htcondenser Documentation

Release 0.2.0

Robin Aggleton

October 27, 2016

Contents

htcondenser is a simple library for submitting simple jobs & DAGs on the Bristol machines.

It was designed to allow easy setting up of jobs and deployment on worker nodes, without the user worrying too much about writing custom scripts, or copying across files to HDFS.

Contents:

CHAPTER 1

htcondenser

1.1 What is it?

htcondenser is a simple library for submitting simple jobs & DAGs on the Bristol machines.

It was designed to allow easy setting up of jobs and deployment on worker nodes, whilst following the Code of Conduct without the user worrying too much about writing custom scripts, or copying across files to HDFS.

Note that this probably won't work for more custom or complicated workflows, but may be a useful starting point.

1.2 What do I need?

An area on `/hdfs/users` that you have read/write permission. Python >= 2.6 (default on soolin), but untested with Python 3.

For developers: To build the docs, you'll need sphinx (`pip install sphinx`). `flake8` and `pep8` are also useful tools, and are available via `pip` or `conda`.

1.3 How do I get/install it?

It depends.

- You can install using `pip` for a given python install instance:

  pip install –user git+https://github.com/raggleton/htcondenser.git@setup

NOTE this **will not work properly** with CMSSW, etc since they use a different `python` and hence look for packages in different places. **TODO: fix this**.

In a similar vein, `conda` users will need to install this for each conda environment.

To do this use the "global" install:

- Manual (global) install (for use with CMSSW, etc): clone this repository, then run `./setup.sh`. This will be required every time you login, so you may want to add it to your `.bashrc`.

1.4 How do I get started?

Look in the `examples` directory. There are several directories, each designed to show off some features:

- simple_job/simple_job.py: Submits 3 jobs, each running a simple shell script, but with different arguments. Designed to show off how to use the `htcondenser` classes.
- simple_exe_job/simple_exe_job.py: Submits a job using a user-compiled exe, `showsize`. Before submission, you must compile the exe: `gcc showsize.c -o showsize`. Test it runs ok by doing: `./showsize`.
- simple_root6_job/simple_root6job.py: Run ROOT6 over a macro to produce a PDF and TFile with a TTree. (**TO FIX:** Requires existing ROOT setup)
- simple_cmssw_job/simple_cmssw_job.py: Setup a CMSSW environment and run `edmDumpEventContent` inside it. For a CRAB-alternative, see cmsRunCondor
- dag_example/dag_example.py: Run a DAG (directed-acyclic-graph) - this allows you to schedule jobs that rely on other jobs to run first.
- dag_example_common/dag_example_common.py has a similar setup but shows the use of `common_input_files` arg to save time/space.

For more info/background, see Usage.

1.5 Monitoring jobs/DAGs

If you submit your jobs as a DAG, then there is a simple monitoring tools, `DAGstatus`. See DAGstatus for more details.

1.6 A bit more detail

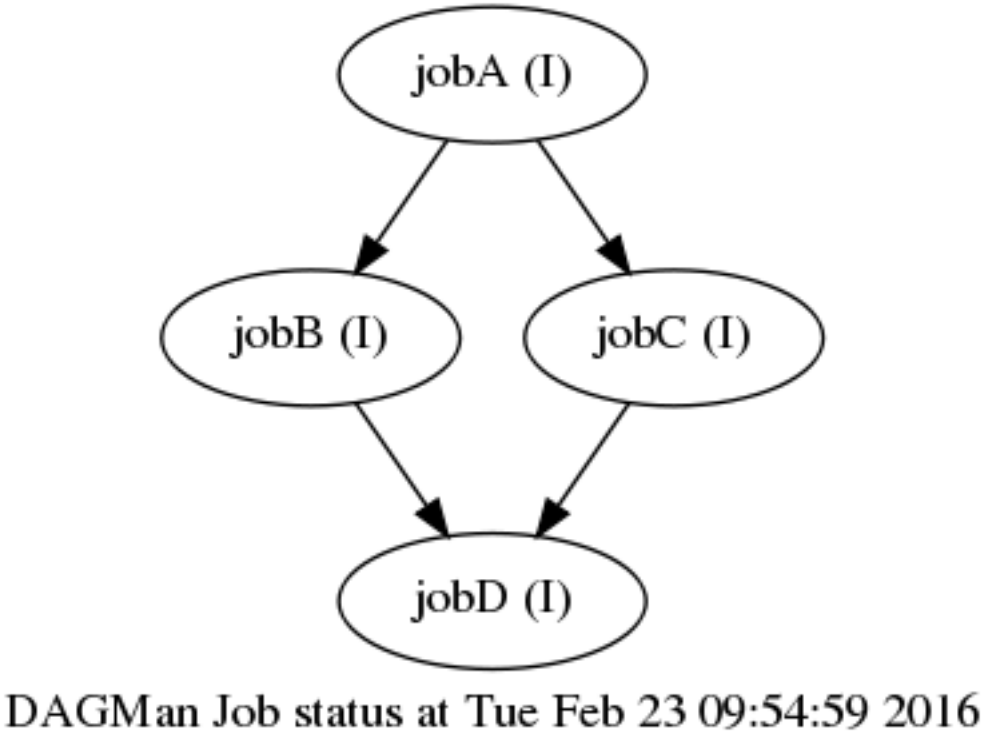
The aim of this library is to make submitting jobs to HTCondor a breeze. In particular, it is designed to make the setting up of libraries & programs, as well as transport of any input/output files, as simple as possible, whilst respecting conventions about files on HDFS, etc.

Each job is represented by a `Job` object. A group of `Jobs` is governed by a `JobSet` object. All `Jobs` in the group share common settings: they run the same executable, same setup commands, output to same log directory, and require the same resources. 1 `JobSet` = 1 HTCondor job description file. Individual `Jobs` within a `JobSet` can have different arguments, and different input/output files.

For DAGs an additional `DAGMan` class is utilised. Jobs must also be added to the `DAGMan` object, with optional arguments to specify which jobs must run as a prerequisite. This still retains the `Job`/`JobSet` structure as before for simpler jobs, to simplify the sharing of common parameters and to reduce the number of HTCondor submit files.

Aside: DAGs (Directed Acyclic Graphs)

Essentially, a way of tying jobs together, with the requirement that some jobs can only run once their predecessors have run successfully.
Graph: collection of nodes joined together by edges. Nodes represent jobs, and edges represent hierarchy. (Note, not the `y = sin(x)` type of graph.)
Directed: edges between nodes have a *direction*. e.g. `A ->- B` means A precedes B, so B will only run once A has finished successfully.
Acyclic: the graph cannot have cycles, e.g. `A ->- B ->- C ->- A`.
For an example see the diamond DAG (`examples/dag_example`):



There, jobB and jobC can only run once jobA has completed. Similarly, jobD can only run once jobB and jobC have completed.

1.7 Full documentation

See htcondenser on readthedocs.

1.8 Common pitfalls

- `ERROR: proxy has expired`: you need to renew your Grid certificate: `voms-proxy-init -voms cms`.
- DAG submits, but then immediately disappears from running *condor_q -dag*: check your *.dagman.out* file. At the end, you will see something like:

```
    Warning: failed to get attribute DAGNodeName
    ERROR: log file /users/ab12345/htcondenser/examples/dag_example_common/./diamond.dag.nodes.log i
    Error: log file /users/ab12345/htcondenser/examples/dag_example_common/./diamond.dag.nodes.log c
    **** condor_scheduniv_exec.578172.0 (condor_DAGMAN) pid 601659 EXITING WITH STATUS 1
```

This is telling you that you cannot put the DAG file (and therefore its log/output files) on a Network File Storage (NFS) due to the number of frequent writes. Instead put it on `/storage` or `/scratch`.

1.9 But I want XYZ!

Log an Issue, make a Pull Request, or email me directly.

1.10 I want to help

Take a look at CONTRIBUTING.

CHAPTER 2

Usage

Here we explain a bit more about the basic **htcondenser** classes.

Full details on the API can be found in htcondenser package

For all snippets below, I've used:

```
import htcondenser as ht
```

2.1 Some basic rules/principles

These go along with the code of conduct and help your jobs run smoothly.

- The worker node is restricted to what it can read/write to:
  - **Read-only**: `/software`, `/users`
  - **Read + Write**: `/hdfs`
- However `/software` and `/users` are all accessed over the network.

Danger: Reading from `/users` with multiple jobs running concurrently is guaranteed to lock up the whole network, including soolin.

- Therefore, it is best to only use `/hdfs` for reading & writing to/from worker nodes.
- Similarly, `JobSet.filename`, `.out_dir`, `.err_dir`, `.log_dir`, and `DAGMan.filename` and `.status` should be specified on `/storage` or similar - **not** `/users`.
- hadoop commands should be used with `/hdfs` - use of `cp`, `rm`, etc can lead to lockup with many or large files.

2.2 Basic non-DAG jobs

There are only 2 basic classes needed: `JobSet` and `Job`.

`Job` represents a single job to be run - the execution of some program or script, with arguments, inputs and outputs.

`JobSet` defines a group of `Job` s that share common properties (e.g. executable), so that they can all share a common condor submit file.

By specifying various options, these classes are designed to handle:

- The transferring of any necessary files (including executable) to `/hdfs`.

- Writing the necessary condor job files.
- Setting up directories for logs, etc.

On the worker node, a wrapper script is run. This handles the transfer of any files before and after execution, and can run a setup script prior to the main executable.

Typically one defines a `JobSet` instance for each different executable to be run:

```
job_set = ht.JobSet(exe='simple_worker_script.sh',
                    copy_exe=True,
                    setup_script=None,
                    filename='/storage/user1234/simple_job.condor',
                    out_dir='/storage/user1234/logs', out_file='$(cluster).$(process).out',
                    ...
                    cpus=1, memory='50MB', disk='1',
                    hdfs_store='/hdfs/user/user1234')
```

Then one defines the relevant `Job` instances with job-specific arguments and files:

```
job = ht.Job(name='job1',
             args=['simple_text.txt', ..., word],
             input_files=['simple_text.txt'],
             output_files=['simple_results_1.txt'],
             quantity=1)

job = ht.Job(name='job2',
             args=['simple_text.txt', ..., other_word],
             input_files=['simple_text.txt'],
             output_files=['simple_results_2.txt'],
             quantity=1)
```

Note that the files specified by `input_files` will automatically get transferred to HDFS before the job starts. This avoids reading directly from `/users`. Files specified by `output_files` will automatically be transferred to HDFS from the worker node when the job ends. Note that any arguments you pass to the job will automatically be updated to reflect any transfers to/from `/hdfs`: *you do not need to worry about this*.

Each `Job` must then be added to the governing `JobSet`:

```
job_set.add_job(job)
```

Finally, one submits the `JobSet`:

```
job_set.submit()
```

The `JobSet` object has several constructor arguments of interest:

- One **must** specify the script/executable to be run, including its path if it's a non-builtin command: `./myProg.exe` not `myProg.exe`, but `grep` is ok.
- The `copy_exe` option is used to distinguish between builtin commands which can be accessed without transferring the executable (e.g. `grep`) and local executables which do require transferring (e.g. `myProg.exe`).
- A setup script can also be defined, which will be executed before `JobSet.exe`. This is useful for setting up the environment, e.g. CMSSW, or conda.
- There are also options for the STDOUT/STDERR/condor log files. These should be put on `/storage`.
- The `hdfs_store` argument specifies where on `/hdfs` any input/output files are placed.
- The `transfer_hdfs_input` option controls whether input files on HDFS are copied to the worker node, or read directly from HDFS.

- `common_input_files` allows the user to specify files that should be transferred to the worker node for every job. This is useful for e.g. python module depedence.

The `Job` object only has a few arguments, since the majority of configuration is done by the governing `JobSet`:

- `name` is a unique specifier for the Job
- `args` allows the user to specify argument unique to this job
- `hdfs_mirror_dir` specifies the location on `/hdfs` to store input & output files, as well as the job executable & setup script if `JobSet.share_exe_setup = False`. The default for this is the governing `JobSet.hdfs_store/Job.name`
- `input_files/output_files` allows the user to specify any input files for this job. The output files specified will automatically be transferred to `hdfs_mirror_dir` after the exe has finished.

2.3 Input and output file arguments

The `input_files`/`output_files` args work in the following manner.

For `input_files`:

- `myfile.txt`: the file is assumed to reside in the current directory. It will be copied to `Job.hdfs_mirror_dir`. On the worker node, it will be copied to the worker.
- `results/myfile.txt`: similar to the previous case, however **the directory structure will be removed**, and thus `myfile.txt` will end up in `Job.hdfs_mirror_dir`. On the worker node, it will be copied to the worker.
- `/storage/results/myfile.txt`: same as for `results/myfile.txt`
- `/hdfs/results/myfile.txt`: since this file already exists on `/hdfs` it will not be copied. If `JobSet.transfer_hdfs_input` is `True` it will be copied to the worker and accessed from there, otherwise will be accessed directly from `/hdfs`.

For `output_files`:

- `myfile.txt`: assumes that the file will be produced in `$PWD`. This will be copied to `Job.hdfs_mirror_dir` after `JobSet.exe` has finished.
- `results/myfile.txt`: assumes that the file will be produced as `$PWD/results/myfile.txt`. The file will be copied to `Job.hdfs_mirror_dir` after `JobSet.exe` has finished, but **the directory structure will be removed**.
- `/storage/results/myfile.txt`: same as for `results/myfile.txt`. Note that jobs cannot write to anywhere but `/hdfs`.
- `/hdfs/results/myfile.txt`: this assumes a file `myfile.txt` will be produced by the exe. It will then be copied to `/hdfs/results/myfile.txt`. This allows for a custom output location.

Rational: this behaviour may seem confusing. However, it tries to account for multiple scenarios and best practices:

- Jobs on the worker node should ideally read from `/hdfs`. `/storage` and `/software` are both readable-only by jobs. However, to avoid any potential network lock-up, I figured it was best to put it all on `/hdfs`
- This has the nice side-effect of creating a 'snapshot' of the code used for the job, incase you ever need to refer to it.
- If a file `/storage/A/B.txt` wanted to be used, how would one determine where to put it on `/hdfs`?
- The one downfall is that output files and input files end up in the same directory on `/hdfs`, which may note be desirable.

Note that I am happy to discuss or change this behaviour - please log an issue: github issues

2.4 DAG jobs

Setting up DAG jobs is only slightly more complicated. We still use the same structure of `Job` s within a `JobSet`. However, we now introduce the `DAGMan` class (DAG Manager), which holds information about all the jobs, and crucially any inter-job dependence. The class is constructed with arguments for DAG file, and optionally for status file (very useful for keeping track of lots of jobs):

```
LOG_STORE = "/storage/%s/dag_example/logs" % os.environ['LOGNAME']
dag_man = ht.DAGMan(filename=os.path.join(LOG_STORE, 'diamond.dag'),
                    status_file=os.path.join(LOG_STORE, 'diamond.status'),
```

Note that like for `JobSet` s, it is best to put the file on `/storage` and not `/users`.

You can then create `Job` and `JobSet` s as normal:

```
job_set1 = ht.JobSet(exe='script1.sh', ...
jobA = ht.Job(name='jobA', args='A')
jobB = ht.Job(name='jobB', args='B')
```

One then simply has to add `Job` s to the `DAGMan` instance, specifying any requisite `Job` s which must be completed first:

```
dag_man.add_job(jobA)
dag_man.add_job(jobB, requires=[jobA])
```

Finally, instead of calling `JobSet.submit()`, we instead call `DAGMan.submit()` to submit all jobs:

```
dag_man.submit()
```

If `DAGMan.status_file` was defined, then one can uses the `DAGStatus` script to provide a user-friendly status summary table. See DAGstatus.

2.5 Logging

The **htcondenser** library utilises the python `logging` library. If the user wishes to enable logging messages, one simply has to add into their script:

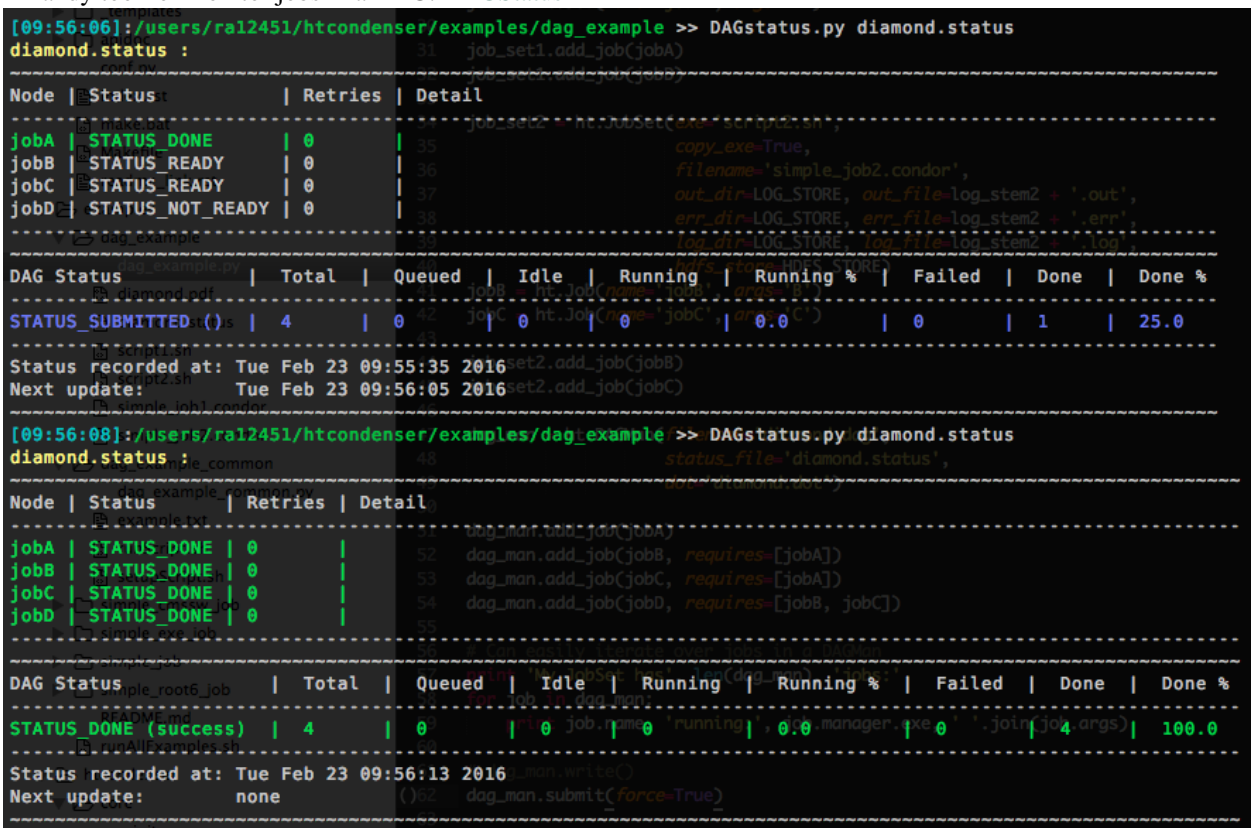
```
import logging

log = logging.getLogger(__name__)
```

where `__name__` resolves to e.g. `htcondenser.core.Job`. The user can then configure the level of messages produced, and various other options. At `logging.INFO` level, this typically produces info about files being transferred, and job files written. See the full logging library documentation for more details.

CHAPTER 3

DAGstatus

A handy tool for monitor jobs in a DAG: DAGStatus



3.1 Usage

Ensure that the `DAGMan.status_filename` attribute is set. Then pass that filename to `DAGStatus` to view the current DAG status. Use the `DAGMan.status_update_period` attribute to control how often the status file is updated.

If you are not using the `htcondenser` library then ensure you have the following line in your DAG description file:

```
NODE_STATUS_FILE <filename> <refresh interval in seconds>
```

See 2.10.12 Capturing the Status of Nodes in a File for more details.

General usage instructions::

```
usage: DAGStatus [-h] [-v] [-s] [statusFile [statusFile ...]]

Code to present the DAGman status output in a more user-friendly manner. Add
this directory to PATH to run DAGStatus it from anywhere.

positional arguments:
  statusFile     name(s) of DAG status file(s), separated by spaces

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  enable debugging mesages
  -s, --summary  only printout very short summary of all jobs
```

3.2 Customisation

It is possible to customise the coloured output to suit your personal preferences. This is done in DAGstatus_config.json. The user must define any colours or styles used in the `colors` object. These can then be used in the `statuses` and `formatting` objects. Any combination of colours/styles can be used, by concatenating with a `+`.

Note that concatenating two colours will only use the rightmost colour.

CHAPTER 4

FAQ

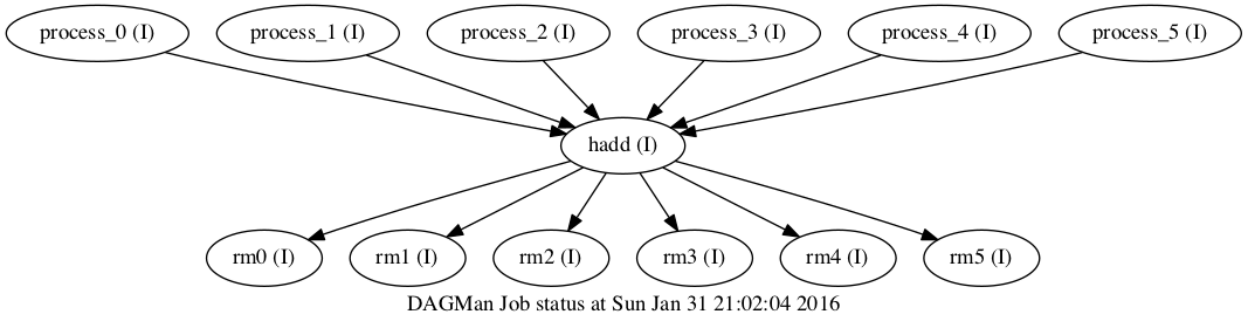
Can a DAG have 1 node/Job?

Yes. You can still have the advantages of auto-retry, `DAGStatus` monitoring, and other DAG options.

What are some cool uses of DAGs?

In addition to the 1-node DAG, you can submit multiple "layers" of processing one go.

For example: say you want to run analysis code over many input files, then `hadd` the files, and finally delete all the intermediate files. This is achievable with a DAG that looks like this:



CHAPTER 5

Changelog

5.1 v0.2.0 (14th June 2016)

- Move setup to `pip` with big thanks to @kreczko https://github.com/raggleton/htcondenser/pull/4:
  - Move python classes out of `htcondenser/core` into just `htcondenser`
  - Rename/move `exe/DAGStatus.py` to `bin/DAGStatus` to aid `pip` deployment
- Use hadoop command to mkdir on HDFS, not `os.makedirs`
- Add check for output file on worker node before transfer
- Add in check to make output dir on HDFS if it doesn't already exist
- Change the readthedocs theme

5.2 v0.1.0 (12th May 2016)

- Initial release.
- Includes classes for jobs and dags.
- Handles transfers to/from HDFS.
- DAG monitoring tool included.
- Basic documentation on readthedocs with examples.

CHAPTER 6

htcondenser package

6.1 Submodules

6.1.1 htcondenser.common module

Functions/classes that are commonly used.

class htcondenser.common.**FileMirror**(*original*, *hdfs*, *worker*)
Bases: object

Simple class to store location of mirrored files: the original, the copy of HDFS, and the copy on the worker node.

htcondenser.common.**check_certificate**()
Check the user's grid certificate is valid, and > 1 hour time left.

Raises RuntimeError – If certificate not valid. If certificate valid but has < 1 hour remaining.

htcondenser.common.**check_dir_create**(*directory*)
Check to see if directory exists, if not create it.

Parameters **directory** (*str*) – Name of directory to check and create.

Raises IOError – If 'directory' already exists but is a file.

htcondenser.common.**cp_hdfs**(*src*, *dest*, *force=True*)
Copy file between src and destination, allowing for one or both to be on HDFS.

Uses the hadoop commands if possible to ensure safe transfer.

Parameters

- **src** (*str*) – Source filepath. For files on HDFS, use the full filepath, /hdfs/...
- **dest** (*str*) – Destination filepath. For files on HDFS, use the full filepath, /hdfs/...
- **force** (*bool, optional*) – If True, will overwrite destination file if it already exists.

htcondenser.common.**date_now**(*fmt='%d %B %Y'*)
Get current date as a string.

Parameters **fmt** (*str, optional*) – Format string for time. Default is %d %B %Y. See strftime docs.

Returns Current date.

Return type str

htcondenser.common.**date_time_now**(*fmt='%H:%M:%S %d %B %Y'*)
Get current date and time as a string.

Parameters **fmt** (*str, optional*) – Format string for time. Default is %H:%M:%S %d %B %Y. See strftime docs.

Returns Current date and time.

Return type str

htcondenser.common.**time_now**(*fmt='%H:%M:%S'*)
Get current time as a string.

Parameters **fmt** (*str, optional*) – Format string for time. Default is %H:%M:%S. See strftime docs.

Returns Current time.

Return type str

6.1.2 htcondenser.dagman module

DAGMan class to handle DAGs in HTCondor.

class htcondenser.dagman.**DAGMan**(*filename='jobs.dag'*, *status_file='jobs.status'*, *status_update_period=30*, *dot=None*, *other_args=None*)
Bases: object

Class to implement DAG, and manage Jobs and dependencies.

Parameters

- **filename** (*str*) – Filename to write DAG jobs. This cannot be on /users, must be on NFS drive, e.g. /storage.
- **status_file** (*str, optional*) – Filename for DAG status file. See https://research.cs.wisc.edu/htcondor/manual/current/2_10DAGMan_Applications.html#SECTION003101200000
- **status_update_period** (*int or str, optional*) – Refresh period for DAG status file in seconds.
- **dot** (*str, optional*) – Filename for dot file. dot can then be used to generate a pictoral representation of jobs in the DAG and their relationships.
- **other_args** (*dict, optional*) – Dictionary of {variable: value} for other DAG options.

JOB_VAR_NAME
str

Name of variable to hold job arguments string to pass to condor_worker.py, required in both DAG file and condor submit file.

JOB_VAR_NAME = 'jobOpts'

add_job(*job*, *requires=None*, *job_vars=None*, *retry=None*)
Add a Job to the DAG.

Parameters

- **job** (Job) – Job object to be added to DAG

- **requires** (*str, Job, iterable[str], iterable[Job], optional*) – Individual or a collection of Jobs or job names that must run first before this job can run. i.e. the job(s) specified here are the parents, whilst the added job is their child.
- **job_vars** (*str, optional*) – String of job variables specifically for the DAG. Note that program arguments should be set in Job.args not here.
- **retry** (*int or str, optional*) – Number of retry attempts for this job. By default the job runs once, and if its exit code != 0, the job has failed.

Raises

- `KeyError` – If a Job with that name has already been added to the DAG.
- `TypeError` – If the *job* argument is not of type Job. If *requires* argument is not of type str, Job, iterable(str) or iterable(Job).

check_job_acyclic(*job*)

Check no circular requirements, e.g. A ->- B ->- A

Get all requirements for all parent jobs recursively, and check for the presence of this job in that list.

Parameters **job** (*Job or str*) – Job or job name to check

Raises `RuntimeError` – If job has circular dependency.

check_job_requirements(*job*)

Check that the required Jobs actually exist and have been added to DAG.

Parameters **job** (*Job or str*) – Job object or name of Job to check.

Raises

- `KeyError` – If job(s) have prerequisite jobs that have not been added to the DAG.
- `TypeError` – If *job* argument is not of type str or Job, or an iterable of strings or Jobs.

generate_dag_contents()

Generate DAG file contents as a string.

Returns DAG file contents

Return type str

generate_job_requirements_str(*job*)

Generate a string of prerequisite jobs for this job.

Does a check to make sure that the prerequisite Jobs do exist in the DAG, and that DAG is acyclic.

Parameters **job** (*Job or str*) – Job object or name of job.

Returns Job requirements if prerequisite jobs. Otherwise blank string.

Return type str

Raises `TypeError` – If *job* argument is not of type str or Job.

generate_job_str(*job*)

Generate a string for job, for use in DAG file.

Includes condor job file, any vars, and other options e.g. RETRY. Job requirements (parents) are handled separately in another method.

Parameters **job** (*Job or str*) – Job or job name.

Returns **name** – Job listing for DAG file.

Return type str

Raises `TypeError` – If *job* argument is not of type str or Job.

get_jobsets()
Get a list of all unique JobSets managing Jobs in this DAG.

Returns **name** – List of unique JobSet objects.

Return type list

submit(*force=False*, *submit_per_interval=10*)
Write all necessary submit files, transfer files to HDFS, and submit DAG. Also prints out info for user.

Parameters

- **force** (*bool, optional*) – Force condor_submit_dag
- **submit_per_interval** (*int, optional*) – Number of DAGMan submissions per interval. The default 10 every 5 seconds.

Raises `CalledProcessError` – If condor_submit_dag returns non-zero exit code.

write()
Write DAG to file and causes all Jobs to write their HTCondor submit files.

6.1.3 htcondenser.job module

Classes to describe individual job, as part of a JobSet.

class htcondenser.job.**Job**(*name*, *args=None*, *input_files=None*, *output_files=None*, *quantity=1*, *hdfs_mirror_dir=None*)
Bases: `object`

One job instance in a JobSet, with defined arguments and inputs/outputs.

Parameters

- **name** (*str*) – Name of this job. Must be unique in the managing JobSet, and DAGMan.
- **args** (*list[str] or str, optional*) – Arguments for this job.
- **input_files** (*list[str], optional*) – List of input files to be transferred across before running executable. If the path is not on HDFS, a copy will be placed on HDFS under *hdfs_store/job.name*. Otherwise, the original on HDFS will be used.
- **output_files** (*list[str], optional*) – List of output files to be transferred across to HDFS after executable finishes. If the path is on HDFS, then that will be the destination. Otherwise *hdfs_mirror_dir* will be used as destination directory.

  e.g. myfile.txt => Job.hdfs_mirror_dir/myfile.txt, results/myfile.txt => Job.hdfs_mirror_dir/myfile.txt, /hdfs/A/B/myfile.txt => /hdfs/A/B/myfile.txt
- **quantity** (*int, optional*) – Quantity of this Job to submit.
- **hdfs_mirror_dir** (*str, optional*) – Mirror directory for files to be put on HDFS. If not specified, will use *hdfs_mirror_dir*/self.name, where *hdfs_mirror_dir* is taken from the manager. If the directory does not exist, it is created.

Raises

- `KeyError` – If the user tries to create a Job in a JobSet which already manages a Job with that name.
- `TypeError` – If the user tries to assign a manager that is not of type JobSet (or a derived class).

generate_job_arg_str()
Generate arg string to pass to the condor_worker.py script.

This includes the user's args (in *self.args*), but also includes options for input and output files, and automatically updating the args to account for new locations on HDFS or worker node. It also includes common input files from managing JobSet.

Returns Argument string for the job, to be passed to condor_worker.py

Return type str

manager
Returns the Job's managing JobSet.

setup_input_file_mirrors(*hdfs_mirror_dir*)
Attach a mirror HDFS location for each non-HDFS input file. Also attaches a location for the worker node, incase the user wishes to copy the input file from HDFS to worker node first before processing.

Will correctly account for managing JobSet's preference for share_exe_setup. Since input_file_mirrors is used for generate_job_arg_str(), we need to add the exe/setup here, even though they don't get transferred by the Job itself.

Parameters **hdfs_mirror_dir** (*str*) – Location of directory to store mirrored copies.

setup_output_file_mirrors(*hdfs_mirror_dir*)
Attach a mirror HDFS location for each output file.

Parameters **hdfs_mirror_dir** (*str*) – Location of directory to store mirrored copies.

transfer_to_hdfs()
Transfer files across to HDFS.

Auto-creates HDFS mirror dir if it doesn't exist, but only if there are 1 or more files to transfer.

Will not transfer exe or setup script if manager.share_exe_setup is True. That is left for the manager to do.

6.1.4 htcondenser.jobset module

Class to describe groups of jobs sharing common settings, that becomes one condor submit file.

class htcondenser.jobset.**JobSet**(*exe*, *copy_exe=True*, *setup_script=None*, *filename='jobs.condor'*, *out_dir='logs'*, *out_file='$(cluster).$(process).out'*, *err_dir='logs'*, *err_file='$(cluster).$(process).err'*, *log_dir='logs'*, *log_file='$(cluster).$(process).log'*, *cpus=1*, *memory='100MB'*, *disk='100MB'*, *certificate=False*, *transfer_hdfs_input=True*, *share_exe_setup=True*, *common_input_files=None*, *hdfs_store=None*, *dag_mode=False*, *other_args=None*)

Bases: object

Manages a set of Jobs, all sharing a common submission file, log locations, resource request, and setup procedure.

Parameters

- **exe** (*str*) – Name of executable for this set of jobs. Note that path must be specified, e.g. './myexe'
- **copy_exe** (*bool, optional*) – If *True*, copies the executable to HDFS. Set *False* for builtins e.g. awk

- **setup_script** (*str, optional*) – Shell script to execute on worker node to setup necessary programs, libs, etc.
- **filename** (*str, optional*) – Filename for HTCondor job description file.
- **out_dir** (*str, optional*) – Directory for STDOUT output. Will be automatically created if it does not already exist. Raises an OSError if already exists but is not a directory.
- **out_file** (*str, optional*) – Filename for STDOUT output.
- **err_dir** (*str, optional*) – Directory for STDERR output. Will be automatically created if it does not already exist. Raises an OSError if already exists but is not a directory.
- **err_file** (*str, optional*) – Filename for STDERR output.
- **log_dir** (*str, optional*) – Directory for log output. Will be automatically created if it does not already exist. Raises an OSError if already exists but is not a directory.
- **log_file** (*str, optional*) – Filename for log output.
- **cpus** (*int, optional*) – Number of CPU cores for each job.
- **memory** (*str, optional*) – RAM to request for each job.
- **disk** (*str, optional*) – Disk space to request for each job.
- **certificate** (*bool, optional*) – Whether the JobSet requires the user's grid certificate.
- **transfer_hdfs_input** (*bool, optional*) – If True, transfers input files on HDFS to worker node first. Auto-updates program arguments to take this into account. Otherwise files are read directly from HDFS. Note that this does not affect input files **not** on HDFS - they will be transferred across regardlass.
- **share_exe_setup** (*bool, optional*) – If True, then all jobs will use the same exe and setup files on HDFS. If False, each job will have their own copy of the exe and setup script in their individual job folder.
- **common_input_files** (*list[str], optional*) – List of common input files for each job. Unlike Job input files, there will only be 1 copy of this input file made on HDFS. Not sure if this will break anything...
- **hdfs_store** (*str, optional*) – If any local files (on */user*) needs to be transferred to the job, it must first be stored on */hdfs*. This argument specifies the directory where those files are stored. Each job will have its own copy of all input files, in a subdirectory with the Job name. If this directory does not exist, it will be created.
- **other_args** (*dict, optional*) – Dictionary of other job options to write to HTCondor submit file. These will be added in **before** any arguments or jobs.

Raises

- `OSError` – If any of *out_file*, *err_file*, or *log_file*, are blank or '.'.
- `OSError` – If any of *out_dir*, *err_dir*, *log_dir*, *hdfs_store* cannot be created.

add_job(*job*)

Add a Job to the collection of jobs managed by this JobSet.

Parameters **job** (`Job`) – Job object to be added.

Raises

- `TypeError` – If *job* argument isn't of type Job (or derived type).
- `KeyError` – If a job with that name is already governed by this JobSet object.

generate_file_contents (*template*, *dag_mode=False*)
Create a job file contents from a template, replacing necessary fields and adding in all jobs with necessary arguments.

Can either be used for normal jobs, in which case all jobs added, or for use in a DAG, where a placeholder for any job(s) is used.

Parameters

- **template** (*str*) – Job template as a single string, including tokens to be replaced.
- **dag_mode** (*bool, optional*) – If True, then submit file will only contain placeholder for job args. This is so it can be used in a DAG. Otherwise, the submit file will specify each Job attached to this JobSet.

Returns Completed job template.

Return type str

Raises IndexError – If the JobSet has no Jobs attached.

setup_common_input_file_mirrors (*hdfs_mirror_dir*)
Attach a mirror HDFS location for each non-HDFS input file. Also attaches a location for the worker node, incase the user wishes to copy the input file from HDFS to worker node first before processing.

Parameters **hdfs_mirror_dir** (*str*) – Location of directory to store mirrored copies.

submit (*force=False*)
Write HTCondor job file, copy necessary files to HDFS, and submit. Also prints out info for user.

Parameters **force** (*bool, optional*) – Force condor_submit

Raises CalledProcessError – If condor_submit returns non-zero exit code.

transfer_to_hdfs ()
Copy any necessary input files to HDFS.

This transfers both common exe/setup (if self.share_exe_setup == True), and the individual files required by each Job.

write (*dag_mode*)
Write jobs to HTCondor job file.

6.2 Module contents

A simple library for submitting jobs on the DICE system at Bristol.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

h

A

C

D

F

G

H

J

M

S

T

W