Introduction

Preparing to run

There is a separate "overview" document which contains a general description of how p3s works and what its components are. For the end user a lot of this detail won't matter. The user is interested for the most part in just running a number of jobs and following their progress, consulting the log files if necessary.

To achive this, the following initial preparations must be done, and it takes just two commands:

- If not already done so, install p3s software simply by cloning the content from GitHub

  ```
  git clone https://github.com/DUNE/p3s.git
  ```

  After you run this, you will end up with a directory "p3s" which will contain a number of subdirectories. The one which will interest you now is p3s/clients.

- This step is CERN-specific. Activate the "Python virutal environment" by running a command as explained below

  ```
  source ~mxp/public/vp3s/bin/activate
  ```

  Verify that the Python is the right version after this step by running "python -V". It shold show 3.5. In addition, it's useful (but not necessary) to try to import "networkx" and "django" just to make sure everthing is indeed configured.

Running a job

- Describe a job. Job description in p3s is done using a fairly simple JSON format (more on that below). It contains a reference to an executable and the environment in which to run.
- Use a dedicated client ("job.py") to submit this job description to the server which will then orchestrate its execution
- Monitor the progress of jobs using a P3S Web page
- Browse and use the output files produced by jobs

In the following we assume that the CERN instance of P3S is used, which entails certain conventions and conviences such as sharing files and scripts via AFS and EOS.

P3S Clients

job.py

This client can be used for the following:

- send a job description (or a number of job descriptions) to the P3S server. This can be done by reading a description of job(s) contained in a JSON file.
- if necessary, adjust job attributes
- delete a job from the server

The following example (with arbitrary file names and variables) demonstrates how JSON is used to describe jobs. Let us assume that we created a file named "myjob.json" with the following contents:

```
[
    {
        "name":         "p3s_test",
        "timeout":      "100",
        "jobtype":      "print_date",
        "payload":      "/home/p3s/my_executable.sh",
        "env":          {"P3S_MODE":"COPY","MYFILE":"/tmp/myfile.txt"},
        "priority":     "1",
        "state":        "defined"
    }
]
```

Note that this format corresponds to a *list* of objects i.e. such file can easily contain a number of jobs; however having just one element in this list is absolutely fine.

Most important attributes are: * the payload, since this is the script that will run. It is recommended that this is a shell wrapper. * the "env" attribute which is the job environment. In particular, it can be used to communicate to the job the names of input and output files, and this is typically done in the wrapper script itself.

For example, let's create a simple test job, and for that we'll need the payload script - which can be named anything but to correlate with the example above let's call it "my_executable.sh".

```
#!/bin/bash
# This script is "my_executable.sh" in the JSON example above
date > $MYFILE
```

It is important that the path /home/p3s/my_executable.sh is readable and executable for other users, otherwise the system won't be able to run it.

Now we can submit this job to the server. Assuming the p3s client software is installed, and we changed to the "clients" directory, the following command can be used

```
./job.py -j ./myjob.json
```

And that's it.

pilot.py

Generates a pilot and performs a small number of other functions. The lifecycle of a pilot is as follows: * The process starts and then attempts to contact the server in order to register. At the time of writing handling timeouts is not imeplemented yet, i.e. the pilot will exit in fail condition if there is not response from the server. This behavior will be improved in future development.

- Upon successful registration, the pilot enters a loop. In each iteration it send a job request to the server. If the request is not granted, the pilot sleeps for a configurable period of time and make another attempts. Having exhausted the maximal number of attempts (also configurable), the pilot exits normally.
- If a matching job is identified by the server, the will receive a message containing the description of the payload (e.g. the path to the script/wrapper/executable that needs to be run). The pilot then initiates job execution using the Python "subprocess" machinery.

workflow.py

Creates and manipulates DAGs and workflows. DAGs serve as templates for actual active workflows.

P3S Interfaces

Server interface

- serverAPI.py: translates calls to methods in scripts to HTTP messages sent to the server utilizing *urllib*. It is used by most all clients.

Please see individual documentation for the clients listed above which is contained in files named like WORKFLOW.md etc. Same information converted into PDF (so it's accessible locally and can be printed) can be found in **p3s/documentation**.

There are additional clients which perform tests or a combination of procedures on these objects, such as * injector.py - scans a directory for new files and created workflow with the new data as input * urltest.py (deprecated) - generic HTTP interface to the server * verifyImport.py - verifies that dependecies are satisfied, i.e. certain packages can be imported

Service Scripts

For the most part these scripts are meant to be used on multiple machines e.g. by utilizing pdsh * pgen.sh serves as a basic generator of pilots, in case multiple pilots need to run on same node. The number of concurrent pilots is configurable on the command line.

- preport.sh reports on the number of pilots running on WNs
- pkill.sh remotely kills pilots