HTCondor and Workflows: An Introduction

HTCondor Week 2015

Kent Wenger
Outline

- Introduction/motivation
- Basic DAG concepts
- Pre/Post scripts
- Rescue DAGs
- Running and monitoring a DAG
Why workflows?
My jobs have dependencies…

Can HTCondor help solve my dependency problems?

Yes!

Workflows are the answer
What are workflows?

› General: a sequence of connected steps

› Our case
  • Steps are HTCondor jobs
  • Sequence defined at higher level
  • Controlled by a Workflow Management System (WMS), *not just a script*
Example workflow

...10k...

Preparation

Simulation

Analysis
Workflows – launch and forget

› Automates tasks user *could* perform manually (for example, the previous slide)...
  – But **WMS** takes care of automatically

› A workflow can take days, weeks or even months

› The result: one user action can utilize many resources while maintaining complex job inter-dependencies and data flows
Workflow management systems

- DAGMan (Directed Acyclic Graph Manager)
  - HTCondor's WMS
  - Introduction/basic features in this talk
  - Advanced/new features in later talk
- Pegasus
  - A higher level on top of DAGMan
    - Data- and grid-aware
  - A talk tomorrow with more details
Outline

› Introduction/motivation
› Basic DAG concepts
› Pre/Post scripts
› Rescue DAGs
› Running and monitoring a DAG
DAG (directed acyclic graph) definitions

- DAGs have one or more nodes (or vertices)
- Dependencies are represented by arcs (or edges). These are arrows that go from parent to child
- No cycles!
Each node represents an HTCondor job (or cluster)

Dependencies define possible orders of job execution
Charlie learns DAGMan

- **Directed Acyclic Graph Manager**

- DAGMan allows Charlie to specify the dependencies between his HTCondor jobs, so DAGMan manages the jobs automatically.

- Dependency example: do not get married until rehab has completed successfully.
A DAG input file defines a DAG:

```
# file name: diamond.dag
Job A a.submit
Job B b.submit
Job C c.submit
Job D d.submit
Parent A Child B C
Parent B C C Child D
```
Basic DAG commands

- **Job** command defines a name, associates that name with an HTCondor submit file
  - The name is used in many other DAG commands
  - “Job” should really be “node”
- **Parent...child** command creates a dependency between nodes
  - Child cannot run until parent completes successfully
Submit description files

For node B:
# file name:  
#   b.submit
universe   = vanilla
executable = B
input      = B.in
output     = B.out
error      = B.err
log        = B.log
queue

For node C:
# file name:  
#   c.submit
universe   = standard
executable = C
input      = C.in1
output     = C.out
error      = C.err
log        = C.log
queue
Input      = C.in2
queue
Jobs/clusters

› Submit description files used in a DAG can create multiple jobs, but they must all be in a single cluster.
  – A submit file that creates >1 cluster causes node failure

› The failure of any job means the entire cluster fails. Other jobs in the cluster are removed.
Node success or failure

› A node either succeeds or fails
› Based on the return value of the job(s)
  0: success
  not 0: failure
› This example: C fails
› Failed nodes block execution; DAG fails
Outline

› Introduction/motivation
› Basic DAG concepts
› Pre/Post scripts
› Rescue DAGs
› Running and monitoring a DAG
PRE and POST scripts

- DAGMan allows optional PRE and/or POST scripts for any node
  - Not necessarily a script: any executable
  - Run before (PRE) or after (POST) job
  - Scripts run on submit machine (not execute machine)
- In the DAG input file:
  - Job A a.submit
  - Script PRE A before-script arguments
  - Script POST A after-script arguments
DAG node with scripts

- DAGMan treats the node as a unit (e.g., dependencies are between nodes)
- PRE script, Job, or POST script determines node success or failure (table in manual gives details)
- If PRE script fails, job is not run. The POST script is run.
Why PRE/POST scripts?

› Set up input
› Check output
› Dynamically create submit file or sub-DAG (more later today)
› Probably lots of other reasons…

› Should be lightweight (run on submit machine)
Script argument variables

- **$JOB**: node name
- **$JOBID**: Condor ID (`cluster.proc`) (POST only)
- **$RETRY**: current retry
- **$MAX_RETRIES**: max # of retries
- **$RETURN**: exit code of HTCondor/Stork job (POST only)
- **$PRE_SCRIPT_RETURN**: PRE script return value (POST only)
- **$DAG_STATUS**: A number indicating the state of DAGMan. See the manual for details.
- **$FAILED_COUNT**: the number of nodes that have failed in the DAG
Outline

› Introduction/motivation
› Basic DAG concepts
› Pre/Post scripts
› Rescue DAGs
› Running and monitoring a DAG
Rescue DAGs

- What if things don't complete perfectly?
- We want to re-try without duplicating work
- Rescue DAGs do this – details in later talk
- Generated automatically when DAG fails
- Run automatically
Outline

› Introduction/motivation
› Basic DAG concepts
› Pre/Post scripts
› Rescue DAGs
› Running and monitoring a DAG
Submitting a DAG to HTCondor

› To submit an entire DAG, run

   `condor_submit_dag DagFile`

› `condor_submit_dag` creates a submit description file for DAGMan, and DAGMan itself is submitted as an HTCondor job (in the scheduler universe)

› `-f(orce)` option forces overwriting of existing files (to re-run a previously-run DAG)

› Don't try to run duplicate DAG instances!
Controlling running DAGs: remove

- `condor_rm dagman_id`
  
  - Removes *entire* workflow
    - Removes all queued node jobs
    - Kills PRE/POST scripts
  
  - Creates rescue DAG (*more on this on later today*)
  
  - Work done by partially-completed node jobs is lost
    - Relatively small jobs are good
Controlling running DAGs: hold/release

- **condor_hold dagman_id**
  - “Pauses” the DAG
    - Queued node jobs continue
    - No new node jobs submitted
    - No PRE or POST scripts are run
  - DAGMan stays in queue if not released

- **condor_release dagman_id**
  - DAGMan “catches up”, starts submitting jobs
Controlling running DAGs: the halt file

• “Pauses” the DAG (different semantics than hold)
  • Queued node jobs continue
  • POST scripts are run as jobs finish
  • No new jobs will be submitted and no PRE scripts will be run

• When all submitted jobs complete, DAGMan creates a rescue DAG and exits (if not un-halted)
The halt file (cont)

› Create a file named **DagFile.halt** in the same directory as your DAG file.
› Remove halt file to resume normal operation
› Should be noticed w/in 5 sec (\texttt{DAGMAN\_USER\_LOG\_SCAN\_INTERVAL})

› Good if load on submit machine is very high
› Avoids hold/release problem of possible duplicate PRE/POST script instances
Monitoring running DAGs:
condor_q -dag

- Shows current workflow state
- The `-dag` option associates DAG node jobs with the parent DAGMan job

```bash
> condor_q -dag

<table>
<thead>
<tr>
<th>ID</th>
<th>OWNER/NODENAME</th>
<th>SUBMITTED</th>
<th>RUN_TIME</th>
<th>ST</th>
<th>PRI</th>
<th>SIZE</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>392.0</td>
<td>nwp</td>
<td>4/25 13:27</td>
<td>00:00:50</td>
<td>R</td>
<td>0</td>
<td>1.7</td>
<td>condor_dagman -f -</td>
</tr>
<tr>
<td>393.0</td>
<td></td>
<td>-1</td>
<td>4/25 13:27</td>
<td>00:00:23</td>
<td>R</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>395.0</td>
<td></td>
<td>-0</td>
<td>4/25 13:27</td>
<td>00:00:30</td>
<td>R</td>
<td>0</td>
<td>1.7</td>
</tr>
<tr>
<td>399.0</td>
<td></td>
<td>-A</td>
<td>4/25 13:28</td>
<td>00:00:03</td>
<td>R</td>
<td>0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

4 jobs; 0 completed, 0 removed, 0 idle, 4 running, 0 held, 0 suspended
Monitoring a DAG: dagman.out file

- Logs detailed workflow history
- Mostly for debugging – first place to look if something goes wrong!

*DagFile.dagman.out*

- Verbosity controlled by the `DAGMAN_VERBOSITY` configuration macro and `-debug n` on the `condor_submit_dag` command line
  - 0: least verbose
  - 7: most verbose

- Don’t decrease verbosity unless really needed
Dagman.out contents

...  
04/17/11 13:11:26 Submitting Condor Node A job(s)...  
04/17/11 13:11:26 submitting: condor_submit -a dag_node_name' '=' 'A -a +DAGManJobId' '=' '180223 -a DAGManJobId' '=' '180223 -a submit_event_notes' '=' 'DAG' 'Node:' 'A -a +DAGParentNodeNames' '=' ''' dag_files/A2.submit  
04/17/11 13:11:27 From submit: Submitting job(s).  
04/17/11 13:11:27 From submit: 1 job(s) submitted to cluster 180224.  
04/17/11 13:11:27 assigned Condor ID (180224.0.0)  
04/17/11 13:11:27 Just submitted 1 job this cycle...  
04/17/11 13:11:27 Currently monitoring 1 Condor log file(s)  
04/17/11 13:11:27 Event: ULOG_SUBMIT for Condor Node A (180224.0.0)  
04/17/11 13:11:27 Number of idle job procs: 1  
04/17/11 13:11:27 Of 4 nodes total:  
04/17/11 13:11:27 Done Pre Queued Post Ready Un-Ready Failed  
| 04/17/11 13:11:27 | === | === | === | === | === | === | === | === | === | === | === |
|------------------|-----|-----|-----|-----|-----|-----|-----|-----|
| 04/17/11 13:11:27 | 0   | 0   | 1   | 0   | 0   | 3   | 0   |     |

04/17/11 13:11:27 0 job proc(s) currently held  

...  

This is a small excerpt of the dagman.out file.
More information

- More in later talk!
- There’s much more detail, as well as examples, in the DAGMan section of the online HTCondor manual.
- DAGMan: http://research.cs.wisc.edu/htcondor/dagman/dagman.html
- For more questions: htcondor-admin@cs.wisc.edu