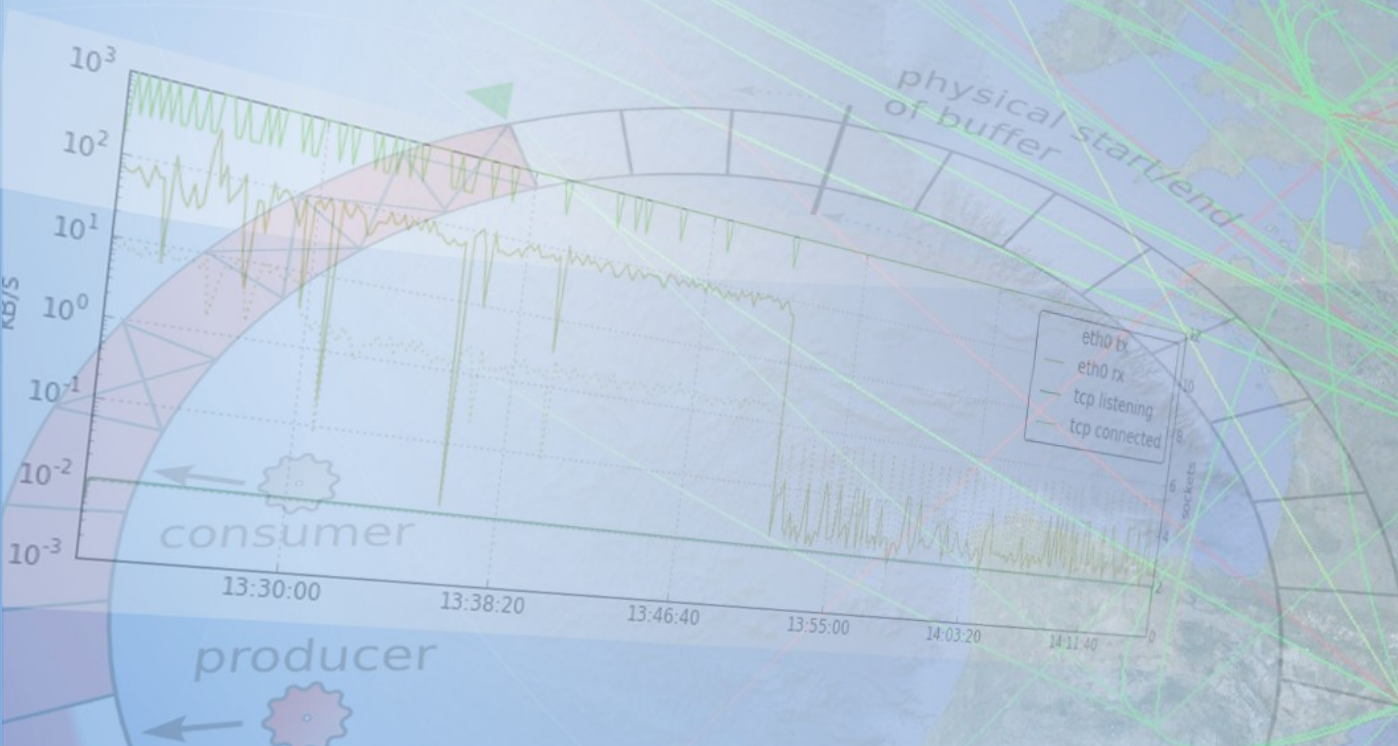


Application of Remote Debugging Techniques in User-Centric Job Monitoring



14th International Workshop on Advanced Computing and Analysis Techniques in Physics Research

Uxbridge, London, UK
September 5-9, 2011

Dr. Tim dos Santos
Bergische Universität Wuppertal

- Scope & Motivation
 - The WLCG and the PanDA Job Brokerage System
 - Grid Job Lifecycle and Failures
- Definition of terms
 - User-Centric Job Monitoring
 - Remote Debugging
- Realisation
 - Job Execution Monitor
 - Implemented Subset (...of Remote Debugging use cases)
 - Current Development

Scope & Motivation

The WLCG and PanDA

- WLCG: World-Wide LHC Computing Grid
 - 141 sites with currently ~270k cores, ~150 PB storage
 - Avg. ~1M jobs/day, average efficiency >95% (central production jobs) / ~80% (user analysis jobs)
- For the ATLAS Collaboration
 - Job brokerage is implemented in „pull“ semantic
 - → PanDA („Production and Distributed Analysis“)
 - Allows submittage of Grid Jobs in an easy-to-use fashion



Grid Job Lifecycle

- Pilot Factories mass-submit small pilot jobs
- Each pilot...
 - Ensures proper health of the worker node it is run on
 - Checks the WN's software environment and resources
 - Pulls a matching user job from a central database
 - Executes it
 - And creates exit summary logs after the job finished and wrote its output
- The Grid Job, in the context of ATLAS, usually performs a „stage-in → init → execute → stage-out“ cycle

Grid Job Failures


- Obvious payload errors (crashes) aside, a Grid job can fail for various reasons
 - Resource excess (wall time, memory, storage, ...)
 - Missing or invalid input files (also: version mismatch between chosen input files and processing software)
 - Missing or invalid contextual metadata files
 - Grid proxy certificate time-out
 - Service failure at the Grid site (typically: storage and transfer of input/output data)
- From the user's point of view, there are 3 classes:

Initialisation failure

Run-Time failure

Time-Outs

Grid Job Failure Classes

- Initialisation failures – „the Job didn't even start“
 - Grid middleware errors
 - Input Data missing / not available on the only sites that provide the needed software (version)
 - Erroneous Job definition
- Run-Time failures
 - Crashes ← in theory, those should all be found locally...
 - Miscalculations ←  this means **logical errors** in otherwise succeeding jobs! „Not detectable“...?
- Time-Outs
 - Wrong job queue chosen (wrong run time expectation)?
 - Site problems? I/O-problems?
 - Looping job? (← problem in user code / endless loops...)

Quick Definition of Terms

User-Centric Job Monitoring

- = Provision of job-related data describing the **progress, health** and **environment** of the job
 - Progress: „How far has my job proceeded?“ „does it even still do something?“ „why is this sub-job so much slower than the rest?“
 - Health: „Will my job make it?“ „Does it produce meaningful output (so far)?“ „Does it efficiently use its resources?“ – or: „Why exactly did it fail?“
 - Env.: „How much CPU/RAM/storage/bandwidth does it use?“ „How much storage/bandwidth/... is available?“ „What other processes run on „my“ WN?“
- Distinguished from Site Monitoring by scope
 - Can provide input for Site Monitoring, though

Remote Debugging

- Wikipedia says:

„Remote debugging is the process of debugging a program running on a system different than the debugger. [...] Once connected [over a network], debugger can **control the execution** of the program on the remote system and **retrieve information** about its state.“

- Main points: One needs to be able to...

- Control the execution.

- Granularity: Job, Module/Script, Function, Line
- **Interactively?** (breaking, stepping) – useful on the Grid...?
- **Semi-Interactively?** (break, get debugging data, continue)

- Retrieve Information.

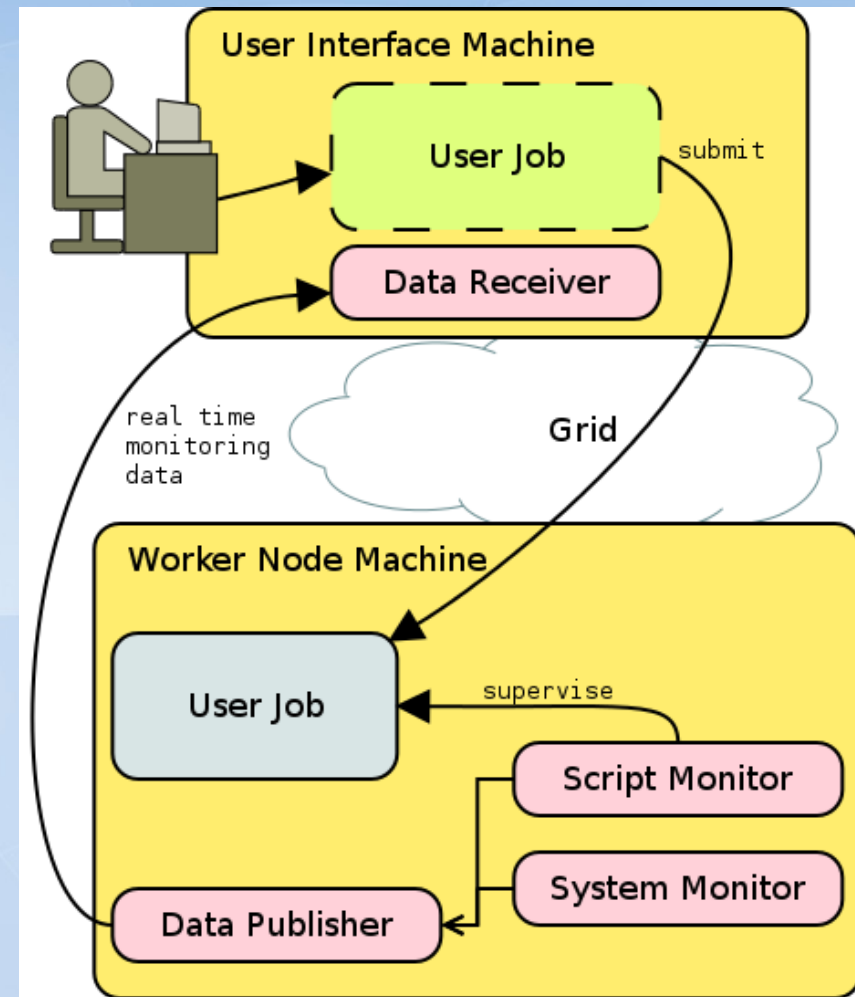
- Environment, Current execution state, (temporary) data, memory contents, ...

if retrieved automatically → **Non-Interactive** rem. debugging

Realisation of a Remote Debugger for the Grid

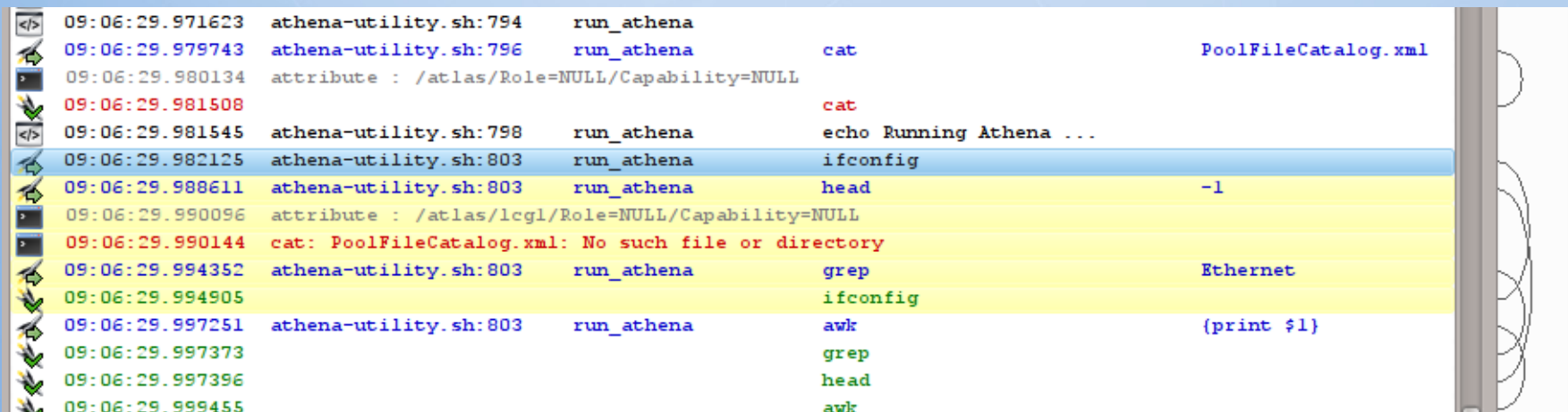
Job Execution Monitor

- Small & lightweight user space tool
- Submitted and run alongside the job on the worker node
- Gathers monitoring data, stores them in log files and can transmit them in real time
 - To the user, if he/she wants
 - To a central instance that generates statistics
- Can be activated by PanDA users with just one parameter
- Will be limited by a central instance (in development) to prevent misuse/excess monitoring/flooding



Implemented Remote Debugging

- The Job Execution Monitor today provides **non-interactive** remote debugging functionality
 - Bash Scripts and Python Code
 - Works out-of-the-box, enabled via submit-time configuration
 - At a moderate performance penalty in such scripts
 - Gives data about script execution progress and exceptions
 - Output streams (stdout/-err) and watched log files
 - Content can be streamed in real time, if desired
 - Time-stamped – allows correlation to rest of mon. data



```
09:06:29.971623 athena-utility.sh:794 run_athena
09:06:29.979743 athena-utility.sh:796 run_athena cat PoolFileCatalog.xml
09:06:29.980134 attribute : /atlas/Role=NULL/Capability=NULL
09:06:29.981508 cat
09:06:29.981545 athena-utility.sh:798 run_athena echo Running Athena ...
09:06:29.982125 athena-utility.sh:803 run_athena ifconfig
09:06:29.988611 athena-utility.sh:803 run_athena head -1
09:06:29.990096 attribute : /atlas/lcgl/Role=NULL/Capability=NULL
09:06:29.990144 cat: PoolFileCatalog.xml: No such file or directory
09:06:29.994352 athena-utility.sh:803 run_athena grep Ethernet
09:06:29.994905 ifconfig
09:06:29.997251 athena-utility.sh:803 run_athena awk {print $1}
09:06:29.997373 grep
09:06:29.997396 head
09:06:29.999455 awk
```

- The Job Execution Monitor today provides **non-interactive** remote debugging functionality
 - For C/C++/Fortran-based ELF binaries
 - Needs to be prepared by compiling the user code with debug symbols and hook-instrumentation (GNU gcc specific)
 - At moderate performance penalty when idle and high overhead when active (white-listed functions/methods, ...)
 - Gives data about execution progress and **memory contents** (variable values, object instances, function arguments)
 - Uses a trigger-system to e.g. create in-depth stack traces on user code failures
 - Is guarded against user code crashes (detect and report) and against internal errors (e.g. due to corrupt pointers/mem)

Example real time data

Timestamp	File	Frame	Called File / Exception	Called Frame / Exception Reason	Code / Command	Arguments / Return Value
10:51:13.887	GaudiHandle.h:194	void StatusCode::StatusCode				
10:51:13.888	ToolHandle.h:121	StatusCode GaudiHandle<IHelloT...				
10:51:13.898	HelloAlg.cxx:103	StatusCode ToolHandle<IHelloToo...			if (m_myPublicHelloTool.retrie...	
10:51:13.981	HelloAlg.cxx:103	StatusCode HelloAlg::initialize	StatusCode.h:131	bool StatusCode::isFailure	if (m_myPublicHelloTool.retrie...	{'this':('const const Status...
10:51:13.982	HelloAlg.cxx:103	bool StatusCode::isFailure			if (m_myPublicHelloTool.retrie...	
10:51:14.048	HelloAlg.cxx:107	StatusCode HelloAlg::initialize	GaudiHandle.h:38	const string (= basic_string<ch...	log << MSG::INFO << m_myP...	{'this':('const const GaudiH...
10:51:14.049	HelloAlg.cxx:107	const string (= basic_string<cha...			log << MSG::INFO << m_myP...	
10:51:14.134	HelloAlg.cxx:110	StatusCode HelloAlg::initialize	StatusCode.h:104	void StatusCode::StatusCode	return StatusCode::SUCCESS;	{'this':('const StatusCode*
10:51:14.135	HelloAlg.cxx:110	void StatusCode::StatusCode			return StatusCode::SUCCESS;	
10:51:14.181	???:-2	StatusCode HelloAlg::initialize				
10:51:16.333	???	???	HelloAlg.cxx:162	StatusCode HelloAlg::beginRun	StatusCode HelloAlg::beginRu...	{'this':('const HelloAlg*', 'Ox...
10:51:16.373	HelloAlg.cxx:167	StatusCode HelloAlg::beginRun	StatusCode.h:104	void StatusCode::StatusCode	return StatusCode::SUCCESS;	{'this':('const StatusCode*
10:51:16.373	HelloAlg.cxx:167	void StatusCode::StatusCode			return StatusCode::SUCCESS;	
10:51:16.374	???:-2	StatusCode HelloAlg::beginRun				
10:51:16.493	???	???	HelloAlg.cxx:115	StatusCode HelloAlg::execute	StatusCode HelloAlg::execute(...	{'this':('const HelloAlg*', 'Ox...
10:51:16.534	HelloAlg.cxx:146	StatusCode HelloAlg::execute	StatusCode.h:104	void StatusCode::StatusCode	return StatusCode::SUCCESS;	{'this':('const StatusCode*



Arguments

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

User Interface (prototype)

The screenshot displays the JLE IDE interface with the following components:

- Source Code:** A Python script named `setup.py` is shown. The current line of execution is `self.custom()` at line 00026. The code includes a `custom` method for customizing DBRelease files.
- System Metrics:** A line graph titled "System Load" showing the system load over time from 13:00:00 to 13:23:20. The left y-axis represents "load" (7.5 to 11.0), and the right y-axis represents a secondary metric (-0.06 to 0.06). A red line indicates the "system load", which peaks around 13:06:40 and then gradually declines. A vertical yellow highlight is present at approximately 13:09:00.
- Debug Log:** A table showing the execution stack and system events. The current frame is `__init__` in `setup.py:26`. The log includes timestamps, file names, frame identifiers, and the called file/exception/command.
- Local Variables and Arguments / Return Value(s):** Two empty tables are provided for inspecting the current state of the program.

Timestamp	File	Frame	Called File / Exception / Command
13:07:29.955269	_33488F5C-2C69-11E0-A22E-001B78E205A2_ .sh:2	<module>	env
13:07:29.959866	_33488F5C-2C69-11E0-A22E-001B78E205A2_ .sh:2	<module>	env
13:07:30.334431	_33488F5C-2C69-11E0-A22E-001B78E205A2_ .sh:2	<module>	cat
13:07:30.338925	_33488F5C-2C69-11E0-A22E-001B78E205A2_ .sh:2	<module>	cat
13:07:30.723340	_33488F5C-2C69-11E0-A22E-001B78E205A2_ .sh:2	<module>	python
13:07:37.122387			
13:07:40.546319	PythonMonitor.9791 launched		
13:07:40.549701	trf.e77a9ece-6eb0-48ac-9ff7-7a0f760262c7.py:3	<module>	setup.py:11
13:07:40.689899	setup.py:19	<module>	setup.py:19
13:07:40.690757			setup.py:66
13:07:40.691339	setup.py:70	<module>	setup.py:20
13:07:40.692122	setup.py:26	<module>	setup.py:29
13:07:47.122495			

X: 01.02.2011 12:58:57.952 Y: -0.017 not connected

Web-Based UI

Job Execution Monitor

detailed system metrics for job PanDA.1300623467

jobId	started	finished	last seen	cloudName	siteName	userName	exitcode	wall time [s]	#events
PanDA.1300623467	29. August 2011 16:25:54	29. August 2011 16:34:27	29. August 2011 16:34:27	CERN	CERN-PROD	Johannes Elmsheuser	0	513	None

Job CPU times

Legend: IO block time [ms] (red), CPU: system time [ms] (orange), CPU: user time [ms] (green)

Job memory usage

Legend: RSS (blue)

WN CPU usage (avg. over cores)

Legend: (red), (orange), (blue)

WN memory status

Legend: Memory (blue), Swap (red)

- One major topic is extension to **semi interactive** remote debugging
 - Obviously, single-stepping through instructions in a Grid job makes not too much sense
 - But: Being able to halt, inspect and then (optionally) continue a Job may be useful
 - This also encompasses download of intermediate output files during run time, the change of monitoring verbosity level on demand, etc.
 - Most important aspect in this project is **security**: authentication / authorization, data integrity, et al.

- Approach: Web-based UI and a messaging channel to the Job (encrypted MQ messaging)
 - For authorized users (Job author? Production role? Site admin?)
 - Providing semi interactive debug actions, for example:
 - „halt the Job, take data / inspect memory, continue the Job“
 - „kill the Job“
 - „add watch for variable X, value Y“
 - ...

- Further Work-in-progress Projects
 - Central controlling instance that can veto (or enforce) monitoring, deciding on a per-job level
 - Aggregated, projected and derived monitoring data (e.g. trend analysis hinting at imminent job failure, daily averaged job performance stats per site, ...)
 - Sophisticated automatic triggers on the Worker Node, providing real time functionality like for example:
 - Adaptive monitoring verbosity
 - Exception tracking (discard „expected exceptions“, caught ones, ...)
 - Reaction on user-defined job milestones / conditions
 - ... and more!

Thank you