PROCEEDINGS OF SCIENCE

Job centric monitoring for ATLAS jobs in the LHC Computing Grid

Erich Ehses[a], Torsten Harenberg[b], Dennis Huning[bd], Peter Mättig[b], Markus Mechtel[b], Tim München*[bd], Martin Rau[a], Peer Ueberholz[c], Nikolaus Wulff[d]

[a] *Fachhochschule Köln, Claudiusstr. 1, 50678 Köln, Germany*
[b] *Bergische Universität Wuppertal, Gaußstraße 20, 42119 Wuppertal, Germany*
[c] *Hochschule Niederrhein, Reinarzstraße 49, 47805 Krefeld, Germany*
[d] *Fachhochschule Münster, Hüfferstraße 27, 48149 Münster, Germany*

As the Large Hadron Collider (LHC) at CERN, Geneva, has begun operation in September, the large scale computing grid LCG (LHC Computing Grid) is meant to process and store the large amount of data created in simulating, measuring and analyzing of particle physic experimental data. Data acquired by ATLAS, one of the four big experiments at the LHC, are analyzed using compute jobs running on the grid and utilizing the ATLAS software framework ATHENA. The analysis algorithms themselves are written in C++ by the physicists using Athena and the ROOT toolkit.

Identifying the reason for a job failure (or even the occurance of the failure itself) in this context is a tedious, repetitive and sometimes unfeasable task. Often, to deal with failures in the RUNNING stage (as opposed to job submission failures or compilation errors in the user algorithms), the job is just being resubmitted. In a grid context, users are never allowed to directly access the worker nodes (WNs) their jobs run on. So, debugging of a user job interactively is not possible. It is even more complicated because the output-sandbox, which contains the jobs' output and error logs, usually is discarded by the grid middleware if the job failed. So, valuable information that could aid in finding the reason of failure is lost. These issues result in high job failure rates and less than optimal resource usage.

As part of the High Energy Particle Physics Community Grid project (HEPCG) of the German D-Grid Initiave, the University of Wuppertal has developed the Job Execution Monitor (JEM). JEM helps finding reasons for job failures by two means: it periodically provides vital worker node system data and collects job run-time monitoring data. By performing a supervised line-by-line execution of the user job, this data is gathered. JEM provides new possibilities to find problems in largely distributed computing grids and to analyze these problems in nearly real-time.

The monitoring information is presented to the user almost instantaneously for realtime analysis and, for completed jobs, additionally stored in the jobs' output sandbox for deeper post-mortem analysis. As a first step, JEM is being integrated into ATLAS' and LHCb's grid user interface Ganga. Jobs submitted in this way are monitored transparently, requiring no additional effort by the user.

In this work, the functionality of and the concepts behind JEM are presented together with examples of typical problems that can easily be resolved using it. Furthermore, we present an ongoing work of classifying problems automatically using expert systems.

XII International Workshop on Advanced Computing and Analysis Techniques in Physics Research
Ettore Majorana Foundation and Centre for Scientific Culture, Erice, Sicily
November 3-7, 2008

PoS(ACAT08)042

This work is divided into four sections. In the first section, the motivation for the development of a user centric job monitoring tool, regarding the LHC experiment ATLAS as context of the jobs to be monitored, will be described. The second section covers the functionality and concepts behind JEM as well as a short description of some typical failures together with their reasons occuring in the aforementioned context which can be found using it. After the proposed interoperation between JEM and a grid centric expert system (GridXP), also developed at the University of Wuppertal, is presented in section three, we conclude with an outlook of things being developed at the moment and opportunities for future work.

1. Motivation

1.1 Grid jobs in the ATLAS experiment

In the ATLAS experiment, grid jobs usually consist of

- an algorithm doing calculations on existing data or data generation using the ATLAS software framework ATHENA[1],
- code setting up ATHENA and the environment,
- ATHENA runtime code providing services like data I/O, logging and bookkeeping,
- a short launcher application, usually implemented as shell-script,
- and of course the data that is being processed.

The user algorithm itself is implemented in C++ and utilizes library functions provided by ATHENA and mathematical libraries such as ROOT[2]. It is this code that does the actual physics work: the generation of physics data (e.g. Monte Carlo or detector simulation), the conversion of this data into other formats for further processing, or the analysis of such data. The ATHENA environment is set up by a collection of shell-scripts. Their main purpose is to set up binary search paths, data sources and other runtime information. ATHENA itself consists of several modules written in Python which prepare the analysis[1] data and repeatedly call the user algorithm with data fragments (e.g. single physic event data). For this, the user algorithm is loaded beforehand as a shared library by ATHENA and used by means of inter-language-calls by the Python interpreter. The actual grid job typically is a short launcher script that loads the ATHENA environment-setup scripts and triggers execution of ATHENA, either written by the user or automatically created by a job submission and management tool like Ganga[3].

To summarize, ATLAS grid jobs are generated or user-written shell scripts running a framework of Python scripts that use algorithms from C++ shared libraries. The fact that this application model consists of several layers depending on each other, implemented by different groups of developers using different programming languages, already implies a number of possible failure conditions and errors. On top of that, one observes the usual errors contained in all human-written software (invalid data and memory access, floating point exceptions, logical errors, etc) and errors

*Speaker.

[1] or simulation, data generation, conversion, etc.

caused by the context, that is, the grid (submission errors, failures to query needed local and remote services, environment errors and so on). Hence, there is a wealth of failure possibilities from which only a small subset is under control of the user. These errors are classified in the next section, considering the possibilities to determine the failure reason with the help of the job monitoring software.

1.2 Job failures

For grid jobs, there are two classes of job failures to distinguish in respect to the possibilities of the user who has to determine the cause of the error. They can be seperated by the status assigned to it by the grid middleware after execution. To avoid confusion, we will explicitely seperate the job into its "grid-part" and the "physics-part" in the following discussion. One has to be aware that both parts combined form each job.

- *Finished*
  If the grid job successfully leaves the state RUNNING, it is being assigned the state FINISHED by the grid middleware. Successful in this respect means there was no middleware- or environment-specific cause of job failure. Nevertheless, the physics job may have failed. The job script terminating with a non-zero exit code most often means the physics job didn't complete successfully. Even so, its grid job status will be set to FINISHED instead of FAILED. Moreover, even with an exit code of zero, the physics job may not have produced the desired output (logical failure). The output of the job (stdout and stderr streams containing all output of both the grid part and the physics part of the job) is usually transmitted to the user via the job's output sandbox, an archive packed by the grid middleware after the job finished, containing this output as well as any data file created by the job (assuming they are correctly declared in the job's description). If the job exits with a nonzero exit code, these files are the only data the user can analyze to determine the cause of the failure. Errors during ATHENAs environment setup, for example, usually result in messages written by the remote shell into stderr, whereas errors in ATHENA itself (meaning in its Python scripts, not during the inter-language-calls into the user algorithm) usually result in stacktrace information also written to stderr. Fatal errors during the user algorithm, that is, errors resulting in the abortion of the current inter-language-call (e.g. floating point exceptions or access violations) result in sparse stderr output from the user algorithm and, additionally, in a stacktrace generated by an error handler triggered by ATHENA, containing the last stack frames of the user algorithm (see fig. 1).

- *Failed* or *Aborted*
  If the grid job fails for grid specific reasons, that means the grid middleware assigns the status FAILED to it, the output sandbox usually is discarded and stderr is not available. This also holds for jobs that got aborted by the user, for example because the job seems to run indefinitely and never finishes. In this case, the job status is set to ABORTED.

Obviously, the user has little information to work with when his job fails. In the first class of failures, the stderr output can be analyzed, but often it contains insufficient data to find the error cause fast and efficiently. The data is either insufficient by itself or hidden in pages of irrelevant

error output created by the several layers of the application described in Sec. 1.1. In the second class of failures, the only information the user has is that the job, in fact, failed. As a result, the neccessity of a tool that monitors the job run is evident for second class failures, but also first class failures can be resolved easier and faster if more information is available, and the information that is available is structured in an efficient way. The following section presents our job monitoring solution JEM that can be used to achieve these goals.

2. The Job Execution Monitor

2.1 Overview

The Job Execution Monitor[4, 5, 6] (JEM) is an application run in user space and submitted with user jobs to grid computing elements (CEs), providing job status, job error and environment data to the user in nearly real-time. During the run of the user job, vital system information on the worker node (WN) the job was scheduled to run on, like CPU and memory usage and free disk space in the job's working directory and temporary directories, is gathered periodically. The scripts the user job itself consists of[2] are run in a supervised line-by-line mode, during which events like function and method calls, returns from functions and methods, exceptions being raised, or shell commands being executed are logged.

2.2 General Structure

JEM consists of two distinguished parts (fig. 2). The first component, JEM UI, is run on the machine used to submit the monitored grid job[3]. It can be run as a stand-alone tool providing a

```
Generating stack trace...
0xf1891ecd in HelloAlg::execute() at ../src/HelloAlg.cxx:124 from /home/atlas029/globus-tmp.wn
0xf3021ef8 in Algorithm::sysExecute() + 0x98 from /griddata/gridsoft/atlas/prod/releases/rel_1
0xf24c208e in AthenaEventLoopMgr::executeAlgorithms() + 0x4e from /griddata/gridsoft/atlas/pro
0xf24c5a9c in AthenaEventLoopMgr::executeEvent(void*) + 0xc6c from /griddata/gridsoft/atlas/pr
0xf24c799b in AthenaEventLoopMgr::nextEvent(int) + 0x33b from\ /griddata/gridsoft/atlas/prod/r
0xf24c1b17 in AthenaEventLoopMgr::executeRun(int) + 0x17 from /griddata/gridsoft/atlas/prod/re
0xf1cf1974 in ApplicationMgr::executeRun(int) + 0x234 from /griddata/gridsoft/atlas/prod/relea
0xf2cf0d34 in <unknown> from /griddata/gridsoft/atlas/prod/releases/rel_14-2/GAUDI/v19r6p4-LCG
0xf379d25c in ROOT::Cintex::Method_stub_with_context(ROOT::Cintex::StubContext_t*, G__value*,
0x0aa725de in <unknown function>
0xf4cd6f14 in Cint::G__CallFunc::Execute(void*) + 0x94 from /griddata/gridsoft/atlas/prod/rele
0xf59f291b in PyROOT::TRootObjectByValueExecutor::Execute(Cint::G__CallFunc*, void*) + 0x23 fr
0xf59f793a in PyROOT::TMethodHolder<PyROOT::TScopeAdapter, PyROOT::TMemberAdapter>::Execute(vo
0xf59facb3 in PyROOT::TMethodHolder<PyROOT::TScopeAdapter, PyROOT::TMemberAdapter>::operator()
0xf59ff588 in PyROOT::(anonymous namespace)::mp_call(PyROOT::MethodProxy*, _object*, _object*)
0xf7e4369c in PyObject_Call at Objects/abstract.c:1861 from /griddata/gridsoft/atlas/prod/rele
0xf7eb7654 in PyEval_EvalFrameEx at Python/ceval.c:3777 from /griddata/gridsoft/atlas/prod/rel
0xf7ebd865 in PyEval_EvalCodeEx at Python/ceval.c:2833 from /griddata/gridsoft/atlas/prod/rele
0xf7ebb76d in PyEval_EvalFrameEx at Python/ceval.c:3661 from /griddata/gridsoft/atlas/prod/rel
0xf7ebd865 in PyEval_EvalCodeEx at Python/ceval.c:2833 from /griddata/gridsoft/atlas/prod/rele
0xf7ebd9f3 in PyEval_EvalCode at Python/ceval.c:500 from /griddata/gridsoft/atlas/prod/release
0xf7edd6a0 in PyRun_FileExFlags at Python/pythonrun.c:1264 from /griddata/gridsoft/atlas/prod/
0xf7eb1b18 in <unknown> from /griddata/gridsoft/atlas/prod/releases/rel_14-2/sw/lcg/external/P
...
```

Figure 1: ATHENA-generated stacktrace of a crash in the user algorithm.

[2] bash- or python-scripts, see sec. 2.3

[3] Typically; However, the JEM UI can be run on any machine.

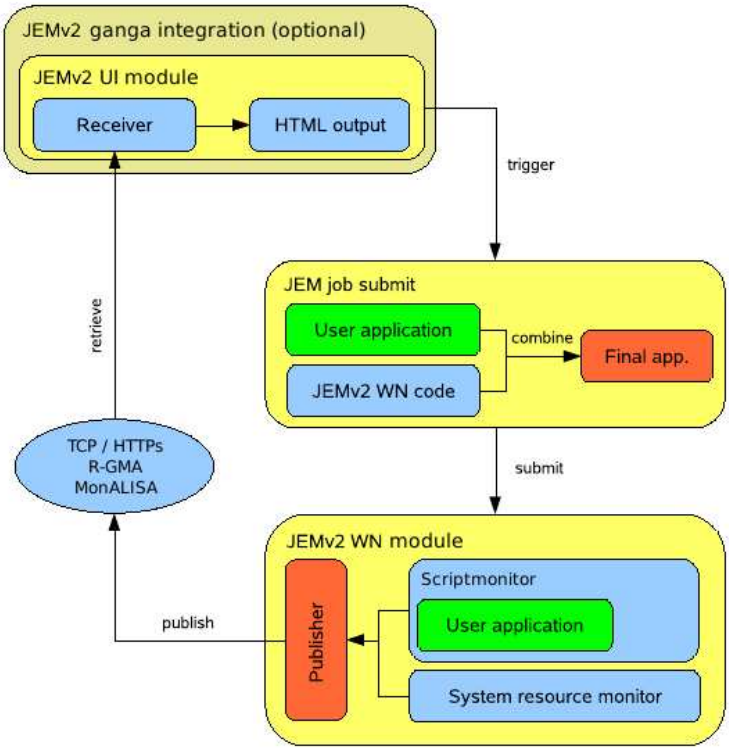

Figure 2: General structure of JEM[7].

simplified job submission user interface, as command-line tool only decorating a usual grid job with the monitoring functionality before submitting, or as a plug-in into ATLAS' job management solution Ganga, adding the job monitoring benefits transparently to Ganga's job submission functionality. The latter is the preferred mode of execution of JEM, requiring no additonal effort by the user besides setting a monitoring flag in the job's description in Ganga.

The second component, JEM WN, is a job wrapper that is submitted to the worker node with the user job and augmenting it with the monitoring functionality. The user job is encapsulated in JEM WN and is run on the WN after JEM initialized and started its data taking and transmission subsystems which will be described below. While the user job is run line by line and environment data is taken periodically, all gathered monitoring data is transmitted from JEM WN to JEM UI in nearly real-time.

Both parts of JEM are mainly implemented in python, having only a minimal set of software dependencies easily provided by typical grid CEs. The transmission of monitoring data from the worker node script to the JEM UI application or -plugin is performed by one of a set of communication modes (named "valves") using different communication backends provided by JEM. There is another ongoing work evaluating the different valves and comparing their applicability for JEMs purpose in terms of data throughput and caused system load on the WN and the UI machines.

2.3 Script monitors

At present, JEM is able to monitor bash- and python-scripts. Calls to other scripts automatically cause the script monitor to fork and monitor the child script as well (if it is a monitorable script as well). The following information is logged during the supervised script execution by JEM WN (for every logged event, besides the data listed in the following paragraphs, the timestamp of the event and the type of script - bash, python - are gathered):

- **function calls** Whenever a (python) function or class-method is called, the called and calling frame are logged with filename, frame name and line number. The arguments passed to the called function are logged as well including a (possibly shortened) string representation of their value.

- **function returns** If the python script execution leaves a function/method, the left frame with file and line number as well as the return value (string representation of the value) is logged.

- **run commands** In bash scripts, the called shell commands, run applications and syntactical constructs (loops, conditions, ...) are logged with their respective exit code.

- **exceptions** Python exceptions are logged with the causing frame (frame name, line number, file name), exception class name and reason (informative string). Furthermore, the values of local variables of the scope where the exception happened are gathered.

The amount of this data that is actually logged can be configured by several criteria to allow the reduction of traffic and load on the sending (WN) and receiving (UI) sides. These criteria include a loglevel setting reducing the overall verbosity of the script monitor, blacklists to skip whole scripts, exception type filters and, as last resort, the discarding of data at random if too much data is generated. There exists an ongoing additional work to make the data reduction more intelligent and expedient.

2.4 System monitor

The system resource monitor (also called "Watchdog") periodically[4] gathers the following monitoring data on the WN:

- CPU usage / system load
- Memory consumption / free memory
- Free disk space in the job's working directory and temp (scratch) space
- The hostname of the worker node
- The point in time this data was taken at

[4]The interval can be configured inside JEM

More often than not, this data is very useful to determine the reason for a job failure that otherwise presents itself the same way a coding error or (grid-) environment error does. On the UI side, the watchdog data can be presented to the user in form of graphs (e.g. CPU usage over time), to visualize trends and to identify temporary problems on the WN that possibly have caused the job failure.

2.5 Examples

The following examples for typical failure scenarios in ATHENA grid jobs can be resolved by using JEM. The failed job is described, the information gathered by JEM in respect to the failures is shown and the therefor possible resulting interpretation by the user is given.

Log file exceeding disk space If the log file written by the grid job heads out, exceeding the available disk space on the worker node, the job cannot continue to run. Aditionally, the middleware stores information in the very same directory and will not be able to finish and write the job information back. A typical error that will be presented to the user in this case is shown in fig. 3.

The job failure caused by the unsuccessful attempt to append to the job's log file is nearly impossible to detect without a real-time job monitoring facility like JEM, because for this failure reason, the middleware instantly aborts the job and the affected job directory is deleted by the batch system. So, no further log output besides status message shown in fig. 3 is available to the user, and this message clearly doesn't point at a disk space problem.

By using JEM, the amount of available disk space in the working directory of the job is recorded and can be presented to the user at regular intervals already during the job run. By comparing several values, a trend can be deduced, that may advert to an imminent job failure caused by exceeding of disk space.


```
**************************************************************
BOOKKEEPING INFORMATION:

Status info for the Job : https://wmslb00.hep.ph.ic.ac.uk:9000/Zd6QUUdfw_-ClFJwHCEM-A
Current Status:     Aborted
Logged Reason(s):
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
    - File not available.Cannot read JobWrapper output, both from Condor and from Maradona.
Status Reason:      hit job shallow retry count (10)
Destination:        ce02.tier2.hep.manchester.ac.uk:2119/jobmanager-lcgpbs-atlas
Submitted:          Sun Dec  7 00:53:08 2008 GMT
---

- Cancelling                =    0
- Ce node                   =    gt2 ce02.tier2.hep.manchester.ac.uk:2119/jobmanager-lcgpbs
- Children num              =    0
```

Figure 3: Example of a grid job whose log file size exceeds the available disk space on the worker node (grid job status as reported by the middleware is shown here).

Exceeding the wall time limit Grid jobs are not permitted to run indefinitely on the worker node. Computing elements typically offer several grid job queues providing different limits of valid job run times. The user has to define an estimate about the runtime of a job in the job definition file.

If this estimate is too low, jobs are killed by the batch system and a signal (SIGUSR1 and later SIGUSR2) is sent to the job. Due to the complexity of the software framework, the signals may be catched and the job is killed in a hard matter, obfuscating the real failure reason (Wall time limit exceeded), or the job simply behaves unexpectedly. Using JEM, the signals and their source can be clearly identified[5], without the need for the user to code additional signal handlers.

ATHENA version mismatch Updates to the ATHENA software framework are released on a regular basis. One distinguishes between major and minor releases. Usually, code and data files are compatible between different minor releases while they cannot be exchanged generally between releases of different major release numbers.

However, as data taking is about to begin shortly, the developers try to unify at least the data format, so that measured and simulated data can be read with different releases (or at the very least, data created with one release should be usable with any newer release).

In some cases, however, code still may crash or behave otherwise unexpectedly if for example a data file written with an older release is analysed with a newer one. These errors are hard to detect (for example, the code runs forever without any meaningful output). In most cases some structures which are accessed by the analysis code just are not filled or are filled with different (mismatching) types of data.

Using JEM's C-Tracer and JEMpole (see 4.2) the user has the possibility to analyse the data structures read by the analysis code just before it crashes or at any other time during runtime and compare it with the expectations made in the analysis code. This is often much easier than digging into the very large repository of the ATHENA framework and to search for the changes made between the two affected versions, especially if one has no idea which data structures cause the problems. They can be easily identified with JEM.

3. GridXP

3.1 Overview

GridXP[8] is a rule-based expert system with a client/server architecture. It was developed at the University of Wuppertal to aid users in finding the reason for job failures and to suggest steps to take to handle the failure. GridXP and another expert system, the Pixel Advisor, developed to help the ATLAS experiment's operators at CERN to deal with error conditions in ATLAS' Detector Control System (DCS), are combined into UnifiedXP, a unified expert system framework sharing core services, the visualisation and user interface[9].

If an error occured during the run of a grid job, a user must review almost all collected information about it to determine what was the problem. GridXP reviews all available information automatically. If it recognises that a job finished with an error it visualises this and describes why it has failed. Futhermore it gives one or more suggestions how the problem might be fixed. A

[5]This holds for signals in Python modules. To see signals in C/C++-Modules, the C-Tracer is needed (see sec. 4.2).

user can rate the advice “positive” if it was helpful or “negative” if it was unhelpful. This rating is used as a quantifier over all advices, eventually changing their displayed order in the suggestions list. Additionally, to aid in finding error reasons, it can combine all data fed into it about a WN to determine if an error originates from the infrastructure or the job itself.

3.2 Proposed interoperation of JEM and GridXP

At present, GridXP utilizes real-time job information provided by means of the Relational Grid Monitoring Architecture, R-GMA[10]. Among the information gained in this way one can find job state transitions like job start- and job end-events, but no detailed real-time monitoring data. A direct interface from JEM into GridXP is planned, that will allow the GridXP ruleset to fully take advantage of the power of monitoring provided by JEM. In case of a job failure, GridXP will be able to consider not only the environment information (worker node system monitoring data and grid status), but also the current execution status of the job, commands executed and possibly exceptions thrown just before the failure. So, GridXP greatly benefits from JEMs input. On the other hand, JEM also benefits from GridXPs possiblilities to automatically interpret the monitoring data and visualize the result to the user in a much more understandable way than by directly looking at the logs.

```
================================================================================================================================
=========
                                                Welcome to ApplicationMgr $Revision: 1.72 $
                                          running on wn064 on Tue Jan  6 12:43:12 2009
================================================================================================================================
=========
ApplicationMgr       INFO Successfully loaded modules : AthenaServices
ApplicationMgr       INFO Application Manager Configured successfully
ApplicationMgr       INFO Updating ROOT::Reflex::PluginService::SetDebug(level) to level=0
StatusCodeSvc        INFO initialize
---- List of all Volume Builders registered with Geo2G4Svc ----
------------------------------------------------------------
 Volume Builder  Extended_Parameterised_Volume_Builder
 Volume Builder  Generic_Volume_Builder
 Volume Builder  Parameterised_Volume_Builder
 Volume Builder  Single_LV_Copy_Builder
------------------------------------------------------------
 default builder is Extended_Parameterised_Volume_Builder
XMLFileCatalog: level[Info] Connecting to the catalog
PoolXMLFileCatalog: level[Info] Xerces-c initialization Number 0
PoolXMLFileCatalog: level[Info] Read-only filesystem
PoolXMLFileCatalog: level[Info] XercesC termination number 0
XMLFileCatalog: level[Info] Disconnected
WARNING: $POOL_CATALOG is not defined
using default `xmlcatalog_file:PoolFileCatalog.xml'
XMLFileCatalog: level[Info] Connecting to the catalog
PoolXMLFileCatalog: level[Info] Xerces-c initialization Number 0
XMLFileCatalog: level[Info] Connecting to the catalog
PoolXMLFileCatalog: level[Info] File PoolFileCatalog.xml does not exist, a new one is created
PoolXMLFileCatalog: level[Info] Read-only filesystem
PoolSvc           WARNING File is not in Catalog or does not exist.
PoolSvc           WARNING Do not allow this ERROR to propagate to physics jobs.
ERROR (pool):
POOL> Unknown storage type requested:
ImplicitCollection  Warning Cannot find persistency storage type. Trying ROOT_StorageType
Warning in <TClass::TClass>: no dictionary for class DataHeader_p3 is available
Warning in <TClass::TClass>: no dictionary for class DataHeaderElement_p3 is available
McEvent.ttbar.7.root   Always Root file version:51800
McEvent.ttbar.7.root   Always Root file version:51800
POOLContainer_DataHeader    Error The requested container:POOLContainer_DataHeader cannot be opened!
InputMetaDataStoreWARNING retrieve(const): No valid proxy for object MetaDataSvc  of type DataHeader(CLID 222376821)
MetaDataSvc       WARNING Unable to load MetaData Proxies
```

Figure 4: Example of a athena job running reprocessing code with a generated data file written by an older release. In this example, simply nothing happens after this step, so the user is lost.

4. Conclusion

4.1 Status of JEM

In its current form, JEM already can be used to monitor typical grid jobs to gather valuable information about job failures or the current job status in nearly real-time. Errors with misleading or hidden messages can be classified by the user more easily. Often, the real-time information even predicts job failures about to occur, allowing to abort and re-submit the job with corrected or optimized settings and parameters. This shortens the overall round-time needed for successful job runs and leads to more efficient resource usage.

Naturally, the benefit of using a monitoring framework like JEM allowing to determine error reasons has its price. The user job augmented with JEM needs longer as JEM's functionality causes a certain performance impact. The exact impact on run time and resource usage is still to be determined (but such efforts are already being taken, see sec. 4.3), but rough estimates can be given here. The initialization of JEM on the worker node that is performed before the user job is launched needs a short constant time in the region of ten seconds. Functions in traced Python modules take roughly 1.5 times longer to execute, depending on their complexity (The longer the functions themselves take the less the impact of JEM becomes). If C/C++-modules are traced as well (see sec. 4.2) the performance impact during the user analysis (Interlanguage-calls into shared libraries containing physics analysis code) at the moment[6] increases by at least a factor of three, up to some 10-fold impact if the user algorithm's memory is inspected deeply. This means, of course, that JEM cannot be just submitted in full verbosity by default for every job submitted. Instead, jobs can be resubmitted with increasing JEM-verbosity when errors should be inspected in more detail.

JEM can be obtained from the project website and used out-of-the-box with only a minimum of configuration needed. The integration into Ganga is nearly production-ready, but is not included in Ganga's releases yet.

4.2 Current Development

JEM is still being developed further. There are several side-projects being worked on, leveraging JEMs functionality and further enriching the monitoring data that can be gathered by using JEM.

Environment monitor JEM already monitors local environment metrics on the worker node the job is launched on (see 2.4), but the status of the grid environment beyond a single machine often can give additional, useful hints at possible error conditions and reasons for job failures. For example, if necessary grid services (information service, workload manager, local storage element) are not available or unreachable by the grid job, it fails. JEMs new environment monitor attempts to supervise just this kind of environment data by periodically polling needed services and assuring the infrastructure is in a usable state.

JEMpole, the Post-Mortem Log Explorer Monitoring information received in real-time on the submission machine or, via the output sandbox if a job finished, is a valuable source for analysing job failures. Often, however, the amount of information is not easy to filter for the user, and the

[6] The C-Tracer code still is being optimized

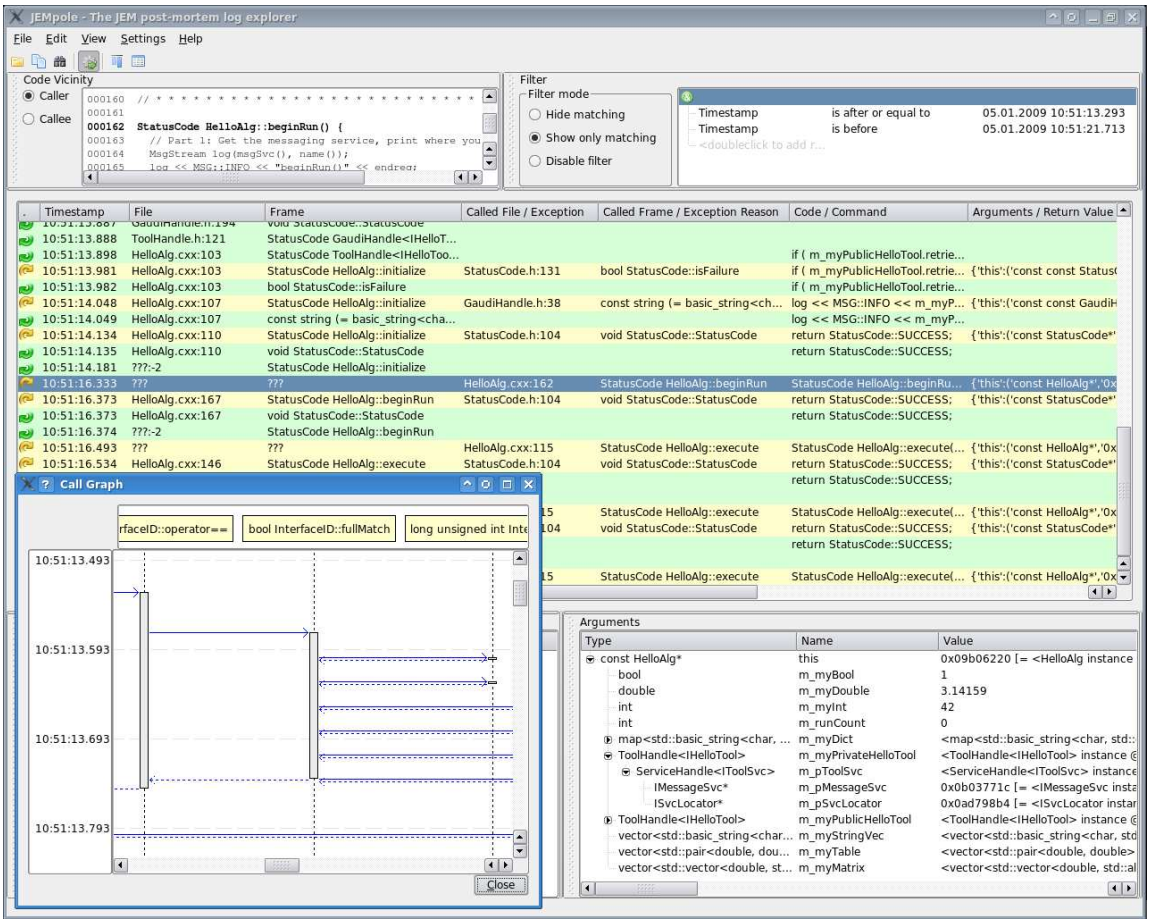

Figure 5: JEMpole main window: Color-coded event list (center), code vicinity view (top left), filter criteria (top right), function call graph (bottom left) and function argument browser (bottom right).

search for specific entries pointing to the reason for a problem with the job is still a challenging task. The newly developed log viewer, JEMpole (fig. 5), attempts to simplify such searches. It allows to browse JEMs logfiles in a color-coded, clearly arranged list that can be sorted and filtered by several criteria. Multiple filters can be linked by boolean expressions, and can contain regular expressions to make the filtering even more powerful. JEMpole can view function arguments or local scope variables, create a call-graph showing the flow of execution just before a job abort, view a caller-report showing what functions call what other functions (and how often) and view the source code around the location of events (calls, returns, exceptions) if available. All these functions help in understanding what exactly happened in the user code at a specific point in time.

C-Tracer Without JEM, all of a grid job's run during the job state RUNNING is a blind spot for the user. Submission- and middleware-errors are reported as such, but errors in the job's run itself only give sparse error messages, or nothing at all. JEM, up until now, reports what happens during running state, as long as the job processes (bash) scripts and python programs. The physics data analysis itself, however, most often consists of compiled binaries (C/C++ shared libraries loaded

and called by the python launcher script). This holds for JEMs main place of action, ATLAS data analysis. With the new C-Tracer library developed specifically for JEM, the blind spot is narrowed even more, as now the job's actions are also reported from inside the compiled user algorithm. The C-Tracer logs function or method calls and corresponding returns in/from C or C++ code and passes this information to JEM, including variable and parameter values and types.

4.3 Outlook

There are several possibilities of future work concerning JEMs development, some of them already planned and scheduled as further projects (Master- and PhD-Theses at the University of Wuppertal). Among them are:

Evaluation and optimization of the data transfer JEM offers several different data transport protocols to choose from, including plain HTTPS transport, R-GMA and MonALISA[11]. Each and every one of them has to be stress-tested for latency and throughput, as well as for network, CPU and RAM usage at transmitter and receiver. With the information obtained in that way, a preferred mode of transportation can be chosen and preconfigured in JEM.

Evaluation of the influence on job performance and throughput Besides the resource consumption mentioned in the previous paragraph, the performance impact of JEM as a whole has to be determined. Measurements are to be taken comparing mediate job runtime with and without JEM in different scenarios (With the C-Tracer and without, with file system logging and without, etc).

Full utilization of JEM data in GridXP When the interface of JEM into GridXP (see 3.2) has been implemented, GridXP itself has to be built upon, implementing a sophisticated rule set to use the monitoring information provided by JEM to create meaningful error description and solution suggestions in GridXP. On top of that, the usage of cutting edge technology to classify errors by their output and by JEM's logging data is to be implemented into GridXP, for example by means of neural networks and fuzzy logic.

References

[1] ATLAS Computing Group. *ATLAS Computing Technical Design Report.*
ATLAS-TDR-017, CERN-LHCC-2005-022
`http://www.atlas.ac.uk/comp.html`

[2] R. Brun, F. Rademakers. *ROOT - An Object Oriented Data Analysis Framework.*
AIHENP conference, Laussanne, 1996.
`http://root.cern.ch`

[3] F. Brochu et al. *Ganga: a tool for computational-task management and easy access to Grid resources.*
`http://ganga.web.cern.ch/ganga/documents/pdf/ganga_cpc09.pdf`
`http://ganga.web.cern.ch/ganga`

[4] D. Igdalov. *Entwicklung eines Systems zur Analyse und Überwachung der Verarbeitung von Rechenanforderungen im LHC Computing-Grid.*
Diploma thesis, Fachhochschule Niederrhein, August 2005.

[5] A. Hammad. *Entwicklung eines Überwachungssystems für verteilte Prozesse im LHC-Computing Grid*. Master thesis, Fachhochschule Niederrhein, December 2005.

[6] D. Igdalov, A. Hammad, S. Borovac, M. Mechtel, T. München et al. *JEM, The Job Execution Monitor*. `http://www.grid.uni-wuppertal.de/grid/jms`

[7] S. Borovac. *A users guide to JEMv2*. Bergische Universität Wuppertal, 2007.

[8] T. Henß et al. *GridXP, A grid centric expert system*. `http://www.grid.uni-wuppertal.de/grid/expertsystem`

[9] T. Henß. *Entwurf und Implementation eines Expertensystems für das Detektorkontrollsystem des ATLAS-Pixeldetektors*. PhD. thesis, Bergische Universität Wuppertal, WUB-DIS 2008-08, 2008.

[10] The EGEE project. *Information and Monitoring Service (R-GMA) – System Specification*. EGEE-JRA1-TEC-490223-R_GMA_SPECIFICATION-v2-0, Juli 2004.

[11] The CERN MonALISA project group. *MONitoring Agents using a Large Integrated Services Architecture*. `http://monalisa.cacr.caltech.edu`