User guide to p3s Workflow Interface and Client

Intro

The protoDUNE prompt processing system (p3s) consists of a central web service (named *promptproc*) 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 the client is to create, manage, adjust and delete workflows 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 describes the topology and general characteristics of edges and vertices of a class of workflows but does not correspond to a running process or processes nor does it have enough information that would be necessary to create a functional workflow. In fact, workflows are created by adding enough parameters to DAGs such that that a proper execution context can be defined (i.e. the environment, paths to executable, location and name of files etc). Workflows therefore are created based on DAGs serving as templates (abstractions) by adding necessary parameters.

Describing DAGs

DAGs and Workflows are stored on the central server of p3s 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 is used as a key.

There are many ways to describe a DAG in a way convenient for the user and suitable for creating a record in p3s. A reasonable requirements is using some sort of text format for describing a DAG so it can be readily edited by prectically any text editor. 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. Internally, p3s is using the Python package *NetworkX* to parse the *GraphML* source and perform a few other basic operations on graphs if needed. The graph information is stored in a database nevertheless.

Option "-g" of the workflow client allows to use a graphML file containing a DAG description and send it to the server. A name can be suppplied via a command line argument, and if absent it is derived from the name of the file containing the graph. DAG names are unique in p3s, and that property is enforced in the database by making the name attribute the primary key. Example of creating a DAG:

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

For a few examples of GraphML files written for p3s please see the directory p3s/inputs. 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.

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. Assuming that "myDAG" is one of such registered DAGs, a workflow is created by

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

The "-n" option allows to specify the name of the workflow, which otherwise (i.e. if not specified) will default to the name of the DAG.

Defining Workflows

Workflows are created based on templates (DAGs) which must be exist on the server by the time a request for a new workflow is sent. A name can be optionally set for a workflow but it's not expected to be unique. Wokrflows are identified in the system by their UUIDs which are automatically generated.

Example of adding a workflow based on a DAG:

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

The above command will create a workflow which is in the "template" state i.e. not marked as ready for execution. If the user wants to create worflow in the "defined" state this can be achieved by adding the "-s" option, i.e.

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

Workflow filename policy

The next important aspect of the workflow creation is the filename policy. 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 one of the three types of input:

- a stirng not formatted in JSON and taking values such as:
- "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
- 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".

Object Deletion

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).

Until serious testing has been completed, please consult the experts about using the delete function - especially if the system is in production.