User guide to p3s Workflow Interface and Client

Author: M.Potekhin

Version: 1.01

Intro

The protoDUNE prompt processing system (p3s) consists of a central web service and a number of clients. At the time of writing, the "workflow client" in p3s exists in the form of a Python script appropriately named *workflow.py*. For brief summary of command line options run the script with "-h" option. The function of this client is to create, manage, adjust and delete workflows (and their templates) in p3s. The client does not provide much of monitoring functionality, which instead mainly exists in the Web interface of the p3s system.

In p3s — just like in many workflow management systems — the Directed Acyclic Graph (DAG) is an abstraction of a workflow. In the graph, vertices represent jobs and edges represent data, since it is data that defines logic of the workflow and relationship among the constituent jobs. Vertex and Edge objects have attributes necessary to describe properties and behaviors of jobs and data elements.

In p3s a **DAG template** describes the topology and general characteristics of edges and vertices of a class of workflows but does not correspond to running processes. It is not expected it have enough information that would be necessary to immediately create a functional workflow (although this is possible). Instead, workflows are created by *adding enough parameters* to DAG templates such that that a proper execution context can be defined, such as

- the environment
- paths to individual executables in the workflow
- location and file names of the data involved

In summary, workflows are created based on DAGs serving as templates (abstractions) by adding necessary parameters. After that, execution of the resulting workflow may be triggered. Same template may be used any number of times to generate any number of workflows, and the user has the freedom to override some parameters of the template in the workflow derived from it. This is similar to class inheritance.

Describing DAGs

DAGs and Workflows are stored on the **p3s server** and are persisted using the backend database by maintaining tables which contain lists of vertices and edges

of the respective graphs. The name of the graph (expected to be unique) is used as a key for accessing the vertices and edges data.

Externally. there are many ways to describe a DAG in a human-readable way convenient for the user and suitable for creating a corresponding record in the p3s database. A reasonable requirement would be to use some sort of text format for describing a DAG so it can be readily inspected edited. It is convenient to leverage an existing XML schema for desribing graphs, *GraphML* — which is supported in a number of software packages and editing and visualization tools.

A GraphML description of a DAG us sent to the server using the **workflow.py** client. Option "-g" of the client instructs it to read a graphML file and send its contents to the server. Optionally, a name for the DAG can be suppplied via a command line argument, and if absent the default is derived from the name of the file containing the graph. Keep in mind that DAG names are unique in p3s Example of creating a DAG from a GraphML file:

```
workflow.py -g myDAG.graphml
```

According to the comment above, in this case the name of the resulting DAG registered in p3s will be the root name of the file e.g. *myDAG*. If you want to create a DAG named "foo" you can simply add "`-n foo`" to the command. For a few examples of GraphML files written for p3s please see the directory *p3s/inputs*. Sorry - the format is not pretty but we didn't design it.

Creating Workflows

A workflow is created based on a pre-existing DAG, stored on the p3s server as a result of running the client with "-g" option as described above. Any number of workflows can be generated based on the same template. Assuming that "myDAG" is one of such registered DAGs, a workflow is created by running the command below. The "`-n`" option allows the user to specify the name of the workflow being created, which otherwise will default to the name of the DAG.

```
workflow.py -a myDAG -n myWorkFlow
```

Workflows are identified in the system by their UUIDs which are automatically generated. So in fact the workflow names are not expected or required to be unique. This allows to easily identify workflows of the same kind in the p3s monitoring Web interface (cf. *"PhotonDetectorWF"* etc). By default, workflows are created the **template** state i.e. not marked as ready for execution. This allows to start a workflow at a latter time by setting its state to **defined**. If the user wants to create worflow in the "defined" state right away (thus prompting p3s to start the execution) this can be achieved by adding the "-s" option, i.e.

```
workflow.py -a myDAG -n myWorkflow -s defined
```

As mentioned above, DAGs are expected to be templates not necessarily containing all of the information which is valid and suitable for execution, for example

the filenames contained in the edges of teh graph can be bogus. Likewise, the executable name contained in some node may not be correct and be just a place holder. Methods of adding the necessary information to DAG in order to convert it to a workflow are explained in the following section.

Object Deletion

NB. Do not attempt to run these commands when system is in production, this is an operation reserved for experts

Deletion of objects in the p3s database is accomplished by running the same workflow client. The "-d" option accompanied by the object key (e.g. UUID for workflows) will cause the client to delete the corresponding object from the database. For both DAGs and workflows, this will cause the deletion not only of the object itself as a record, but also of its vertices and edges. This is achieved by running the client in this manner:

```
workflow.py -d -w workflow -u 1234567890
```

where "-d" stands for delete, "-w" stands for "what" since both DAGs and workflows can be deleted via this interface, and "-u" stands for UUID of the object to be deleted. If for example one wants to delete a DAG which is named "myDAG" from the system, the necessary command would look like this:

```
workflow.py -d -w dag -n myDAG
```

It is easy to see that DAGs are referred to by their names as opposed to UUIDs (which they don't have).

From DAG to Workflow

Workflow filename policy

In most practical cases, the DAG template will contain dummy names for the files referenced in it. When a workflow is created this is expected to be fixed by supplying necessary information. For that reason, an important aspect of the workflow creation is the filename policy i.e. how the actual files are named. By default, p3s will automatically generate names for all edges in the workflow based on UUID and the extension registered in the "Datatype" table in the database. For example, a use may define type "TXT" (can be any name) which implies the file extension ".txt". The system will know to generate filenames with this extension wherever it enounters files tagged with "TXT".

Extra information, overriding or filling placeholders in the DAG can be provided by utilizing the "-f" option in the client. This option accepts two types of input, non-JSON and JSON.

Non-JSON

Values to be used with this option (-f):

- "sticky", in which case a workflow inherits the file names from its parent DAG
- "inherit:name", in which case filenames will be automatically generated based on the supplied name and DAG topology

JSON

Values to be used with this option (-f):

- a JSON-formatted string which can specify the filenames for any of the DAG's edges if desired
- a name of a JSON file containing same information (must contain ".json" to be properly identified)

The example given above can therefore be extended to something like:

```
workflow.py -a myDAG -n myWorkflow -s defined -f myFileInfo.json
```

Job Information

It works similar to adding file information to a workflow as explained above. Information supplied in JSON format (either on the command line or in a file supplied with -j option and having json extension will be included in the job object on the server side.

An extended example which includes both file and job information to round out a workflow based on a DAG named "source1":

```
./workflow.py -v 2 -a source1 -n sourceTest -s defined -f ../inputs/fileinfo1.json
-j '{"filter":{"payload":"env"}}'
```

In this example, file names will be created according to the extra info contained in "fileinfo1.json" while the job information, in particular payload, is supplined in-line (it could have been fed to the script as a JSON file as well). Purely for demonstration purposes, the payload of the DAG node names "filter" has been defined as the standard shell command "env".

A working example

Assuming you are working at CERN and are in the "clients" directory of the p3s repo the following commands will produce a simple but working example of a workflow with an origin, three payloads in the middle and one sync job on the output.

The first command defined the template for p3s while the second instantiates a workflow based on the template with a few modifications

```
./workflow.py -g ../inputs/3filters/3filters.graphml
./workflow.py -a 3filters -f ../inputs/3filters/fileinfo_3filters_mod.json -s defined
```

Appendix

Editing GraphML files

In addition to text editors that can be used for editing these files, there are a number of tools for exploring, editing and visualizing graphs, such as *GePhi* and a few others. For example, it is possible to import and export graph info from/to a spreadsheet or a comma-separated file, and edit elements of a graph in a GUI. Using a simple test editor is still perhaps the most efficient way to do it, although the particular XML schema of GraphML is not very pretty and takes some getting used to.