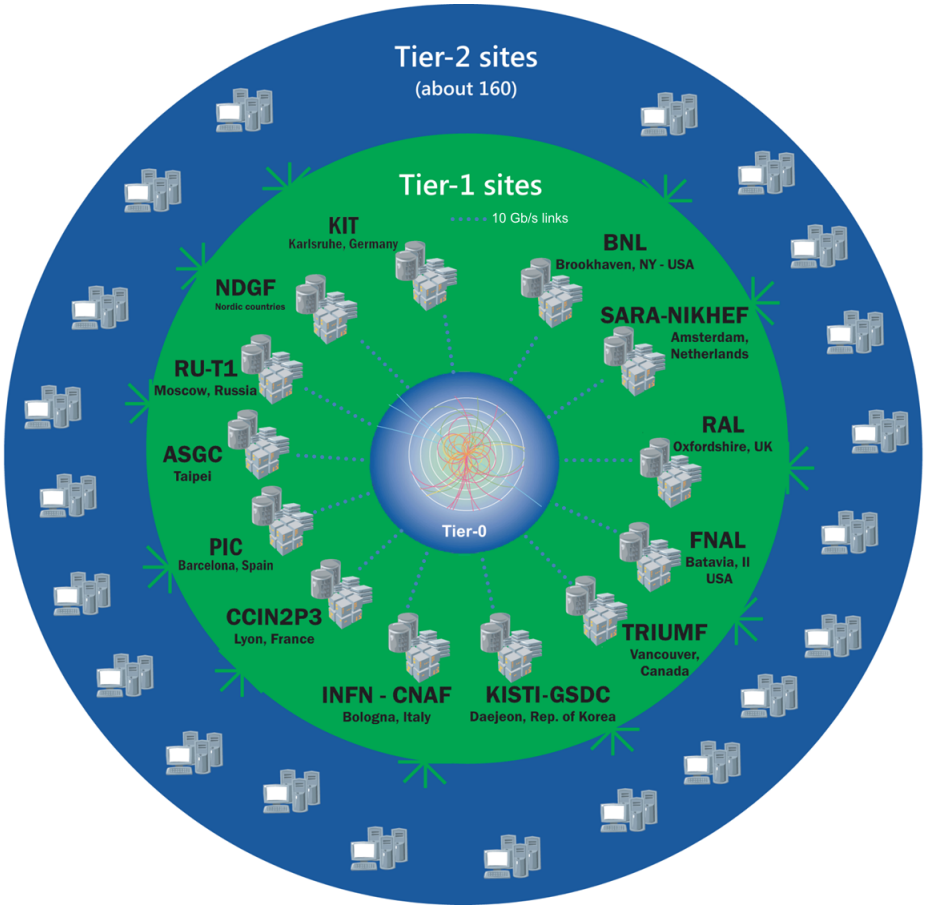


Figure 3.1.: Tier topology of the WLCG [24].

a ring network topology, with the Tier-0 at the center. This network is called the LHC Optical Private Network [23].

3.3. ATLAS Computing

As this thesis has been conducted in the computational environment of the ATLAS experiment, it shall be introduced in the following [25]. Most importantly, ATLAS uses its own infrastructure to manage the huge amount of data from the detector and from the simulation processes (see 3.4), namely PanDA, which is introduced in 3.3.2.

3.3.1. Tasks and jobs

In the ATLAS Grid context the mass production jobs (see Section 3.4) have a far bigger share of resource usage than analysis jobs. Efficiently handling and accounting for these amounts of jobs requires organizing the jobs in *tasks*, which are logical units in the production system.

Jobs belonging to a task usually share the same binaries and differ only in the individual input files

from a larger dataset or in a random seed parameter, that is needed for MC computation. The number of jobs in a task can be as low as ten per task, but huge tasks with more than 1000 jobs are also not uncommon.

3.3.2. PanDA

ATLAS uses its own Grid job submission and distribution system called Production and Distributed Analysis (PanDA) [26]. It has been developed as a data driven workload management system, that is scalable at an LHC experiments data processing dimension. PanDA is realized as a centrally managed, prioritized task and job queue, using the CERN Oracle cluster as its main back end storage database. The middleware is realized in Python and several Apache web servers function as HTTP front ends, that allow other systems to interface with it. The system is overall highly automated to reduce manpower and maintenance.

PanDA is tightly integrated in the ATLAS Data Distribution Management (DDM) system and schedules the Grid jobs in such a way, that the necessary input data has to be available locally on the Grid site. This omits waiting times, as a job does not request a file transfer from a remote site to the local SE before it can start. For mass production (see: 3.4) PanDA is connected to the production database and automatically fetches job and task descriptions from this system.

If compilation of source code for any of the jobs of the current task or user workset is required, PanDA will first schedule a so called build job. This build job simply compiles a given source code resource and, if everything has been built successfully, stores the created binaries on a SE. The PanDA server then signals, that the rest of the jobs are cleared for scheduling and execution. Build jobs usually have one of the highest priorities, as they do not require much CPU and storage resources and are a prerequisite for the further jobs of this task.

Pilots PanDA introduced the concept of pilot jobs, which are pre stage job wrappers using traditional Grid job transmission methods to be submitted to the Grid, which after initialization is done, request job metadata from the central server.

The pilots are continuously submitted into the Grid by so called pilot factories [28], that abstract the specific Grid middleware flavors in use throughout the WLCG. As these pilots have virtually no required resources they get evenly brokered to all computing elements. Pilot factories are stationed at CERN and some instances are also distributed throughout the WLCG, for example at the Tier-2 computing center in Wuppertal, to provide fault tolerance.

Once a pilot runs, it performs various tests to make sure what kind of local resources are available and that these are functioning within proper parameters. After that, the PanDA central database server is queried for a new job matching the local resources. All data necessary to run the job is then transmitted to the pilot, which starts to set up the job by downloading the needed datasets from the local SE and by setting up the specified ATLAS analysis software cache.

After all this is done, the payload job is executed and the pilot waits for it to finish. Occasionally the pilot notifies the central PanDA server, that it is still alive and therefore, the payload job can also assumed to be still running. A schematic of the PanDA workflow can be found in Figure 3.2.

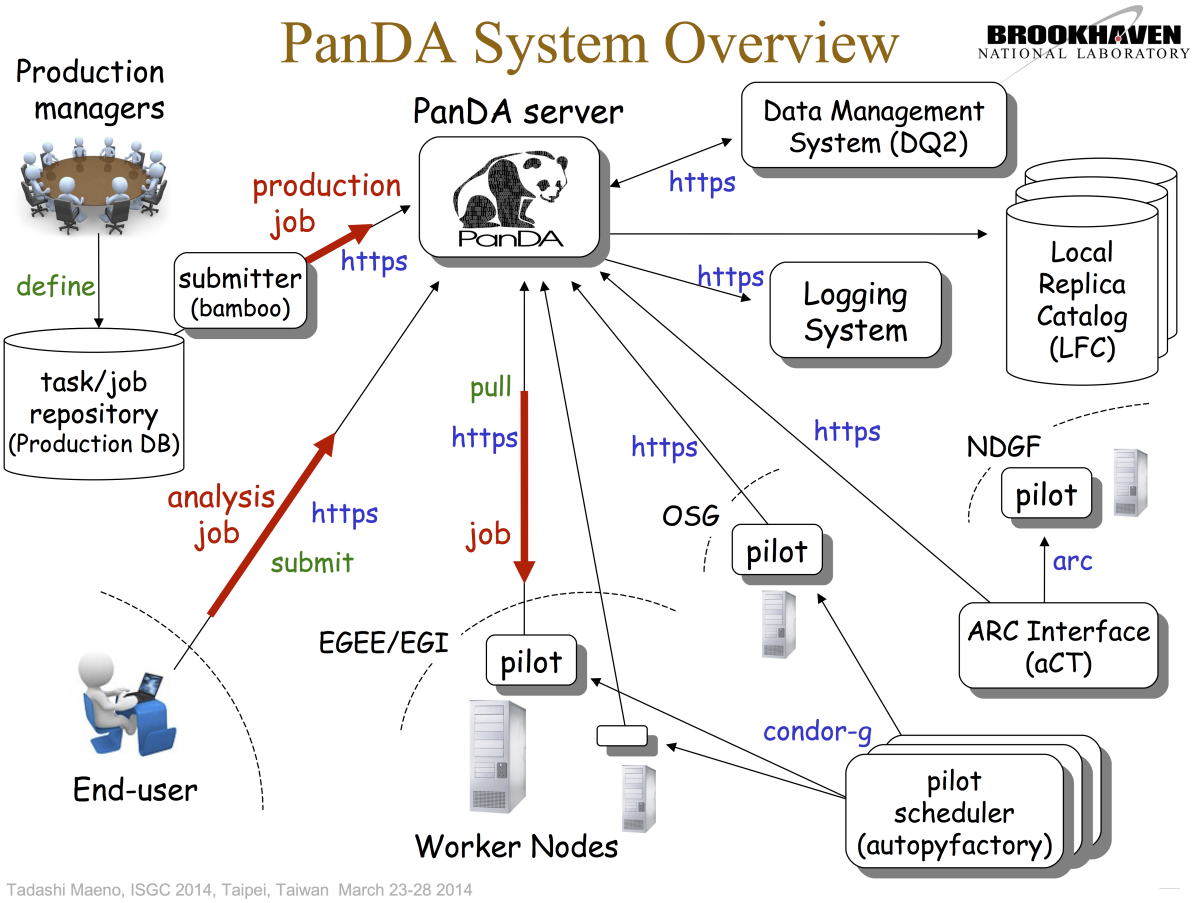


Figure 3.2.: PanDA job distribution work flow [27].

3.3.3. Distributed Data Management

In ATLAS, distributed data management in the Grid is based on Don Quijote 2 (DQ2) [29]. Here, files are organized and registered in datasets, which are themselves organized in data containers, that exist in Space Tokens, which are reserved spaces on one SE, which are associated with a particular working group inside ATLAS.

Files are the smallest unit of DQ2, but only whole datasets can be transferred between different sites. On top of these datasets stands the concept of data containers, which can contain more than one dataset and provide a way to logically group several datasets.

The file catalogue database, called LCG File Catalogue (LFC) is a database located at the CERN computing centre and contains a mapping of logical filenames to their physical path(s). The LFC is queried to find the exact location of one file before it can be accessed.

DQ2 has been replaced by Rucio [30] in 2014, as DQ2 experienced scaling problems[3] and the system structure proved too inflexible to allow further developments [31]. Rucio scales better and is more streamlined to actual ATLAS file access patterns. It integrated seamlessly into the existing client

[3] particularly due to unused file versioning

infrastructure and most users were able to switch systems without troubles.

3.3.4. Monitoring

For a large computational system like ATLAS computing, a wide range of monitoring and internal accounting tools is necessary to control and steer the flow of jobs and tasks in the Grid. Central to most of these monitoring tools is the PanDA database as reference data, which provides accounting data about finished and running jobs in the Grid. Most monitoring dashboards regularly pull the data they need from the PanDA database to store them in a relation that is more favourable to their data mining model. They then use their own, optimized data schemas for analysis and presentation.

Examples for that are the PandaMonitor [32], which monitors all current jobs, or the historical dashboard [33], which allows to query job info of already finished jobs. The historical dashboard, for example, can be used to derive data about job failure rates and therefore to calculate efficiencies of resource usage. The pilot factories as integral part of the ATLAS computing efforts are centrally monitored [34].

Another integral system to any ATLAS Grid operation is the data distribution and its management. Data flow between Grid sites is monitored as well as the necessary distribution of data replicas, especially the frequently used data sets, that are needed for ongoing analysis efforts.

The Site Status Board (SSB) [35] and ATLAS Grid Information System (AGIS) [36] store metadata about all Grid endpoints relevant to ATLAS, their pledged resources, their actually available resources and their reliability. These endpoints include the CEs at the sites, their batch queues and their availability as well as the SEs at the sites. For all these endpoints the theoretically available pledged amount is stored as well to later compare the actually provided and used resources against the pledge.

3.3.5. Athena

Athena [37] is a software framework derived from the Gaudi project [38], which was originally developed by the LHCb collaboration. It provides users with input and output services for different data formats and allows user analysis code to loop over input data in an event wise way, while applying several different analysis algorithms and filters. Athena can be scripted using Python, which makes it highly configurable and allows different behaviours without recompilation. One example analysis software is HepMCAnalysis [39], which is used in Part IV and its use is explained there.

Around the central Athena framework, ATLAS uses job transforms [40], which are able to automatically configure Athena and chain several production steps after each other while keeping track of event counts, storing the proper metadata and configuring everything appropriately.

3.4. Mass production in ATLAS

For the ATLAS experiment, aside from the user analysis, two large groups of automated mass production computing need to be distinguished: online and offline experimental data reconstruction and MC simulations.

The server, which handles mass production requests, is closely connected to the PanDA-server. New production requests are automatically processed in such a way, that the appropriate task and job

descriptions are pushed to the central Production and Distributed Analysis (PanDA) database, from where the jobs are pushed further into the Grid.

3.4.1. Reconstruction

The live experimental data from the detector trigger farm (see 2.2.2) is available in a bytestream format called RAW and needs to be stored and transformed, before any physics analysis can be done. After the RAW files are stored as a primary copy at the Tier-0 tape library and backup copies are distributed over all Tier-1 data centres, the RAW data is transformed into more sophisticated data formats.

First, the raw binary bytestream from the detector is transformed into the Event Summary Data (ESD) format, which is, with its ca. 500kB/event, about a factor of three smaller than the original RAW format, which has a size of 1.6MB/event. The ESD data format is smaller, than the previous one, as the binary RAW data is replaced with reconstructed physics objects, that allow particle identification, track re-fitting, jet calibration etc.

In a second step, the ESDs are then transformed into the Analysis Object Data (AOD) format, which is with only 100kB/event again about a factor of five smaller. AOD data contains further information about the event and its participating particles, replacing the original data about the individual sensors, where these were originally registered. They also contain a summary of reconstruction data, which is detailed enough to support typical analysis action[4]. They are sufficient for user analysis and several physics analysis groups use their own style of AOD files, which suits their analysis better.

For each AOD, that is produced, so called TAG files are filled. These files contain metadata about several event classes, making it easier to select large amounts of special events later on, that are distributed over several AOD files. TAG files contain data at the rate of ca. 1kB/event.

Some analysis groups transform the data further into their own Derived Physics Data (DPD) formats, which are easier to use in terms of producing histograms from the input data or doing quick statistical analysis.

3.4.2. Monte Carlo production

With the design and construction of a detector, detector models are created to simulate and understand the expected behaviour, even before the machine is built. Here the particle collision events are simulated and further on, the interaction of these particles with detector material is simulated. These are called MC calculations, referring to the same named casino, as the necessary calculations require a huge amount of random numbers to be generated.

Dataset IDs For MC production, each physics configuration has been assigned an identification code [41], the dataset identifier (dataSetID), to better store and reference them. DataSetIds distinguish physics configurations by their physics process, main particle decays, particle density functions, theoretical corrections and the generator software used.

Logically attached to each dataSetID is a describing dataset name, which is also found in the task name of the output dataset, and a configuration file for the generator software. This configuration file describes parameters of the physics process to the generator software.

[4] For example re-evaluation of calorimeter cluster positions or track refitting, among others.

Table 3.1.: Software tag types.

Software tag name	description
eTag	event generation
sTag	detector simulation
rTag	reconstruction
dTag	digitization
mTag	dataset merging

Different generator software packages require differently formatted configuration files. These are maintained and updated on different time scales and produce, due to specialization, slightly different results. Therefore different DATASETIDs are used to distinguish between them.

ATLAS software tagging system To have stable MC production, ATLAS software releases are tagged. Each tag contains information about the underlying individual software packgage version being used and fixes basic configuration file collections of these individual software packages. Tags are named for the type of software stage they describe (see Table 3.1).

Each MC process has an associated software tag, indicating the software version, that was used. Datasets, which are generated by non event generation software, are described by a chained string of tags, so that the whole software version chain, that was involved in creation of the dataset, can be understood and reproduced if necessary. Administrative data about the different tags and datasets is stored in the Atlas Metadata Information System (AMI) [42].

As a DATASETID describes the physics process and the configuration to be used and the software tag the exact version, datasets generated with the same set of DATASETID and software tag are basically the same, with respect to randomized input values. With enough statistics, the results of analyses based on different files with the same DATASETID and software tag combination become virtually indistinguishable.

Event generation The first stage of MC production is the event generation, where the pure physics process is simulated without interaction with other matter.

For this, several generators like PYTHIA [43], Sherpa [44] or AlpGen [45] exist, which all specialize in different physics processes or have different theoretical corrections included in their internal models. Important input parameters into this stage are the centre-of-mass energy of the collider, which is to be used, the initial particles of the reaction[5], an initial random generator seed from which all further seeds can be generated deterministically, the desired physics process to be simulated and various other parameters about the particle beam.

As all particle interactions are described by various probability distributions and complex field equations, the random generator is used heavily to distribute all interaction results according to the specified distributions. Not all possible particle reactions are considered per individual MC production,

[5] In case of the LHC's current use cases it would be either two protons, two lead ions or a proton and a lead ion.

but a small subset, for example W-boson or $t\bar{t}$ pair production and their different decay modes to later study on.

Additionally, for each event, a so called Pile Up can be added, which are additional low energy collisions, that happened in parallel with the original event, as more than two initial hadrons collided from each particle bunch. Pile Up consideration is important as with increasing luminosity more and more parallel events are happening, which make the isolation of the one interesting event more difficult.

Detector simulation In the detector simulation stage the physical properties of the existing detector are modeled as closely as possible using the particle-matter-interaction modeling software Geant4 [46]. It is configured with a very fine granular model of the detector and its physical properties. Each part of the detector is represented in small, simple geometrical objects like cylinders or cubes with appropriate material properties. For each of those objects the material, density, electric current and magnetic flux of and cross section between different particles is stored.

These jobs require huge amounts of memory as large parts of the detector geometry need to be kept in memory. Also the energy, momentum and particle balance for each particle interaction with detector matter has to be accounted for. They are therefore quite expensive to compute as the particle accounting takes lots of time and the higher memory footprint depletes available memory for other jobs on the same WN. For some physics analysis use cases it is enough to skip this step and simply use the generated events at parton level.

Digitization After the detector simulation the actual hardware reaction inside each sensor of the detector is simulated. If a simulated particle travels through a sensor the probability of its interaction with the sensor material is calculated, as well as the sensors response. The resulting data is in the form of RAW datasets, which are the same format as actual data from the detector itself.

Reconstruction In the last MC stage, the results of the digitization stage are handled as if they were RAW data from the detector. They then can be pushed through the same chain, that is described in 3.4.1 to compare them against real data. As the originally simulated particles are still known, the solution granularity and overall particle detection efficiency of the detector can be calculated by comparing the reconstructed data from MC versus reconstructed actual physics data.

Fast Simulation Instead of full and expensive detector simulation, ATLAS can also use Fast Simulation [47], taking event generator output data and applying general smearing functions to some particles to simulate a faster version of detector simulation and reconstruction. This process is not as precise as a full detector simulation, but a lot faster and therefore saves a lot of computing resources.

3.5. Evolution of ATLAS Grid Computing

Over the last decade, Grid computing in the ATLAS context had to evolve and adapt to changing hardware, like the introduction of 64bit CPU architectures. With increased luminosity the average count of collision events per bunch crossing increased. This increased Pile Up proved to be problematic, for example due to problems in track fitting where combinatorial problems need to be solved which

scale worse than linear with the number of hits in the detector. This section shows a few projects, that break with the original design paradigms discussed in 3.2.1 and improve various aspects of ATLAS Grid computing.

3.5.1. Multi core jobs

Over the last decade, the core count per CPU increased to deliver more computing power per chip. This presents a problem as peripheral infrastructure like RAM or network bandwidth did not necessarily increase as well, or at least not in terms of price performance. This lead to the fact, that the amount of available RAM per computing core is decreasing, due to cost effects. This presents a problem for working memory intensive computing like detector simulation.

ATLAS has started development efforts to adapt its core software package Athena (see 3.3.5) to share common data in RAM, which then allows multiple processes to have a significantly smaller memory footprint. Multi core jobs have to be assigned to special multi core queues, both by PanDA and by the underlying batch system. If the resources for these queues are dynamically handled, it is possible, that multi core jobs have to wait until all desired cores are free. One long running job, blocking one of the cores, can lead to large waiting times and therefore to significantly decreased core occupation rate, which is inefficient. Therefore, intelligent CPU resource allocation is necessary to prevent these inefficiencies.

3.5.2. FAX

As explained, PanDA job brokering ensures, that jobs are only sent to sites, where all necessary input data is locally available. Should this information be incorrect or the local SE has a problem to deliver the input data to the WN, the pilot tries a second time and then fails the job.

Federated ATLAS XRootD (FAX) [48] is an ATLAS internal adoption of XRootD[6] and allows the file to be obtained from somewhere else, should FAX be able to get a proper handle on the file. With FAX, it will be possible to gain access to more CPUs from opportunistic resources like small computing centres without a local SE but sufficiently broad network connection.

3.5.3. Event Service

The Event Service (ES) [50] is a new form to efficiently and quickly distribute event data to running Grid jobs by providing event level granularity access to input data. It is motivated by having a way to handle short lived computing resources efficiently. These resources can be free slots at high performance computing centres, spot market commercial clouds or volunteer computing[7].

The ES handles data flows, data storage, monitoring and bookkeeping at event level granularity. Remote data repositories are used for input data flows, without the need to locally pre stage the whole dataset, therefore reducing the local storage footprint, but on the other hand relying on a powerful network connection. The output data is pushed continuously to remote object stores, further reducing the local storage footprint. Also, at most one event computation is lost, should the computing resource suddenly be unavailable again.

[6] XRootD is a "high performance, scalable fault tolerant access to data repositories" [49].
[7] See ATLAS@home [51]

Part II.

JEM

Introduction

After introducing the physics background for the LHC, its experiments and the technical aspects of Grid computing in Part I, the Job Execution Monitor (JEM) and its environment, in which the main projects of this thesis have been conducted are introduced in this second part.

Chapter 4 provides an overview of the general architecture of JEM, its development history and its main use cases. In the following chapter the recent improvements in the project, since its last major discussion in [52], are introduced.

As a job execution monitor, that technically runs as the user jobs parent, by wrapping it as a subprocess, JEM can employ a wide variety of possible tools to gather measures about the payload job and its behaviour. Additionally JEM can execute its own subprocesses to analyse the results of the payload job, which enables some central features for the Live Monte Carlo Validation, described in Part IV.

Together with a collection of services, running at a central location, the JEMserver, the JEM monitoring functionalities can be controlled, the monitoring data can be received, stored, analysed and displayed.

Why is JEM necessary?

As described in Chapter 3, the Grid has been designed in such a way, that it is essentially an non-transparent black box to a Grid user with respect to the current internal status of their jobs. The only feedback channels available to a Grid user were limited to the job exit code, log files contained in the output sandbox or self implemented methods of sending debug or state information of the current job back home. Most of this job status data is only available, once the job has finished and the physics results and its log files have been written to the Grid storage system or to the jobs output sandbox.

Starting from a D-Grid initiative [53], a Job Monitoring System (JMS) was designed and implemented. This project later became JEM and has evolved since (see Section 4.3). The projects goal was to provide the Grid user with additional channels to receive monitoring data from running jobs and to provide live data channels that are not limited to the previously mentioned ones.

Therefore, first JMS and later on JEM, tried to provide Grid users with tools to deeper understand the progress of their running jobs and to shed some light into the black box, that was the original concept of a Grid job.

4. JEM overview

This chapter provides an architectural overview (see Section 4.1) of all central components of the JEM project and how they interact when applied to the different use cases presented in Section 4.2. JEM has been in development from 2005 on, its development history is described in detail in Section 4.3, and has constantly evolved over time to meet changing performance and usage criteria.

With the growth of the JEM project a nomenclature problem arose: while JEM generally means the whole project it also means the main program `JEM.PY`, which runs one of its working modes (see 4.1.1). The JEMSERVER generally refers to the central and dedicated machine, which hosts all JEM services, necessary to provide the project with its infrastructure.

Since the last major documentation of JEM in [52] several additions have been made and new features have been implemented. These are presented in the following chapter.

With the numerous improvements of the last years (see Section 4.3), JEM is now able to provide real time monitoring data on various aspects of the user's job execution and the hardware it is running on. This monitoring data can now also be displayed to users without access to special JEM UI software, that interprets the custom binary data format JEM is using. It is rendered using dynamic web pages containing graphic representations of the jobs monitoring data and is explained in 5.1. This data allows the user to inspect and understand job behavior, especially in case of errors or problems during the job run time and allows the user to react in time to save resources.

4.1. JEM architecture

In this section, an overview of the different JEM modules, systems and working modes is given. The JEM systems and modules of the JEM ecosystem are basically grouped into four categories:

- JEM working modes (further details in 4.1.1)
- JEMSERVER processes (further details in 4.1.2)
- JEM services (further details in 4.1.2)
- PANDA pilot extension (further details in Chapter 11)

Figure 4.1 shows a summary of all currently relevant systems and modules, while older modules, that have been deprecated or are currently not in active use, have been left out, as to not congest the overview. It should be noted, that the boundaries of these categories are not always clear and do overlap, however, clarifications are provided in the following subsections.

JEM working modes are explained in detail in 4.1.1 and the PANDA pilot extension module, as a central part of JEM's ACTIVATIONSERVICE is explained in Section 11.2 as it is a central part of the Live Monte Carlo Validation.

The category JEM services contains the various regularly scheduled tasks, which are being run by the JEM ServicesWorker. It is explained in detail in 4.1.1. The services are themselves grouped by their main purpose inside the JEM ecosystem, while all tasks listed under "various", provide database maintaining or monitoring data post processing functionalities. The pilot module is described in Chapter 11, as it is a central part of the ActivationService.

4.1.1. Working Modes

In 2010, the concept of working modes has been introduced to the JEM code base, where now a central `JEM.py` module is the single entry point to almost all of JEM's main processes. It provides a central, common infrastructure for all its different working modes:

- JEM environment validation
- Bootstrapping
- Save binary module import[1]
- Central configuration management
- Central logging setup

JEM has been designed as a WN Grid job wrapper so that the worker node mode and the driving `JEM.py` base module are tightly integrated. For future development, this tight coupling would need to be loosened to have better maintainability and more simple structures for developing further features. While there are more, the most commonly used working modes are explained in this chapter. The ActSvcRuleEvaluater working mode, as central part of the ActivationService, is described in detail in Section 11.5.

WN and UI

The WN working mode is the remote part of the central JEM use case. It is the part of JEM, which runs wrapped around the payload job, in the Grid.

The UI is a graphical front end module, which is able to show graphical representations of received monitoring data, either live, or post mortem from either a special ring buffer dump file or a database dump of recorded events. Both are described in [52] together with the shared memory for fast IPC. The trigger and valve architectures for filtering and transmitting monitoring data are also described in [52].

The WN mode uses a wide variety of monitors to inspect the machine and the payload job in regular time intervals. These are necessary to gather all relevant data on the worker node and transmit it for further evaluation to a central JEMserver, where the monitoring data is stored, analysed and finally presented.

Several additions, in particular new trigger modules (see Part III), new data monitors (see also Part III) and the MC validation handling (see Part IV) have enhanced the WN working mode since its last description in [52].

[1] Binaries can be loaded via HTTP from a foreign location, if they are not available locally.

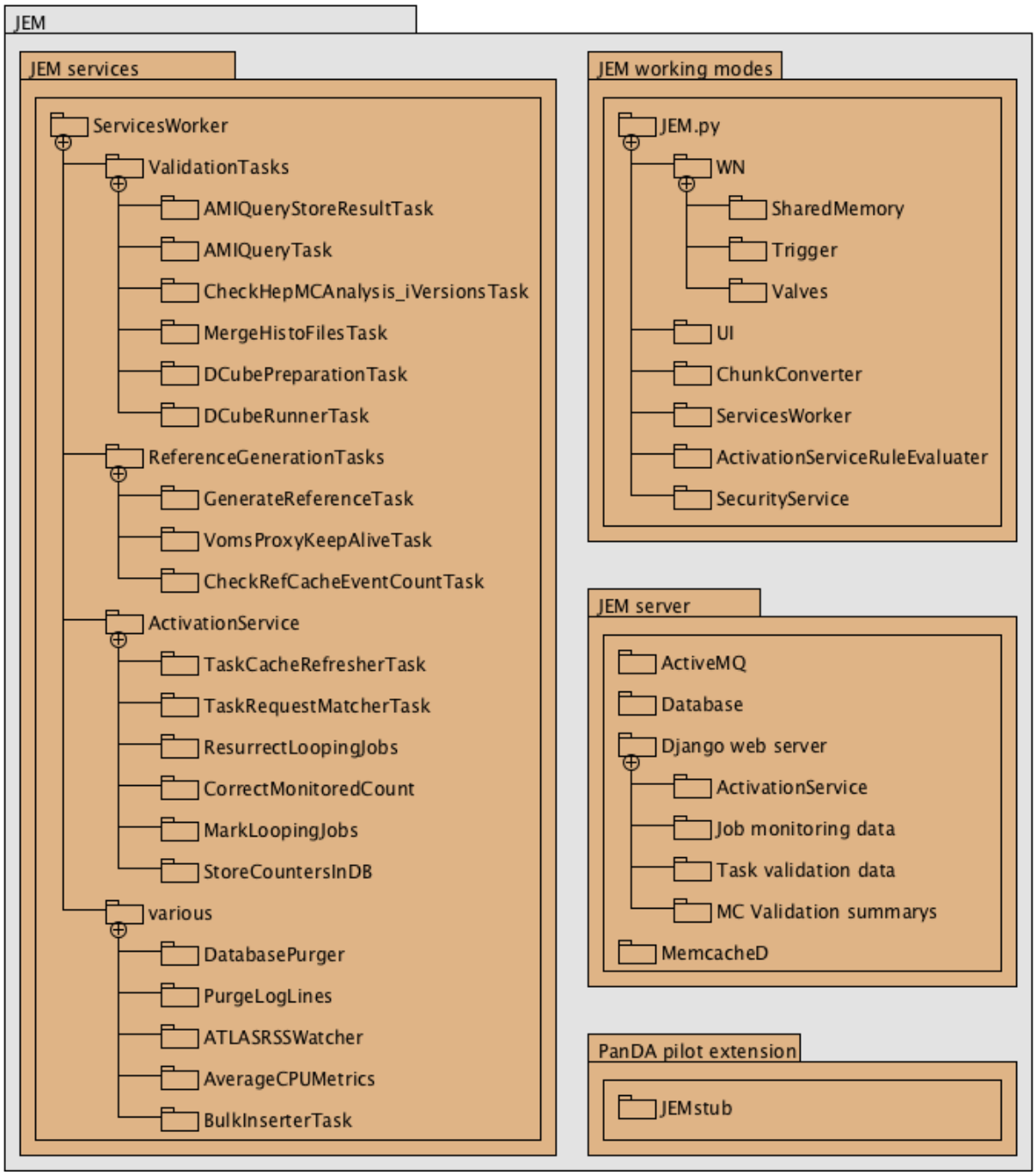


Figure 4.1.: JEM systems and modules overview.

ChunkConverter

To access and visualize live job monitoring data, with a JEM UI instance or with the Django [54] web site, monitoring data needs to be transmitted from the worker node instances running in the Grid to their respective destinations. The preferred method to do this in JEM is to use a central ActiveMQ messaging server located on the JEMserver to collect and distribute monitoring data which is received using the Streaming Text Oriented Messaging Protocol (STOMP) [55] protocol.

In the Grid, on the worker node, the *StompValve* is responsible for sending any data chunks, which have been cleared for transmission via STOMP by, e.g. the *ApproveSelectedTrigger*, assuming, that the *StompValve* is configured appropriately. Per default the ActiveMQ instance running at the JEMserver is selected and once cleared for transmission, all data chunks are forwarded in their binary format to all the configured ActiveMQ servers.

On the JEMserver a special working mode, the *ChunkConverter*, is running, which regularly checks for newly arriving chunks at the ActiveMQ server, translates them from their binary format, and finally stores them in the Django object related database. Additionally some metadata about the gathered job data is generated and is stored as well. This includes internal accounting information about the amount of running jobs per user, per site and per cloud.

Before the change to the database layout and web monitoring data presentation based on Django, this "stomp to database" translation was executed by a working mode called *Spy*, which also rendered the first version of the job monitoring overview pages from the old database. With the new database layout and Django's vastly improved capabilities of building and maintaining web pages and data presentations, the old monitoring overview pages were removed.

ServicesWorker

For recurring tasks a ServicesWorker working mode has been added, which periodically launches different tasks. These tasks are derived from a base *Task* class, which provides all necessary functionality for time scheduling, command line configuration management and IPC via POSIX socket.

Upon start the tasks are dynamically loaded from a *–taskList* command line parameter, containing a double comma („,") separated list of the task names and their configuration string. The configuration string for each task is passed as a JSON [56] dictionary, which is evaluated during initialization and the arguments are then passed to each task. The JSON representation allows several parameter values in several different data types to be securely represented on the command line. The task then sets its own run interval upon initialization, so that the ServicesWorker knows how often to call the tasks `run` method and execute the tasks functionality.

Each ServicesWorker task is responsible to set up special logger handling. This is necessary if Django based database access is needed. The Django logger set up does not work nicely with the JEM logging infrastructure because Django provides its own logging configuration that can cause problems with the one used by JEM.

As several ServicesWorker are running at the JEMserver, the tasks are usually grouped for their respective system, e.g. the ActivationService, reference generation or validation evaluation (see figure Figure 4.1). This also allows easy distribution of resource intensive tasks to several different machines, as there is no restriction on the number of individual tasks that are allowed to be run simultaneously.

4.1.2. JEMserver and its services

The JEMserver is, in the context of this thesis, the logical collection of all essential services, that need to be centrally available to support the different JEM use cases, as can be seen in Figure 4.1. The JEMserver acts as a central access point for all collected job monitoring data, where the received data is also stored and processed. Currently this is only one physical machine.

The various JEM services and the essential ServicesWorker tasks are described in the following subsections. However, the database layout, the Memcachd [57] and the various data presentations provided by Django are detailed later in Chapter 5.

ActiveMQ server

The ActiveMQ [58] server provides reliable handling of multiple connections and an organized access to received data using topics, which are created per monitored job. Hence a UI instance and the JEMserver can access monitoring data reliably and simultaneously.

For using live monitoring, data has to be transferred from a JEM monitoring instance, running on a WN in the Grid, to the monitoring user who submitted the job. As main method of transport of monitoring data from the payload job, running in the Grid, STOMP support has been added to JEM. The previously used R-GMA [59] and MonALISA [60] methods did not provide suitable throughput, speed, reliability and stability and are not used further.

STOMP messages are being received by an ActiveMQ messaging server running at the central JEMserver, from which they are translated into the monitoring database by the ChunkConverter (see 4.1.1).

MySQL database

The database, currently implemented in MySQL [61], acts as back-end to the Django monitoring data handling and keeps track of the different validation activities, as explained later in Chapter 12. Its data is replicated automatically to a second MySQL server to prevent data loss by redundancy in case of hardware failure.

MarkLoopingJobsTask

This task periodically checks all monitored jobs, marked as "running", if the `last_seen` entry in the database is not older than a given threshold. The `last_seen` field is updated every time new information about the job is received, be it either by heartbeat messages or by monitoring data. If the time difference is exceeds the configured threshold, the job is marked as "looping". This indicates, that either a job end event has been missed or that the job has crashed. It can also indicate, that there was a network connectivity problem, which resulted in missing monitoring data.

ResurrectLoopingJobTask

Accompanying the previous mentioned task, this task changes the ‚JEM internal, status of monitored jobs in the status "looping" back into "running" once new monitoring data arrives. Jobs, which have been in the "looping" state too long, get set to "finished" with a special error code indicating that the

original error code is not known. The looping condition is directly displayed on the job monitoring overview web page and therefore gives the use a direct feedback about their job status.

DatabasePurger

The *DatabasePurger* has been created to delete old monitoring data should the accumulated size of the database reach a limit. As the resources at the JEMserver are limited, it is necessary to keep a minimum amount of free disk space at the server to guarantee stable operation.

4.2. JEM Use Cases

Initially JEM was designed as a monitoring tool for individual Grid job users to better understand job behaviour in an otherwise closed environment. The log files in the output sandbox were the only monitoring information a user could get, besides the exit code, being transmitted by the Grid system. Often, these log files were only available if the local batch system of the site, the particular job was running on, was able to store them properly. With high job failure rates that could reach 10% for relatively unsupervised analysis jobs in the early years of productive Grid operations, the necessity for job monitoring arose and generated the Grid user use case (see 4.2.1). Even today, Grid job failure rates remain high, as can be seen in Figure 4.2.

With additional monitoring functionality and the possibility to communicate data efficiently to a central server, the versatility of JEM grew and is now suited as a multi purpose tool for a wider variety of monitoring purposes. In the following subsections user groups are described, which can benefit from adopting aspects of JEM functionality into their workflows.

4.2.1. Grid users

As the normal Grid analysis user was the main motivation behind JEM development, almost all the features available to the worker node module of JEM reflect this in the monitoring functionalities they provide:

- Script monitors for *Python* and *Bash* with line by line execution traces
- In depth, line by line, binary execution monitoring, as long as the binary is appropriately prepared with debug symbols
- A system monitor for various WN system metrics like CPU, memory, disk and network usage
- Live log file parsing and complex string pattern matching
- Live log file transmission
- Binary execution tracing using GNU Debugger (GDB) [62]
- User job memory inspection using GDB
- Secure back channel for communication and control of the user job

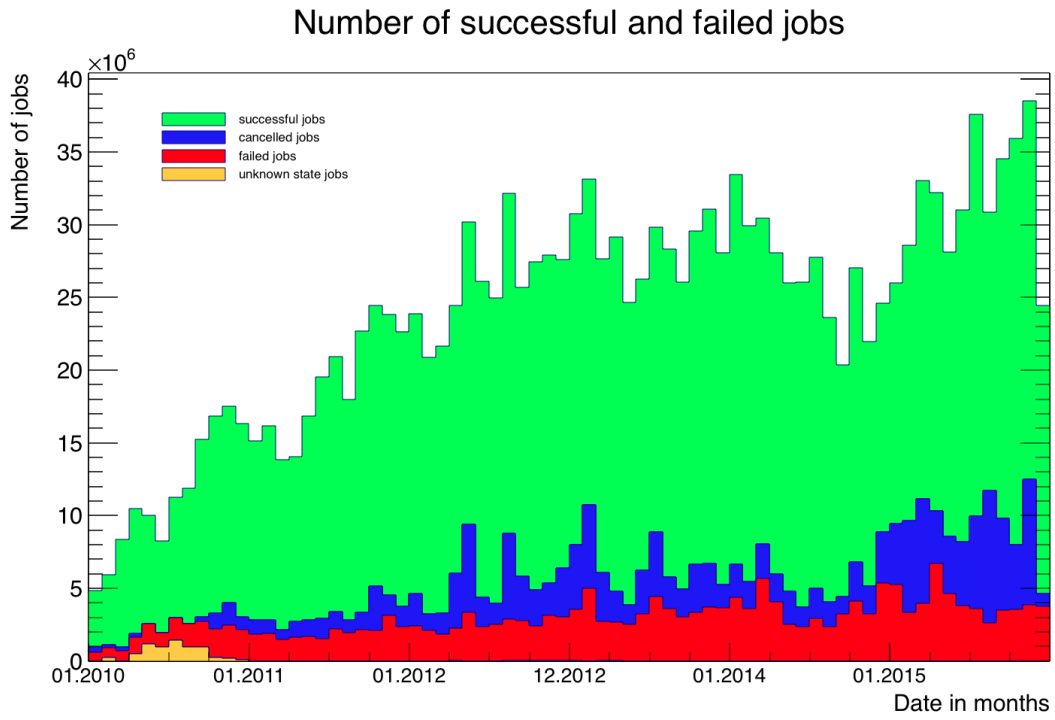


Figure 4.2.: Distribution of successful, failed, cancelled and unknown jobs of all types from 2010 through 2015, grouped by month for the German cloud. The number of jobs in unknown state is small and negligible in this context. Plot derived from [33].

Hence, JEM can help any Grid user to gather detailed performance data about their jobs by providing system metrics, live log file analysis and log file excerpts, which are being pushed forward to the user during the Grid job's runtime.

Debugging functionality, like script monitors for Bash and Python, a binary execution tracer and online stack trace and memory inspection from an attached GDB provide the user with runtime data that would not be available without JEM, neither during job runtime nor afterwards. This can help to understand erroneous behaviour of Grid jobs, which sometimes cannot be accurately replicated on a local developing machine. Either because of different hardware or software environments or due to certain errors are rare enough so that the attempt to catch them locally by running hundreds of long running jobs is neither possible nor would it be practical.

All of the above mentioned data sources are highly configurable and can be specifically activated, depending on the necessary level of inspection demanded by the Grid user.

4.2.2. Site administrators

With the possibility to gather live network and file system monitoring data from running jobs, JEM can be used to regularly gather job performance data. This data then can be aggregated at the JEMserver or any site local ActiveMQ server for further analysis. As this data can be aggregated on a per site basis, this can help to supplement the site local monitoring tools with data provided by JEM. Especially with data, which is usually not accessible to site monitoring tools. These have no concept of the inner workings of the computing jobs, that are supposed to be run and their observational boundary is usually the physical machine itself.

For example, the RSS of running jobs can be easily monitored and could then be fed into the local batch system to supervise overcommitment of memory resources on multi core nodes. In [63], a small analysis of gathered JEM monitoring data was presented. It was shown, that a significant amount of jobs exceeded the limit of 2 GB RSS per single-core job, which has been defined as the maximum allowed memory footprint for the VO atlas [64].

This, in turn, can lead to significant WN performance decrease, if too many jobs use up the available memory and the machine has to start swapping. In severe cases it has been observed that WNs can become completely unusable if enough high memory jobs deplete all physical memory and all swap memory. This usually means that no new processes can be started, not even a shell from which the machine can be shut down gracefully. Then, only a hard reboot of the machine by a local site administrator, with a complete loss of all results of all currently running jobs, recovers the machine back into a working status.

4.2.3. Computing shifters

Due to the complex and diverse nature of the WLCG operations, ATLAS keeps a constant presence of several shifters on duty to monitor and supervise all Grid computing activities, the service availability of the Grid itself and its various participating sites. They use various monitoring tools provided and maintained by ADC to have a steady overview of current Grid activities and problems. The shifters keep track of all occurrences in several JIRA [65] and *eLog* [66] instances.

Even during normal operations a huge variety of problems can occur and need to be attended to. These problems can generally be separated into several different categories:

- problems caused by errors, which were centrally introduced during production task submission (e.g. wrong or missing configuration files, software failures, incompatible software versions, etc.)

- central service problems (e.g. job/task broker service problem, data distribution service problems)

- site local problems, caused by faulty hardware at the site, a misconfiguration of site local Grid services or missing files.

Shifters need to determine the cause of the problem, assign tickets to the responsible experts and follow up on the solutions.

In most of these cases shifters need to inspect pilot or job log files to get a better understanding of the origin for the problem. Output files of Grid jobs can in general only be evaluated once the job is finished and the log files have been written to the output sandbox. In the case of ATLAS, and its PanDA based Grid scheduling system (see Chapter 3), the log files are not passed to the user by writing to an output sandbox. Instead these are stored in a special volatile space token in the Grid storage system by the job wrapping PanDA pilot. The lifetime of the log file space token varies, depending on available space and configured values in the pilot factories. Nevertheless, job log files are only available after the job has finished and the pilot script has had a chance to store all relevant log files, including the pilot log files, in the designated space token [67].

Faulty jobs with a long runtime present a difficult problem to debug, if the underlying PanDA pilot and the batch system are not properly synchronized to the timeout limit for the current queue. If the pilot does not know the correct timeout value, the batch system will automatically terminate the running job with the pilot as its wrapper and no log files will be written. Also a severe overuse of available resources can cause the batch system or the local site administrator to terminate the Grid job without the pilot having a chance to store log files.

A LogFileSaver has been added to the JEM worker node mode to save and store log files from long running jobs. If enabled, the LogFileSaver will regularly archive and compress a list of configured files and send them via HTTP to the JEMserver. Here, the latest copy is kept available and can then be easily retrieved via HTTP requests.

If, for example a set of jobs from a possibly faulty task takes more time to finish than expected, subsequently started jobs could be instrumented with JEM and an activated LogFileSaver. This would allow the task submitter to determine in-time, whether the job has run into some kind of error[2] condition from which they cannot recover. Already running jobs could then be terminated to prevent further waste of resources, if the diagnosed failure is endemic to the task itself.

For certain simulation jobs, with specialized overlays, the individual job runtime can easily exceed 24 hours. A quick peek into some log files, while the job is still running, can save time and resources, once an error is identified. Therefore, selective usage of JEM could give shifters a better understanding of possibly faulty job behaviour and appropriate steps could then be initiated.

[2] A possible explanation for such an erroneous state could be an infinite loop, which has been observed in Grid jobs doing detector simulation.

4.2.4. Monte Carlo validators

MC production represents a significant amount of the used resources the Grid. For efficiency reasons and due to the large amounts of data to be generated and processed, the production is organized in huge computing tasks, which can contain thousands of jobs and can require tens of thousand of CPU hours along with significant amount of storage space.

Therefore, faulty, misbehaving or misconfigured production tasks can cause significant amount of wasted resources, as the produced data needs to be discarded or fixed. The effects are even more severe when errors in early stages of the production chain are not detected and faulty data sets are used in subsequent production steps.

The current situation has been discussed in a series of ATLAS physics group meetings and it has been decided to centrally validate updated software by comparing it against known reference samples. However, no centrally organized and automated approach at generating and comparing results had been realized. Additionally the number and quality of analyses has not been standardized throughout the different working groups.

The implementation details of JEM's answer to this use case are further described in Part IV. The JEM ecosystem, has been extended to serve as a Live Monte Carlo Validation tool. The realized automated approach allows continuous quality control of MC production results. It also can show errors and problems in new tuning sets for event generators or new software versions of MC software.

4.2.5. HLT operators

The HLT, as described in Section 3.3, is being prepared in a special setup, which runs exclusively at the Tier-0 at CERN. Recent discussions with HLT developers have shown, that the recomputation of trigger setups is very time critical and usually has to be finished within 24 to 30 hours. As the usual HLT task contains $O(1000)$ jobs and a job fails, it has to be decided quickly if the entire task is broken or only a small faction of the jobs.

Therefore, a reliable and quick access to all necessary log files is necessary to decide whether the running task should be aborted or not. The JEM LogFileSaver provides exactly this functionality and some early tests have been conducted. It is described in detail in Section 7.1.

4.3. History of JEM

In this section the JEM project history and the contributions of the different developers over the last ten years, since the original foundations of JEM are discussed. An overview of the project evolution by the different developers is summarized in Table 4.1.

The predecessor project to JEM, the JMS, was first planned in 2004 as part of the D-Grid initiative. For this project, the black box of a Grid job and their intransparent failures were deemed a hurdle to succesfull Grid operations and a job centric monitoring system was created.

The first JMS implementation was written by Ahmad Hammad as part of his master thesis [68] in 2005 and consisted of a first bash script monitor and a communication of monitoring data via R-GMA back to the user. During the same time Dimitri Igdalov worked, as part of his diploma thesis, on the JMS internal IPC using POSIX sockets [69].

Table 4.1.: The main JMS / JEM developers and their contributions to the project.

Developer	year(s)	Main work on JMS / JEM project	reference
Ahmand Hammad	2005	• JMS • bash script monitoring • communication via socket and R-GMA	[68]
Dimitri Igdalov	2005	• JMS IPC on worker nodes using sockets	[69]
Andreas Baldeau	2007	• new Bash monitor with modified Bash binary	[70]
Dr. Stefan Borovac	2007	• rename to JEM • maintainability improvements • IPC with named pipes • Python script monitor improvements	[71]
Martin Rau	2007	• JEM Ganga [72] integration	[73]
Dr. Markus Mechtel	2010	• expert system prototype	[74]
Dr. Tim dos Santos	2009 - 2012	• JEM refactoring • shared memory / trigger implementation • *CTracer* • new Python monitor • updated bash monitor • upgraded system monitor	[52]
Raphael Ahrens	2010 - 2013	• *SecurityService* and backchannel • ring buffer developments	[75]
Frank Volkmer	2010 - 2015	• Live Monte Carlo Validation • BackTraceMonitor • Django web front end • LogFileSaver	this thesis

In 2007 Andreas Baldeau added a new bash monitor [70], which uses a modified version of the bash binary itself, and that therefore allows a far better understanding of bash script execution. The same year, Dr. Stefan Borovac took over the lead project development. JMS has been renamed and has since then been called JEM. He improved maintainability, added a Python script monitor and changed the worker node IPC to named pipes [71]. Also in 2007 Martin Rau integrated JEM into the Ganga job submission front end [73].

Dr. Markus Mechtel experimented with an expert system prototype, which used job error codes and different job behaviour metrics, to predict future job failures as part of his PhD thesis [74]. In 2009 Tim dos Santos started a major refactoring [52] of the existing code base, due to severe limitations in the R-GMA monitoring data transportation mode and the named pipes IPC solution on the worker node side. The worker node part has been switched to a shared memory based ring buffer for fast communication and added the concept of triggers and valves for local data analysis and data transportation. To better understand the execution and behaviour of compiled binaries a `C` execution tracer was added to JEM.

From 2010 to 2013 Raphael Ahrens added a secure back channel prototype (see Section 5.4, [75]) and further improved the shared ring memory by fixing bugs in the ring buffer wrap around functionality and the correct handling of critical behaviour, once the ring buffer starts to fill up.

The main additions by the author since 2010 on are documented further down in Chapter 5, as well as in Part II and Part III.

5. Recent JEM improvements

After introducing JEM's general architecture and its use cases in the previous chapter, an overview of the recent JEM improvements and developments since the last major discussion of this project in [52] are provided in this chapter. The project aspects discussed in this chapter consist of improvements to the general JEM systems and while all of them enhance the JEM monitoring capabilities they are not necessarily central parts to the work presented in this thesis.

Originally designed as a monitoring layer for individual Grid user jobs with the capability to trace script execution on the worker node, JEM has been enhanced, partially rewritten and modified to expand its original monitoring and remote debugging capabilities. These new capabilities now include pattern based log file evaluation (see Chapter 7), whole log file dumps, and a BackTraceMonitor and Live Monte Carlo Validation, which is discussed in detail in Part III and Part IV.

JEM is now an arbitrarily, but highly configurable, addable Grid job wrapper, which can provide monitoring data and additional functionality from specially selected, individual Grid jobs that have not been configured at task definition or job definition level to be instrumented with JEM. This has been realised in a way that all decisions regarding JEM can be done in a highly dynamic way by a rule base driven ActivationService (see Chapter 11), which individually decides for each Grid job if an instrumentation with JEM is desired or not and if special monitoring is required.

Section 5.1 introduces the new web front-end and the database layout, which comes naturally from the chosen framework. After that, the Memcachd and its internal uses are discussed in Section 5.2. Some simple overhead measurements of jobs instrumented with JEM versus uninstrumented jobs are presented in Section 5.3.

5.1. JEM web front end

To present the results of gathered monitoring data to the user, additionally to the UI working mode, a web front end for JEM has been developed. This allows users from basically anywhere to check the health and the status of monitored jobs, without the need to launch a JEM UI instance. As most of JEM is written in Python, a web presentation layer based on Python, namely Django, has been chosen. Django allows easy prototyping new visualizations and it also has a powerful object relational database mapper to access and modify data more directly.

Starting from a simple overview of jobs, which are or were running (see Section 5.1.1), the web server now serves as an HTTP endpoint for various submodules of the JEM ecosystem:

- Job monitoring data (see Section 5.1.1)
- ActivationService (see Chapter 11)
- LogFileSaver (see Chapter 7)

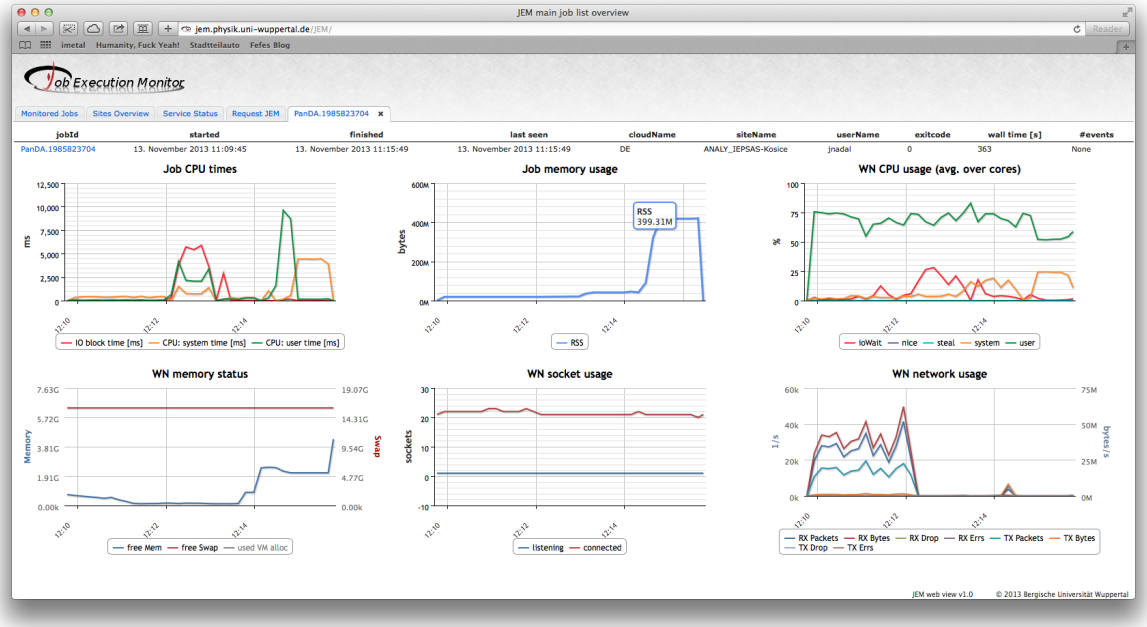


Figure 5.1.: Screen shot of monitored data, being updated continuously as the job is running.

- JEM Live Monte Carlo Validation (see Part IV)
- JEM binary module downloads (JEM development version)
- secure backchannel (see Section 5.4)

The Django web front end has been extensively supplemented with JS [76] libraries like jQuery [77], Highcharts [78] and Datatables [79]. This adds usability and gives a more dynamic user responsiveness with less direct rendering of web pages on the server. Also, a Memcachd server has been added to the running JEM services, which is supported by Django and used for caching of the content of rendered pages.

The web front end has been designed in such a way that static and dynamic content are separated from each other. This allows dynamic JS code in the browser to load, and dynamically reload the monitoring data in an AJAX like manner. Hence the load on the web server can be reduced and the web pages gain a higher responsiveness and better usability.

5.1.1. Monitoring data presentation

As JEM was designed as a job centric monitoring system the main website[1] presents a searchable list of all monitored jobs known to the system. There are too many jobs[2] to be displayed on one single HTML page, therefore, the jobs are presented in a dynamic table, animated by the DataTables plug-in for the jQuery framework. This allows easier searching and filtering of jobs by various filter categories:

[1] `http://jem.physik.uni-wuppertal.de/JEM/`
[2] about 1,027,514 (date: 01.11.2014)

- user name
- site
- cloud
- time frame
- exit code
- job wall time

From the overview table, each job links to an individual job overview page, which gives in depth system metrics and further metadata information about the job and its progress. This data automatically refreshes itself once per minute should the job be in a running state. An example is shown in Figure 5.1.

5.1.2. Database layout

The JEMserver internally uses a MySQL database to store monitoring data in a table layout, which is driven by the object relational database mapper from Django. This allows easy and quick development. Together with the django-south database evolution plugin [80], quick prototyping of new data relations between different object classes can be managed.

The main object class, referenced by all other monitoring data classes refer to is the *Job* object. It contains metadata about each monitored job, like the `jobId` and the `taskId`, the `cloud`, `site` and `queue` its running on, timing information about `start`, `finish` and the `last received` heartbeat and the `exitcode` once the job has finished. For each monitoring data type there is a specific table. These tables are explained in Table 5.1, where each entry contains one set of data, together with a timestamp and a foreign key relationship to its corresponding *Job* object. Additional data for remote identifier caches of running jobs, received job log lines and accounting information for validation tasks is as stored in the JEM database as well.

Although, the ActivationService is a different Django application, its accounting data is also stored in the same database. This logical separation allows for easy schema migrations and updates. Also, it is possible, that the ActivationService data can be moved into its own database on a possibly different machine, which might be necessary for security or performance reasons.

BulkInserter

As Django, at least in the used versions 1.3 and 1.4, has no bulk methods for SQL `INSERT` statements, a utility class, the *BulkInserter* has been developed. It allows to store several Django database abstraction objects, from which then a larger SQL `INSERT` statement is generated. This prevents several single `INSERT` statements from using up to much resources. This is especially relevant in the ChunkConverter. Here, a high number of parallelly monitored Grid jobs can create a serious load on the database through single monitoring data insertion statements.

Table 5.1.: Additional monitoring data tables.

Table name	description of monitored content
NetworkStatusMetric	status data about sent and received packets, bytes, errors and packet drops
NetworkSocketStatusMetric	number of listening and connected sockets per protocol
CpuStatusMetric	CPU usage for user, nice, system, idle, iowait and steal per core
CurrentJobMetric	process data regarding open files, resident set size, CPU usage and I/O delays
SummaryMetric	current load and averages for the last 5 and the last 10 minutes
JobIdentifiers	Contains the keys to all variable length strings that are transmitted in all text related blocks

5.2. MemcacheD

A lot of the results being presented on the main JEM monitoring result website do not change regularly, but can be quite computational expensive to retrieve from the back end database. Therefore, the web front end has been extended with a fast memory based caching system, Memcachd. This eases the load of the database and the website rendering systems. It provides an easy to use and fast key-value store, that automatically expires its entries after a time out, given at storage time. If the cache needs to store new entries once the cache is full, the oldest entries are purged automatically.

Django provides an easy integration of Memcachd's Python interface into its own caching infrastructure, which gave further reason to chose this software package. Memcachd can also be easily distributed over several machines. This way it provides a parallel, synchronized cache and therefore a future upgrade possibility of the JEM infrastructure, which could improve the whole system performance while maintaining scalability.

As described later in Chapter 11, before the ActivationService uses a centralized evaluation process, the ActSvcRuleEvaluater[3], these ActivationService status counters needed to be synchronized between the different WSGI processes. With the key-value store of Memcachd, these values could be shared, therefore providing a simple IPC solution. The IPC became gradually more used by different JEM subsystems.

Memcachd found further use in the command object relay, which is used by the ActSvcRuleEvaluater. Here the output data of commands is stored in the memory cache to be accessed by the commands originating process upon return. This allowed quickly prototyping of new commands, as no new permanent data structures in a database needed to be implemented.

This cache and its IPC capabilities could have been implemented to use a database as a more permanent cache for data store. However, due to the limited machine resources available to the project during the development phase, the database as the most critical part to data evaluation and presentation has been protected from additional load.

[3] see Section 11.5

It has to be noted, that the MEMCACHD system is still a volatile cache. The developed IPC solution should therefore, at a later stage in development, be replaced with a proper IPC solution, which is designed for this kind of task.

5.3. JEM overhead measurements

Naturally, as an additional layer between the payload job and the PANDA pilot, JEM imposes a small computational overhead on the job runtime and degrades its performance, measured in events over CPU time.

To measure this effect a simulation test run has been set up locally on a typical WN of the Pleiades Cluster of the Bergische Universität Wuppertal in 2011. The worker node has been set offline in the local batch system to provide a clean environment without CPU load noise from normal computing operations. JEM was configured in the default monitoring mode, which generates regular system metrics, but no code trace executions or log file scannings were activated.

Each test run consisted of five individual job runs. The job to be tested ran three times instrumented with JEM, in the beginning, in the middle and in the end. In between, at slots two and four, the job was not instrumented with JEM. The jobs running with JEM used a stub PANDA pilot to mimic the normal JEM job set up on a worker node. The first JEM run assured, that no side effects from a slow file system or any of the JEM libraries are not available in time from the JEMSERVER due to some network problems.

All timing data and the exit codes for all five processes were send to the central JEMSERVER via HTTP call to a specially set up DJANGO endpoint, which stored the data in the JEM database for further evaluation. Figure 5.2 shows the overall distribution of the test results of all 164 jobs. The linear regression gives a minimal, but generally constant overhead of 1.2% in job runtime and a constant overhead of 23.2 seconds with a correlation factor R^2 of 0.986.

The figure also shows, that at least 2 datapoints are lying below the marked $f(x) = x$ line, which means, that these instrumented jobs were faster with JEM activated.

5.4. SecurityService

As described in his diploma thesis [75], Raphael Ahrens has extended JEM with a prototype of a secure back channel for command data flow. It is the only system, described here in this thesis, not worked on by the author. Nevertheless it deserves mentioning, as a central new part of JEM. If fully deployed, it could completely change the current Grid computing paradigm for user jobs.

It allows control communication to be sent from a verified external source to a JEM worker node instance running in the Grid. The back channel was added as an experimental feature to study the possibility of configuring and controlling remote instances of JEM dynamically during runtime.

As most worker nodes have no public IP address and/or are shielded by firewalls or NAT, it is usually not possible to send any data to a running Grid job once it is running due to the lack of establishing a communications channel to this worker node. Therefore, to initiate a back channel connection to a JEM worker node the JEM process on the worker node has to register itself for receiving messages at a central known instance. It then regularly polls this server for an awaiting communication request.

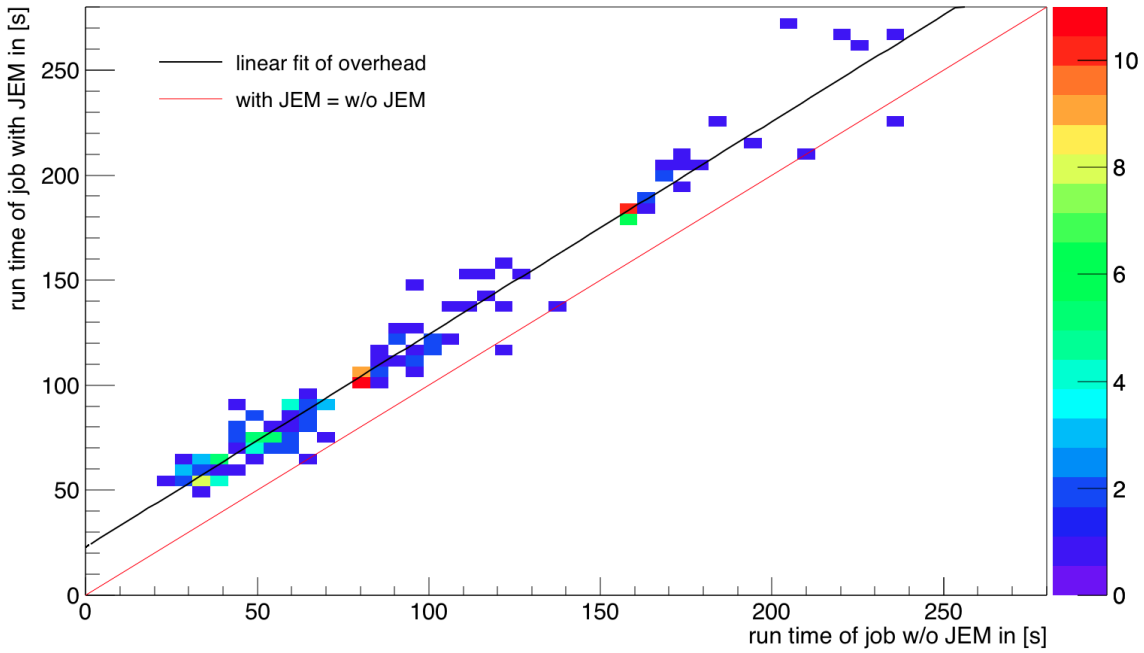


Figure 5.2.: Comparison of job run times with and without JEM.

This service has been realized as a JEM working mode and uses STOMP over an ActiveMQ server as transport channel for its communication.

All communication is authenticated by the users certificate via HMAC signing, to prevent malicious misuse of this back channel. As finally mentioned in [75], the cryptographic measurements taken to secure the back channel should be reviewed by an external person to make sure that all necessary security precautions have been taken correctly.

This feature allows the extension of some of the use cases described in Section 4.2. By having a certain (larger) amount of Grid jobs instrumented with JEM in a dormant state and only later enabling certain monitoring functionalities if the need arises, monitoring features could be quickly activated and used.

For any Grid user, already planning to monitor jobs, this allows the user to instrument a larger amount of jobs, therefore, giving broader coverage without compromising resources on the JEM-SERVER. Especially for computing shifters, this might be an interesting possibility to dynamically inspect Grid jobs of a certain misbehaving class during runtime. For example jobs of a task with recently failed jobs.

Part III.

Analyzing backtraces over the Grid

Introduction

The JEM back trace monitor (BackTraceMonitor) is an addition to the monitoring capabilities of JEM in a prototype state.. It allows the user to extract stack traces of running binaries. For the *CTracer*, described in [52], it was always necessary to have the binaries specially prepared.

Chapter 6 discusses the motivation for the developments of this process inspection method and its connection to the use case described in 4.2.3. In Chapter 7 additions and new analytical evaluations to already existing log file inspection and transmission techniques are explained, that preceded the BackTraceMonitor development , but are nevertheless related to the same use case.

In Chapter 8, a detailed description of the stack trace extraction, their transmission procedures and the visual inspection by a stand alone application for the user is given. This part then closes with a result discussion in Chapter 9 a an outlook to future development possibilities.

6. Motivation

Faulty computing processes, running on remote hardware are usually difficult to understand. Especially if the logging facilities do not show conclusive evidence what kind of error occurred or which kind of behavior led to the error condition. If the computing job ends up in some kind of infinite loop, it will in case of most Grid environments, likely result in a forceful termination of the job, once the overall time limit has been reached. Additionally, large scale production jobs are hard to debug if the error is believed to be related to faulty input data, which then lead to a faulty state.

The JEM project was contacted by the ATLAS MC coordinators to help investigate faulty job behaviour in long running minimum bias overlay simulation jobs. These are a special kind of jobs, which are quite computing intensive and have a very long runtime per event. As simulation jobs usually have a long initialization phase, due to the large geometry of the ATLAS detector it is not very economical to have a low events per job ratio. As some of these long running jobs regularly, but unpredictably, hit the batch queue time limit and were therefore terminated by the local batch system, it was believed that faulty input data led to some kind of infinite loop in the computing jobs. For a better understanding of these errors, a successful log file retrieval is necessary and the various log file analysing techniques, described in Chapter 7, were developed.

As it was unclear which algorithm is responsible for the suspected infinite loop it is very difficult to selectively increase logging granularity to a higher level. An overall increase of logging granularity would have degraded the general job performance too much. Also, Athena automatically decreases logging verbosity after a few events have been processed, to a degree that not even individual event numbers are acknowledged. Athena also does not printout timestamps per individual log line, making it very hard to correlate logging events with the actual progress of the job.

As the suspected infinite loops occurred very rarely and the task execution is quite expensive, in terms of needed computing time in the Grid, a method was searched to understand job execution flow of very long running jobs with insufficient logging output[1]. After discussion with ATLAS MC coordination a prototype was designed, that allows these infinite loops to be identified online, without changes to the underlying Athena framework or the job configuration. This prototype and resulted in a new JEM WN monitor: the BackTraceMonitor.

Some of these infinite loops are believed to occur due to faulty data structures, where particles in the linked lists have a faulty circular connection. Either by automatic heuristic analysis or by manual inspection of the stack traces these loops can be identified. Also, with targeted memory excerpts the number of the event, the job is currently processing, can be extracted and the problem can be inspected further by checking the particular event in the input files, containing the raw event data.

A job, properly instrumented with JEM and the BackTraceMonitor can manage that. With monitoring data transmission to the JEMserver, the stack trace excerpts can be monitored online.

[1] The understanding and debugging of these kind of problems will be even more difficult, once these jobs are being parallelized into multiple processes.

7. Logfile inspection

As described in [52], JEM has been equipped with a *FileWatcher* monitor, which allows the user a quick peek into the progress of their jobs. It reads the last n number of lines from a set of output files and stores them as file line chunks in the ring buffer every m seconds. These can then be transmitted back to the JEMserver to give the users a quick peek into the progress of their jobs, if they are cleared for transmission. The interval m, the number of lines to read back n and the names of the files to be watched can be configured.

This feature was never meant to transmit entire log files in real time back to the JEMserver, as a small number of jobs with very verbose log files could easily use up the available bandwidth and more importantly, the processing capacity of the ChunkConverter at the JEMserver. Two additions are described in this chapter, to improve the existing monitor: the LogFileSaver in Section 7.1 and the live log file inspection using pattern matching in Section 7.2.

7.1. LogFileSaver

For manual, almost real time, inspection of log files the JEM log file saving plugin (LogFileSaver) monitor has been added to the JEM WN module. It regularly, by default every five minutes, collects a list of files, preconfigured before runtime and then archives and compresses them. This archive then gets transported to the central JEMserver using a POST request[1].

The `storeLogs()` HTTP endpoint at the JEMserver is realized as a Django view, which gets the file data from the POST request and then stores it on the file system in a special directory, named after the jobs PanDA identifier, which is transmitted as metadata with the POST request. Any HTTP request received by this view, that does not adhere to the expected data format gets silently discarded.

Another HTTP endpoint, `getLogsLink()`, returns a list of HTTP links to the base directory containing all received files, to the most recently received file and to the folder containing all unpacked files. Following the link to the unpacked files allows quick access to the most recent log files using only a couple of clicks. While everything is conveniently accessible in the web browser all relevant files can also be downloaded to a local machine. Additionally the log files can be sent to any other HTTP endpoint, as long as it properly understands the HTTP header fields, containing the file data, used by the LogFileSaver.

This is a cruder implementation of the existing *FileWatcher*, as a lot of the data is transmitted redundantly, however, it consumes a lot less computing resources at the JEMserver as the log line data is not processed by the ChunkConverter. It can also be modified later on to automatically peek into intermediate analysis job results by transmitting not log files but for example created histogram files.

[1] The remote URL at the JEMserver is `http://jem.physik.uni-wuppertal.de:8080/JEM/storeLogs`

7.1.1. PurgeLoglinesTask

As the storage of job log files consumes a serious amount of disk storage resources at the JEMserver the *PurgeLoglinesTask* has been created. It regularly checks all received log file directories for old versions of received archives and deletes them, as the newest archive holds more current information, thereby obsoleting the older archives. Additionally, this task regularly checks the overall size consumption of the entire log file directory and starts deleting old directories, should a preconfigured size threshold be exceeded. This ensures, that the received log files do not use up too much disk space on the JEMserver.

7.2. Online log file analysis

Additional to the manual log file inspection, JEM is able to inspect log files on the fly for the occurrence of patterns, indicating certain problems as they happen. The online log file analysis searches in all log files, which are being observed by the *FileWatcher* monitoring module by having two specialized triggers checking the generated chunks.

7.2.1. Token and Pattern

The pattern matching system is based on tokens and pattern, similar to the techniques used in compilers [81]. There, tokens are generated by doing a lexical analysis of source code, which is to be compiled. For this application, similarly, a lexical analysis is performed by checking each log line against a defined set of regular expressions, which are each associated with a token.

The generated tokens are then handed over to a syntactical analysis for pattern matching. As the generated tokens contain metadata about the location of their origin, namely the log filename and the line number the pattern analysis can take this information into account Therefore, patterns can span multiple log files.

7.2.2. RegExTokenTrigger

Token objects from `CHUNK_TYPE_WATCHED_FILE_LINES` chunks, which have been generated by the *FileWatcher*, are generated by the *RegExTokenTrigger*. It is written in `C` for fast regular expression performance and stores detected tokens, together with a time index of the current log line, the line number and up to three group matches from the regular expression. As seen in Listing A.1 the found token are encoded into *LlfpmTokenBlock*, which are transmitted using a `CHUNK_TYPE_LLFPM_TOKEN_CONTAINER` chunk.

The trigger architecture has been designed in such a way, that each trigger needs an accompanying Python module containing configuration data. From this *RegExTokenTriggerConfig* module the list of token is given to the *RegExTokenTrigger* during initialization as a Python list of dictionaries, where each dictionary represents a token. See Listing A.2 for the Python representation and Listing A.3 for a shortened example of the known token list.

In the `postRegister()` method[2] the given token data is transformed from Python into `C` structures and stored in memory. It is referenced by the `tokenList` pointer in the `RegExTokenTrigger`

[2] It is derived from the base trigger structure, see [52].

structure, as can be seen in Listing A.4. The main structure also contains some internal counters for info and debugging purposes, as well as a small cache for the shared memory string identifiers of all known filenames. There is also a caching structure, the `tokenCache`, to store all matched tokens for a file line chunk, so that one single token chunk can be generated in the end.

For each `CHUNK_TYPE_WATCHED_FILE_LINES` chunk the `examine()`[2] method of the *RegExTokenTrigger* is called, which updates the internal `senderCache` and in turn calls `analyzeLine()` for each `BLOCK_TYPE_FILE_LINE` block. Here all known regular expressions from the `tokenList` are tested against the line and upon success are stored in the `tokenCache`. Should the regular expression have matched a grouped wild card, the results of up to three results are stored and later on transferred to a *LlfpmTokenBlock*. If at least one token has been identified for the whole `CHUNK_TYPE_WATCHED_FILE_LINES` chunk a `CHUNK_TYPE_LLFPM_TOKEN_CONTAINER` chunk is created and written to the ring buffer.

7.2.3. SimplePatternTrigger

The second stage of the live log file analysis is realized in the *SimplePatternTrigger*. It acts similarly to the syntactical analysis of a compiler. The token chunks generated by the *RegExTokenTrigger* are read and fed to all known patterns. Once a pattern is marked as complete the result is propagated further.

A pattern is characterized by having a unique `patternId`, a describing `name`, and a list of token on which the pattern matching result can be based, as can be seen in Listing A.5. A *SimplePattern* is currently the only specialization of this interface and is basically a simple AND-conjunction of the occurrence of all token in the internal `tokenList`. Therefore meaning, that all of the tokens of this pattern, regardless of order have to be at least encountered once before the pattern member `isMatched` can be set to true.

The *SimplePatternTrigger* is realized in Python. During the initialization phase, it registers itself for `CHUNK_TYPE_LLFPM_TOKEN_CONTAINER` chunks and gets a list of all known token and patterns. It also generates an internal dictionary, the `__tokenDict`, which maps all known token identifiers to a list of all patterns. This allows faster lookup. The patterns are stored at a well known location in a simple list, as shown in Listing A.6.

Upon being triggered, the `examine()` method of the *SimplePatternTrigger* is called. The token information is extracted from the *LlfpmTokenBlock*s and each token is then fed to all patterns it is associated with via the internal `__tokenDict` dictionary. For this each pattern has a `newToken()` method, which internally evaluates the pattern and marks it as matched once that the case. If at least one matched pattern is discovered a `CHUNK_TYPE_LLFPM_PATTERN_CONTAINER` chunk is created and for each matched pattern a *LlfpmPatternBlock*, see Listing A.7, is attached to the chunk. This chunk can then be transmitted to the user to inform the user about the pattern's event.

The *SimplePatternTrigger* operates on generated token chunks and not on mere line data information. Therefore, more complex patterns, spanning multiple lines, and, more importantly, multiple files, can be detected. Together with the time index of each token, this functionality could be used to check for progress of the job by searching output lines for event indices.

Worker Node main module pattern callback

The main WN module has a callback method `handlePattern()`, which only gets called once the *SimplePatternTrigger* detects a finished pattern. As its only parameter the method gets the identifier of the finished pattern, which is stored locally in `__handlePatternList`. In the WN main loop the method `__workPattern()` is regularly called, which checks the `__handlePatternList` and is able to react to the occurrence of certain patterns. This has been used to selectively start the BackTraceMonitor, which is explained in the next chapter.

7.2.4. PostMortemLogAnalyzer

The pattern matching functionality of the *SimplePatternTrigger*, as described above, has been integrated into a small stand alone Python application: the PostMortemLogAnalyzer. This allows one to check log files of finished jobs, without using the complete JEM ecosystem.

It accepts a list of one or more PanDA job identifiers as parameters and then checks the local working directory, whether a subfolder named after the job identifier already exists. If not, the PostMortemLogAnalyzer attempts to download the log files from the PanDA server and stores them in an appropriately named new directory.

If suitable input files have been either found or downloaded in the local working folder, a `Controller` class is initialized, which then in turn initializes all known tokens and patterns. The `Controller` provides a `analyzeFiles()` method, which gets a job identifier as a parameter and then subsequently opens all log files in the specific folder, parses them line by line and feeds them through the internal pattern matching engine. After each job identifier and the corresponding log files have been analysed, a summary of all found and matched patterns is given, therefore providing a quick tool for pattern based log file analysis.

The PostMortemLogAnalyzer is a single Python script without any import dependencies on any JEM code. Therefore it can easily be distributed to analysis users or MC production operators. Finished jobs can now be quickly investigated for the programmed known patterns. As long as the patterns are sufficiently unique, these problems can also be either detected or ruled out. In the second case other problems are the most likely source of error and further investigation can be directed into this direction.

8. Inspecting stack traces from payload jobs

As described in Chapter 6 the goal of the subproject, described in this part, is to develop a method that grants insight into the behaviour of running jobs, which have not been specially prepared for the *CTracer*. The BackTraceMonitor is currently in active development. It automatically interacts with the GNU Debugger (GDB) by using the GDB internal Python module, which is explained in Section 8.1. The WN monitoring module and the actual stack trace generating script are explained in Sections 8.2 and 8.3.

For the analysis of the generated stack trace data, a standalone Python application is in development, which displays the stack trace data and allows navigation and searching. This application is explained in Section 8.4.

8.1. GDB Python interface

To automatically interact with a GDB instance, attached to a running process, the GDB internal Python interface `gdb` [82] is very useful. It provides a lot of convenience to any programmer as the basic data concepts of GDB debugging are accessible as Python class instances:

- Inferiors (processes known to GDB)
- Frames (corresponding to function calls and their stack frame)
- Blocks (containing symbols per frame)
- Symbols (information about symbols; name, type, values, location, etc.)
- Events (allows Python code to react to certain events)
- Threads (allows the inspection of different threads of the Inferiors)

This interface only works inside GDB, meaning the `import gdb` Python statement will fail in any normal Python interpreter, except when the internal Python interpreter is invoked from a running GDB instance. To use it, one needs to start GDB, source a Python script, which then contains the necessary any desired, complex logic to investigate one or more processes.

While the Python GDB extension is still a prototype it is well documented and allows automated control of GDB, rather than command line parsing from and to the GDB interface using Python's `subprocess` module. GDB has a command, called `source`, which can load an arbitrary file and interpret it. If the file is a Python file GDB invokes its internal Python interpreter and control of the GDB instance is delegated to the Python interpreter.

8.2. The BacktraceMonitor WN Module

The BackTraceMonitor, similar to many other JEM WN modules, runs as a thread in the WN monitoring process. It acts as a launcher and watchdog for the GDB.

The payload job does not run as a child of the WN monitoring process. It consists of several subprocesses and the PID of the actual payload job has to be determined. The BackTraceMonitor checks the `/proc` file system for all processes running as children of the main JEM process. The command line entry for each process is checked by a regular expression, which currently matches to any command line containing the string `"athena.py"`. It is assumed that there is only one process matching the search criteria and its PID is then stored in a local file[1].

If no matching process can be found the search is performed in frequent intervals. Once the process is identified GDB is launched and the BackTraceMonitor starts to regularly check if the GDB process is still alive. Via the *stdin* pipe GDB is told to `source` the `GdbBackTraceScript.py`, which is explained in detail in the following section. If the end of the subprocess has been detected the local PID file is deleted and the BackTraceMonitor finishes. Currently, the backtrace monitoring of only one process is supported.

8.3. The GdbBackTraceScript

The *GdbBackTraceScript* is a Python script, which can only run inside the internal GDB Python interpreter as it uses the GDB specific Python modules. First it sets up the common JEM logging facilities by connecting to the logging process. After that, it attaches itself to the shared memory ring buffer using JEM environment variables and introduces the BackTraceMonitor to the *WNcore* with a `CHUNK_TYPE_MONITOR_STARTED_EVENT` chunk.

After initialization of the *BacktraceMonitor* class, its `run()` method is called, which, in a regular intervals calls the `sample()` method. the `sample()` method gathers data as long as the process, which is to be monitored, is running. Each time, a `CHUNK_TYPE_BACKTRACE` chunk is created at first, so that the data blocks can then be appended later on.

As the PID of the process to be monitored is known from the parent process, the GDB command `attach <PID>` is executed. Once attached, the complete stack trace of the attached process is extracted. First the *gdb.newest_frame()* method is called to get a reference `frame` of the function frame, where the process is currently working in. Then, recursively, all parent frame references are gathered by calling `frame.older()` until *None* is returned.

For each frame the full name, containing its function name is extracted. As the stack pointer is stored in the name of the frame as a hexadecimal text representation of its address, it is extracted using string operations. With the stack pointer and the name of the frame a *BTStackframeInfoBlock* can be filled, containing additionally the depth of the current stack frame.

Also, once the stack pointer is known, a memory excerpt with fixed length (currently 200 bytes), is taken and stored in a *BTMemoryInfoBlock*, which is also appended to the `CHUNK_TYPE_BACKTRACE` chunk. For a description of the *BTStackframeInfoBlock* and the *BTMemoryInfoBlock* see Listings 8.1 and 8.2. Once all data has been gathered, the chunk is released to the shared memory and GDB is detached until the next data sampling period.

[1] The filename, containing the PID of the process to be instrumented, is `.JEM.gdb.cmd`

```
class BTStackframeInfoBlock(PayloadBlock):
    block_visible_name = "stackframe-info"
    block_type_name = "BLOCK_TYPE_BT_STACKFRAME_INFO"
    block_fields = [("stackId", "I"),
                    ("stackLevel", "I"),
                    ("frame", "I"),
                    ("eip", "I"),
                    ("savedEip", "I"),
                    ("argList", "I"),
                    ("locals", "I"),
                    ("previousSp", "I"),
                    ("ebxReg", "I"),
                    ("ebpReg", "I"),
                    ("esiReg", "I"),
                    ("ediReg", "I"),
                    ("eipReg", "I"),
                    ("frameName", "#s", "s")]
```

Listing 8.1: *BTStackframeInfoBlock* block description.

```
class BTMemoryInfoBlock(PayloadBlock):
    block_visible_name = "stackframe-info"
    block_type_name = "BLOCK_TYPE_BT_MEMORY_INFO"
    block_fields = [("stackId", "I"),
                    ("memOffset", "I"),
                    ("memLen", "I"),
                    ("frameMem", "#s", "s")]
```

Listing 8.2: *BTMemoryInfoBlock* block description.

8.4. Analytical front-end

The analytical front-end, the BackTraceViewer, has been developed as a *curses*[2] [83] based stand-alone Python application. Due to the fact that *curses* and Python are widely deployed on all relevant operating systems, the BackTraceViewer can easily be downloaded and executed locally. The input database file, which is read by the BackTraceViewer is generated by the ChunkConverter, the process is described in 8.4.2. Its contents are displayed and methods for comfortable navigation are provided. The BackTraceViewer regularly checks the input database file for updates, which are added by the ChunkConverter should the job still be running.

8.4.1. User interface

As a *curses* application the BackTraceViewer is naturally controlled via keyboard. It uses four panels to display the stack memory, the current stack trace, application logging information and general information. The stack trace can be navigated up and down with the keys 'q' and 'a'. With 'w'

[2] *curses* is a terminal control library, used mostly for Unix-like systems It enables the construction of text based user interface applications.

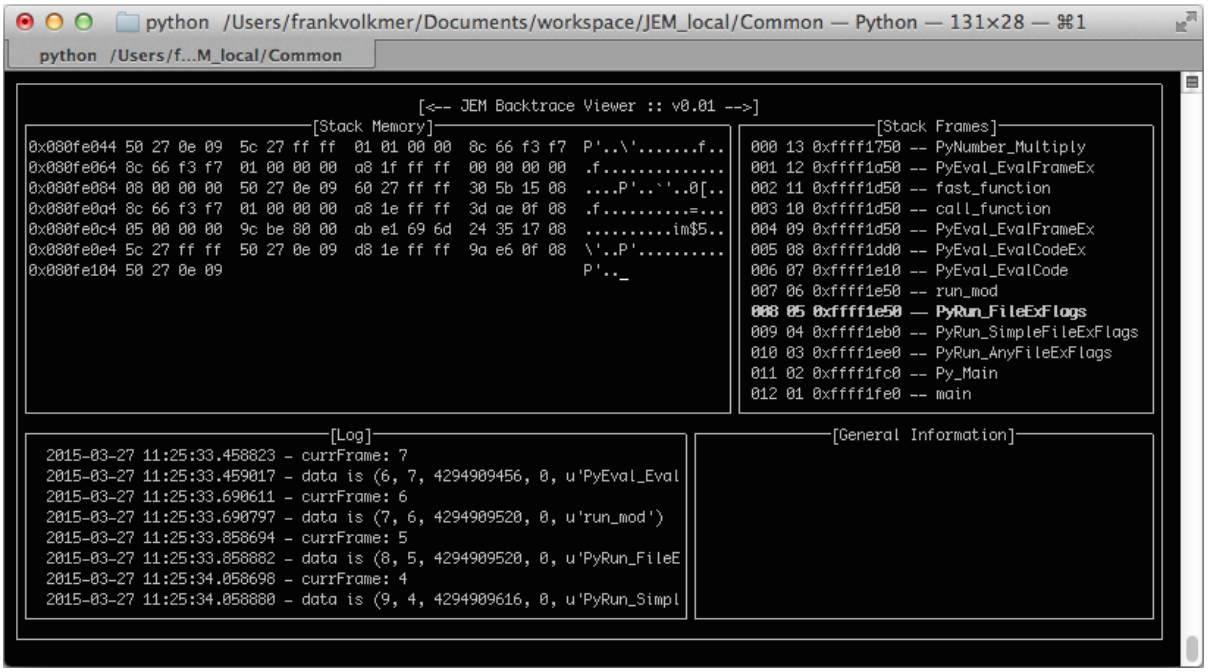


Figure 8.1.: BackTraceViewer front end, showing a stack trace and one functions stack memory content.

and 's' the next and previous available stack trace can be selected respectively. With 'o' the application can be closed. Figure 8.1 shows an exemplary screen shot.

It allows therefore, easy navigation in the received data. With this data representation, a user can check the stack traces for signs of looping behaviour and can verify this assumption be reading the stack memory for recurring data patterns.

The BackTraceViewer is still in a prototype phase and needs to be developed further. Missing is for example an heuristic detection of looping behaviour or an automatic recognition of the current event number. As this information can be gathered, simply by analyzing the stack trace and the stack memory excerpts, it should be possible to do this locally in the BackTraceViewer. Otherwise additional detection intelligence needs to be added to the BackTraceMonitor.

8.4.2. ChunkConverter modifications

It has been decided to automatically export all stack trace data into a dedicated SQLite3 [84] database, to have complete offline access. This database consists of a single file and can easily be sent around for inspection of monitored jobs to third parties. The database files are created in a location, which is web accessible[3]. The standard functionality of storing chunk data in the normal monitoring database is still available but currently disabled by configuration option.

As with all other transmitted data chunks, the ChunkConverter reads them from the ActiveMQ server, decodes the binary format of the chunks and stores them. In this case, chunk data is

[3] `http://jem.physik.uni-wuppertal.de/btmDB/jobId.jem.btm.db` where <jobId> is the PanDA job id.

not stored in the general JEMserver MySQL database as access to this database is restricted to local applications but in separate SQLite3 database files per `jobId`.

The ChunkConverter checks its local cache dictionary `__openDBConnections` for open database connections for the chunks `jobId`[4]. If no open connection exists, it is checked if a database file in the `btmDB` directory already exists. Should no database file exist it is created using the Python SQLite3 API and the database tables `stackFrames` and `stackMem` are created, each one having corresponding fields to the member variables of the stack info block and the memory info block. Once a correct handle to the database file has been established the stack data and the memory data is inserted into the database file via a SQL `INSERT` statement.

[4] The database connections in the local cache gets regularly closed after a timeout.

9. Discussion

Although, the BackTraceMonitor subproject shows promising results, it remains in a prototype state due to a shift in development focus of the JEM project. It needs further attention before it can be deployed into a production environment, it shows, however, that direct stack inspection is possible and feasible with reasonable amount of work and that the data can be visualized and exploited for further information. The time a job is being suspended, while GDB inspects and copies the stack trace into Python memory, is less than a second and quite negligible compared to production job runtimes measured in hours.

Due to ongoing efforts to parallelize Grid jobs into several processes by ATLAS, the limitation of the BackTraceMonitor to only support the GDB inspection of one payload subprocess needs to be rethought and enhanced to support such a scenario. To utilize the full introspective potential of GDB inside Python, its interface needs to be enhanced, as some functionality is still missing.

Although the BackTraceMonitor provides redundant functionality to the *CTracer*, it is far from complete compared to the comprehensive functionality of the *CTracer*. Rather, it provides a different approach to binary monitoring, that should be investigated further.

9.1. Future prospects

For future updates the search and extraction of the current event number is planned. This number has to be stored somewhere in the processes stack space, and if not, it will be at least referenced from there, so that a memory extraction of the heap, it is pointed at, is also possible. After a review of the Athena code base the appropriate function names can be identified and therefore the memory location of the event number.

With the current event number known, the time per event can be measured. Either, locally in the job in the BackTraceMonitor or post mortem in the BackTraceViewer. For each event the algorithm stack can be evaluated. Should the job run into some sort of looping behaviour the event number is known and can be used to reproduce the erroneous behaviour externally in a debugging environment.

Additionally, if gathered log lines from the LogFileSaver are available, they could be shown instead of the stack memory to correlate log events with the stack trace of the monitored process.

Part IV.

Live Monte Carlo Validation

Introduction

In this part the implementation of a new central JEM use case (see, 4.2.4) is described and discussed: the JEM Live MC Validation.

In Chapter 10 the motivation for the presented system is explained, together with a deeper analysis of the driving use case. In the following chapters the ActivationService, as a central JEM component for centrally instrumenting Grid jobs with JEM, is introduced, followed by the actual task validation system, in the next chapter. The reference file cache, as central store for validation related histogram files, is introduced next, followed by Chapter 14, which describes the web interface, necessary for controlling and initiating validation related tasks in the JEMserver system.

Chapters 15 and 16 discuss the user's experience and constructive criticism by the validation shifters and the JEM operator, current technical limitations and possible ways to solve them in the future. Finally Chapter 17 provides possible alternative solutions to the one being introduced in this part and discuss their feasibility.

System overview

With the introduction of a highly configurable ActivationService, the central JEMserver is able to instrument specially selected Grid jobs with JEM. As JEM can be enhanced with additional scripts, any job post processing can easily be defined and implemented. This is especially important, if special post processing is only required for a small subset of all jobs in a large computing task, which can not be configured using traditional ATLAS production tools. Output files of MC production jobs are usually quite large. It is a time and performance advantage to work with them, while they are available at the Grid jobs local working directory, instead of accessing these later with comparably huge costs in network, disk space and computing capacities.

Therefore, for this use case, selected Grid jobs are post processed with a special software that produces quality histograms from already available output files, immediately after they are available. Under the premise, that these quality histogram files are by several orders of magnitude smaller than the original output files these files are then sent back to the JEMserver. There they are stored, merged and finally compared against a previously defined reference file by using various statistical methods. A web interface allows authenticated validation personal to selectively instrument the required Grid jobs for quality histogram creation and to initiate further actions within the JEM Live Monte Carlo Validation system.

10. Motivation

In this chapter the general use case for the JEM Live Monte Carlo Validation is motivated, by firstly explaining, why MC software in general needs to be regularly validated in Section 10.1. Secondly the necessary subtasks for an online task validation in the computing Grid in Section 10.2 are described. The ATLAS MC working group established a unified generator validation group to standardize validation procedures, which is introduced in Section 10.3. Also, the important role of JEM as a central infrastructure part for generator validation, together with a key example why automated validation is necessary is presented there.

10.1. Necessity to validate Monte Carlo software

As described in 3.4.2 MC computing is an integral part of experimental high energy physics and the ATLAS experiment is no exception. The whole amount of software packages used in mass production, together called MC software, are constantly being developed, improved and adapted to newest physics results.

For different parts of the MC production process various software packages have emerged during the history of high energy physics computing. Each package specializes on different aspects of the generation and simulation process necessary to understand the model.

As some of event generators, that are being used, are quite old and have been in use in various high energy physics experiments over the last decades, some of them[1] are still written in Fortran77 and are currently being rewritten in more modern languages like `C` or `C++`. Also, with advances in theoretical physics and new results from high energy particle experiments, the physics described in event generators, has to be updated and improved. Aside from that, improvements in memory usage and usage are always necessary to keep in the limits of available resources or the obtain results faster.

To ensure that results are consistent between the different generator software packages, their different versions and the obtained data from high energy physics experiments agree with each other, the different software packages need to be validated against each other and data. Some of these validations can be done using automated functional tests, as is done for example in the ATLAS RTT [85] framework. It runs the tests of the nightly builds of a lot of ATLAS software packages for different target releases and for all supported combinations of target platforms, bit widths and compiler versions. In these automatic nightly tests some, computationally cheap, statistical validations are performed.

However, to fully evaluate all possible branches with enough statistical significance, a lot of computational resources have to be used to satisfactorily describe the reality of the experiment. Event counts in the order of at least 100,000 are usually used to ensure, that statistical validity has been achieved. Creating and computing this amount of events on a nightly build test cluster would be impossible.

[1] For example PYTHIA was written in Fortran up until version 8.1, from where on development continued in `C++`.

These calculations can take several hours to finish and require memory and computing resources usually not found in nightly test systems. This often requires that these validation computations can not be run locally on developing machines or even dedicated validation machines, but instead have to be run in the Grid as specialized validation tasks.

10.2. Online validation procedures

In general one has to match several criteria to get an automated, reliable MC validation which uses the resources of the Grid to compute its validation results. These are:

- organizing the validation tasks
- creating the statistical output needed for any validation by running jobs in the Grid
- generate quality histograms describing the underlying physics process
- transferring the output to a central server for the following steps
- aggregating the generated output to levels of statistical significance
- comparing the merged validation task output to known references for the specific physics use case
- notify dedicated persons on comparison results, especially if significant deviations have been detected

While all of these criteria can be matched by a variety of different software solutions, some of which are discussed further in Chapter 17, JEM (see Part II) already provides key elements of the needed infrastructure. Together with guidance from ATLAS MC coordination the project described in detail in this part has been realized, based on the afore mentioned criteria.

10.3. ATLAS Monte Carlo validation group

The ATLAS MC working group [86] is responsible within ATLAS to initiate and control MC mass production. As such they are also responsible for the software, that is being used to generate the needed results. While the general Grid infrastructure is provided by ATLAS Distributed Computing (ADC), the ATLAS MC group has to concentrate on the production and analysis software running in the Grid. This includes updates, maintenance of configuration setups and, important for this work, the validation of the results of the certain MC stages as described in 3.4.2.

As described by the MC working group convener in summer 2013, the event generators have been maintained by the responsible experts, usually concentrating only on their areas of expertise and interest. This lead to situations, where errors are discovered rather late as no one was double checking all the appropriate control plots to completely validate a software release. An example is shown in Section 10.3.2.

Table 10.1.: Excerpt of mc_1 and mc_2 dataset details from AMI.

Dataset Parameters	mc_1	mc_2
logicalDataset-Name	mc12_8TeV.110819.AlpgenPythia-_P2011C_ZeebbNp2.evgen.-EVNT.e1477	mc12_8TeV.200334.AlpgenPythia-_Auto_P2011C_ZeebbNp2.-evgen.EVNT.e1930
generatorName	Alpgen+Pythia+Photos	Alpgen+Pythia+Photos+Tauola
Transformation-Package	17.2.4.8	17.2.8.13
totalEvents	45,000	250,000
totalSize	837,925,638 B	4,841,404,212 B
PDF	CTEQ6L1 - LO with LO α_s	CTEQ6L1 - LO with LO α_s
generatorTune	Perugia2011C	Perugia2011C
generator	PYTHIA 426.2	PYTHIA 426.2

Due to missing automated systems to control MC result quality, errors remained undetected in official releases, sometimes up to a couple of weeks. This lead to erroneous MC files, which were then used further down the chain by the analysis physicists, which in turn resulted in lots of wasted resources, both in computing and in working time. The organizational overhead to identify all user of an erroneous dataset and its derivatives is also huge.

Once discovered that inputs to ongoing analysis efforts were not correct, the whole analysis chain needed to be reevaluated to estimate the impact of discovered errors on current analysis results. Usually some parts of the analysis needed to be recomputed as previous results were deemed to be incorrect. This is obviously quite inefficient and costly a lot of both computational and analysis physicists labor resources.

10.3.1. Adding JEM as an automation tool for validation

As mentioned above, each software package expert validated ones own tool, usually with their own set of scripts and analyses. To improve generator result quality, inside the ATLAS MC working group, a new position has been created: the MC generator and sample validation coordinator. Under this new responsibility generator validation has been unified, by using a fixed set of validation control plots and standardized procedures to validate the generator outputs.

As a first step it was agreed upon, to use the histograms created by HepMCAnalysis (see Section 12.2.2) for validation. In close communication with the generator validation coordinator the functionality of JEM has been extended by the methods described in Section 10.2. JEM Live Monte Carlo Validation has since become an integral part of generator validation in ATLAS.

10.3.2. Example of an erroneous Monte Carlo production

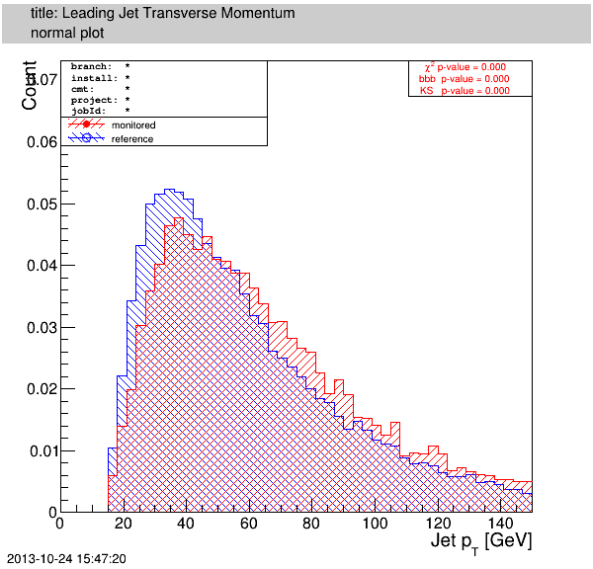
To give one example of an erroneous MC production, a sample produced in May 2013 is presented. Figure 10.1 shows the DCube comparison plots of two datasets: mc_1 as the reference set in blue and

mc_2 as the faulty dataset under inspection in red. As these were no official validation samples, the event count differs from the recommended 100,000. The second sample mc_2 has been described as having a broken b-parton shower by having cuts, that were set to high for b-partons. The details of the two datasets are shown in Table 10.1.

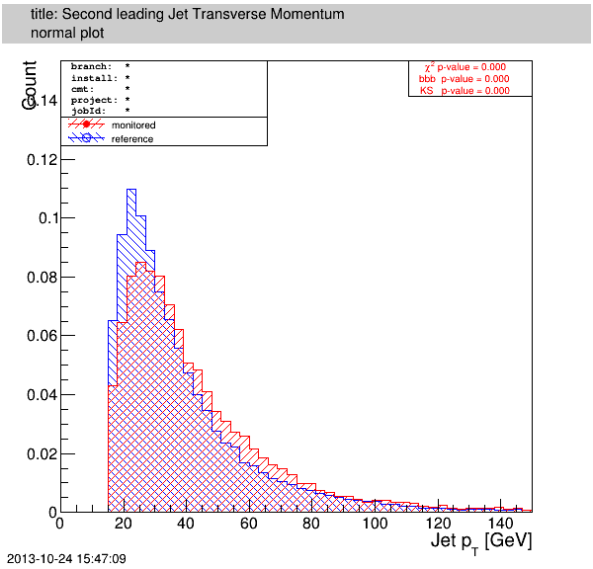
While the error itself is not to be discussed here, it resulted in a serious derivation from the chosen reference sample mc_1, in which the b-parton shower is implemented correctly. The difference is clearly visible both to the human eye, as well as to the statistical tests that were performed. This example was given by the MC working group convenor to illustrate the need for automatic validation as the "problem was easily detected after one minute... once somebody looked, 2 months later"[2].

For the faulty dataset 250,000 useless events were generated, simulated and reconstructed, which resulted in wasted resources. Also, an unknown amount of analysis users used this sample and had to check and possibly recalculate their physics results. Therefore, not only time and resources are wasted, by recreating the needed data, also the wrong usage in users analysis binds manpower and generated unnecessary friction.

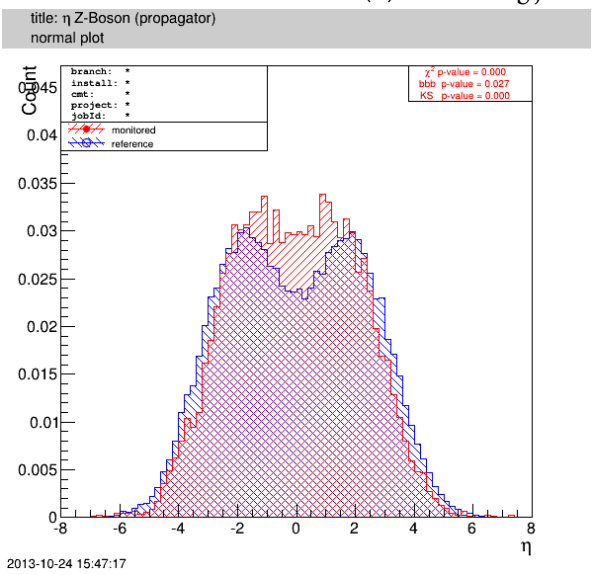
[2] Dr. Thorsten Kuhl, June 2013, MC Working group convener



(a) Leading jet transverse momentum.



(b) 2nd leading jet transverse momentum.



(c) Z-Boson propagator.

Figure 10.1.: Key plots, which show a clear discrepancy between the two datasets mc_1 (blue) and mc_2 (red).

11. Activation Service

Central to any automated activation of JEM in Grid jobs is the JEM ActivationService [63], which is described in detail in this chapter. The motivation to implement this service and the problems it has been designed to address are explained in Section 11.1. The JEMstub as the central pilot plugin (see 3.3.2), responsible for communication with the ActivationService (see Section 11.3), is introduced in Section 11.2 followed later by a description of rule based request evaluation in Section 11.4.

The rule evaluation engine has been refactored. Instead of a static rule base, being instantiated and evaluated in every activation request from the web front end of the ActivationService to a separate process, described in Section 11.5. The next two sections, 11.6 and 11.7, show in what way JEM instruments selected tasks for the desired validation or monitoring purposes and how match criteria can be stored in the database to create appropriate rules once the criteria are matched by new tasks. Finally, Section 11.8 sums up the developments, done for the ActivationService, and put them in a critical perspective. An overview diagram showing all relevant ActivationService systems and interactions can be found in Figure 11.1.

11.1. Motivation

As explained in Section 4.3, JEM started as a user centric job monitoring system, which sent its monitoring data back directly to the Grid user, who had launched the job. With the implementation of a central ActiveMQ messaging server to collect monitoring data and store them in a central database, the necessity to control access to these resources arose. This could have been realized by simply limiting access to the messaging server, however, this would not have resulted in a well working solution for all users, monitoring jobs on the Grid, as the messaging server does not determine, which messaging streams to block and which to allow. Also, even a very small fraction of all Grid jobs being instrumented with JEM creates a significant load on the central JEM systems as multiple services like the ActiveMQ server, the ChunkConverter and the web front end heavily use the database. To control access to these resources a centrally controlled approach has been implemented, which in the meantime has proven to be a precious tool for further use cases (see Section 4.2).

Once the ActivationService was available it became possible to extend JEM's monitoring capabilities to production jobs, which was not practically feasible beforehand, as the complex system, which feeds the production jobs into the Grid could not have been easily upgraded to launch a monitoring job wrapper instead of the payload job itself. As production tasks usually contain hundreds or thousands of jobs, monitoring every single one of them would, depending on the use case, probably not provide as much additional information as only monitoring a small subset. Nevertheless each JEM monitoring instance requires Grid resources that need to be preserved and activating job monitoring for too many jobs would significantly reduce Grid performance.

In summary, the ActivationService has been designed with several different purposes in mind:

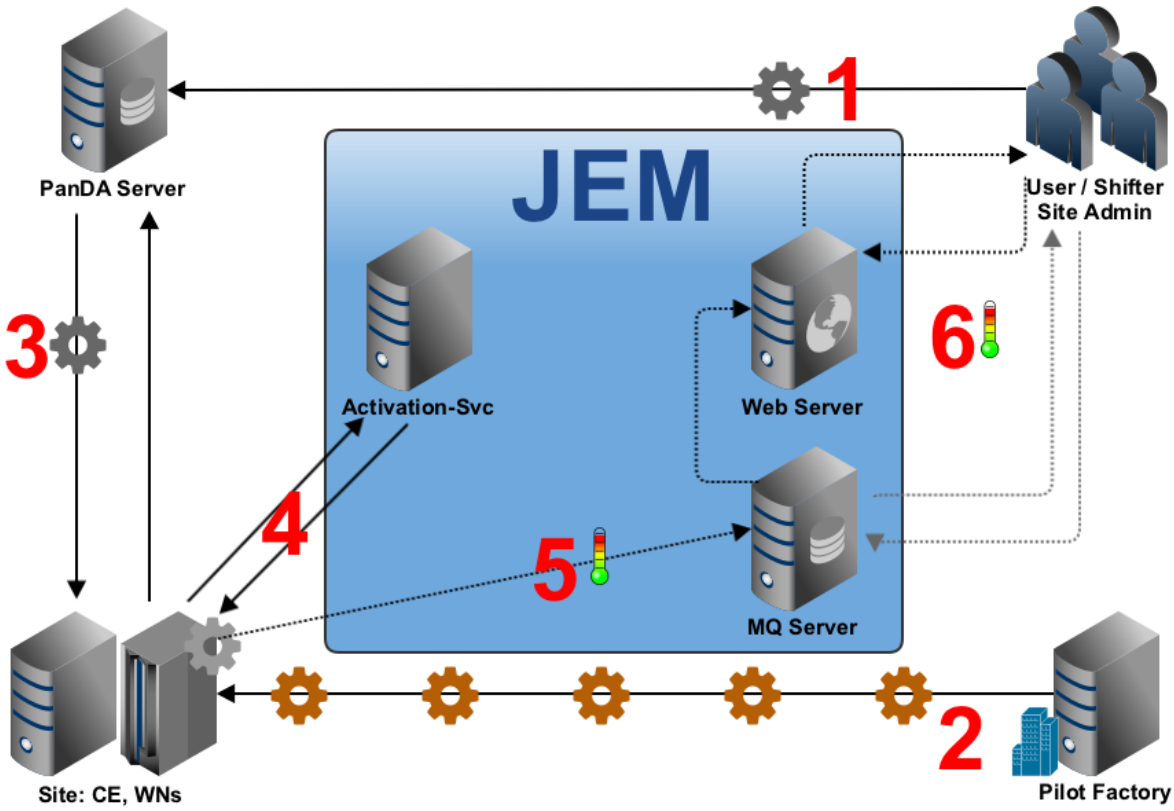


Figure 11.1.: Overview of the JEM ActivationService. In (1) a user submits a job to the Grid, which gets picked up by a pilot in (3), which itself was launched by a pilot factory in (2). In (4) the ActivationService is queried and upon a positive answer, the job can be instrumented with JEM, monitoring data can be transmitted via ActiveMQ in (5) to the JEMServer and in (6) the job's user can inspect the monitoring data. Plot produced, using [87, 88].

```
from JEMstub import updateRunCommand4JEM
cmd = updateRunCommand4JEM(cmd, job, jobSite, tolog, metaOut=metaOut)
```

Listing 11.1: JEMstub public interface to PanDA pilot.

to easily invoke JEM's capabilities from a central location, to control and configure the instrumentation of Grid jobs with JEM to protect the JEMserver infrastructure from too many simultaneously running, instrumented Grid jobs.

11.2. The JEMstub pilot module

As described in Section 3.3, ATLAS uses a pilot based, centralized pull method to distribute jobs in the WLCG. After the pilot has set up its local environment it queries the central PanDA database for any free jobs matching the information sent by the pilot.

This implies, that the information at which specific computing element a certain job will be running is only known to the PanDA system. After a pilot, running on a worker node of this computing element, has queried the central PanDA server for this job and the job data has been brokered to this pilot. This fact puts some serious time constraints on time critical activation of validation tasks which are further discussed in 11.6.1.

In 2010 a small plugin module has been added to the PanDA pilot, which is responsible for querying the JEM ActivationService via HTTP: the JEMstub [89]. This module consists of a very minimal interface (see Listing 11.1), where the JEMstub module gets the command line parameters that the pilot would launch as a string and returns itself a modified command line, which consists of all parameters necessary to launch JEM instead of the original payload command. Additionally, the `updateRunCommand4JEM()` method gets the pilots `job` and `jobSite` objects to extract metadata about the payload job and the current site for the ActivationService query as well as a `tolog()` callback method to use the pilots logging facilities. `updateRunCommand4JEM()` then returns the modified `cmdline` string, which should be executed by the pilot in case the ActivationService allows JEM instrumentation of this job. This method has an extra `metaOut` parameter, where a dictionary gets passed, where `updateRunCommand4JEM()` can set answer data from the ActivationService for the pilot, which in turn gets stored with the job metadata in the central PanDA database. This return data informs the pilot whether JEM was activated or not. In case of a denied request, the reason is given and is available from the panda job overview web page.

Once called, `updateRunCommand4JEM()` tries to extract the command line arguments that are meant for JEM (if they are present) and will interpret them (seeTable 11.1). These command line parameters are usually only present if the job is a user job, meaning it has been created and entered into the PanDA system via *pathena*, *prun* or Ganga by a physics analysis user[1].

Two command line parameters, that have originally been designed to control JEM activation behaviour before the ActivationService was introduced are specially parsed by the pilot (see table: Table 11.1). The `--enable-jem` parameter is passed forward as a `userRequestedJEM` field to the ActivationService. Special `UserRules` (see 11.4.3) can decide upon this information and

[1] It however can as well be used by a production user, although it is quite difficult to transport all necessary parameters through the various stages of the production system (see 12.3.1 and [90]).

Table 11.1.: JEM command line options.

command line argument	explanation
`--enable-jem`	enable JEM wrapping with standard worker node configuration
`--jem-config`	a semicolon separated list of key value pairs to configure JEM (further described in [91])

the key-value pairs from the `--jem-config` parameter will be exported to the environment. JEM's configuration infrastructure layer will then interpret these (see Table 11.1).

From there the ActivationService is queried via POST method[2], which will evaluate the jobs request based on a rule engine, which is described further down in Section 11.4. If the ActivationService has returned a non negative answer (see Table 11.2), a JEMstub object is created, which gets all parsed user parameters and the answers of the ActivationService and will then construct the command line replacement.

To ease use and to prevent command line bloating by having too many too long key-value pairs in the `--jem-config` parameters and in the response of the ActivationService some of the most common parameters are coded into shorter ones [91]. To parse these correctly, the JEMstub downloads the zipped JEM library tarball from the JEM software repository which contains all relevant Python files, including the startup `JEM.py` script to launch JEM, which is then extracted into the pilots local working directory. The repository URL and the JEM version to be used are also specified either by special command line parameters or by the answer of the ActivationService.

In this downloaded JEM library a module, which is then able to parse and decode the shortened command line options, is included so that these options can be stored properly in the process environment and JEM can be configured accordingly. The old command line given by the pilot will be written into a special shell script which is made executable. As a pilot sometimes runs several subjobs, the payload containing shell script has a timestamp in its filename to avoid, that earlier versions of `payload.sh` get overwritten, which happened to be a problem once multijobs[3] got instrumented with JEM.

The command line replacement consists of call to `JEM.py`, using the pilots Python version, with a special `--payload` parameter which gets the payload shell script filename as executable. When all of this has been executed by `updateRunCommand4JEM()` without an error the command line replacement will be returned to the pilot. It will, after some further pilot related setup work, finally launch JEM, which then will in turn launch the original payload as its subprocess. Since there is no command line option to explicitly disallow the usage of JEM, the ActivationService will be queried for every call to `updateRunCommand4JEM()` and its answer will finally decide whether JEM will be activated or not.

Additionally the JEMstub module provides a static method, which the pilot calls at the end of each job: `notifyJobEnd2JEM()`. This method gets the job metadata dictionary of the pilot as a

[2] Timeouts are selected very strictly and carefully to rather miss an ActivationService response than to block the pilot execution time for too long.

[3] Multijobs are jobs where the pilot launches several production jobs under the same PanDA job id.

Table 11.2.: ActivationService answering commands.

rule result string	JEMstub action
`YES`	JEMstub will enable JEM with the given configuration from the ActivationService response
`NO`	JEMstub will not enable JEM
`DONTCARE`	JEMstub will not enable JEM unless `--enable-jem` has been set by the user

parameter and extracts all necessary information the ActivationService needs to mark this job internally as finished. It is not guaranteed, that this method is called every time a job is finished as the pilot can crash, the batch system can terminate the batch job and with it, the pilot, network problems might occur or the worker node might have a power down event. Therefore the ActivationService needs to have fallback methods, as described in Section 11.3 to finally decide, that a job is finished and the internal accounting can be updated appropriately.

AGIS provides a parameter on a per-site basis called `allowJEM` which specifies if the pilot should query the ActivationService for jobs running on this site. This parameter has been added so that individual sites can prevent the usage of JEM in the case that JEM would not work properly on this site or security considerations might apply. One reason could be, that a restrictive firewall does not permit outbound HTTP or STOMP connections which would block most of JEM's use cases. Also this parameter can be changed in bulk for all sites to disable JEM for the whole Grid should misuse be detected or severe problems arise.

11.3. Activation Service web front-end

The ActivationService web front-end is the receiving part of the activation request, sent by the JEMstub as described above. It delegates the request evaluation to the ActSvcRuleEvaluater, which is described in Section 11.5, and returns the result to the HTTP request.

Internally the ActivationService updates counters in the Memcachd, which keep track of the number of running jobs per site and per user, which is necessary for the user rules and site rules, which are explained further down in 11.4.3 and 11.4.4. The kept counters are shown in Table 11.3.

As mentioned previously, the pilot notifies the ActivationService, once a job is finished. With this information the ActivationService can decrease the necessary counters in the Memcachd to keep the internal counters balanced. Once a JEM instrumented job is no longer sending data, without receiving a job end notification, the ActivationService will mark the job internally as looping. This can happen, for example due to network problems.

If a looping job is not marked as running again after a fixed time out period the active jobs counter in the ActivationService is decremented so that new jobs can be instrumented with JEM and the job is marked as finished without a known exit code. This is done by special ServicesWorker tasks that constantly keep track of all monitored jobs and the last update of the internal metrics. If a job is actively monitored and sending back data via the ActiveMQ the ChunkConverter updates a

Table 11.3.: ActivationService Memcachd counter.

Memcachd key	value and explanation
`running`	counter of all running jobs
`monitored`	counter of all currently monitored jobs
`<siteName>_running`	counter of all currently running jobs on site `<siteName>`
`<siteName>_monitored`	counter of all currently monitored jobs on site `<siteName>`
`<userdn>_running`	counter of all currently running jobs on site `<userdn>`
`<userdn>_monitored`	counter of all currently monitored jobs on site `<userdn>`
`knownSites`	comma separated list of all known sites
`knownUsers`	comma separated list of all known userdn
`<jobid>`	job metadata, including ActivationService request time

`lastSeen` field in the monitored job data object representation in the database.

11.4. Rule based request evaluation

The instrumentation of jobs with JEM by the ActivationService has to follow several policies. These are implemented in various rule classes, that are evaluated by a rule engine, which is itself implemented in the ActSvcRuleEvaluater. The result of a rule base evaluation for a given request gives an answer, whether this request should be accepted or denied.

The rule base keeps its rules in several dictionaries: `siteRules`, `userRules` and `taskRules`. These rule classes are evaluated in a special order, which represents the precedence each class takes over the following (see Figure B.2). Rules that are evaluated later can change the result of previous evaluations if the current rules priority is higher than the previous rule results generating rule priority. The `globalRules` list contains rules like the `JEMMaxMonitoredRule` which puts a fixed global limit on the number of jobs that are marked as instrumented. Other, finer grained rules that limit usage of certain JEMserver subsystems are also possible, for example for the LogFileSaver (see 7.1) or the *BacktraceMonitor* (see Part III).

11.4.1. Rule evaluation result

Each individual rule evaluation returns an instance of the type `RuleResult` as described in Listing 11.2. A rule evaluation result instance specifies in its member variable `self.activate` how the JEMstub (see Section 11.2) should react (see Table 11.2 for details). Each rule result object keeps a reference to the rule that produced this result in `self.rule`. Therefore further rule evaluation results can be compared against each other by deciding, which rule has the higher priority.

In `self.config` each rule can set a configuration string, that gets evaluated by the JEMstub and is there passed down to the JEM instance to be started. This allows a high amount of configurability of JEM on the worker node besides the default settings. For informative purposes, each rule should also set `self.reason`, which gets displayed in the pilot log. If another JEM version than the default version, `latest`, should be used, for example `dev`, a rule can set a version to be used in `self.version`.

```
class RuleResult:
  def __init__(self, activate = ns.DONTCARE, reason = "", rule = None,
      version = 'latest', config = ''):
    self.reason = reason
    self.activate = activate
    self.rule = rule
    self.version = version
    self.config = config
```

Listing 11.2: RuleResult class member.

11.4.2. Task rules

In the rule base, task rules are stored in a dictionary, the TASKID acting as reference key. As a small but nearly constant look up time is desired, task rules are checked first before any user rules or site rules apply.

The current job's TASKID is extracted from the request and is then checked against all known TASKIDs in the rule base. As task rules have been implemented for the Live Monte Carlo Validation use case (see 4.2.4), more than one validation rule per task does not provide any additional benefit, since all the jobs of a production task only differ in their statistically spread input data.

If a task rule has been successfully evaluated the rule result is returned immediately. This reflects the maximum precedence, task rules take over any other rule, even over rules protecting the JEM infrastructure. This can be assumed, as task rules are designed and used in such a way that none of the usual JEM monitoring functionalities are used for Live Monte Carlo Validation and, therefore they do not block resources, that are normally used by traditional job monitoring.

Should a use case arise, in which Live Monte Carlo Validations are being used side by side with JEM live system monitoring capacities, the rule evaluation strategies would need to be redesigned. To provide the High Level Trigger (HLT) group with insight into log files (see 4.2.5) of their jobs a more general task rule, the *TaskConfigRule*, has been created. This rule class allows it to send an arbitrary JEM configuration string to instrumented jobs of the specified task, without creating a VALIDATIONTASK and invoking all the validation related activities.

Currently only one task rule per TASKID is allowed. This could, however, easily be changed as the lookup value to the TASKID key can be changed to a list of rules that then gets evaluated in order should any such use case arise.

11.4.3. User rules

The `userRules` dictionary holds all rules relevant to the user running this job in a tuple. In this context a user is defined as the string, that is part of the jobs USERDN, which is sent by the pilot as part of the jobs metadata to the ACTIVATIONSERVICE (see Section 11.2). As a USERDN usually contains more information than the simple user name like the institute or the country, user rules could also be used to control JEM activation based on these criteria. Originally this was designed to specify JEM for certain analysis users that want to monitor their Grid jobs.

These rules are mainly used to grant JEM and pilot developers a constant positive answer for developing purposes. As there is currently no functionality to change this `userRules` list dynamically

during runtime, changes have to be entered manually by JEM administrators on dedicated requests.

The dynamic rule base described further down in 11.5.3 has been added to serve the dynamic nature of task rules and Live Monte Carlo Validation. These systems could as well be used to keep `userRules` in a dynamic state. This, however, has not been done yet and remains a project for future expansion of JEM.

11.4.4. Site rules

To define specific JEM activation behaviour for sites, a queue name of a specific site and its site name have to be given. Together they build a tuple that acts as the key in the site rules dictionary of the ActivationService rule base. For each of these entries, multiple site rules can be specified.

The general idea behind site specific rules is, that a small percentage of running jobs is to be instrumented with JEM to supplement local site monitoring tools with job runtime behaviour.

Site rules usually only specify that a certain small percentage of all running jobs for one site should be activated or that a small fixed number of jobs should be running with JEM enabled. These site rules can also be used, to prevent usage of JEM at whole sites if security considerations have been raised. This behaviour can, however, also be controlled centrally by the ADC operators using the `allowJEM` switch implemented in AGIS, as described in Section 11.2.

A possible application could be to use JEM as additional data source for local site performance monitoring as described in Section 4.2.

11.4.5. General rules

Several general rules exist, which get evaluated to check the internal resource usage of the JEM services to prevent over saturation of the available physical resources. These rules include the *JEMMaxMonitoredRule*, which vetoes further job activation if the `monitored` counter, kept in the Memcachd, exceeds its preconfigured value. Operational experience has shown, that, on the currently used hardware, the JEM services can handle up to 100 parallel monitored jobs, which submit monitoring data via ActiveMQ, that the get stored in the monitoring database.

Additionally there is a prototype of the *JEMDatabaseLoadRule*, which vetoes further job monitoring activation, if the `dbload` value in the Memcachd exceeds its configured threshold value. The *JEMDatabaseLoadRule*, however, never was used in production, as the gathered database load values were to erratic and unstable, compared to assumed load from actual current usage.

11.5. ActSvcRuleEvaluater

The ActSvcRuleEvaluater has been designed and implemented as a command execution engine with a synchronized access to the ActivationService rule base for modification purposes. The main idea was to centralize the data state holding of the ActivationService as it became more and more difficult to keep a consistent rule base state in the Django web front end. The issue arose, when more types of dynamic rules were added and as the rule base needed to be dynamically altered during runtime.

Also the static rule base needed to be loaded from scratch every time a new request reached the ActivationService HTTP endpoint, as each time the web server launched a WSGI process running

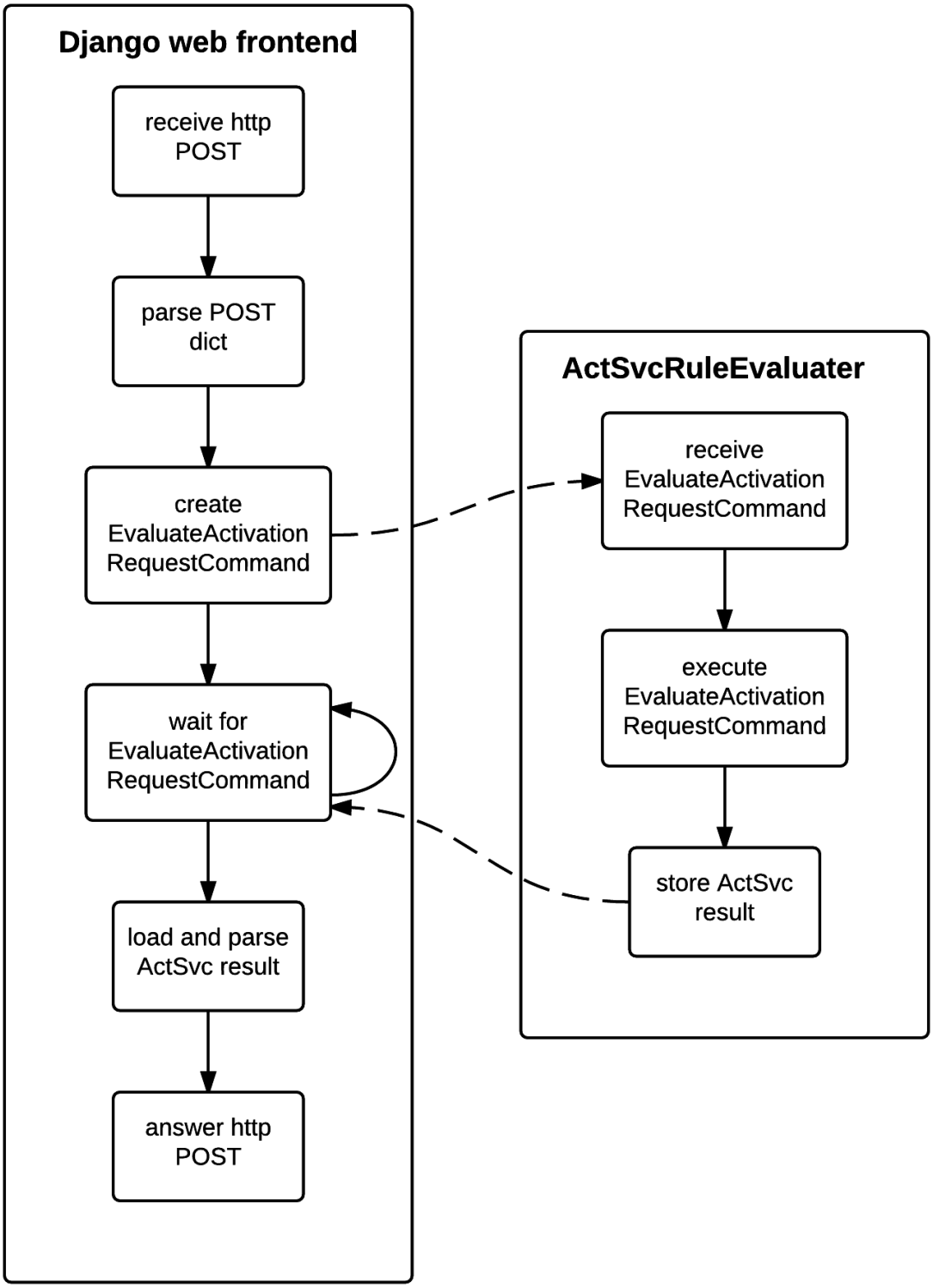


Figure 11.2.: JEM ActivationService overview (components and call paths). Produced with [92].

```
class Command(object):
  def __init__(self, requiresRuleBase, data = None, threadedExecution =
      True, memcacheKey = None):
  def createFromJson(jsonStr):
  createFromJson = staticmethod(createFromJson)

  def toJson(self):
  def execute(self):
  def _executeThreadWrapper(self):
  def send(self):
  def waitForJson(self):
  def setRuleBase(self, ruleBase):
  def requiresRuleBase(self):
  def isThreadedExecution(self):
  def isFinished(self):
```

Listing 11.3: Command object interface.

the Django web front end code. This was a very inefficient way of handling rule evaluations, especially considering, that more and more complex rules would slow down each request even more as they needed to be initialized for every request. In Figure 11.2 an overview of the data flow between the various subsystems is shown.

The ActSvcRuleEvaluater runs as a JEM working mode and benefits from the configuration management and the logging facilities of the `JEM.PY` infrastructure. Internally the ActSvcRuleEvaluater is a parallel command execution engine where each incoming command is received, parsed and then executed as thread. On start up a POSIX datagram socket is created at a well known location in the JEMserver file system[4] and any foreign process can then send commands, serialized as JSON datagrams to the ActSvcRuleEvaluater.

11.5.1. Command objects

The commands being interpreted by the ActSvcRuleEvaluater need to be derived from a common `Command` base class, see Listing 11.3. This base class provides the basic methods for commands to send themselves to the ActSvcRuleEvaluater as a JSON string and to be reconstructed from this string back to Python objects. During the initialization several values are stored in the command object:

`memCacheKey`
: MemCacheD key to the command result data once the `execute()` method is finished

`data`
: a dictionary providing arguments for the commands execution

`isFinished`
: initially `False`, it is set to `True` upon exit of the commands `execute()` method[5]

[4] usually `/tmp/ActSvcEval.sock`

[5] This is needed for synchronization purposes on rule base changes.

Table 11.4.: `Command` class overview.

Command	functionality
`ActivationServiceHit`	a pilot has queried the ActivationService and is waiting for a response
`ActivationServiceEndHit`	a Grid job has finished and the pilot notifies the ActivationService about this, together with the exit code and other job metadata
`AddRule`	a rule is to be added dynamically to the rule base of the ActSvcRuleEvaluater. Further command execution will be synchronized on this command.
`RemoveRule`	a rule is to be removed dynamically from the rule base of the ActSvcRuleEvaluater. Further command execution will be synchronized on this command.
`GetRuleBase`	a JSON excerpt of the rule base will be created and written to the rule's MemcachD answer key

`requiresRuleBase`
: tells the ActSvcRuleEvaluater if the command needs a rule base for its proper execution

`threadedExecution`
: if set to `False` the ActSvcRuleEvaluater will wait for all other running commands to finish before executing this one

Initially at the `memCacheKey` location the string 'empty' is stored in the Memcachd. This provides a default value to read should the command execution have failed. Together with the class name of the current command object, the `memCacheKey` value and the data dictionary are stored in a dictionary, which is then transformed into a JSON string in the `Command`'s base `toJson()` method. This is then the marshalled string representation of a `Command` object, which is then transmitted as a POSIX socket datagram to the ActSvcRuleEvaluater. As the socket communication is unidirectional, the Memcachd location is used as the back channel to transport the command result back to the originating process.

Of the above list, only the first two values are stored in the JSON string, once the `Command` is sent to the socket. All other values are either only relevant during or after command execution,like `isFinished`, or are implicitly known on command reconstruction time, as they are hard coded into the constructor call of the derived `Command` class, for example `requiresRuleBase` and `threadedExecution`. At the ActSvcRuleEvaluater the received command strings are unmarshalled using a static method of the base `Command` class: `createFromJson()`. It accepts a JSON encoded string representation of a Command and returns a Command object, which then can be executed by the ActSvcRuleEvaluater. The commands that are currently implemented are described in Table 11.4 and adhere to the interface described above.

11.5.2. Command execution

After a `Command` has been properly reconstructed, the execution engine sets the rulebase via the `setRuleBase()` method and checks for the commands synchronization options via `isThreaded-Execution()` before executing the `Command`. Also each `Command` object is stored in an internal bookkeeping list that gets regularly cleared. For this, each `Command` objects's `isFinished()` method is checked and upon `True`, the `Command` is removed from the list.

If a command requires no synchronization, meaning that no rule base change is anticipated, the execution engine will execute the command in a thread. Otherwise the ActSvcRuleEvaluater waits for all currently running `Command`'s, by regularly checking the internal bookkeeping list and only then executing the synchronized `Command` when no further `Command`'s are running anymore.

As described above, each `Command` has a `memCacheKey` member variable, where each individual `Command` may store results to be transmitted back to the originating sender of the `Command` object. The data format of this response field is `Command` implementation specific, but is usually a JSON marshalled Python dictionary, containing for example the rule evaluation result for an `Activation-ServiceHitCommand`.

The sending process, which originally created the `Command` object, may call the `waitForJson()` method to get the response JSON from the ActSvcRuleEvaluater. This method internally polls the MemCache at the `memCacheKey` location and returns the result once the default value of '`empty`' has changed.

11.5.3. Modification of the Activation Service rule base

To modify the ActivationService rule base, several constraints have to be considered. As request evaluations, especially of task rules, require a change in state, modifications of the rule base needs to be synchronized to prevent inconsistencies.

For this, specially marked rule base altering commands, like the `AddRuleCommand` or the `Remove-RuleCommand`, signalize the ActSvcRuleEvaluater to wait with further command execution until all other currently running commands are finished, then execute the special rule and then, finally, resume normal operations. During this waiting time newly arriving commands are kept in the system queue of the POSIX socket and will be read by the ActSvcRuleEvaluater once the synchronized command is finished.

During the synchronized modification time, rule base evaluations, which are also handled by commands, are suspended. This will result in a small increase of waiting time for further incoming `ActivationServiceHit` command object, this is, however, not a problem, as the changes in the rule base are faster than any timeouts of ActivationService requests on the PanDA pilot.

To prevent any loss of rule base state during a power failure or ActSvcRuleEvaluater process crash the rule base is periodically and after every rule change, written to a backup file on disk. This file is read at any ActSvcRuleEvaluater initialization to load existing rules if they are present. This can be prevented by setting a command line option during start up so that the ActSvcRuleEvaluater starts with a clean default rule base.

11.6. Requesting JEM for specific tasks

If one wants to directly validate specific physics processes, which run in the Grid, using JEM, the corresponding tasks need to be instrumented with JEM and modified in such a way that the needed quality histograms are being created and transmitted to the JEMserver (see Section 12.3). To do this, the ActivationService needs to know at the time it gets queried by a job of a desired task that is to be instrumented, that this job should get a positive answer. For this, there needs to be an appropriately configured `TaskRule` indexed with the correct taskID in the rule base of the ActivationService.

`TaskRules` are added to the rule base dynamically by using `AddRuleCommands`, which are usually created by the web front end in response to action taken by a registered validation user (see Chapter 14). These rules can, however, be as well created from the JEMserver administrator, using a Django shell. This might, for example, be in response to a request from an interested party, that needs MC validation but is not added to the system as an authenticated user.

Another method to validate tasks that have been finished already is described in Section 12.6, where the existing result files are downloaded and the quality histograms are created locally by an appropriately configured Athena run.

The practical problems of getting to know the taskID in time to enter it into the system before too many jobs have already started running is discussed in 11.6.1. A solution for this problem is then introduced in Section 11.7. The technical details of the web front end are explained in detail in 14.2.2.

11.6.1. Time constraints

To instrument a production task with JEM a `TaskRule` (see 11.4.2) needs to be created and for that the taskID has to be known at the time of the `TaskRule` creation. During tests of the Live Monte Carlo Validation is has been observed, that this is quite difficult to handle manually as the taskID for a given task is only known after the production system has processed the request and the PanDA system has picked the task and its jobs up from the previous stage.

Production system operators feed templates of the tasks to be generated into the production system using a web interface. From there on, the production system interprets these templates and generates configurations for the task and its jobs. These new production tasks are then delivered to PanDA, which can take an arbitrary amount of time, depending on the current work load of both systems. Therefore this time frame cannot be predicted reliably from an outside observer. If a lot of preprocessing needs to be done to generate huge amounts of jobs for lots of tasks, the production system might take some time to process the request. As there is no outside monitoring of the batch queues of the production system available, one has to guess when the processed task will be delivered to PanDA.

The practical problem now is, that the time frame between the taskID being created in PanDA and the jobs for this task being submitted to free pilots in the Grid varies a lot. It can be as small as a couple of minutes if the priority for the generated jobs is high enough. As the panda job and task monitoring systems do not really provide live views off the existing panda database but, due to internal caching, the presented data can itself be as old as the last cache refresh, which is usually in the order of 10 to 30 minutes. Even if one manages to get the right data it might still be too late as the data was already too old.

If the Grid at this moment has enough free resources and the priority of the submitted jobs is high

enough then these new jobs can be scheduled for immediate running and will be delivered to waiting pilots before it is shown in any monitoring system. This happens especially fast with validation tasks, which only contain 10 or 20 jobs and usually have the maximal priority allowed.

So in summary, the time between submitting a task request to the production system can be untransparently long and submission time to the Grid can be quite short. This makes it practically difficult for an outside JEM operator to catch the right moment to learn the taskId and enter the metadata of a task into the JEM system especially if the task scheduling happens outside of normal working hours of the current time zone. A solution for this problem is described further down in Section 11.7.

For the general production system user this described time delay is usually not a problem as there is no need to target any specific actions on the time period between tasks entering the PanDA system and their jobs starting to run in the Grid. Also these processing times in the Bamboo and PanDA systems are usually small compared to the actual runtime of the task in the Grid, especially if it is a large production task.

11.7. Requesting JEM for validation tasks by matching criteria

As described in Section 11.6.1 it is very difficult to manually find the correct time frame to instrument jobs of a desired task, especially if the task is very small and has a high priority, as dedicated validation tasks usually are.

This section describes a method to store a request for a task, that has not yet been started, in the database and regularly checks the PanDA system for new tasks, which then get matched against the existing requests. If a request is matched an `AddRuleCommand` for a `TaskRule` is created and configured with the requests data. Task requests, a schema is shown inTable 11.7, are added via the requestJEM web interface which is explained in detail further down in chapter Chapter 14 and are stored in the JEM ActivationService database.

Necessary data for this matching includes the amount of jobs to instrumented, respectively the amount of events one is interested in, the software tag and the dataSetId. In the case of ATLAS this minimal set of data is sufficient to reliably identify a desired production task.

11.7.1. TaskCacheRefresherTask

The task cache refresher task (TaskCacheRefresherTask) is a ServicesWorker task and regularly checks the PanDA Monitor web API [93] for all tasks in the states `running`, `pending`, `holding`, `submitting`, `waiting` and `submitted`. These tasks are either already running or in one of the preparatory stages before running, which makes their jobs possible targets for an instrumentation with JEM. Tasks in the states `failed`, `lost`, `aborted`, `obsolete`, `deleted` and `archived`, `finished` and `done` are not queried as they are already finished or in some sort of erroneous state that is unsuitable for JEM anyway.

The task data is returned in a JSON format together with other metadata that is mainly used by the PanDA Monitor page itself, which also queries this API internally as data source for the web pages it

presents. In the TaskCacheRefresherTask the tasks are parsed from the response and are either added to the task cache if they do not exist yet or the existing entry is updated.

Table 11.5.: *TaskEtagDsIdCacheEntry* schema.

field name	value explanation
ts	timestamp of last modification in JEM system
taskId	taskID
eTag	eTag of the task
dsId	dataSetID of the task
taskName	name of the task (as defined by dsId)
requestJobs	the number if requested jobs
doneJobs	the number of done (finished) jobs
state	the current task state
events	the total number if events of the task
priority	the PanDA scheduling priority of the task
project	the tasks associated project
physRef	the physics reference of the task
physGroup	the name of the tasks requesting physics group
pandaTs	the timestamp of the last task modification in the PanDA system
swVersion	the ATLAS software version being used for this task
validationPossible	a boolean indicating whether JEM Live Monte Carlo Validation instrumentation is possible

All tasks in the task cache are looked up by their `taskId` and their `state` field is updated accordingly. Together with all entries a `timestamp` is saved, marking the last change in `state` or `doneJobs`, so that old and finished entries can be purged after a timeout period of a couple of days. This period is configured by a command line parameter of the TaskCacheRefresherTask and prevents the task cache from growing too much in size as more tasks are added over time.

Resolving the `swVersion`

The `swVersion` and the `validationPossible` fields are not filled from the PanDA Monitor response as this data is not available there. The `swVersion`, however, provides crucial information to JEM whether a Live Monte Carlo Validation is possible with this task or not, as HepMCAnalysis needs to of a sufficient minimal version[6] to be run with current ATLAS software packages. The only information related to software versions is coded in the `eTag` field of the PanDA Monitor response. Therefore the JEMserver needs to decode the `eTag` and resolve it into an ATLAS software version and then check the HepMCAnalysis version available in the corresponding ATLAS software production cache.

[6] The minimal software version for HepMCAnalysis needs to be at least 00-00-85.

Table 11.6.: HepMCAnalysis version cache schema.

field name	value explanation
ts	timestamp of last change
atlasVer	version string of the atlas version
hepMCVer	the used HepMCAnalysis_i version

This is done by two ServicesWorker tasks, the *AMIQueryTask* and the *AMIQueryStoreResultTask*, which query the AMI database using the AMI command line client. As this AMI client is only available in a proper ATLAS software environment which does interfere with Django's internal logging facilities, the original task has been split in two separate task that communicate using JSON encoded files to exchange which data is to be fetched and what the results are. The *AMIQueryStoreResultTask* uses the results from the AMI query and stores the correct ATLAS software version in the *TaskEtagDsIdCacheEntry* entry and then checks the *HepMCAnalysis_iVersion* table, as described inTable 11.6 for the corresponding HepMCAnalysis version.

Only if this gathered version is greater than the minimal required version the `validationPossible` field in the *TaskEtagDsIdCacheEntry* is set to `True`. To keep the data in *HepMCAnalysis_iVersion* up to date, the *CheckHepMCAnalysis_iVersionsTask*, a ServicesWorker task, regularly checks all available ATLAS software caches. It correlates the HepMCAnalysis version used in this cache by reading the distributed CVMFS, which contains all ATLAS software caches.

11.7.2. RequestTaskCacheMatcher

The request task cache matcher (RequestTaskCacheMatcher) is a ServicesWorker task that regularly checks all unfinished MatchTaskRequest entries from the database against all *TaskEtagDsIdCacheEntry* entries in the JEM internal task cache. Task matching requests are stored in the database with the schema show in Table 11.7.

Upon running, the RequestTaskCacheMatcher gets all unfinished requests from the database, which means all entries where the `taskId` is set to `0` and `finished` is set to `False`. Then, for each of these entries, the *TaskEtagDsIdCacheEntry* table is filtered for the entries matching `eTag` and `dsId` to get possible candidates for fulfilling the request. These cache entries are then further filtered for all unfinished tasks, meaning all tasks not in the states `done`, `finished` or `failed`. The remaining candidates are then checked for the most not already done jobs, which is the difference between `requestJobs` and `doneJobs`. This difference has to be larger than the `requestJobs` field from the MatchTaskRequest entry, otherwise the request will not be matched as not enough events will be available. If more than one task candidate is found, the result list is ordered in descending order by the amount of not yet done jobs and the task at at the top of the list is selected as the most likely candidate to have all requested jobs instrumented.

MatchTaskRequest entries have a limited lifetime, configured by command line parameter of the RequestTaskCacheMatcher, of 5 days after which they are automatically deleted. This prevents old MatchTaskRequests to be checked over and over again if the targeted tasks for example never started or directly transitioned into an erroneous state. Validation operators who create

Table 11.7.: MatchTaskRequest schema.

field name	value explanation
ts	timestamp of entering into the task request database
taskId	taskID of the task the request has been matched to (0 on entering)
eTag	desired ETAG to be matched against
dsId	desired dataSetID to be matched against
requestJobs	number of jobs to be instrumented
finished	boolean to indicate if the request has been matched yet
referenceFile	foreign key to the , which is to be used for validation
config	the HepMCAnalysis configuration as JSON string, which is to be used by the worker node JEM instance

validation task requests need to actively follow their lifetime and eventually resubmit new ones once the old ones expire.

11.8. Summary

This chapter has shown how JEM activation on WNs is handled by several different actors: the PanDA pilot with its JEMstub module, the JEM ActivationService and the underlying rule base being evaluated by the *ActSvcRuleEvaluater* as well as how requests for task validations can be stored that later on automatically result in the creation of the proper *TaskRule*'s.

This instrumentation is centrally controlled and has to happen before the pilot launches the pilot job as JEM needs to run between the pilot and the payload job being monitored. The proposed method allows JEM to be configured very specifically for any use case, and in relevance to the job, which should be monitored.

The ActivationService could have been added as a submodule into Athena, which is the dominant framework, being used in ATLAS and therefore a large fraction of all Grid jobs are using it. This, however, would have had several disadvantages:

- As mentioned above not all Grid jobs are using Athena and some special jobs might use their own framework.
- Any scripts that are used around Athena, which are being used for setup or pre- and post-processing would not be able to be monitored by JEM. This is contradicting its original use case.

So, the PanDA pilot is the smallest common denominator: all Grid jobs, running in an ATLAS context are PanDA jobs and are therefore using a common infrastructure.

As it is now clear how JEM is able to identify the tasks, the following chapter shows, how the validation tasks are instrumented on the worker node, how the gathered data is transferred to the JEMserver, how it is then evaluated there and the results are presented.

In the future, the *RequestTaskCacheMatcher*'s functionality could be enhanced by adding a string representing a regular expression that tries to match the physics reference string of the a task cache entry. This way JEM validation users would not need to look up specific DATASETIDs but could, for example, validate all tasks with "Zee" in the task name, meaning $Z \rightarrow e^+e^-$ decays, against a small highly specific Z boson validation sample.

12. Task validation operations

As it is at this point now clear, from the last chapter, how individual jobs of interesting tasks can be automatically selected and then be instrumented with JEM, this chapter now focuses on one central part of the MC validation use case: the actual histogram generation and their comparison with a known set of reference files.

Before tasks and some of their individual jobs can be validated, a job needs to be prepared to produce the desired quality histograms, which is described in Section 12.3. To then evaluate the received results of validation task jobs, JEM uses several ServicesWorker tasks running on the JEMserver as described further down in Section 12.4. The following section 12.5 then describes how the gathered results are presented on the JEM web pages. An offline validation mode, the so called direct validation, is finally presented in Section 12.6, which explains how tasks can be validated using the JEM infrastructure once a task is already finished.

12.1. Motivation

As discussed in Chapter 10 the new generator validation procedures inside the ATLAS MC working group need to use clearly defined, automatic procedures. That is the reason why these procedures have been implemented as a collection of server processes on the central JEMserver. In summary, these processes are the core of the central use case, for which this generator validation project has been started. During the design time, it has been considered to move some of the validation evaluation procedures to the worker node, once the payload job is finished and has produced a quality histogram output file. This possibility, and why it was ultimately discarded, is later discussed in more detail in Section 17.4.

12.2. Use of external software

As described in [52], JEM uses an external software package to facilitate logging, Bash script monitoring or cryptography for the *SecurityService*. For the Live Monte Carlo Validation use case two additional external software packages are necessary, that are introduced in the following.

12.2.1. DCube

DCube [94] is a software package specifically designed to compare histograms of two ROOT [95] files which follow a common topological schema. This schema assumes a flat topology, where the root node contains several analysis directories, which themselves contain ROOT histograms. These are assumed to be named the same, both in a reference and a monitored files.

Internally DCube is XML driven, operating as much as possible on a XML data representation of the original ROOT file. It uses different statistical testing methods of ROOT to get measures of agreement between two histograms.

DCube is used by JEM to generate the comparison plots for validation purposes. It was considered to write a histogram comparison tool, that is specifically tailored to the needs of the presented use case. However, DCube proved to be stable and configurable enough to provide the needed functionality. The plotting methods of DCube have been modified, to better suite the needs of the validation use case. The plots now also contain a ratio plot under the comparison plot and are normalized by default, as the amount of events upon which two validation files are based can vary.

See further description in of DCube use in Sections 12.4.2 and 12.4.5.

12.2.2. HepMCAnalysis

HepMCAnalysis [39] is a framework, originally intended to validate MC generators and perform generator studies. It provides several benchmark physics processes that automatically fill ROOT histograms with the results. HepMCAnalysis has been designed to be robust, by having minimal dependencies on other ATLAS software packages, to be simple, by providing an easy to understand C++ API, and to be scalable, by being readily extendable with third party analyses.

The tool is able to be run with Athena and can be added as post execution filter, which then analyses already generated generator output. It is used by JEM for reference file generation, both online and offline. See further description of usage in Sections 12.3.1 and 13.3.1.

12.3. Instrumenting validation jobs with JEM

To create the necessary quality histograms from a production job, the Athena payload job itself needs to be modified with the post production module HepMCAnalysis. As the quality histogram producing method runs as part of the payload job itself, the command line of this job needs to be modified before the payload jobs gets launched.

JEM uses its own configuration infrastructure (see Chapter 4), which parses commands from different sources. These configurations are usually used, in case of the worker node module, to specify which monitors are to be launched and how the monitoring data is to be sent back to the JEMserver. However, as shown in 4.1.1 the JEM worker node mode is very tightly integrated in the main `JEM.py` module as JEM makes every possible effort to ensure that the payload job gets launched, even if the JEM monitoring modules, for some reason, do not get started.

As the payload job script needs to be modified before the payload job gets launched, JEM is still in its bootstrapping stage and most importantly the quality histogram generation cannot be run as a JEM monitoring module, as for example the *BackTraceMonitor* (see Part III). Therefore the shell script, which has been created by the JEMstub module of the pilot, needs to be modified before the payload job can be launched.

12.3.1. Modifying the payload job script

Athena provides several possibilities to be modified and configured, so that the desired code is run. Usually Athena gets a so called `jobOptions.py` Python file that is imported by the Athena main

```
postExecStr = "postExec="
strList = [
  'from HepMCAnalysis_i.HepMCAnalysis_iConfig import HepMCAnalysis_i',
  'myAna=HepMCAnalysis_i(`HepMCAnalysis_i`,file=`%s`)' % validationOut
]
confDict = simplejson.loads(conf.validationConf)
for hepMCAna in confDict:
  strList += ['myAna.%s=%s' % (hepMCAna, confDict[hepMCAna])]

strList += ['from AthenaCommon.AlgSequence import AlgSequence']
strList += ['topAlg=AlgSequence()']
strList += ['topAlg+=myAna']

for appendStr in strList:
  postExecStr += "'%s',," % appendStr
# remove trailing ,,
postExecStr = postExecStr[:-2]
# add '\n'
postExecStr += '\n'
```

Listing 12.1: Creation of the Athena `postExec` parameter.

process and itself includes several, physics process specific, Python configuration files.

To instrument the payload Athena process with HepMCAnalysis one needs to ensure that the original process runs undisturbed and only when all output files have been created normally, HepMCAnalysis is called and works with these files. For this purpose Athena provides a special command line parameter: `postExec`.

The Athena `preExec` and `postExec` parameters require Python code that is then executed before or after the main Athena job options file. As the parameters are given in a shell environment, the Python code characters, that have special meaning for the shell interpreter, like escape characters, string delimiters and line breaks, need to be masked with a special syntax [90].

For production jobs the Athena command line is usually constructed by the production system before being transmitted to PanDA and from there on to the pilot where it is finally executed. Therefore the `pre-` and `postExec` parameters need to pass two database systems[1] which additionally impose restrictions on usable characters due to their respective SQL syntaxes.

Listing 12.1 shows, how the `postExec` parameter is built in the JEM code. `validationOut` contains the filename of the desired output file, which has been determined earlier and usually adheres to the following schema: "`EvGen_HepMCAnalysis-<jobId>.root`", where `jobId` is the PanDA id of the current job. The configuration parameter `conf.validationConf` contains a JSON encoded dictionary of the HepMCAnalysis configuration, which is to be used.

Once the `postExec` parameter has been built, the `payload.sh` file needs to be modified. To do that JEM searches for the line containing the call to the Athena executable and appends the constructed `postExec` string. As Athena allows more than one `postExec` parameter to be used, the appending at the end of the command line ensures, that the JEM `postExec` parameter gets executed as the last one, therefore allowing previous post execution work to finish normally. After

[1] the production system database and the PanDA central database

this payload file modification is finished the payload job is launched and the usual JEM bootstrapping process continues as usually.

12.3.2. Indirect job output validation

There is also a prototype of a different quality histogram creation mode, which is internally dubbed as *indirectValidation*, in which HepMCAnalysis is not appended to the original payload job. Instead a second, small Athena payload job is started after the original payload job has finished successfully. This second job is set up in such a way, that in a subprocess first an ATLAS software environment is set up, suitable for the desired HepMCAnalysis version is run. After that an Athena process with only the configured HepMCAnalysis as job option is launched. The HepMCAnalysis of this job is configured to use all files in the local directory that match the the output file pattern[2] of the original payload job.

This has been added in case the desired version of HepMCAnalysis is either not available in the currently set up ATLAS software environment or there is a problem in HepMCAnalysis that would crash the whole payload job and therefore HepMCAnalysis can not be used.

12.3.3. Sending back result files

Once the payload job is finished and MC validation has been activated, either in its traditional use case or in *directValidation* mode, JEM checks for the existence of the output file. If the output file is found, it is opened, read by JEM and then internally encoded in base64. A HTTP connection[3] to the JEMserver is opened and the encoded file is sent via POST method.

Files ending on "`.root`" are being automatically deleted by the pilot. This prevents job result ROOT files from being stored in the log file dataset. Therefore, the quality histogram output file, which had been created by HepMCAnalysis earlier on, also gets deleted. JEM renames the result file by appending "`.jem`" to the filename and the file can then later be salvaged by accessing the automatically created job log files, which the pilot automatically creates after a job has finished. Should the HTTP transmission fail, the file can then be added manually to the JEM validation process by a JEMserver administrator.

12.4. Server side actions

On the server side, the JEM validation system uses DCube (see 12.2.1). To use DCube properly in the proposed environment several automated steps need to be conducted:

- a ValidationTask needs to be created (12.4.1)
- the DCube runs need to be prepared (12.4.2)
- quality histogram ROOT files need to be received and stored properly (12.4.3)
- received ROOT files need to be merged (12.4.4)

[2] Files, which contain the strings "`EVNT`" and "`root`" in their filename.
[3] `http://jem.physik.uni-wuppertal.de/JEM/validation/receiveHisto`

Table 12.1.: ValidationTask database schema.

field name	value explanation
`taskId`	the taskId or the internal validation taskId in case of direct validation(see 12.3.2)
`referenceFile`	the name of the reference file from the reference file cache to be used
`monitoredFile`	the name of the monitored file from the reference file cache to be used, if it is a direct validation task
`validJobs`	the number of jobs that are about to be instrumented (corresponds to the expected amount of individual quality histogram files to be received)
`validJobsFinished`	the number of finished jobs of this task (corresponds to the number of already received quality histogram files)
`compareTime`	timestamp of the last DCube validation run
`state`	current state of the validation task
`lastMergeAndValidation`	number of already merged and validated individual quality histogram files

- DCube needs to be run to produce validation results (12.4.5)
- the produced results need to be processed for the presentation (12.4.6)
- the produced results need to be presented (Section 12.5)

JEM internally uses a state machine to keep track of the progress of the different validation tasks. An overview of the states, their abbreviations and representations is shown in Table 12.2 and its transitions are described in Figure B.3. All validation tasks are initially entered into the database in the state ST.

As each of the three described service tasks require either directly ROOT or launch DCube processes that use ROOT, one has to make sure that an appropriate ROOT environment has been set up and the *PyROOT* [96] Python wrapper is available and working with the currently used Python environment.

12.4.1. Create a ValidationTask

Internally each task validation is represented by a ValidationTask object (see Table 12.1), stored in the JEM database. The `taskId` either represents the taskId, which means that individual Grid job results will be used to generate the monitored ROOT file, or it represents an internal taskId, which means that no Grid job had been run and the reference and monitored ROOT files have both been gathered from the reference file cache. This second mode is called *directComparison* and is described in detail in Section 12.6.

All actions modifying the ValidationTask are driven by a state machine on which the various ServicesWorker tasks, described further down react. The state transitions are visualized in Figure B.3 and the different states are shown in Table 12.2.

Table 12.2.: ValidationTask states.

state abbreviation	state name and explanation
ST	started
WG	waiting for DCube generation
GE	DCube generation
WF	waiting for files
WM	waiting for merging
ME	merging
WV	waiting for DCube validation
VA	DCube validation
FI	finished
ER	error

12.4.2. Preparing DCube

Before DCube can run any ROOT file comparisons, it needs to create a configuration XML file. This configuration file later on specifies the name of the histograms, the location of the histograms in the subsequent DCube comparison runs. Also the plot options[4] and the statistical tests to perform are specified.

At first the responsible ServicesWorker task, the *DCubePreparationTask*, checks, that DCube is available and that all necessary directories and symbolic links have been set up. Then the reference file is copied from the reference file cache location to the local working directory.

A newly created validation task starts in the state ST until the next *DCubePreparationTask* picks it up and starts the configuration file generation process while changing the validation task state to GE. Upon successful finish the state of the validation task will be changed to WF, indicating that this task is now waiting for monitored ROOT files. If the tasks notPanda field has been set to True then the state is automatically changed to WV as no files are expected due to a comparison of two already existing files from the reference file cache. To minimize waiting times, the *DCubePreparationTask* launches DCube as a subprocess and stores its PID and a timestamp in a local cache. On each run it checks whether one of the started subprocesses has finished since the last run. If a DCube run does not finish within a preconfigured time frame, it will be killed and tried again[5].

Once a successful finish of the DCube configuration generation subprocess is detected, the ValidationTask state will be set to WF, indicating that it is now waiting for files.

12.4.3. Receiving a quality histogram

Upon receiving a HTTP POST request at the JEMServer histogram receiving endpoint (see 12.3.3), the request is checked for the expected metadata: jobId, taskId, vFileName, vFileData and

[4] norm for normalized or logy for logarithmic y axis

[5] As there is no upper limit to the number of retries, one has to actively check the log files if a validation task does not leave the state WG.

vFileLength. If the corresponding ValidationTask object is existing, the state changed to WM, indicating that a new merging is to be done.

If the ValidationTask object has not yet been created, the received file is not stored and a HTTP 400 status is sent back. This prevents a denial of service attack on the hard drive space of the JEMserver handling the histogram receiving functionality, as anybody can send data to this HTTP endpoint and it would get stored without proper accounting.

12.4.4. Merging results

Upon each arrival of a new ROOT file from a finished validation task job, the file is written into the ValidationTask's local working directory by the Django web front end. After that the ValidationTask state is set to WM, indicating that it is now waiting for a merging operation if the state is not already set to WV or VA.

Every time the *MergeHistoFilesTask* runs, it checks for tasks in the state WM and changes these tasks state directly to ME, indicating that it is currently merging. It then collects a list of all ROOT files in the current ValidationTask's working directory. Exempt from this list are two ROOT files, namely reference.root, the reference file of this ValidationTask, and monitored.root, the merged file of possible previous merging steps, which gets overwritten with the next merging. All other files are then merged by recreating the topological structure of an already received ROOT file and by then merging each individual histogram.

The internal structure of the all ROOT files is the same as they all were created by HepMCAnalysis. As new analyses are added with newer versions, DCube will always use the smallest common denominator and use only the histograms, that are available in both files. If one reference file proves to be too old, it can always be recreated using the offline reference file creation mechanisms described in Section 13.3.1.

Although the merging process is quite fast, usually in the order of a couple of seconds, it overwrites the existing *monitored.root* file containing all previous merging results. To prevent an *DCubeRunnerTask* (see 12.4.5) from picking up a partially merged *monitored.root* the merging process itself uses its own state ME.

12.4.5. Running DCube comparisons

Similar to the *DCubePreparationTask* the *DCubeRunnerTask* is a ServicesWorker task that looks for ValidationTask in the states WV which are ordered for compareTime so that the tasks with the longest duration since the last run get processed first. It launches one DCube subprocess per run, which uses the generated configuration file, that has been generated by the *DCubePraparationTask* and the ValidationTask state is changed to VA indicating that a validation run is now in progress.

As DCube comparison run usually takes up to 3 minutes, each DCube run is launched as a child process and the PID is stored in an internal bookkeeping list. The task checks on each run if tasks exist in the bookmarking list and whether they have finished since its last run. Once all result files have been received and the actual DCube run was the final validation run after the final file merging and if a successful exit has been determined, the post processing stage is run and the ValidationTask state is changed to FI.

Otherwise the *DCubeRunnerTask* checks in the end after the DCube validation run, if the number of local ROOT files has increased beyond the ValidationTask's `lastMergeAndValidation` entry. If the number is greater, a new file has arrived in the meantime and `state` is set to `WM`, otherwise it is set back to `WF`.

12.4.6. DCube post processing

As DCube only provides a log file and an XML file containing the validation results, a post processing is necessary to transform the results to a data format that is easier to compute by the web front end. The Django template tag layer, which renders the result website only understands Python data structures[6]. Therefore, the log file `dcube.log` is parsed using several regular expressions and the results of the various histogram comparison tests are accumulated. Also, the filenames of the histogram and the difference plots are gathered, as DCube puts all generated plots in a `plots` subfolder and generates filenames using its own schema.

Various summaries of the results are generated as well, like the sums of successful and failed comparisons of all comparisons and one summary per analysis group. For each histogram a validation mismatch severity S is calculated out of all test results:

$$S = \frac{\sum_{\mathrm{t}} V_t \cdot w_{\mathrm{t}}}{\sum_{\mathrm{t}} V_t} \ , \tag{12.1}$$

S is the weighted sum of the individual statistical tests of each histogram comparison, where V_t is the value of the statistical measurement t and w_{t} the weight factor that is defined in the code, and whose current default values are set to 1 for each test. The calculated severity S gives a plain measure of the consistency of both histograms. Histograms are considered consistent and, if $S < 0.25$ and are represented in green. For $0.25 < S < 0.5$ a orange coded warning is given and values of S above 0.5 are considered problematic and should be reviewed. With S all histograms of an analysis group can be presented in an ordered fashion, having the most violated histograms first.

12.5. Presenting results

The main way of presenting the results of a ValidationTask is by rendering a website[7] with the results of the output generated by DCube. To do that, the generated JSON output of the post processing stage of the *DCubeRunnerTask* is transformed so that the validation result data can be stored in the context dictionary of the Django template render function. The reordering is necessary to have the data prepared for the rather limited logic of the Django Template Language [97].

The generated validation result page consists of four parts:

1. a metadata overview, containing (see Figure 12.1):
   - information about the internal validation task representation in the JEM system
   - links and information about the PanDA task which was validated

[6] see Section 12.5

[7] available via: `http://jem.physik.uni-wuppertal.de/JEM/validation/<taskId>`

- the reference file which was used

2. a table for all analysis groups inside the compared ROOT files, providing color code fields for a quick recognition of the results (see Figure 12.1)

3. all analysis groups in alphabetical order, where the most violated histograms, according to their severity as described in 12.4.6 (see Figure 12.2)

4. a summary table where detailed results for all individual statistical tests of each analysis group are displayed

If the website is accessed before the first validation run has been finished, only the information about the internal validation task representation in the JEM system is shown, together with the links to the '`reference.root`' and, if already available, the '`monitored.root`'. The result data is also available by directly downloading the validation result JSON file from the JEMserver[8].

12.6. Direct reference file comparison

If the physics case that needs to be validated against a reference already has a file representation in the reference file cache (see Chapter 13), there is no need to launch a special validation task in the Grid to run a JEM validation process. Instead one can simply enter a modified ValidationTask into the system, which copies all necessary files from the reference file cache to the working directory and then launches first the *DCubePreparationTask* and then immediately thereafter the *DCubeRunnerTask*.

As ValidationTask's working directories are named after the related taskId this new type of ValidationTask's needs its own taskId, which is not related to any taskId to avoid conflicts. The taskId is simply an auto incrementing primary key on the PanDA database table for production tasks [98]. At the time of this thesis new tasks are generated with taskIds upwards of 1,400,000 [99]. Therefore, JEM gives out so called internal `taskId`'s in the range between 10,000 and 999,999 to prevent collisions with actual taskIds being used in live production. Each ValidationTask API exposes a method, as a property, called `isNotPanda`, which is checked at various locations during the JEM validation process to allow special handling of this direct validation case.

Direct validation ValidationTasks are either entered into the JEM system by directly modifying the ValidationTask table in the database or by using the web interface which is described in detail further down in Chapter 14. These tasks are, as well as normal ValidationTasks, created in the state `WG`, but once they get picked up by the *DCubePreparationTask* it is checked if the working folder has already been set up by the web interface or if this still needs to be done. Additionally the files specified in the fields `referenceFile` and `monitoredFile` are copied from the reference file cache. Once the DCube preparation stage is completed the ValidationTask's `state` is directly set to `WV` and will then get picked up by the next run of the *DCubeRunnerTask*.

[8] `http://jem.physik.uni-wuppertal.de/validation/<taskId>/result.json`

12.7. Summary

This chapter has shown how JEM generates the quality histogram files on the worker node and how these results, after they have been transmitted to the JEMserver, are automatically evaluated. This happens by, using various ServicesWorker tasks, on the server side that gather, merge and prepare the data for presentation on a website, while being driven by a state machine store inside each ValidationTask object.

The following chapter now explains how JEM stores the reference files, necessary for any validation on the JEMserver and how they can be created offline, by downloading physics result datasets and running a specially configured Athena.

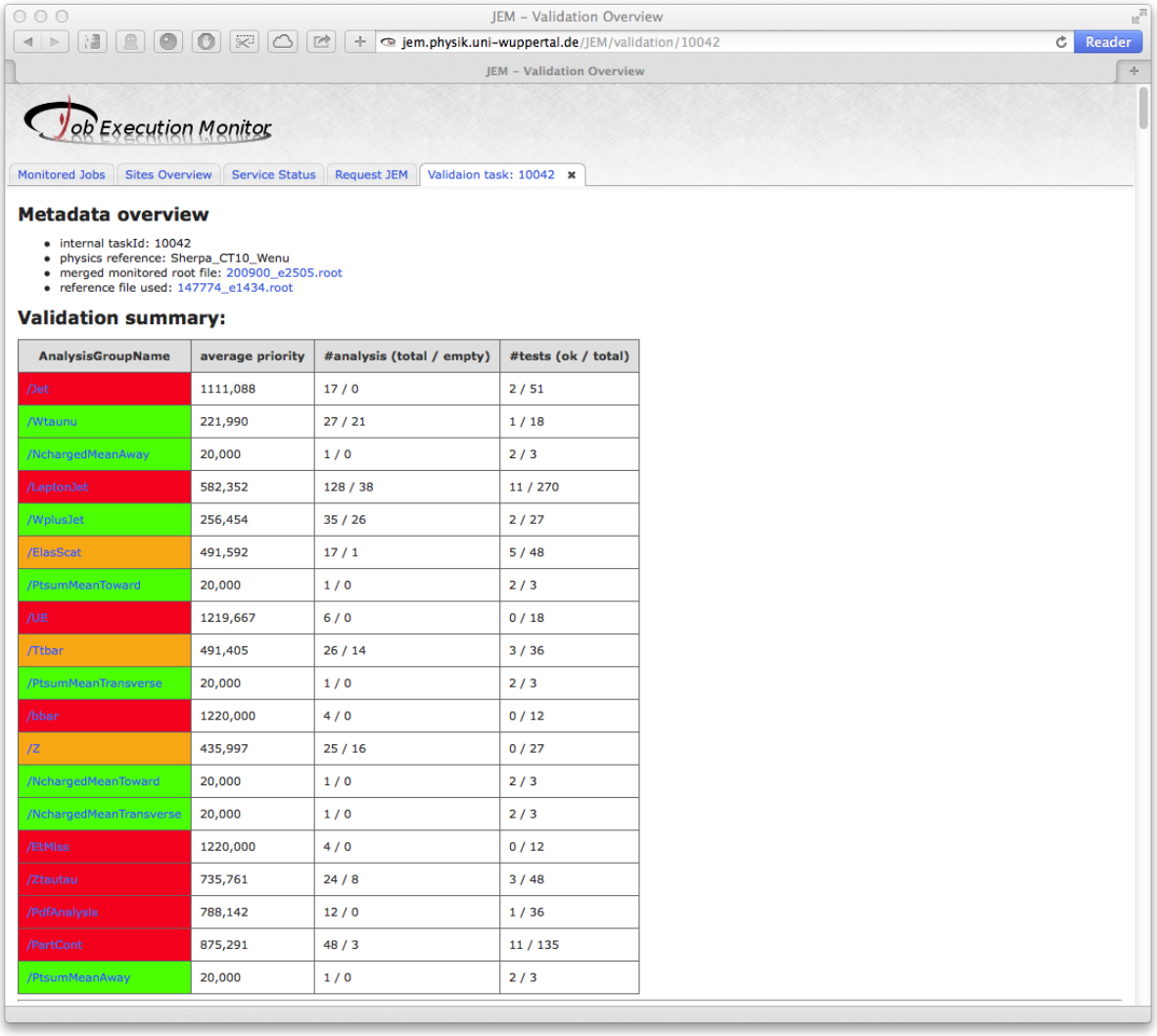


Figure 12.1.: Validation result page header, which gives an easy and direct overview of all validated analysis.

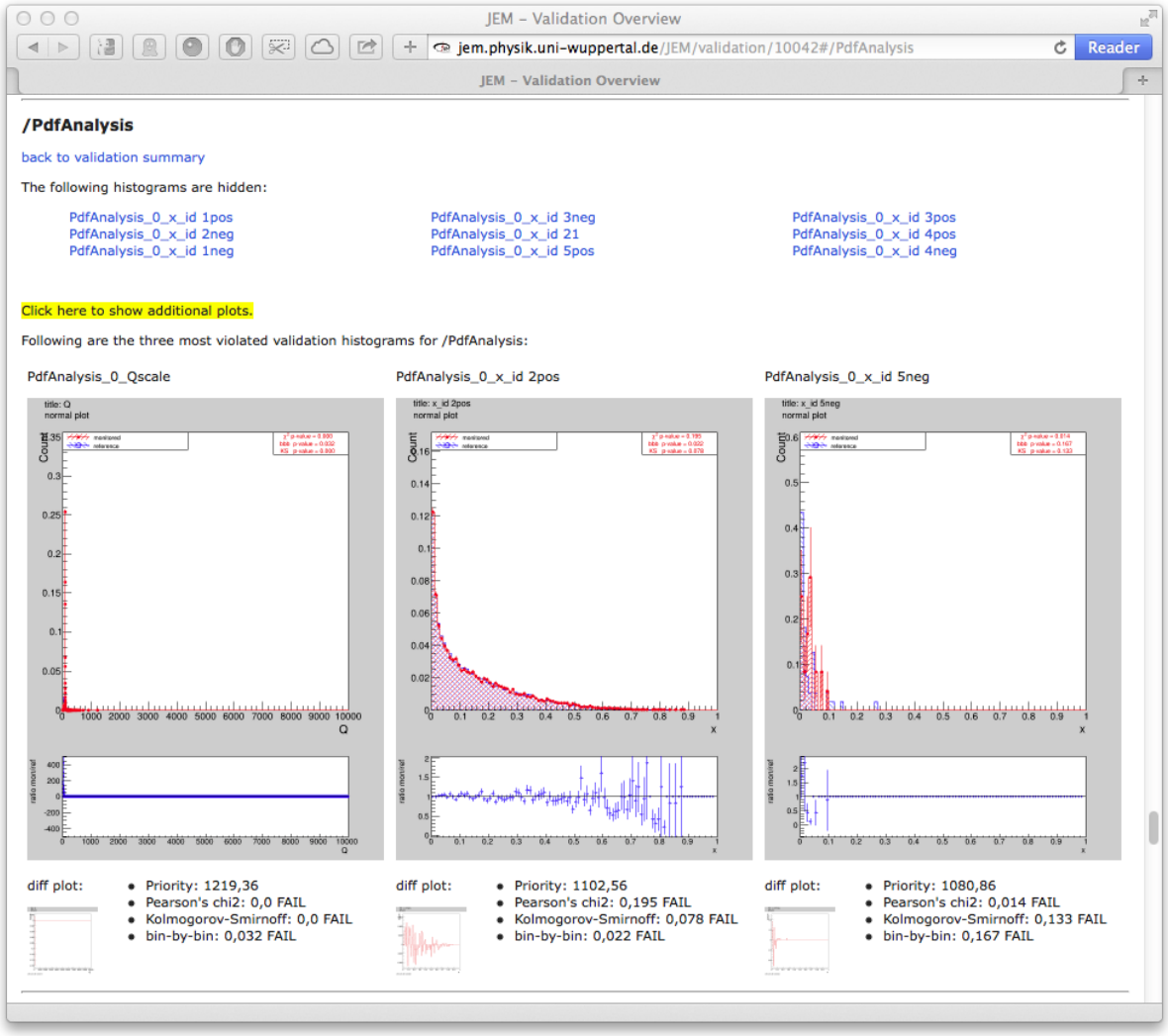


Figure 12.2.: Validation result page with result plots for the `/PdfAnalysis`.

13. Reference file cache

As reference files are necessary for any validation operation, this chapter describes the storage of all reference files inside the JEM system, namely in the JEM reference file cache (see Section 13.2). Additionally the offline generation of new reference files from existing data sets is explained in Section 13.3. Operational experience with the handling of reference files are discussed in Chapter 15.

13.1. Motivation

Important for any validation effort is a set of known and well understood reference histograms, that covers all relevant physics use cases, that the validator's monitoring results can be compared to. To easily produce validation results, the reference files need to be available and indexed with proper metadata to decide, which files are suitable candidates for current validation efforts.

This also implies, that reference files need to adhere to a common internal structure to allow easy automation of validation processes and that the metadata is stored in a commonly known format. With these goals in mind the reference file cache has been designed and implemented in the JEM system.

As each validation is based on the comparison of a huge number of different histograms, which describe various aspects of the underlying generators and the generated physics process these histograms naturally should be grouped and stored together. ROOT allows to store several histograms in one file by creating an internal a file system like structure.

One could also do this individually for each histogram, but this would certainly complicate things by handling a varying amount of individual files per validation process.

13.2. Metadata

Reference files are uniquely identified by the combination of their DATASETID, their appropriate ETAG[1] and the software version, which created the histograms from the original dataset.

The first one describes the physics process and the second one the used software version to create the original dataset. As such they are stored in a predefined file system location which is accessible by the HTTP server so that they can be downloaded by third parties if so intended. As usually files form the same physics process are compared, for different software versions this

Currently only one reference file per DATASETID and ETAG configuration is allowed in the system. This should, however, be reconsidered in the future to allow multiple instances, which represent the different software version, which were used to create the histograms so that differences between these

[1] Reference filenames adhere to the following naming pattern: *<dataSetId>_<software tag>.root*.

Table 13.1.: metadata schema.

field name	value explanation
`dataSetId`	the DATASETID of the underlying physics process
`eTag`	the ETAG of the software generating the data
`filename`	filename under which the file is stored in the file system
`swGenTag`	the software generation tag of the software that generated the original histograms
`swConfig`	a JSON dump of a Python dictionary containing all HEPMCANALYSIS analysis groups that were activated during file creation
`physRef`	a name describing the physics process being identified by the `dataSetId`

can be validated as well. If a file is added to the reference file cache and an older version with similar metadata already exists, the older one will simply be overwritten.

For each reference file there is a database entry in *ActivationService_referencefile* as described in Table 13.1. Some of the data is redundant like the `dataSetId` and the `eTag` as they are also encoded in the `filename`, this, however, allows easier searching and filtering on all entries. `swGenTag` describes the ATLAS software tag of the software, which generated this file. As HEPMCANALYSIS can be configured, which of the available analysis should be used, this configuration is stored in `swconfig`. In `physRef` an explanatory text string, associated with the DATASETID is stored to allow easier searching for reference files.

Two independently created reference files with the same DATASETID and ETAG should result in exactly the same output file, if the same random generator seed has been used to create the underlying physics datasets and the same selection of files from the original dataset is chosen. They should be different if a different seed is used, but the overall form of all distributions should still agree up to a small statistical error.

13.3. Reference file generation

While the reference file cache can be easily populated by the JEM system administrators, there need to be easier ways, that day to day JEM validation operators are able to contribute to this database and fill in more reference files. The JEM web page has the ability to generate file generation requests (see Chapter 14), that are stored in the database and are then handled by the GENERATEREFERENCETASK.

13.3.1. ReferenceGenerationTask

The GENERATEREFERENCETASK works on REFERENCEGENERATIONREQUESTS as described in Table 13.2 by downloading individual result files from an already existing dataset from the Grid. After that, ATHENA is run with a specialized job options file, which launches HEPMCANALYSIS, to

Table 13.2.: ReferenceGenerationRequest metadata schema.

field name	value explanation
`ts`	a time stamp of the last `state` change
`state`	current state of the generation task (see table Table 13.3)
`eTag`	the ETAG of the desired dataset to be used
`dsId`	the DATASETID of the desired dataset to be used
`dataSetName`	the full name of the dataset to be used
`taskId`	the TASKID of the originating dataset to be used

Table 13.3.: ReferenceGenerationRequest states.

state abbreviation	explanation
`RQ`	request for generation is pending
`DL`	currently downloading files
`DF`	downloading is finished
`GR`	generation is running
`GF`	generation is finished
`TR`	files are being transmitted
`FI`	generation is finished and new file is registered in table
`FR`	generation is finished and earlier downloaded `EVNT` have been removed
`ER`	a non recoverable error occurred

produce a new reference file, which then gets stored in the reference file cache. Similar to the validation evaluation described in Chapter 12, the ReferenceGenerationRequest uses internally a state machine, stored in the `state` field, to keep track of the progress. The states are explained in Table 13.3 and a diagram of state changes is shown in Figure B.4.

As described in Listing 13.1, which shows the main `run()` method, the GenerateReferenceTask serially initiates and checks each of the three states in file generation: downloading, generating and post processing. These are explained in the following.

Currently there is no locking on state changes, so the system is only designed to be run by one ServicesWorker instance at a time. To distribute the load over different machines a simple locking mechanism needs to be added to prevent duplicated effort and inconsistencies in the generated data.

13.3.2. Identifying and downloading candidate data sets

The `run()` method starts with `startADownload()` which gets all requests in the state `RQ`, orders them chronologically and picks the oldest one. This method is then responsible for setting up a working directory in the filesystem where all necessary files get downloaded to and where the file generation can later be run.

```
def run(self):
    self.startADownload()
    self.checkForFinishedDownloads()
    self.checkForUnFinishedDownloads()
    self.startAGeneration()
    self.checkForFinishedGeneration()
    self.transmitToJEMServer()
    self.checkAndRemoveData()
```

Listing 13.1: GenerateReferenceTask main `run()` method.

To do this, the `taskId` of the ReferenceGenerationRequest must be known. If a `taskId` is unknown, at this time the `startADownload()` queries the PanDA web API for all finished generator tasks with the appropriate `dsId` and `eTag` and selects one with enough finished jobs, that the needed number of result files can be downloaded from the dataset. Once the `taskId` is known the working directory is created and is named after the full dataset container name that the particular task created, which includes the taskId embedded into the end of the container name.

`startADownload()` will then start a download in a subprocess by running a shell script, that first sets up the local ATLAS software environment and then launches `dq2-get`[2] to download the files from the Grid to the working directory of this file generation. The state of the ReferenceGenerationRequest is changed to `DL` to indicate the download in progress. As `EVNT` files usually contain 5,000 events per file and the JEM validation is based on reference file, which use 100,000 events, usually 20 files are randomly chosen to be downloaded from the dataset if more than 20 files exist. If a reference file based on more than 100,000 events should be created or each individual file contains more or less than then usual 5,000 events, the numbers mentioned above are scaled appropriately.

The PID of the subprocess is stored in a bookkeeping dictionary, along with a reference to the ReferenceGenerationRequest object. `checkForFinishedDownloads()` regularly checks if there still is a process for all PIDs in this dictionary and if not, whether this process has finished successfully. This method also checks if the number of downloaded files in the working directory, containing `EVNT` in their filename, has reached the required limit. If not, the state is changed to `DL`, otherwise it is changed back to `RQ` and `startADownload()` will pick it up in the next run.

Running this downloading subprocess, requires a valid Grid proxy certificate. As these usually have a limited lifetime, a ServicesWorker task, the *VomsProxyKeepAliveTask*, regularly uses the existing proxy certificate to generate a new one, which is stored at a well known location. Therefore, the local user, running the GenerateReferenceTask, always has access to a valid Grid certificate and can run applications like DQ2.

13.3.3. Generating new reference files

Once all necessary files are downloaded `startAGeneration()` picks the oldest one, by querying all ReferenceGenerationRequest's in the state `DF` and ordering them descendingly after their `ts`.

For each request a new subprocess is created that runs a shell script in which first the ATLAS setup is executed to load the ATLAS software environment and then, second, Athena is launched in the

[2] see 3.3.3

requests working directory, with a special `validation.py` (seeListing A.9), as a job option, that first collects all `EVNT` from the local directory and then sets them as ATHENA's `svcMgr.EventSelector.InputCollections`, which allows ATHENA to read and understand them as input files.

The state of the request is changed to `GR`,indicating a running generation, and the PID of the subprocess is stored in a local bookkeeping data structure.

`checkForFinishedGeneration()` regularly reads the bookkeeping dictionary and checks if one of the entries has finished. If the generation has been finished successfully, the request's state is set to `GF`, otherwise the nature of the error is analysed and if the GENERATEREFERENCETASK finds it to be recoverable, the request state is set to `DF` to have a rerun at a later run of the main `run()` method.

13.3.4. Handling finished files

`transmitToJEMServer()` gets all requests from the database that are in the state `GF`, ordered by `ts`, and picks the oldest one. The newly created reference file from this request is then copied via SCP to the central reference file cache location on the central JEMSERVER and is registered in the database table. After this, the state is changed to `FI`, indicating that the file generation is finished and the file is available for validations.

Finally, after a configurable timeout period[3] all downloaded `EVNT` files from the generation's working directory are deleted by `checkAndRemoveData()`. This prevents, that copies of files, that exist in the Grid anyway, use up too much space on the local machine working on the file generation requests. A rerun of a file generation, once either more statistics are needed or the generating software has been updated is always possible later on.

13.4. Summary

As this chapter now has shown how reference files are stored in the JEM reference file cache and how a new generation of such files can be initiated if they are missing, the main functionalities of all JEM Live Monte Carlo Validation processes have been explained. The following chapter explains how a user of the system can initiate these actions using the REQUESTJEM validation web interface.

[3] default value is set to 21 days

14. Web interface for validation control

As the previous chapters explained the internal structure and mechanics of the JEM Live Monte Carlo Validation, this chapter discusses the user interface and how it enables previously registered users to initiate actions in the system and keep track of their progress. In Section 14.1 the main motivation for an external control interface why it was implemented in this specific way is explained. Following that, the dynamic expanding dialogue forms of the main requestJEM are explained in Section 14.2, as well as the overview status pages of the different JEM MC validation processes, which run in the background (see Section 14.3).

14.1. Motivation for an external control interface

Crucial to the MC validation use case (see Section 4.2) is the ability to expose its functionality to the MC validation shifter, who initiates and oversees validation processes, via a complete and comprehensible user interface, that has to meet several criteria:

- It needs to be easy to use.
- All relevant functionality must be accessible.
- Configuration of the individual tasks should be as intuitive as possible.
- Feedback about the status of initiated tasks needs to be accessible.
- Access needs to be restricted to registered, and therefore trusted, users to protect the JEM validation systems from abuse and malicious use.

To meet all these criteria, several options have to be considered, which either could be building a stand alone application, that runs at the validation shifters machines and communicates the commands and results back to the JEMserver, or of creating user interfaces at the JEMserver itself, where commands are entered directly. As a stand alone application also involves designing a data exchange protocol between the application and the JEMserver, a web site based interface was chosen to omit additional work. Also the existing Django infrastructure used for the ActivationService provided existing development experience and therefore faster prototyping for the development of the validation result presentation (see Section 12.5). Web sites offer a better visual user experience and are a common tool within ATLAS to facilitate control over server processes.

Alternatively, the functionality now provided by the web interface could also have been encapsulated into a set of scripts, which are placed and maintained at a centrally accessible location, for example at

the lxplus login machines at CERN. However, these scripts would then lack the afore mentioned advantages of web pages.

14.2. Live Monte Carlo Validation control

The web interface has been designed to grant external users control over Live Monte Carlo Validation functionality, as not only people with direct shell access to the central JEMserver are using the system. As discussed in the previous section, a web interface proved the easiest and most user friendly solution. This functionality is restricted to authorized users to prevent malicious use of limited JEM computing resources. Users can be added by the JEMserver administrators.

The requestJEM pages have been designed as continuous stream of dynamic HTTP forms that load new form data from the server upon entering new data using AJAX technologies provided by jQuery. This results in a continuous user experience and an easy, self-explanatory flow of control.

14.2.1. Dynamic form building

To build the dynamic forms, the requestJEM URL[1] is mapped to the `requestJEM` view in the Django back-end, which returns the main template of the page, containing the layout and the necessary JS includes. The JS code running in the clients browser, the front end, then requests the first form to display from the Django back end `requestJEMForm`. `requestJEMForm` holds the central page logic for authenticating the user and validating the final form inputs. If the user is already logged in, as indicated by the existent session variable `userName` then the first form in the form flow is returned: *ValidationTypeSelectorForm*. Otherwise the the *UserForm* is returned, which asks the user to login using user name and password. Upon submitting the *UserForm* `requestJEMForm` checks the given username and password combination against the list of registered users and, if correct, the user is logged in by storing its username and its eMail address in the session. Otherwise a new *UserForm* is returned and the user has to try again.

As the generated HTML code for the various forms has unique identifiers for the `<input>` tags, they can easily be selected by jQuery and event callbacks can be registered. These callbacks are coupled to state change events at their associated HTML tags, so that once an option has been selected the next form can be loaded. Therefore the web form grows with each further selection as the action request gets more complete.

Central to all event handler registration functionality in the front end is the JS function `refreshFormHandlers()`, which is called upon each successful event handling. Most of the callbacks enlarge the form by requesting further form data from the back end using the front end function `appendForm()`.

Others simply request metadata, like the task name or the amount of available job, for a taskId or a combination of dataSetId and etag. `appendForm()` queries the `/ActSvc/getForm` URL to get a further form data from the Django back end and expects a form name as an argument. Additionally a JS callback function can be provided that executes additional code after the form data has been received and integrated into the page's HTML.

[1] `http://jem.physik.uni-wuppertal.de/ActSvc/requestJEM`

Once one of these handlers has fired, based on the state of the current inputs, that a further form needs to be loaded, `appendForm()` is called with the form name to be loaded as an argument. This method gets the name of the form to be called as its first argument and can get an anonymous JS function as second argument, which is called after the internal AJAX called has returned the form. Internally this method uses `$.ajax()` to call the Django `getForm()` view. `getForm()` internally tries to load the form, given by the `formName` attribute in the POST data, by importing it from the ActivationService `form` module. If the form was successfully imported, it is sent back and rendered as a HTML table `<td>` lines, otherwise a 403 HTML error code is returned. Additionally the ActivationService web back end stores all form names, from the forms returned, in a list which is stored every time in the Django session dictionary, which itself is kept persistent in the database, for later form validation.

As `appendForm()`'s AJAX call receives the rendered form answer from the back end, the received content is appended at the end of the current form containing `<tbody>` element of the input table. If the `callback` parameter of `appendForm()` was not `null`, the callback is executed. These callbacks are used to add additional event handling code to the elements just received.

At the end of each successful `appendForm()` call `refreshFormHandlers()` is called to update all event handlers and the form behaviour. The JS function `appendForm()` keeps track of all requested form from the back end by storing them in a `$.data()` element after they have been appended. This prevents the same form from being added multiple times, if some event should fire multiple times.

If a form has been completely assembled and the validating JS code in the front end has determined, that all fields are filled out properly, the *Submit* button is released and the user can submit the form data to the Django back-end. At the JEM web server a special Django view function `validateForms` receives all aggregated form data and tries to validate it. This happens in several steps. First a class object of the empty `SumForm` is created and all form fields from all forms stored in the `form` session variable are added to the `SumForm`'s `base_fields`. From this newly generated form an instance is created and all input form data received from the front end is given to the `SumForm` constructor. The Django form logic then allows for an internal validation, that each form field holds valid input data.

14.2.2. Workflow on the requestJEM pages

The form flow, which has been technically described in the previous subsection, allows for multiple combinations of forms to be aggregated together. Depending on the choices in the different forms, made by the user, several final actions can be initiated by `validateForms`. These include:

- validate a task running in the Grid
- store a validation request for a task that is about to be run in the Grid
- generate a reference file from already existing output files
- start a validation run, by only using files from the reference file cache
- start a batch validation of multiple reference files stored in the reference file cache

For all the above mentioned actions an eMail is sent to the user, by using the address stored in the session, as confirmation, that the action has been successfully executed. Also links to further information and action progress are included.

The whole form flow is visualized in Figure B.1, while a complete list of all forms their functionalities are listed in Table 14.1. A screenshot example of a finished form flow is shown in Figure 14.1.

14.3. Task progress overviews

As the JEMserver handles various tasks for the Live Monte Carlo Validation, as described in Sections 12.4 and 13.3, there needs to be a system to provide feedback about the current state of these tasks to the tasks initiator. additional to the validation result pages, described in Section 12.5.

ValidationTask's[2] and ReferenceGenerationRequest's[3] have their own simple and paginated list pages providing an overview of all known entities of the respected class, ordered descendingly by timestamp of the last action or the last internal status change. Therefore the most actual changes are always on top and can be easily followed. Without these overview pages the validation users would only have the confirmation mail sent by the `validateForms` Django back end function as the presented information is otherwise only available to JEM administrators with direct database access. The current rule base is also available[4] for interested users and provides an overview of the ActivationService status. This allows a validation user to check if the intended tasks to be instrumented with JEM do receive the correct set of configuration parameters.

As these pages provide no critical information about the inner workings of the Live Monte Carlo Validation system, they are available without password protection. The caching infrastructure of Django caches the rendered pages for a short time to ease the database query load.

14.4. Summary

The web front end provides an easy and convenient interface to reach and use the methods and functionalities of the JEM validation services. It also provides feedback on the status of the various tasks being performed in the background.

While the overview pages described in Section 14.3 provide the necessary data, they could be enhanced in future updates by adding sort and search capabilities with the DataTables plugin [79], as it is used by the main JEM job overview page (see 5.1.1).

[2] `http://jem.physik.uni-wuppertal.de/ActSvc/getValidationTasks`
[3] `http://jem.physik.uni-wuppertal.de/ActSvc/getReferenceFileGenerationRequests`
[4] `http://jem.physik.uni-wuppertal.de/ActSvc/rulebase`

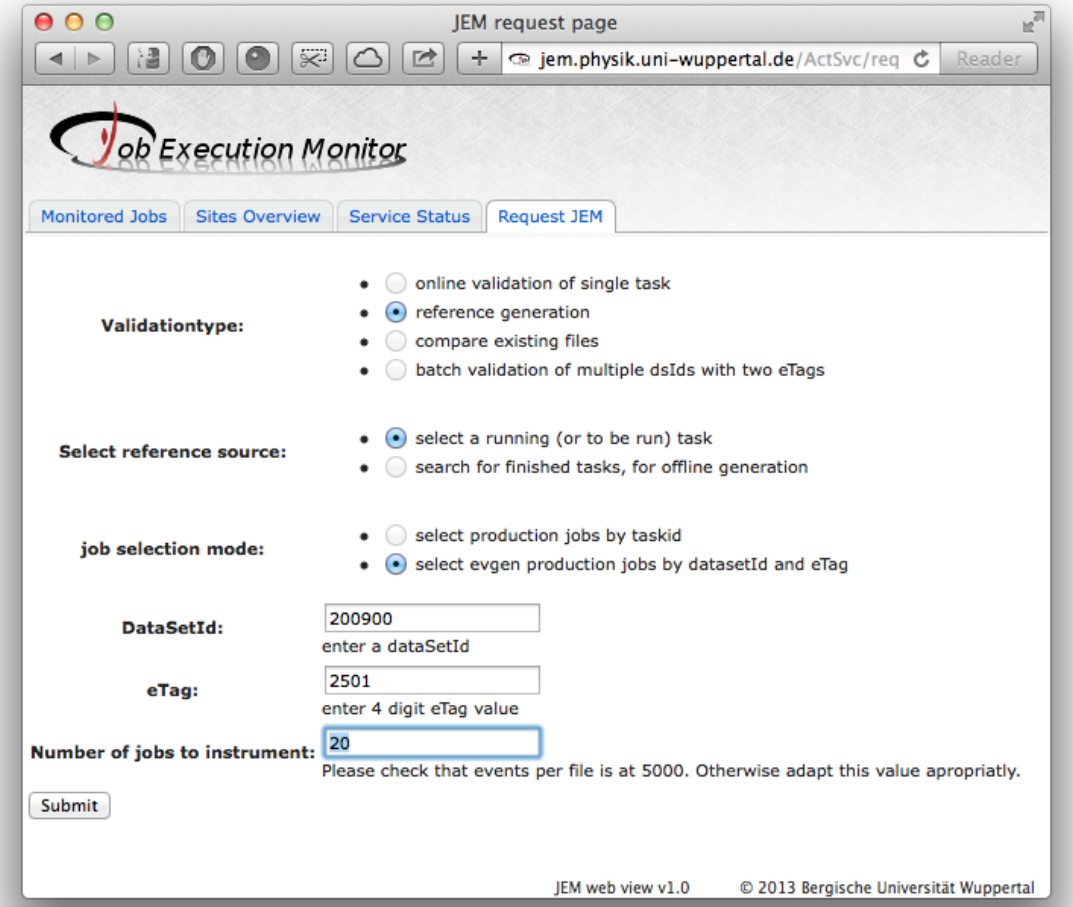


Figure 14.1.: Example of a filled form flow on the REQUESTJEM page. Here a request for a reference file generation is entered and about to be submitted to the JEMSERVER.

Table 14.1.: requestJEM forms and their functionalities. The last two have special functionality.

form name	function in form flow
UserForm	provides fields for user name and password credentials. This form is only used for login purposes.
ValidationTypeSelector	selects the main validation task to be specified: validation of Grid task, reference file generation, comparison of existing files or batch validation of multiple existing files
JobSelectionForm	selects how JEM should choose production jobs. Either by taskId or dataSetId
ReferenceTypeSelectionForm	similar to JobSelectionForm
ReferenceComparison-SelectionForm	select two reference files from the JEM reference file cache for direct comparison
BatchValidationSelection-Form	selects two etags and a list ofdataSetIds which should then get compared
MCTaskIdForm	selects a task running in the Grid by its taskId and select how many jobs should be instrumented
HepMCAnalysisConfigForm	provides activation fields for each individual HepMCAnalysis analysis
DsIdEtagRequestForm	selects a task running in the Grid by an etag and a dataSetId and select how many jobs should be instrumented
ReferenceGenerationTaskId-Form	selects a taskId and the number of files to download for offline reference file generation
ReferenceSourceSelection-Form	selects if the task from which the new reference file should be generated is still running or is already finished
ReferenceGenerationSearch-Form	provides fields to search for finished tasks by entering etag and dataSetId data
MyBaseForm	general base form for all other forms, which provides special HTML table rendering functionality
SumForm	empty form in which all other used forms of the current session get aggregated before being validated

15. Operational Experience

After explaining the technical backgrounds and functionalities of the JEM Live Monte Carlo Validation in the previous chapters, this chapter discusses the gathered operational experience since the first prototype of the system went online in early December of 2013.

In Section 15.1 the different validation processes that have been done, what kind of difficulties have been encountered and how they have been solved are described. The validation shifters have been asked to comment on the JEM user experience and their answers are presented and discussed in Section 15.2. The JEM operator experience as expert shifter for the JEM system itself is presenter in Section 15.3, which also includes the special manual reference file handling, that was occasionally necessary.

15.1. Successful validations

Since the first prototype activation of the JEM Live Monte Carlo Validation more than 512[1] different ValidationTasks have been initiated by the validation shifters and the JEM expert and have been successfully finished. This results in 197,660[1] individual histogram comparisons, that had to be evaluated by validation shifters. Table 15.1 shows a list of all event generators, that have been successfully validated using JEM MC validation services, are currently in validation or in a preparatory state.

Also, more than 414[1] ReferenceGenerationRequests, as described in Section 13.3 have been initiated by the validation shifters to create the necessary reference files for their validation efforts. Some of these files have either been used in multiple validation tasks or had to be reproduced due to missing statistics or updates, that have been made to HepMCAnalysis. These updates consisted of minor bug fixes that were found during the Sherpa validations, like correct event weighting, and of several additions and improvements in plots that were needed for proper understanding of the underlying physics case.

Especially the Sherpa [44] validations were used for testing of the prototype functionalities of JEM and provided a deeper insight into the needed layout for the ValidationTask result page described in Section 12.5. Some of the suggestions for the current versions include more links on the page itself for better navigation inside the page, the possibility to shrink the plots and have them display full resolution on demand, by clicking on them, and the hiding of plots, so that only the most severe statistical violations are shown.

The most difficulties and discussion with the validation shifters were had during final development of the requestJEM user interface, described in Section 14.2. The final workflow of the web forms, as described in the previous chapter, has been constantly updated and revised in close dialogue with the validation shifters.

[1] As of 01.08.2015

Table 15.1.: Event generators being validated, using JEM MC validation services.

generator name	reference version	status
Sherpa 1.4.5	1.4.3	validated
Sherpa 2.0.0	1.4.3	stalled, continue with 2.1.0
Sherpa 2.1.0	1.4.5	bugged, continue with 2.1.1
Sherpa 2.1.1	1.4.5	validation in progress
PYTHIA 6.428	6.427	validation in preparation
PYTHIA 8.183	8.175	validated
PYTHIA 8.185	8.183	validation in progress
Herwig++ 2.7.0	2.6.3	validation in preparation
EvtGen 1.2	1.1	validated for C++ generators validation in progress for Fortran generators
aMC@NLO	not yet defined	generator version in preparation

15.2. Validation shifter experience

One validation shifter[2], using the system intensively, with self-proclaimed four years of software validation experience in different parts of the ATLAS experiment, remarked, that

> "the automation of the process of submission and plotting is very useful"

and that it

> "shows that JEM can be used for comparisons of a big numbers of samples."

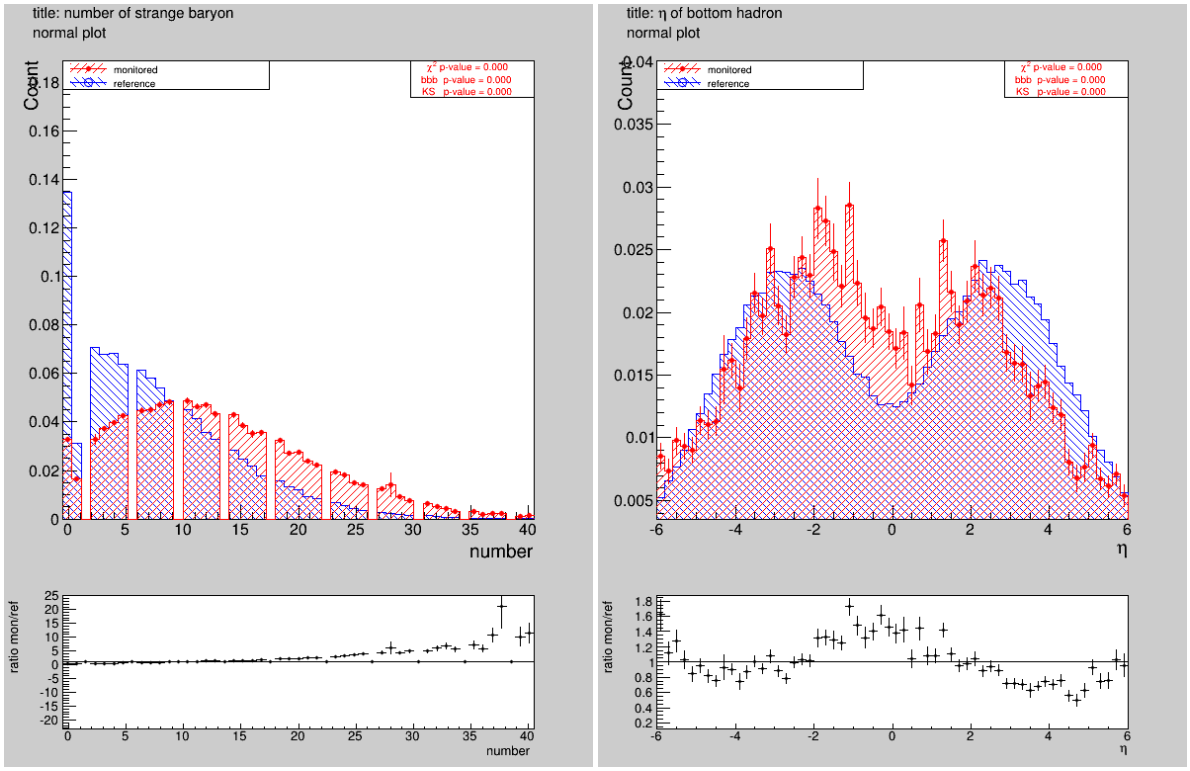
The validation shifters

> "have found different discrepancies by looking at the plots and investigated the origins, sending feedback for the main authors of the generators"

This resulted in various bugs being fixed and a new validation round has then been initiated. See for example the Sherpa validations for versions 2.0.0. and 2.1.0, mentioned in Table 15.1, which were stalled or contained errors and finally version 2.1.1 has been validated using JEM Live Monte Carlo Validation. Two examples of discrepancies in the Sherpa 2.1.0 validation are shown in Figure 15.1. It was further remarked, that it

> "is useful to have in JEM the automatic checks (chi2, KS) on the histograms for a fast validation, although still all of the histograms must be checked by eye"

[2] Leonid Serkin, M.Sc.



(a) Number of strange baryons.

(b) η of bottom hadrons.

Figure 15.1.: Key plots, which show a clear discrepancy between Sherpa 1.4.5 used as reference(blue) and Sherpa 2.1.0 (red) in the W+jets: 0,1,2j@NLO+3,4,5j@LO; $W \rightarrow e\nu$ decay process.

This manual checking is necessary, as some discrepancies are sometimes expected. For example when a software update in the validated version removes an error that created a deviation in the currently used reference version or a not validated previous version (see the mentioned, but not validated Sherpa version in Table 15.1). Also the slightly statistical differences between otherwise largely agreeing, and therefore validated, generator versions, can cause deviations, that are detected by the statistical tests of the validation process. So the statistical test might react to errors, that can be considered negligible by manual human monitoring. However, for most standard operations, it has been mentioned,

> "that JEM requires an expert [...] to be present and able to respond to any inquiries about the plots or samples in a very fast way."

This is mostly due to special reference file handling, which are not yet available to validation shifters as the functionality has not yet been implemented into the RequestJEM interface, see 15.3.1 for further details. The shifter also mentioned some missing features, which will be discussed in further detail in Chapter 16:

- "the selection of more than one sample for comparisons", which is a consequence of the afore mentioned missing advanced reference file handling capabilities of the RequestJEM interface.
- "selective changes in the ratio binning per histograms", which would require a far more complicated interface to HepMCAnalysis.

The shifter concluded with a positive summary, that said, that "in general as a user of JEM, I am happy with the current version, which has been largely improved over time", "is looking and working pretty fine" and that "JEM has incorporated all the proposed changes and updates to the user interface".

15.3. JEM operator experience

Since deploying the first prototype version of the JEM Live Monte Carlo Validation system, the author acted as JEM expert shifter to the validation shifters for any technical questions and assistance. As the first public version was a feature incomplete prototype, bug fixing and development was tightly integrated with requests and feedback from validation shifters. The fact that the system matured during the first two months of operation required a constant overview of all relevant subsystems, which needed occasional restarting due to errors or updated features.

Also the internal state machines of the ValidationTasks and ReferenceGenerationRequest tasks were not working perfectly in the beginning and got stuck in some erroneous state. This required some manual database intervention to reset some states or restart some of these tasks.

15.3.1. Reference file handling

Most manual interventions by the JEM expert, which were not failure recoveries or bug fixes, were needed for special handling of reference files, that could not be facilitated by validation shifters, due to a lack of control options in the requestJEM interface. Therefore the JEM expert was required to intervene from time to time to

- reproduction or merging of reference files due to lacking statistics from too few events (see 15.3.1)
- manually merge several reference files for some special validation cases (see 15.3.1)
- due to updates in HepMCAnalysis.

The first two cases are known limitations of the requestJEM interface and will be added at a future point. Especially the last case requires manual access to the machine running the ReferenceGenerationRequest tasks as the locally available version of HepMCAnalysis needs to be updated.

Merging for statistics

When reference files are generated from existing datasets the maximum number of available events is naturally given. Sometimes this number might not be high enough and the histograms in the generated reference files do not offer enough statistical significance. In such a case several ReferenceGenerationRequest tasks can be started for different PanDA tasks of the same physics process[3] and the resulting files can then be manualy merged by the JEM expert.

For example, for ValidationTask 10190[4], which is a validation task for PYTHIA, in which PYTHIA 8.183 is validated against PYTHIA 8.175. The underlying physics process is based upon

[3] same combination of dataSetID and etag

[4] `http://jem.physik.uni-wuppertal.de/JEM/validation/10190`

minimum bias and soft energy physics and therefore this process requires more events to gain the necessary statistical relevance, when the histograms need to be compared.

Both needed files, *185491_e2801.root* for reference and *185491_e2803.root* for validation, were merged from two ReferenceGenerationRequest task results each. All four base files were already based on 2,400,000 events each, were merged properly and put manually into the reference file cache, yielding two new files based upon 4,800,000 events each.

Merging for inclusive samples

For at least one example the merging of several different reference files was necessary to create the validation resource for a ValidationTask. In this case, ValidationTask 10112[5] was describing a $7TeV$ W+jets with 0,1,2j@NLO and 3,4,5j@LO; $W \rightarrow e\nu$ decay process.

The reference file *200900_e2506.root*[6] had to be validated against a combination of ref_1[7], ref_2[8] and ref_3[9]. The three files have been described as *"orthogonal cuts in the phase space"* by the validation shifter, which allows a simple merging to happen regardless of their differences. They were merged using the ROOT command `hadd` to produce *404142_e1735.root*, which was then added manually to the reference cache.

The implied dataSetID 404142 was chosen of the last two digits of the individual dataSetIDs of ref_1, ref_2 and ref_3 and therefore reflects their merging. However, this dataSetID is not correct, as it is not correlated with the real physics of the generated file. But as it was necessary to store the file within the reference file cache this solution was the most practical at the time. For future versions of the reference file cache, a new category for special merged files, should be added and indexed properly.

[5] `http://jem.physik.uni-wuppertal.de/JEM/validation/10112`
[6] from dataset mc12_valid.200900.Sherpa_CT10_Wenu.evgen.e2506
[7] from dataset mc11_7TeV.167740.Sherpa_CT10_WenuMassiveCBPto_BFilter.evgen.e1735
[8] from dataset mc11_7TeV.167741.Sherpa_CT10_WenuMassiveCBPto_CJetFilterBVeto.evgen.e1735
[9] from dataset mc11_7TeV.167742.Sherpa_CT10_WenuMassiveCBPto_CJetVetoBVeto.evgen.e1735

16. Future technical advances

The current working version of the JEM Live Monte Carlo Validation is quite specifically tailored to support the generator stage (see 3.4.2) of the MC production chain. This happened, as described in Chapter 10, mostly due to close contact and coordination with members from the ATLAS MC Generator group. Also, the impact of errors and mistakes at the generator stage is higher than in later stages on end results for physics analysis users.

The JEM the Live Monte Carlo Validation has, however, been designed in such a way that additional MC stages can easily be added, which is discussed in this chapter. In the next chapter, possible alternative solutions, which would be able to achieve the same goals are discussed.

16.1. Current technical limitations

Although the current JEM Live Monte Carlo Validation system has been widely accepted in the ATLAS community for generator validation, the tools provided still have some technical limitations that need to be addressed.

As mentioned in 15.3.1, there are still some limitations to the provided functionality regarding reference file handling. One of the most missed features by validation shifters is the possibility to merge reference files or to upload their own versions. Before this can be achieved the information schema for the reference files, as described in Table 13.1 would need to be updated to store several additional properties:

- a creator field, to distinguish beetween files uploaded by shifters and files being created by a ReferenceGenerationRequest
- a version field, to have several versions of files, for example based on different event counts
- another reference file naming schema, as more than one file versions would not work within the current file naming schema
- a comment field, that contains explanation, for this particular file version

Once these features have been added to the reference file cache, the requestJEM interface can be enhanced to by adding control functionality for reference file uploading and merging tasks. For the second one a dedicated ServicesWorker task would probably be the best solution as the database entries can be used for bookkeeping on the different merge actions.

Another missing feature of the web interface is the possibility to freely configure a standard JEM task instrumentation with an arbitrary configuration string. Ideally this upgraded web interface would also dynamically do a consistency check on the provided options to make sure, that the JEM instrumentation of Grid jobs does not fail due to configuration problems.

The web interface itself would also need to get an internal API, with which the different form flows are defined, as these are currently hard coded together with a lot of error prone glue code in the `requestJEM.js` library. An abstraction layer to the form flow would also allow to implement some sort access control list, which could hide or activate different control options based on a set of roles, a web interface user has. This would prevent any erroneous action being taken by an individual in parts of the requestJEM page that is not fully understood.

It might be a worthy feature to incorporate the `jsRoot` [100] prototype into the ValidationTask result page. `jsRoot` is a JS abstraction layer around *libROOT*, which is the core library of ROOT. This would, in principle, allow a web browser to directly inspect and navigate ROOT files and have direct look at the contents of histograms.

Finally, Rivet [101], is currently being added as a second option to HepMCAnalysis for generating reference files. As this is, at the time frame of this thesis, not completely finished and still far away from productive use, it shall only be mentioned here. There are, however, some complications that need to be addressed, like the necessity to adapt the reference file cache to handle files from different sources. Due to the fact, that Rivet and HepMCAnalysis use completely different analyses and have different topological layouts in their ROOT files the DCube ServicesWorker tasks serving the ValidationTasks need to be adapted to prevent mixing of the two different file types.

The ValidationTasks could later on be equipped with some load balancing logic, to prevent, that each newly received monitored ROOT file from a finished Grid job triggers a DCube comparison run, if the system is under a very high load. One could try a weighted priority approach, where a fraction of the first and the final arriving monitored result files get higher priority for merging. Therefore, one can then see early if some distributions do not agree by wide margin and the final file obviously needs to be merged to complete the validation run.

In the future the *MergeHistoFilesTask* should take the *monitored.root* file and move it after a proper renaming (see Section 13.2) to the reference file cache and register it in the database. Therefore the reference file cache will grow with each successful validation run.

Currently only one reference file per dataSetId / etag combination is allowed in the reference file cache. It might prove beneficial to weaken this limitation by also adding version management. Problems with normalization might occur, so that it might be necessary to have several reference files for the same process but with different underlying event counts. Also this would help to identify problems or differences in the quality histogram generating software.

16.2. Adding further Monte Carlo stages

The output data formats of subsequent MC stages are not binary compatible with each other and the validation goals will differ for each stage. Therefore, one cannot simply apply software packages like HepMCAnalysis, which is generator stage specific, to generate the full amount of quality histograms to satisfy all validation needs.

So, to add other stages, a similar software like HepMCAnalysis needs to be provided. It needs to be able to be either run the same way as HepMCAnalysis is now used by JEM, by adding it with a `"postExec"` parameter to the Athena execution, or by having the software run by JEM as a monitored subprocess, which then produces the desired output.

The internal handling of reference files would need to be updated to not only use etags but the

full range of supported software tags and their subsequent combinations. Any access to the reference cache then would also need to check the extended software tag attributes to return the appropriate files. As long as other MC stage quality software produce a similar ROOT output file as HepMCAnalysis does, the rest of the JEM comparison infrastructure could be used quite unchanged.

Depending on the quality histogram generating software being used for any new stage, new specialized ServicesWorker tasks, analogous to the *DCubePreparationTask* and the *DCubeRunnerTask*, would need to be implemented and the ValidationTask table, described in Section 12.4 would need to be extended to reflect the different validation tasks depending on the MC stage being validated.

For future developments the *directValidation* mode, introduced in 12.3.2, might prove useful, as it provides an infrastructure for validation of other MC stages, such as simulation or reconstruction, that provide no plugin like HepMCAnalysis for Athena or any other physics framework that is being used.

17. Possible alternative solutions

JEM's ability to selectively instrument Grid jobs provided a solid basis for a design idea about MC validation. Furthermore JEM already provided a stable and easily expandable web service based on Django and an associated database for validation task metadata handling and accounting of reference files. Therefore JEM already provided a good infrastructure to start with.

Alternative, possible implementations and ideas are worthwhile to be discussed, as is done in the following.

17.1. Automatic quality histogram creation

A big part of the work for the Live Monte Carlo Validation efforts, were the modifications of the ActivationService to dynamically instrument selected Grid jobs to create the necessary quality histograms, which are later used on the JEMserver for the validation. Another solution to be considered might be a general post production step for any Athena job, or at least for a fixed amount of jobs per task, that produces quality histograms automatically for each task.

Similar to the used HepMCAnalysis for this project specialized post production steps for the several output data typed would need to be realized. The quality histograms could be stored in the output dataset or, due to their limited size could be simply written to the working directory of the Grid job and be stored in the log file data set associated to each job. As these log file data sets have a limited life time the quality histograms would need to be obtained in this time frame or be later reproduced.

The ongoing effort to create the Event Service (ES) (see 3.5.3), which automatically gathers metadata about each generated and each experimentally produced event could also be used as an infrastructure basis to store quality data in the form of histograms at a central location. This, however, still requires an infrastructure to compare the generated quality histograms to appropriate reference files and notify interested parties of problematic results.

17.2. Automated Grid validation

Instead of using JEM to selectively instrument jobs and generate validation output from distributed Grid jobs, a centrally controlled system could be used with the target to generate quality histograms for all available tasks and then automatically cross compare all reasonable combinations. For each new task an automated Grid validation system would need to check if the combination of dataSetId and software tag has already resulted in a successful production of quality histograms. If not, a new validation task would need to be created which has the output data set of the targeted task as input data set and then reads the required amount of files to generate the quality histograms.

One has to discuss, whether one job would be able to handle all input files in one instance or if several jobs are necessary. Operational experience has shown, that ReferenceGenerationRequest use up several GB[1] of disk space to create the quality histograms. While this amount of disk space in itself might be problem, as it exceeds locally allowed disk usage, the runtime of these jobs can last up to a day.

In the later case this would result in several files with quality histograms that would need to be merged at a central location. If merging is necessary it would also be possible for each of these tasks to automatically check the number of existing quality files in the output dataset. If all files are present, except the last one of the job itself, all of them could be downloaded to the job and then be merged. This could, however, require some external synchronization if the Grid storage catalogues are not entirely up to date.

This would require a specially reserved amount of computing resources dedicated to this certain class of Grid jobs, that are specialized for this kind of automated validation and would also require validation shifters to oversee validation results. Validation results could be stored in the usual Grid storage systems. A central, metadata indexed, catalogue of generated files should be considered, to allow easy and transparent access without searching through the complete ATLAS file catalogues.

This approach would transfer the basic ideas of the realized system into a dedicated validation system, which would span several subsystem of ATLAS computing. As most of these components are critical for ongoing computing efforts, agile development would be difficult and a small project like JEM would not be able to handle the organizational overhead.

17.3. Dedicated validation cluster

Another possibility would be to handle every validation task on a dedicated MC validation cluster. Without the complex Grid infrastructure and the Grid job handling overhead, it would be a lot easier to have an automated output file gathering and comparison system as files can easily be moved through the local area network or on a properly setup distributed file system.

Such a validation cluster would probably need to be at a large Tier-1 computing centre to have the appropriate hardware and, more importantly, the network resources available to handle the large amounts of data that need to be downloaded in order to generate the quality histograms, should these not be available locally.

This validation cluster would impose additional costs on the existing computing budgets and would also require additional computing resources to be provided and managed, which make this approach too costly and too ineffective.

17.4. Worker node based validation evaluation

During the early design stages of the JEM Live Monte Carlo Validation project, a live validation on the WN has been discussed and has finally been discarded for several reasons, which are discussed here.

The output of one job does not provide enough statistical relevance to give meaningful validation output so a merging of various validation results at a centralized process is necessary anyway. This

[1] up to 145GB

implies a central validation result data aggregation infrastructure, that has now been implemented, however, not on the basis of validation results but on the basis of quality histograms.

Next, it is a lot easier and statistically safer to merge quality histograms before validation than to merge comparison results. As the statistical variations in unmerged quality histograms would produce a vastly higher error on the results of comparison methods like the Kolmogorov-Smirnoff-Test [102].

Also the reference files are usually based on a number of events that is at least an order of magnitude greater than any single output of one MC job, which means that appropriate scaling has to be done. This introduces another source of error.

To validate the generated output JEM would then need to get a handle on the appropriate reference file, most certainly by downloading it from the central JEMserver. The validation process itself could either happen by using DCube, which in this case would need to be sent to the WN alongside JEM, or by using custom ROOT scripts. However, as pointed above, the generated output of one MC job usually produces way less events than the histograms in the reference file are based upon, which would introduce scaling issues at the comparison point. Also the amount of data to be transferred would increase by the number of jobs times the size of the reference file. On a small scale this would not introduce any serious problems concerning bandwidth. However, if this system would be deployed broadly, the reference files would need to be put on a special content delivery endpoint to access them performantly.

As any action that significantly adds to the job runtime have always been a sensitive political point while discussing job monitoring by instrumenting jobs with JEM or similar software with ADC coordinators any calculations and data transfers that are not really necessary on the worker should be done centrally on dedicated hardware. This might, however, change at a future point, once this method of gathering validation data is widely accepted and has left its childhood stages.

All of the above points have been considered during the design process for the validation evaluation part of the JEM Live Monte Carlo Validation process and have ultimately not been realized. Another important point to notice is that any complex software added to the WN has to be written a lot more carefully and failure proof as errors are hard to understand, very difficult to debug remotely and need to be recovered automatically by JEM on the WN. Especially if a foreign software package like DCube is to be used, which is not fully integrated into JEM itself.

Part V.

Summary & Outlook

18. Summary

After introducing the experimental and physics environment in Part I the new development and the changed design of the WLCG since its original planning stages have been discussed. It describes a computing Grid that has changed remarkably in some ways from its original designs, while specialising to the unique requirements of computing in high energy physics. The computing model changed from a "push" to a "pull" method in the ATLAS specific computing environment.

In Part II, JEM is introduced as a highly configurable job-oriented monitoring wrapper for Grid jobs. Its abilities have been substantially enhanced to suite the needs of the Live Monte Carlo Validation.

To better understand the inner workings of a jobs binary, running in the Grid, the BackTrace-Monitor prototype is detailed in Part III. It allows explicit analysis of process memory, both in stack and heap space, simply by attaching a Python scripted GDB instance to the binary.

The main focus of this thesis lies in the detailed description and analysis of the Live Monte Carlo Validation, which is explained in Part IV. There, after discussing the main motivation, the ActivationService is introduced, a powerful service, which is queried by every ATLAS Grid job. The answer to these queries can order the pilot job wrapper to instrument the Grid job with JEM, which allows JEM to monitor and control the Grid job. With the help of the ActivationService, jobs of tasks, that are to be validated, can be selectively instrumented with JEM, which is configured in such a way, that quality histograms are automatically created once the job is finished. These are then sent back to the JEMserver, where they are merged with all the other quality histogram files from other instrumented jobs.

The merged quality histograms can then be compared against a known reference set of histograms that are stored in the JEM reference file cache. During these validation operations, statistical tests are performed to get a measure for the goodness of the validation procedure. The results are refined and displayed together with plots of the histogram comparisons on a web page where validation personal can inspect and qualify agreements and disagreements between the monitored histograms and the reference ones. The controlling actions inside the Live Monte Carlo Validation system are initiated and supervised via a web interface by authorized shifting personnel. As described in Chapter 15, this project has been established as an important tool for ATLAS generator validation and is being further evaluated.

19. Outlook

Aside from the technical outlook for the Live Monte Carlo Validation system, described in Chapter 16, the JEM ecosystem itself can and should be extended further. For these future developments various points should be considered.

Coupling with PanDA pilot JEM, as a job monitor, provides a broad range of functionality, that is not necessary for most correctly running jobs. However, as job failures can happen in groups, it would be beneficial to instrument jobs with monitoring features, that are already running and this is currently not possible. As the PanDA pilot is already a job wrapper some functionality from JEM could be moved into the pilot as a monitoring module. It could be made available in such a way that a pilot could invoke monitoring capabilities from a JEM library should the need arise. This would allow a user to react a lot faster to failing jobs by instrumenting similar jobs of the same type or on the same site and to react accordingly.

Decoupling of `JEM.py` With the addition of the ServicesWorker, the ChunkConverter, the tight integration and close coupling of the Worker Node (WN) working mode into the main `JEM.PY` module became obvious and at time problematic. While no unmanageable problems were encountered, development and maintenance would be far easier should the main `JEM.PY` driver be split up into one, that provides basic functionality for the new working modes and a version that has the necessary WN features already integrated into the main driver.

Use of Django commands The ServicesWorker working mode described in this work is not very well suited to use the JEM internal logging facilities, while also using Django functionalities. Instead most of the individual tasks could be realized as Django *Command*s, that are naturally provided by Django and could the be regularly run as `cron` tasks. This is true for all tasks related to ValidationTask handling and for all tasks, that handle the offline file generation, which only change the state of the tasks state machine and then act accordingly.

Move of hardware to CERN As, at least the Live Monte Carlo Validation use case, now provides a central service to the Physics Group of ATLAS, the hardware necessary to run this feature could be moved to the data centre at the main CERN site. This would allow higher levels of stability and scalability as the main CERN computing centre can professionally provide better hardware failure prevention, than any Tier-2 site could manage. Also, the large virtualization cluster resources, which would be needed for offline reference file generation, can be scaled easier to individual needs. The huge network bandwidth available, especially to all Tier-1 computing centres, where most of the generated MC production is stored, would allow a far more efficient offline reference file generation.

Additionally, it would allow easier parallel development to other ATLAS projects, once the different service machines are closer to each other and share a common CERN infrastructure like authentication and shared file systems.

Part VI.

Appendices

A. Code listings

```
  def run(self):
    clearCmdCnt = 0
    while not self.__stop:
    try:
      # read data from socket
      cmd = self.__getNextCommand()
      clearCmdCnt += 1
      # idea: keep trackof running commands
      if cmd is not None:
        if cmd.requiresRuleBase():
          cmd.setRuleBase(self.__ruleBase)
          if cmd.isThreadedExecution():
            cmd.execute()
          else:
            waitForSyncCount = 0
            while len(self.__runCommandList) > 1:
              waitForSyncCount += 1
              self.__clearCmdList()
              time.sleep(0.1)
              if waitForSyncCount == 10:
                logger.error('tried 10 times to wait for commands, will run
                    unsynchronized')
                break

            # cmd list should now be empty
            cmd.execute()
            if clearCmdCnt >= 10:
              # every ten cmd executions: try to clear list
              clearCmdCnt = 0
              now = time.time()
              if now - self.__runCommandListLastCleared > self.
                  __runCommandListClearingIntervall:
                self.__runCommandListLastCleared = now
                self.__clearCmdList()
  except:
    log_last_exception(logger.error)
```

Listing A.8: Activation Service Rule Evaluater (ActSvcRuleEvaluater) `run()` method, cleared of comments and logging outputs.

```
class LlfpmTokenBlock(PayloadBlock):
    block_visible_name = "llfpm-token"
    block_type_name = "BLOCK_TYPE_LLFPM_TOKEN"
    block_fields = [("lineno", "I"),
                    ("fileNameID", "I"),
                    ("tokenID", "I"),
                    ("match1Id", "I"),
                    ("match2Id", "I"),
                    ("match3Id", "I")]
```

Listing A.1: *RegExTokenTrigger* token info output.

```
class Token(object):
    def __init__(self, lineNo, fileName, tokenStr, requestMatch = 0):
        self.__lineNo = lineNo
        self.__fileName = fileName
        self.__tokenStr = tokenStr
        self.__requestMatch = requestMatch
        self.__resultStrings = []

    def appendResultString(self, resultStr):
        self.__resultStrings.append(resultStr)

    def getTokenStr(self):
        return self.__tokenStr

    def getResultStrings(self):
        return self.__resultStrings

    def __str__(self):
        return "Token: %s:%d (%d) -- %s" % (self.__fileName, self.__lineNo,
            self.__requestMatch, self.__tokenStr)
```

Listing A.2: Python token representation.

```
tokenList = [
    {'id': 'DNS_1_1',
     're': re.compile(r'Could not set service name')},

    {'id': 'DNS_2_1',
     're': re.compile(r'.*getaddrinfo failed')},
    {'id': 'DNS_2_2',
     're': re.compile(r'Name or service not known')},
     ....
]
```

Listing A.3: Known token list example.

```
typedef struct {
    Trigger          base;
    int              lineCount;
    int              tokenFoundCount;
    int              tokenCount;
    tokenRegExp *    tokenList;
    senderInfo       senderCache[SENDERIDCACHELEN];
    matchedToken     tokenCache[MATCHED_TOKEN_CACHE_SIZE];
    int              tokenCacheCnt;
} RegExTokenTrigger;
```

Listing A.4: `RegExTokenTrigger` main structure.

```
class Pattern(object):
    def __init__(self, name, patternId, tokenList):
        self.__name = name
        self.__isMatched = False
        self.__id = patternId
        self.__patternInfo = {}
        self.__tokenList = tokenList
    ...
    def newToken(self, token):
        raise NotImplementedYet()
    def checkMatched(self):
        raise NotImplementedYet()
    def resetMatched(self):
        raise NotImplementedYet()
    def getPatternInfoData(self):
        raise NotImplementedYet()
```

Listing A.5: General Pattern interface.

```
patternList = [
    SimplePattern(
        'stage in failed, file was missing  ',
        0x0001,
        ['STAGEIN_1', 'STAGEIN_2', 'STAGEIN_3']),
        ...
]
```

Listing A.6: Example list of *SimplePatterns*.

```
class LlfpmPatternBlock(PayloadBlock):
    block_visible_name = "llfpm-pattern"
    block_type_name = "BLOCK_TYPE_LLFPM_PATTERN"
    block_fields = [("patternID", "I")]
```

Listing A.7: *SimplePatternTrigger* pattern info output.

```
from AthenaCommon.AppMgr import ServiceMgr
ServiceMgr.MessageSvc.OutputLevel = DEBUG

evtMax = -1
theApp.EvtMax = -1
EvtMax = evtMax

#load pool support
import AthenaPoolCnvSvc.ReadAthenaPool

# input
import os
fileList = []

for fileName in os.listdir(os.getcwd()):
    print 'considering ' + fileName + ' for generating reference'
    if 'EVNT' in fileName and 'pool.root' in fileName:
        print 'using ' + fileName
        fileList.append(fileName)

svcMgr.EventSelector.InputCollections = fileList
from HepMCAnalysis_i.HepMCAnalysis_iConfig import HepMCAnalysis_i

myAna = HepMCAnalysis_i("HepMCAnalysis_i", file = 'out.root')
myAna.JetAnalysis = True
myAna.WplusJetAnalysis = True
myAna.ZAnalysis = True
myAna.ZtautauAnalysis = True
myAna.WtaunuAnalysis = True
myAna.ttbarAnalysis = True
myAna.bbbarAnalysis = True
myAna.UEAnalysis = True
myAna.EtmissAnalysis = True

myAna.UserAnalysis = False
myAna.ElasScatAnalysis = False

myAna.LeptonJetAnalysis = True
myAna.ParticleContentAnalysis = True
myAna.PdfAnalysis = True

from AthenaCommon.AlgSequence import AlgSequence
job = AlgSequence()
job += myAna
```

Listing A.9: HepMCAnalysis configuration script used by the generate reference task (GenerateReferenceTask) to create a new reference file.

B. Flow diagrams / state machines

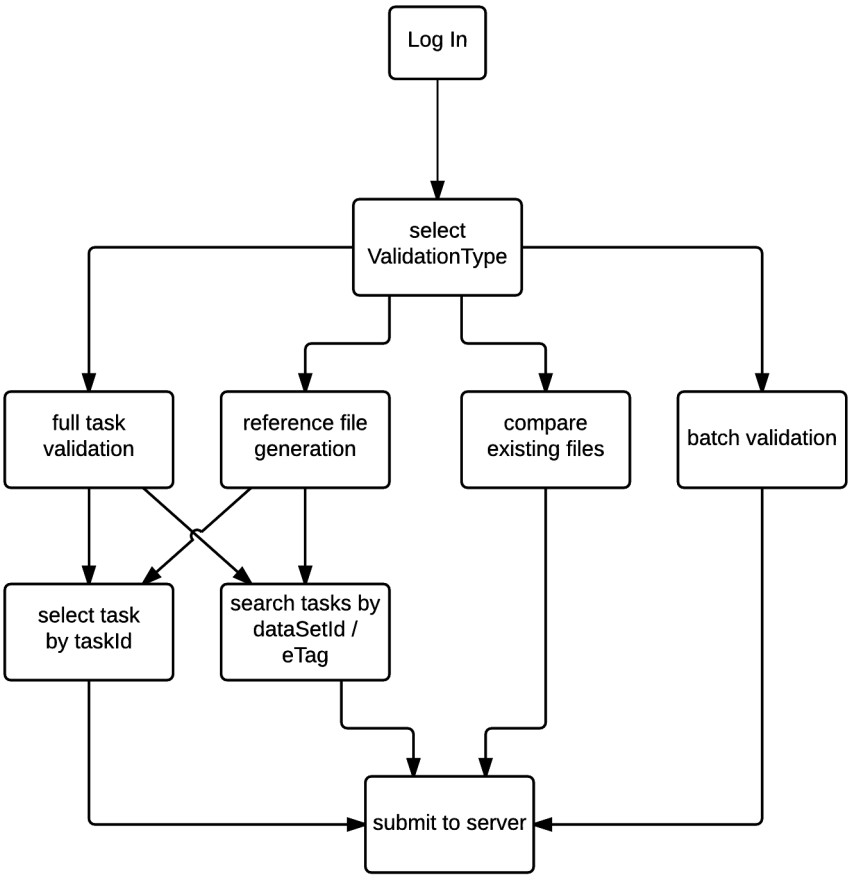


Figure B.1.: Work flow of JEM request web interface (requestJEM) Hypertext Transfer Protocol (HTTP) forms. Produced with [92].

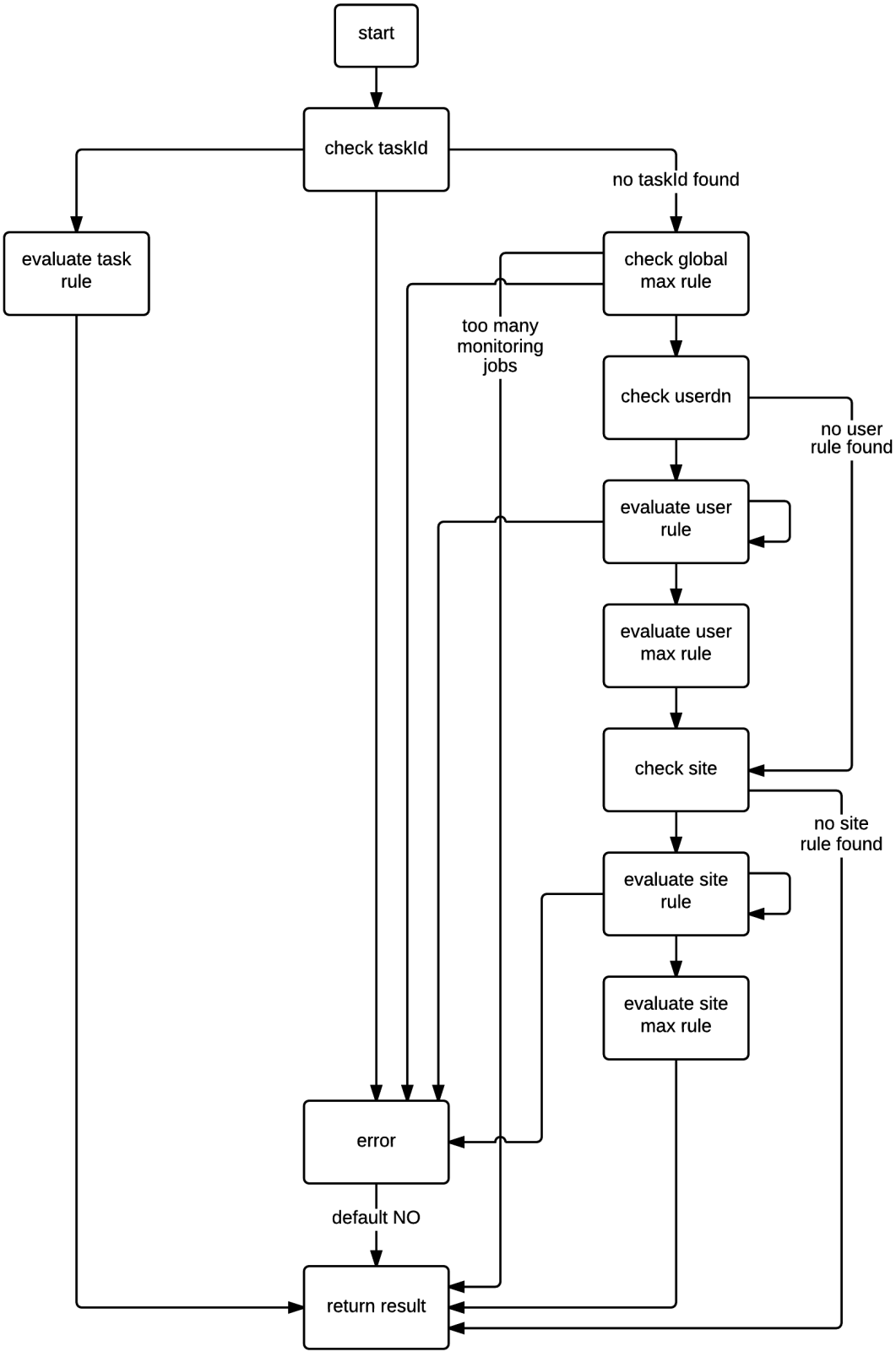


Figure B.2.: Rule evaluation flow. Produced with [92].

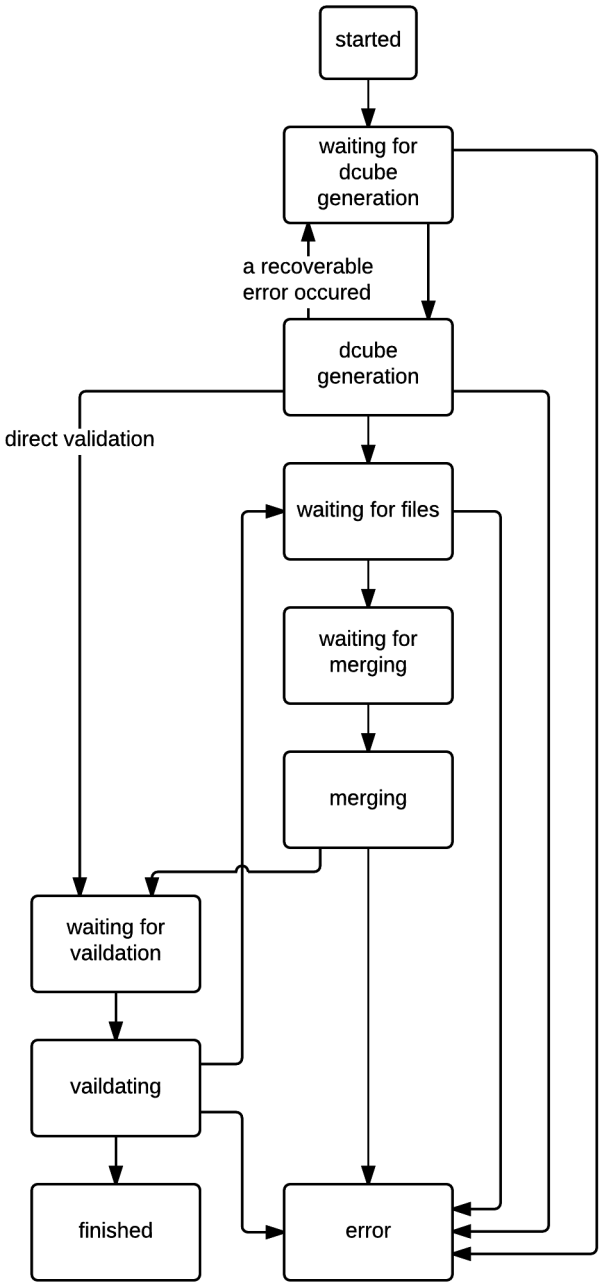


Figure B.3.: States and state transitions of a validation task. Produced with [92].

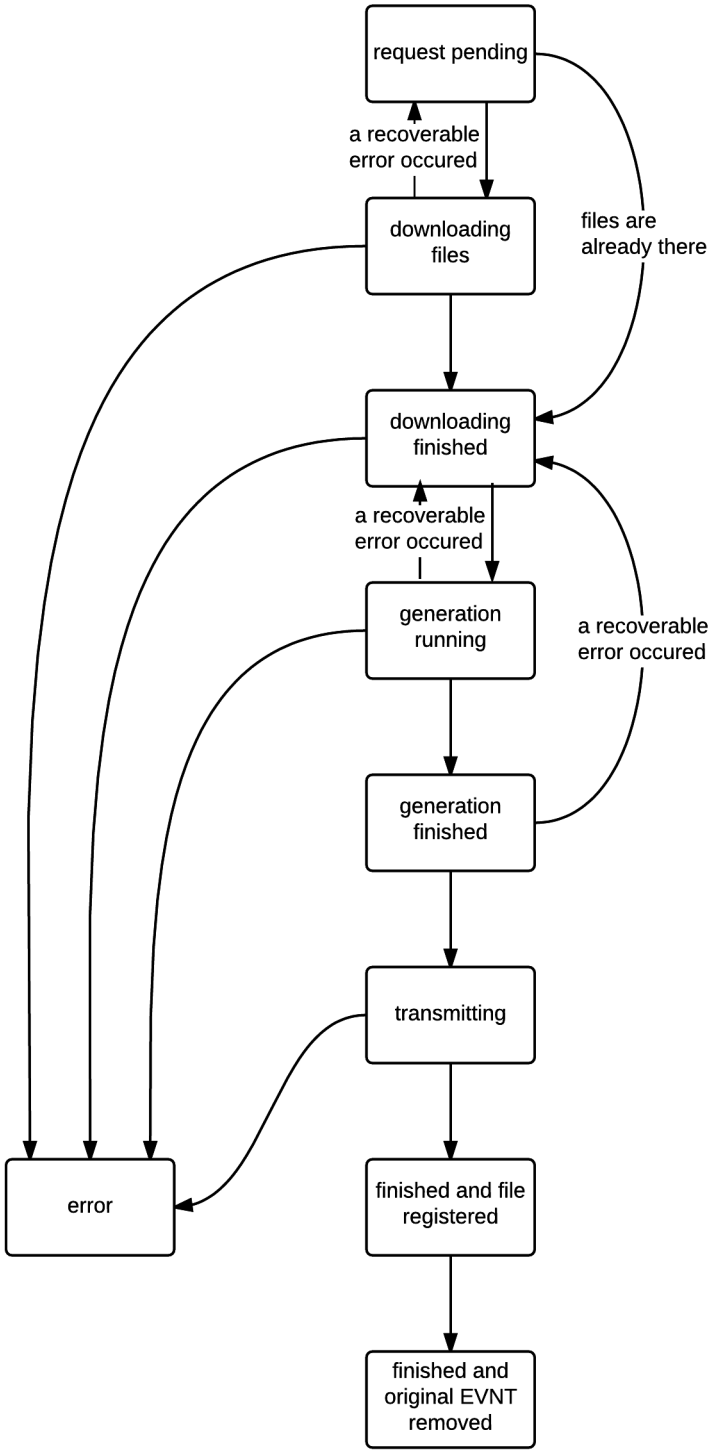


Figure B.4.: State machine for reference file generation. Produced with [92].

C. Acronyms

JEM.py	JEM main launch module
ActivationService	JEM Activation Service
ActiveMQ	ActiveMQ message broker
ActSvcRuleEvaluater	Activation Service Rule Evaluater
ADC	ATLAS Distributed Computing
AGIS	ATLAS Grid Information System
AJAX	Asynchronous JavaScript and XML
ALICE	A Large Ion Collider Experiment
AMI	Atlas Metadata Information System
AOD	Analysis Object Data
API	Application Programmer's Interface
ASIC	Application Specific Integrated Circuit
Athena	Atlas Control Framework, based on Gaudi
ATLAS	Multi purpose experiment, located at CERN, formerly known as A Toroidal LHC Apparatus
BackTraceMonitor	JEM back trace monitor
BackTraceViewer	JEM BackTraceViewer
base64	binary-to-text encoding schema, represeting data in ASCII
CE	Computing Element
CERN	Conseil Européen pour la Recherche Nucléaire – European high energy physics centre
ChunkConverter	JEM chunk converter
CMS	Compact Muon Solenoid
CPU	Central Processing Unit
CVMFS	CERN VM File System
dataSetId	dataset identifier
DCube	XML driven histogram comparison framework
DDM	Data Distribution Management
Django	Python based web framework
DPD	Derived Physics Data

DQ2	Don Quijote 2
ES	Event Service
ESD	Event Summary Data
etag	event generation software tag
FAX	Federated ATLAS XRootD
FPGA	Field Programmable Gate Array
FTS	File Transfer Service
Ganga	lightweight Grid job management tool
GDB	GNU Debugger
GenerateReferenceTask	generate reference task
HepMCAnalysis	HepMCAnalysis quality histogram generation software
HLT	High Level Trigger
HMAC	Keyed-Hash Message Authentication Code
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IPC	Inter Process Communication
JEM	Job Execution Monitor
JEMserver	central JEM server
JEMstub	PanDA pilot JEM module
JIRA	issue tracker for software teams
JMS	Job Monitoring System
jQuery	jQuery, a fast JavaScript library
JS	JavaScript
JSON	JavaScript Object Notation
LEP	Large Electron-Positron Collider
LFC	LCG File Catalogue
LHC	Large Hadron Collider
LHCb	LHC b-Physics experiment
LogFileSaver	JEM log file saving plugin
MatchTaskRequest	match task request object
MC	Monte Carlo
Memcachd	distributed memory cache demon
MONARC	Models of Networked Analysis at Regional Centres
NAT	Network Address Translation

NFS	Network File System
PanDA	Production and Distributed Analysis
PID	Process ID
POSIX	Portable Operating System Interface
POST	HTTP POST method
PostMortemLogAnalyzer	JEM post mortem log file analyser
R-GMA	Relational Grid Monitoring Architecture
RAM	Random Access Memory
ReferenceGenerationRequest	reference file generation request object
requestJEM	JEM request web interface
RequestTaskCacheMatcher	request task cache matcher
Rivet	Rivet
ROOT	Data Analysis Framework
RSS	Resident Set Size
scp	SSH based Secure Copy
SE	Storage Element
ServicesWorker	JEM ServicesWorker
SM	Standard Model of particle physics
SPS	Super Proton Synchrotron
SQL	Structured Query Language
STOMP	Streaming Text Oriented Messaging Protocol
TaskCacheRefresherTask	task cache refresher task
taskId	PanDA task identifier
UI	User Interface
URL	Uniform Resource Locator
userdn	user distinguished name
ValidationTask	validation task object
VO	Virtual Organization
WAN	Wide Area Network
WebDAV	Web-based Distributed Authoring and Versioning
WLCG	Worldwide LHC Computing Grid
WN	Worker Node
WSGI	Web Server Gateway Interface
XML	Extended Markup Language

D. List of Listings

E. List of Tables

F. List of Figures

G. Bibliography

[1] David Griffiths. *Introduction to Elementary Particles*. Wiley-VCH, 2nd edition, 2008.

[2] Wikimedia Commons. *Standard Model of Elementary Particles*, 2006. URL https://commons.wikimedia.org/wiki/File:Standard_Model_of_Elementary_Particles.svg.

[3] Wikimedia Commons. *Elementary particle interactions*, 2008. URL https://commons.wikimedia.org/wiki/File:Elementary_particle_interactions.svg.

[4] Chris Quigg. *Spontaneous Symmetry Breaking as a Basis of Particle Mass*, 2007. URL http://arxiv.org/abs/0704.2232.

[5] Georges Aad et al. *Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. Phys. Lett. B*, 716(arXiv:1207.7214. CERN-PH-EP-2012-218):1, 2012. URL https://cds.cern.ch/record/1471031.

[6] Lyndon R Evans et al. *LHC Machine. J. Instrum.*, 3, 2008. URL https://cds.cern.ch/record/1129806.

[7] Julie Haffner. *The CERN accelerator complex. Complexe des accélérateurs du CERN*, 2013. URL https://cds.cern.ch/record/1621894. General Photo.

[8] Joao Pequenao. *Computer generated image of the whole ATLAS detector*, 2008. URL https://cds.cern.ch/record/1095924.

[9] The ATLAS Collaboration, et al. *The ATLAS Experiment at the CERN Large Hadron Collider. Journal of Instrumentation*, 3(08):S08003, 2008. URL http://stacks.iop.org/1748-0221/3/i=08/a=S08003.

[10] Joao Pequenao. *Event Cross Section in a computer generated image of the ATLAS detector.* Technical report, ATLAS, 2008. URL http://cds.cern.ch/record/1096081.

[11] Ian Bird. *Computing for the Large Hadron Collider. Annu. Rev. Nucl. Part. Sci.*, 61:99, 2011. URL https://cds.cern.ch/record/1447125.

[12] Ian Foster. *What is the Grid? A Three Point Checklist*, 2002.

[13] Ian Foster et al. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[14] Ian Foster et al. *Globus: A Metacomputing Infrastructure Toolkit. International Journal of Supercomputer Applications*, 11:115, 1996.

[15] E. Laure et al. *Programming the Grid with gLite*. In *Computational Methods in Science and Technology*, page 2006, 2006.

[16] M. Romberg. *The UNICORE architecture: seamless access to distributed resources*. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 287–293. IEEE, 1999. URL http://dx.doi.org/10.1109/hpdc.1999.805308.

[17] Patrick Fuhrmann. *dCache: the commodity cache*. In *Twelfth NASA Goddard and Twenty First IEEE Conference on Mass Storage Systems and Technologies*, 2004.

[18] Tools.ietf.org. *RFC 7530 - Network File System (NFS) Version 4 Protocol*, 2015. URL https://tools.ietf.org/html/rfc7530.

[19] Tools.ietf.org. *RFC 2518 - HTTP Extensions for Distributed Authoring – WEBDAV*, 2015. URL https://tools.ietf.org/html/rfc2518.

[20] H Matsunaga, et al. *Data transfer over the wide area network with a large round trip time*. *Journal of Physics: Conference Series*, 219(6):062056, 2010. URL http://stacks.iop.org/1742-6596/219/i=6/a=062056.

[21] Christoph Eck, et al. *LHC computing Grid: Technical Design Report. Version 1.06 (20 Jun 2005)*. Technical Design Report LCG. CERN, Geneva, 2005. URL https://cds.cern.ch/record/840543.

[22] M. Aderholz et al. *Models of networked analysis at regional centres for LHC experiments (MONARC). Phase 2 report*, 2000.

[23] Edoardo Martelli et al. *LHCOPN and LHCONE: Status and Future Evolution. LHCOPN: LHC Optical Private Network*. Technical Report CERN-IT-Note-2015-003, CERN, Geneva, 2015. URL https://cds.cern.ch/record/2017201. The paper is part of the CHEP2015 proceedings, which will appear in the open access Journal of Physics: Conference Series (JPCS).

[24] *Documents & Reference | WLCG*, 2015. URL https://espace2013.cern.ch/WLCG-document-repository/images1/WLCG/WLCG-TiersJun14_v9.png.

[25] *ATLAS Computing: technical design report*. Technical Design Report ATLAS. CERN, Geneva, 2005. URL https://cds.cern.ch/record/837738.

[26] T Maeno. *PanDA: distributed production and distributed analysis system for ATLAS*. *Journal of Physics: Conference Series*, 119(6):062036, 2008. URL http://stacks.iop.org/1742-6596/119/i=6/a=062036.

[27] T Maeno. *PanDA's Role in ATLAS Computing Model Evolution*, 2014. URL https://cds.cern.ch/record/1668644.

[28] J Caballero, et al. *AutoPyFactory: A Scalable Flexible Pilot Factory Implementation*. Technical Report ATL-SOFT-PROC-2012-045, CERN, Geneva, 2012.

[29] M Branco, et al. *Managing ATLAS data on a petabyte-scale with DQ2*. *Journal of Physics: Conference Series*, 119(6):062017, 2008. URL http://stacks.iop.org/1742-6596/119/i=6/a=062017.

[30] C Serfon, et al. *Rucio, the next-generation Data Management system in ATLAS*. Technical Report ATL-SOFT-PROC-2014-009, CERN, Geneva, 2014. URL https://cds.cern.ch/record/1955476.

[31] Vincent Garonne, et al. *The ATLAS Distributed Data Management project: Past and Future*. *Journal of Physics: Conference Series*, 396(3):032045, 2012. URL http://stacks.iop.org/1742-6596/396/i=3/a=032045.

[32] J Schovancova, et al. *The next generation of the ATLAS PanDA Monitoring System*. Technical Report ATL-SOFT-PROC-2014-003, CERN, Geneva, 2014. URL https://cds.cern.ch/record/1695129.

[33] E Karavakis, et al. *Common accounting system for monitoring the ATLAS Distributed Computing resources*. Technical Report ATL-SOFT-PROC-2013-009, CERN, Geneva, 2013.

[34] *ATLAS pilot factory monitor*, 2015. URL http://apfmon.lancs.ac.uk/.

[35] Julia Andreeva, et al. *Automating ATLAS Computing Operations using the Site Status Board*. Technical Report arXiv:1301.0101. ATL-SOFT-PROC-2012-048, CERN, Geneva, 2012. URL https://cds.cern.ch/record/1450176.

[36] Alexey Anisenkov, et al. *AGIS: The ATLAS Grid Information System*. Technical Report ATL-SOFT-PROC-2012-052, CERN, Geneva, 2012. URL https://cds.cern.ch/record/1456348.

[37] P Calafiura, et al. *The Athena Control Framework in Production, New Developments and Lessons Learned*, 2005. URL https://cds.cern.ch/record/865624.

[38] G. Barrand et al. *GAUDI - A software architecture and framework for building LHCb data processing applications*. In *Proceedings of CHEP 2000*, 2000.

[39] Sebastian Johnert. *Measurement of the $W \to \tau\nu\_\tau$ cross section in proton-proton collisions at ATLAS and the development of the HePMCAnalysis tool*. Ph.D. thesis, Hamburg U., 2012. URL http://www-library.desy.de/cgi-bin/showprep.pl?thesis12-009.

[40] G A Stewart, et al. *ATLAS Job Transforms*, 2013. URL https://cds.cern.ch/record/1605832.

[41] */Generators/MC15JobOptions/trunk/share – atlasoff*, 2015. URL https://svnweb.cern.ch/trac/atlasoff/browser/Generators/MC15JobOptions/trunk/share.

[42] Jerome Odier, et al. *Evolution of the Architecture of the ATLAS Metadata Interface (AMI)*. Technical Report ATL-COM-SOFT-2015-088, CERN, Geneva, 2015. URL https://cds.cern.ch/record/2015212?ln=de. May 17th.

[43] T. Sjastrand et al.strand et al. *High-Energy-Physics Event Generation with PYTHIA 6.1*, 2000. Computer Phys. Commun. 135 (2001) 238 (LU TP 00-30, hep-ph/0010017).

[44] T. Gleisberg et al. *Event generation with SHERPA 1.1*, 2009. JHEP02 (2009) 007.

[45] M.L. Mangano et al. *ALPGEN, a generator for hard multiparton processes in hadronic collisions*, 2003. JHEP 0307:001.

[46] J. Allison et al. *Geant4 Developments and Applications*. In *IEEE Transactions on Nuclear Science*, 2006.

[47] Elzbieta Richter-Was, et al. *ATLFAST 2.0 a fast simulation package for ATLAS*. Technical Report ATL-PHYS-98-131, CERN, Geneva, 1998. URL https://cds.cern.ch/record/683751.

[48] R Gardner, et al. *Data Federation Strategies for ATLAS using XRootD*, 2013. URL https://cds.cern.ch/record/1609593.

[49] *Home Page | XRootD*, 2015. URL http://www.xrootd.org/index.html.

[50] Paolo Calafiura, et al. *The ATLAS Event Service: A New Approach to Event Processing*. Technical Report ATL-SOFT-PROC-2015-027, CERN, Geneva, 2015. URL https://cds.cern.ch/record/2016132.

[51] David Cameron. *ATLAS@Home: Harnessing Volunteer Computing for HEP*, 2015. URL https://cds.cern.ch/record/2007503.

[52] Tim dos Santos. *New approaches in user-centric job monitoring on the LHC Computing Grid*. Ph.D. thesis, Bergische Universität Wuppertal, 2011.

[53] T Harenberg. *HEP CG: The High Energy Physics Community Grid Project Inside D-Grid. Overview over the High Energy Physics Community Grid in Germany's D-Grid Initiative. PoS*, ACAT:025, 2007. URL https://cds.cern.ch/record/1116554.

[54] *The Web framework for perfectionists with deadlines | Django*, 2014. URL https://www.djangoproject.com.

[55] *STOMP*, 2014. URL http://stomp.github.io/stomp-specification-1.2.html.

[56] Tools.ietf.org. *RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format*, 2014. URL https://tools.ietf.org/html/rfc7159.

[57] *memcached - distributed memory object caching system*, 2014. URL http://www.memcached.org/.

[58] *Apache ActiveMQ (tm) – Index*, 2014. URL http://activemq.apache.org.

[59] *Relational Grid Monitoring Architecture*, 2015. URL http://www.nesc.ac.uk/events/ahm2003/AHMCD/pdf/148.pdf.

[60] Harvey B. Newman, et al. *MonALISA : A Distributed Monitoring Service Architecture. CoRR*, cs.DC/0306096, 2003.

[61] *MySQL :: The world's most popular open source database*, 2014. URL https://www.mysql.com/.

[62] *GDB: The GNU Project Debugger*, 2014. URL https://www.gnu.org/software/gdb/.

[63] *Application of rule-based data mining techniques to real time ATLAS Grid job monitoring data*, 2014. URL https://cds.cern.ch/record/1450171.

[64] *EGI Operations Portal*, 2015. URL http://operations-portal.egi.eu/vo.

[65] JIRA, 2015. URL https://www.atlassian.com/software/jira.

[66] *ELOG*, 2015. URL http://midas.psi.ch/elog/.

[67] P Nilsson et al. *The ATLAS PanDA Pilot in Operation. Journal of Physics: Conference Series*, 331(6):062040, 2011. URL http://stacks.iop.org/1742-6596/331/i=6/a=062040.

[68] Ahmad Hammad. *Entwicklung eines Überwachungssystems für verteilte Prozesse im LHC-Computing Grid*, 2005.

[69] Dmitri Igdalov. *Entwicklung eines Systems zur Analyse und Überwachung der Verarbeitung von Rechenanforderungen im LHC Computing-Grid*, 2005.

[70] Andreas Baldeau. *Skriptüberwachung im Job-Execution-Monitor für das LHC Computing Grid*, 2007.

[71] Dr. Stefan Borovac. *A users guide to JEMv2*. Bergische Universität Wuppertal, 2007. URL http://www.atlas.uni-wuppertal.de/grid/jms/JEMv2-guide.pdf.

[72] Frederic Brochu, et al. *Ganga: a tool for computational-task management and easy access to Grid resources. CoRR*, abs/0902.2685, 2009.

[73] Martin Rau. *Erweiterung der Benutzerschnittstelle für LHC Grid Jobs um die Monitoring-Funktionalität*, 2008.

[74] Markus Mechtel. *Studies on Resonances of Top Quark Pairs and Development of a Grid Expert System*. Ph.D. thesis, Bergische Universität Wuppertal, 2011.

[75] Raphael Ahrens. *Entwurf und Implementierung eines sicheren Rückkanals für die Grid-Monitoring Software JEM*. Master's thesis, Fachhochschule Köln, 2013.

[76] *Standard ECMA-262 ECMAScript Language Specification*, 2011. URL http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[77] *jQuery*, 2014. URL http://jquery.com/.

[78] *Highcharts - Interactive JavaScript charts for your webpage*, 2014. URL http://www.highcharts.com/.

[79] *DataTables (table plug-in for jQuery)*, 2014. URL http://www.datatables.net/.

[80] *South*. Technical report, 2015. URL http://south.aeracode.org.

[81] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus*. Oldenbourg Wissenschaftsverlag, 1997.

[82] *Python - Debugging with GDB*, 2015. URL https://sourceware.org/gdb/current/onlinedocs/gdb/Python.html#Python.

[83] *Announcing ncurses 5.9 - GNU Project - Free Software Foundation (FSF)*. Technical report, 2015. URL https://www.gnu.org/software/ncurses/ncurses.html.

[84] *SQLite Home Page*, 2015. URL https://www.sqlite.org/.

[85] P Sherwood et al. *The ATLAS RunTimeTester*. Technical Report ATL-COM-SOFT-2012-153, CERN, Geneva, 2012. URL https://cds.cern.ch/record/1471395.

[86] *AtlasProductionGroup < AtlasProtected < TWiki*, 2014. URL https://twiki.cern.ch/twiki/bin/view/AtlasProtected/AtlasProductionGroup.

[87] *Online Diagram Software and Flow Chart Software - Gliffy*, 2015. URL https://www.gliffy.com.

[88] *Clipart - Thermometer*, 2014. URL https://openclipart.org/detail/110065/thermometer.

[89] *Changeset 7362 for pilot3/JEMstub.py - panda*, 2014. URL https://svnweb.cern.ch/trac/panda/changeset/7362/pilot3/JEMstub.py.

[90] *RecoTRF - Special Syntax*, 2014. URL https://twiki.cern.ch/twiki/bin/viewauth/AtlasComputing/RecoTrf#Special_syntax_for_preExec_postE.

[91] *JobExecutionMonitor < AtlasComputing < TWiki*, 2014. URL https://twiki.cern.ch/twiki/bin/viewauth/AtlasComputing/JobExecutionMonitor.

[92] *Flow Chart Maker & Online Diagram Software | Lucidchart*, 2015. URL https://www.lucidchart.com.

[93] *PanDA Monitor:*, 2014. URL http://pandamon.cern.ch/tasks/listtasks1.

[94] *DCube Documentation*, 2014. URL https://twiki.cern.ch/twiki/bin/view/Sandbox/DCubeDoc.

[95] Rene Brun et al. *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/.

[96] *PYROOT | ROOT*, 2014. URL http://root.cern.ch/drupal/content/pyroot.

[97] *Built-in template tags and filters | Django documentation | Django*, 2014. URL https://docs.djangoproject.com/en/dev/ref/templates/builtins/.

[98] *PandaDB < AtlasComputing < TWiki*, 2014. URL https://twiki.cern.ch/twiki/bin/viewauth/AtlasComputing/PandaDB.

[99] *Panda assigned production jobs*, 2014. URL http://panda.cern.ch/server/pandamon/query?job=*&cloud=ALL&type=production&jobStatus=assigned&hours=12.

[100] Bertrand Bellenot. *ROOT I/O in JavaScript. J. Phys.: Conf. Ser.*, 396:052011. 8 p, 2012. URL https://cds.cern.ch/record/1515907.

[101] Andy Buckley, et al. *Rivet user manual*. Technical Report arXiv:1003.0694. MCNET-10-03, 2010. URL https://cds.cern.ch/record/1246848.

[102] Frederick James. *Statistical Methods in Experimental Physics: 2nd Edition*. World Scientific Publishing Company, 2nd edition, 2006.

All URLs have been checked and accessed at November 19, 2016.