Task Management in the New ATLAS Production System

CHEP 2013
October 14-18
Amsterdam

K De
D Golubkov
A Klimentov
M Potekhin
A Vaniachine

on behalf of the ATLAS Collaboration
The ATLAS Production System is the top level workflow manager which translates physicists' needs for production level processing and analysis into actual workflows executed by PanDA WMS across over a hundred Grid sites used globally by ATLAS.
Next Generation of ATLAS Production System

As the production workload increased in volume and complexity in recent years (the ATLAS production tasks count is above one million, with each task containing hundreds or thousands of jobs) there is a need to upgrade the Production System to meet the challenging requirements of the next LHC run while minimizing the operating costs. The rate of the ATLAS Production System task submissions grown exponentially. This created new requirements for scalability and management tools.
Production System and PanDA

- In the new design, the main subsystems are the Database Engine for Tasks (DEFT) and the Job Execution and Definition Interface (JEDI). Based on users' requests, DEFT manages inter-dependent groups of tasks (Meta-Tasks) and generates corresponding data processing workflows. The JEDI component then dynamically translates the task definitions from DEFT into actual workload jobs executed by the PanDA WMS:
PanDA Meta-Task as a way to manage complexity of the workflow

• In recent years, ATLAS production managers started to create workflows to be processed in PanDA. For example, the following highly simplified diagram illustrates a chain of PanDA tasks and the data that defines their relationship:
Motivations for system evolution

We need to address the following:

• The concept of Meta-Task as a group of logically related tasks, required to complete a specific physics project or sub-project. Absent in the original product which goes back many years (the original Production System), it emerged based on operational experience with PanDA and its workflow. It effectively became the central object in the high-level workflow management, but…

• …it is currently modeled in spreadsheets, which act as surrogate database and GUI, require active maintenance and limit the scalability of the overall Production System.

• Meta-Tasks must be properly modeled, introduced and integrated into the system to guarantee that it delivers adequate scalability and performance going forward, as well as a productive environment for the Production Managers and users involved in the physics analysis.

• Analysis tasks are currently handled outside the Production System, however they also have workflow characteristics. Having a universal system supporting the management of workflows across production and analysis will bring benefits to the ATLAS community.
Meta-Tasks: what do we want to see?

- **Automation**: we need the capability to process Meta-Tasks with minimal human intervention beyond task definition. Right now it is a labor intensive semi-manual process.

- **Automatic recovery** from simple failure modes

- But we also need the capability to have **operator intervention** and Meta-Task recovery: there must be adequate tools for the operators and managers to direct the Meta-Task processing

- **Dynamic** job definition (e.g. making it dynamic as opposed to static once the task is created) – functionality implemented in JEDI

- **Maintainability**: the code of the existing Production System was written "organically", in order to actively support emerging requests from the users, and it starts showing its age

- **Scalability**: in general, given the dramatic rise in the number of tasks defined and executed, we must ensure a lot of headroom going forward.

- **Ease of use**: there is currently a great amount of detail that the end user (Physics Coordination) must define in order to achieve a valid task request. We must automate and facilitate the task creation process, whereby cumbersome logic is handled within the application, and the user interface is more transparent and friendly.
Next Generation Production System = DEFT + JEDI

The next generation Production System (ProdSys2) is a tandem of two subsystems, which play complementary roles and represent two different levels in managing the overall workflow in PanDA:

- **DEFT: Database Engine For Tasks:**
  - DEFT is responsible for formulating the Meta-Tasks and the underlying tasks (collections of jobs). Meta-Tasks can include chains of tasks, bags of tasks and other types of task groupings, complete with all necessary parameters. DEFT keeps track of the state of the Meta-Tasks under its management, and their constituent tasks, by using a database as its persistence mechanism. It accounts for data dependencies among the tasks by maintaining records of datasets and their states. It provides the interface for each Meta-Task definition, management and monitoring throughout its lifecycle.

- **JEDI: Job Execution and Definition Interface**
  - JEDI is using the task definitions formulated in DEFT to define and submit individual jobs to PanDA, keep track of their progress and handle re-tries of failed jobs, job redirection etc. In addition, JEDI interfaces data management services in order to properly aggregate and account for data generated by individual jobs (i.e. general dataset management)
DEFT and JEDI working in tandem
DEFT and JEDI as an assembly line
DEFT: Meta-Task as a Graph Model

Graph model is commonly used to describe workflow:
• It allows implementation of the “chain”, “bag” and “bag of chains” topologies
• Liberates the designers of the workflow from limitations of the current “spreadsheet” model
• In the following, we will generally use the concept of tasks as a group of jobs using one or more datasets as input, and one or more datasets as output:

An example of a Hypothetical Complex Meta-Task
The Graph Model

Crucial features of the DEFT Meta-Task Model

- In accordance with operational practice in ATLAS, the dependencies between two adjacent tasks, which we model as nodes of a graph, are best conceptualized as datasets, which then become the edges of the graph.
- In order to handle “bag” and other complex topologies, we introduce “pseudo-tasks” representing the entry and exit points. It is actually an established technique in handling workflows.
The DEFT Equation

We model tasks as workflow elements acting on an array of input datasets, while producing a number of output datasets. The tasks have *states*, and undergo state transitions based on their inputs and applicable rules. Each task in a Meta-Task can be represented as a vector of variables describing its internal state and state of each input and output. The role of DEFT is to apply rules to transform this vector. Importantly, some of the parameters will be modified by JEDI.

\[
\begin{pmatrix}
y_1 \\
y_2 \\
y_3 \\
... \\
y_N \\
\end{pmatrix}
= D \otimes
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
... \\
x_N \\
\end{pmatrix}
\]
How do we represent and document the objects created according to the Graph Model, in human-readable format?

- The need to represent Meta-Tasks and their components in a way amenable to reading, editing and archival by humans was realized early on in the project.
- Do we need to build this format from scratch? Probably not.

**It’s the model!**

- Since we consider the Graph Model as the optimal way to represent the workflow in its various states, it is a reasonable approach to try and identify a natural way to represent the graph.
- This leads to realization that there are already standard languages and schemas that do exactly that. Some are XML-based, some are not.

<table>
<thead>
<tr>
<th>Language</th>
<th>Format/Language</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphML</td>
<td>XML</td>
<td>Supported for I/O by NetworkX, with some caveats</td>
</tr>
<tr>
<td>DGML</td>
<td>XML</td>
<td>Microsoft Product, used in Visual Studio</td>
</tr>
<tr>
<td>GXL</td>
<td>XML</td>
<td>Top-heavy. Somewhat complex syntax</td>
</tr>
<tr>
<td>GML</td>
<td>Custom</td>
<td>Supported for I/O by NetworkX. No clear advantage to GraphML, and would require a custom parser</td>
</tr>
<tr>
<td>DOT</td>
<td>Custom</td>
<td>Too graphics-oriented, unwieldy syntax</td>
</tr>
</tbody>
</table>
DEFT: example of a workflow template in GraphML

```xml
<key attr.name="comment" attr.type="string" for="edge" id="comment" />
<key attr.name="meta" attr.type="string" for="edge" id="meta" />
<key attr.name="state" attr.type="string" for="node" id="state" />
<key attr.name="tag" attr.type="string" for="node" id="tag" />
<key attr.name="comment" attr.type="string" for="node" id="comment" />
<key attr.name="meta" attr.type="string" for="node" id="meta" />

<key attr.name="last_update" attr.type="string" for="node" id="last_update"/>

<graph id="simulation template 1" edgedefault="directed">
  <node id="entry">
    <data key="state">disarmed</data>
    <data key="tag">entry</data>
    <data key="comment">Entry point/Null task</data>
    <data key="meta">meta-task id</data>
  </node>
  <node id="evgen">
    <data key="state">armed</data>
    <data key="tag">e1119</data>
    <data key="comment">Event Generation</data>
    <data key="meta">meta-task id</data>
  </node>

  <node id="Simulation 1">
    <data key="state">disarmed</data>
    <data key="tag">s3201</data>
    <data key="comment">Simulation 1</data>
    <data key="meta">meta-task id</data>
  </node>

  <node id="Simulation 2">
    <data key="state">disarmed</data>
    <data key="tag">s2_1111</data>
    <data key="comment">Simulation 2</data>
    <data key="meta">meta-task id</data>
  </node>
</graph>
```
The DEFT Prototype

- DEFT exists as a functioning, proof-of-integration prototype (CLI utility)
- Integration of NetworkX, GraphML, PyUtilib and DB Oracle schemas
- Capability to import and export workflows in GraphML format, as well as to persist data in RDBMS, and access and modify data transparently across these containers
The DEFT Prototype

- First implemented as a CLI utility, which allows the user to translate GraphML-formatted description of a Meta-Task into objects in RDBMS
- Capability to support workflows described by DAG of any complexity, not just “chains” and “bags”
- Implementation of rules for state transition of tasks
- Straightforward template capability, cloning and copying of tasks
- Essential functions available in both CLI and the Web UI application that is now being developed
• **JEDI** is a component of the PanDA server with *necessary logic to dynamically define jobs* based on task definitions coming from **DEFT**.

• Its principal function is dynamic creation of jobs to be processed by the PanDA WMS, based on task descriptions (which can be more abstract since they miss the details defined by **JEDI** during its analysis of the current allocations, policies and status of resources), which contain parameters for a potentially large number of jobs.

• **JEDI** is required to *optimally divide and direct the workload* according to a variety of factors, from resource allocations to specifics of the payload job configuration to the current status of remote computing resources managed by PanDA.

• In contrast to **DEFT**, where *the user interface and user control* are prominent elements of its functionality, **JEDI** operates automatically for the most part. Nevertheless, monitoring capabilities are essential in order to debug the system and execute proper operational support later on.

• **JEDI** has been rolled out end of summer 2013, and has effectively crossed into “beta” and pre-production by now.
Knowledge Management

How are jobs defined?

- In most cases, PanDA jobs can require extremely complex configuration
- This is impossible or at least impractical to set by hand
- The “magic” mentioned in the previous slide, whereby JEDI creates jobs from “hints” contained in the task description, is possible by providing **job templates** appropriate for each class of tasks. Each template encapsulates the knowledge pertaining to a particular class of tasks and can be managed in a database.
- The database to support this functionality is now being created.

At the task level, there is a similar problem with a large number of parameters required to define a task

- Solution: encapsulate the knowledge in templates, and expose fewer parameters to the lay user.
- DEFT has a human-readable format for task description that facilitates this
Modification of entries in the database, with certain elements being used as tokens, is the principal mechanism of *communication between DEFT and JEDI*. One simple use case can be described as follows:

- The operator uses the DEFT UI to fetch a workflow template (stored as a Meta-Task prototype in the database) and formulate a **Meta-Task**, which results in new entries in the table containing tasks.
- During a periodic database sweep, JEDI picks up **task definitions** and processes the tasks for submission. This includes **dynamic job definition** for each task.
- Tasks are executed and their status (as well as the status of the data being produced) is continuously updated by JEDI in the database. DEFT makes this information available via its monitoring system. It also controls the sequence of events by applying its set of rules to each state of the workflow (for example it may suspend processing automatically at a certain stage, pending QA by physicists).
- The operator may take action with regard to a Meta-Task, such as to **suspend the execution** of one of its elements pending further decision. The operator can also choose to cancel and unwind the task and its associated data, or maybe clone a task to provide a refined set of parameters needed in a more precise analysis.
**JEDI: Core Functionality**

- **JEDI** features *plug-in design* for its job definition engine, in order to be able to handle widely varying types of tasks. Examples include Group Production Tasks, ROOT-based analysis tasks, tasks with file-level splitting etc.

- An interesting feature of JEDI is generation of scout jobs for certain types of tasks, whereby a computing resource is being “probed” by running a limited number of jobs first. This allows for the **resource to be validated** and metrics to be collected and subsequently used in decision-making in JEDI.

- JEDI also includes logic for job “retry” and reassignment, in order to increase execution efficiency on the Grid.

- When used at scale, PanDA handles its data in units termed “*datasets*, which are collections of files”. In case a small number of files become corrupt or lost due to storage malfunction or some error condition in the data management system, the dataset can become essentially incomplete. JEDI contains provisions to “repair” such broken dataset by recreating the missing pieces of data, where possible.

- More sophisticated functionality is being implemented to enable fine-grained workflow, e.g. event-based processing, file-level splitting etc. The planned “*event server*” is an extreme example of fine-grained workflow management that may open venues for efficient utilization of opportunistic resources.
JEDI: progression of the task status in the rules engine
# DEFT: the WEB UI front-end prototype

## Meta-Task Monitor

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Template</th>
<th>Status</th>
<th>Created</th>
<th>Updated</th>
<th>Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>11312</td>
<td>mc13.8TeV.Pythia.Ztau332</td>
<td>187</td>
<td>Running</td>
<td>2013-09-03 18:07</td>
<td>2013-09-04 07:02</td>
<td></td>
</tr>
<tr>
<td>11313</td>
<td>mc13.8TeV.Pythia.Ztau334</td>
<td>187</td>
<td>Suspended</td>
<td>2013-09-03 18:07</td>
<td>2013-09-04 07:02</td>
<td></td>
</tr>
<tr>
<td>11314</td>
<td>mc13.8TeV.Pythia.Ztau335</td>
<td>187</td>
<td>Running</td>
<td>2013-09-03 18:17</td>
<td>2013-09-04 04:02</td>
<td></td>
</tr>
<tr>
<td>11315</td>
<td>mc13.8TeV.Pythia.Ztau336</td>
<td>187</td>
<td>Running</td>
<td>2013-09-03 18:07</td>
<td>2013-09-04 07:02</td>
<td></td>
</tr>
<tr>
<td>11317</td>
<td>mc13.8TeV.Pythia.Wmumu115</td>
<td>203</td>
<td>Running</td>
<td>2013-08-17 10:03</td>
<td>2013-08-20 01:42</td>
<td></td>
</tr>
<tr>
<td>11318</td>
<td>mc13.8TeV.Pythia.AlpgenJ101</td>
<td>081</td>
<td>Cancelled</td>
<td>2013-09-01 00:02</td>
<td>2013-09-02 01:22</td>
<td></td>
</tr>
</tbody>
</table>
Conclusions

• There are strong motivations for the upgrade of the ATLAS PanDA Production System
• Salient features of the new system include native support of complex workflows, efficient user interface and a very sophisticated engine for dynamic job definition aimed at optimizing resource utilization
• The job definition component (JEDI) has already been tested with realistic ATLAS workflows.
• Design of the Task Engine (DEFT) has been completed and its core functionality implemented in a prototype. Its WEB UI is currently under active development.
• Work is being done to finalize the product by creating the DEFT UI and integrating DEFT and JEDI.
Backup slides
Choice of Schema

• There is an obvious advantage in choosing the schema that’s is standardized, enjoys support and has parsers already written to handle its specifics.

• **GraphML** appears to be very simple, human-readable and enjoys parser support in many existing visualization and analysis software products.

• It allows us to standardize on the workflow description, visualization, editing, documentation and versioning with essentially zero effort.

• Capable graphic editors, such as GePhi, already exist and can be used immediately to create, visualize and edit Meta-Task templates (see backup slides).

NetworkX

• “**NetworkX** is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.”

• While its functionality is quite rich, and allows all sorts of graph analysis and exploration, the minimal subset of methods is quite easy to learn and use immediately.

• Reads GraphML, JSON, and other documents and creates an in-memory model of the graph automatically. Likewise, serializes graphs into a variety of formats, like GraphML, JSON etc.

• Visualization can be implemented by a few supported Python packages which need to be installed separately, such as matplotlib etc.

Persistence in RDBMS

• Persisting graphs in RDBMS had been addressed before; we revisited existing approached and chose the “Adjacency Table” approach as the most scalable and easy to implement. Two tables are created, one for nodes and another for edges. See Twiki for details: [https://twiki.cern.ch/twiki/bin/viewauth/Atlas/TaskModel#Graphs_in_RDBMS](https://twiki.cern.ch/twiki/bin/viewauth/Atlas/TaskModel#Graphs_in_RDBMS)
DEFT/JEDI Communication

• A few features of the DEfT/JEDI tandem design: in managing the workflow in DEFT, it’s unnecessary (and in fact undesirable) to implement subprocesses and/or multi-threaded execution such as one in products like PaPy etc. That is because DEFT is NOT a processing engine, or even a resource provisioning engine, as is the case in some Python-based workflow engines. It's a state machine that does not necessarily need to work in real time (or even near time). Near-time state transitions and resource provisioning is done in PanDA.

• There are a few ways to establish communication between Deft and Jedi, which are not mutually exclusive. For example, there may be callbacks, and database token updates, which may in fact co-exist. If a message is missed for some reason, the information can be picked up during a periodic sweep of the database. In summary, DEFT and JEDI will work asynchronously.

• Essentially, both components perform a database sweep.

• Since there are no in-memory processes keeping track of all Panda tasks and jobs at any given time, this provides us with better scalability going forward, compared to some other workflow management solutions.
Task visualization, editing and monitoring

• Basic visualization tools are already available, such as GePhi and matplotlib add-on to NetworkX (cumbersome installation though). Editing is available in GePhi complete with a GUI interface, and of course GraphML files can also be edited using any text editor.

• For more polished look and more dynamic and better user experience, we can develop a browser-based frontend utilizing jsPlumb, WireIt, Raphael etc – but we need to budget manpower for that, since the considerable power of these graphics systems comes with significant complexity of logic and API
Backup slides: examples of workflow visualization and editing in GePhi
Backup slides: examples of workflow visualization and editing in GePhi
Backup slides: examples of workflow visualization and editing in GePhi
Backup slides: examples of Javascript tools to aid in building Meta-Task GUI in DEFT