The Particle Physics Data Grid
Final Report

Miron Livny
The Condor Project
Computer Sciences Department
University of Wisconsin-Madison

The main objective of the Particle Physics Data Grid (PPDG) project has been to implement and evaluate distributed (Grid-enabled) data access and management technology for current and future particle and nuclear physics experiments. PPDG has been underway since August 1999, funded initially by the DOE Next Generation Internet program, and recently by a combination of DOE HENP and the Mathematical, Information, and Computational Sciences (MICS) office. It is has been a collaborative effort between computer scientists and physicists at DOE HENP laboratories (Argonne, Berkeley, Brookhaven, Fermilab, Jefferson and SLAC), Caltech, the San Diego Supercomputer Center (SDSC) and the Condor Project at the University of Wisconsin. PPDG has focused on integrating middleware and tools under development by members of the collaboration.

Within the broad vision of Grid-enabled data management and access for HENP, the specific goals of PPDG have been to:

- Design, implement, and deploy a Grid-based software infrastructure capable of supporting the data generation, processing and analysis needs common to the physics experiments represented by the participants
- Adapt experiment-specific software to operate in this Grid environment and to exploit this infrastructure.

To accomplish these goals, the PPDG has focused on the implementation and deployment of three critical services:

- Reliable and efficient File Replication Service
- High Speed Data Transfer Services
- Multi-Site File Caching and Staging Service
- Reliable and recoverable job management services

The focus of our activity has been the job management services and the interplay between these services and distributed data access in a Grid environment. We have studied the special needs and requirements of the HENP community in this area and developed software to study the interaction between HENP applications

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

and distributed data storage fabric. We developed a number of prototypes that enabled us to study and experiment with different approaches to interfacing HENP applications to remote storage devices and staging data from/to such devices to local data caches. A number of independent research and developed activities (e.g. The Pluggable File System (http://www.cs.wisc.edu/condor/pfs/), The Network Storage Appliance (http://www.cs.wisc.edu/condor/pfs/), and the formulation of the frame work of Data Placement (DaP) jobs) were triggered by these activities.

One key conclusion from these early studies was the need for a reliable and recoverable tool for managing large collections of interdependent jobs. In the attached document we provide an overview of the current status of the Directed Acyclic Graph Manager (DAGMan) we developed and list its main features and capabilities. DAGMan has proven to be an important contributor to Grid middleware and adopted by most HENP Grid projects (e.g. The Grid Physics Network (www.GriPhyN.org) and the EU Data Grid Project (eu-datagrid.web.cern.ch/eu-datagrid/).

The Directed Acyclic Graph Manager (DAGMan)

Condor Project
Computer Sciences Department
University Of Wisconsin-Madison

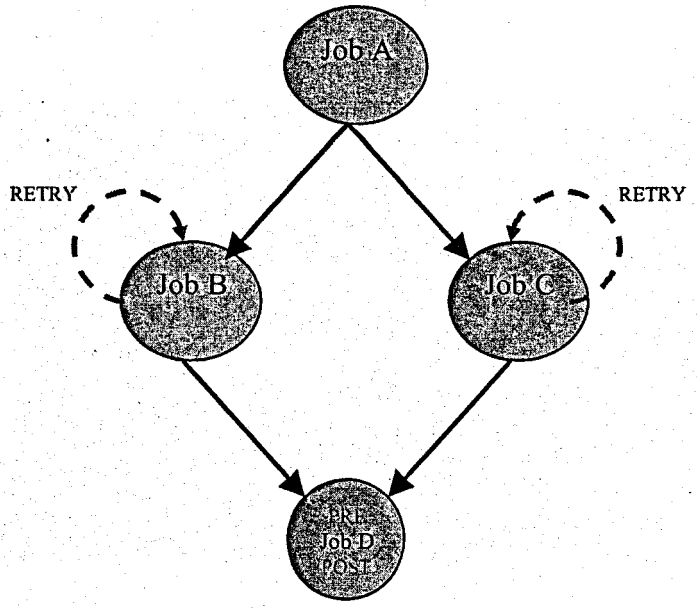
Overview

The Directed Acyclic Graph Manager (DAGMan) is a service developed for managing the execution of multiple distributed jobs with interdependencies. DAGMan accepts a declaration of the jobs to be run and the constraints on their order, and manages their execution in a fault-tolerant manner. DAGMan assumes that re-execution of old work is expensive, and is designed to continue right where it left off in the aftermath of crashes and other failures.

DAGMan relies on Condor in two important ways. First, DAGMan uses Condor as a service for reliably executing jobs. DAGMan need not worry about the many ways that an individual job may fail in a distributed system, because Condor assumes all responsibility for hiding and retrying such errors. Thus, DAGMan need only concern itself with the details of ordering and task selection, and ensuring the fault-tolerance of the *sequence* of jobs, rather than the individual jobs themselves.

Second, Condor is also responsible for making DAGMan itself reliable. To accomplish this, the DAGMan service is run as a special Condor job which simply executes locally, at the submission site. Once started, DAGMan turns around and submits jobs of its own back to the same Condor queue it is running in.

```
# example.dag
JOB     A   job-A.condor
JOB     B   job-B.condor
RETRY   B   3
JOB     C   job-C.condor
RETRY   C   3
JOB     D   job-D.condor
PRE     D   prepare-D.sh
POST    D   double-check-D.sh
PARENT  A   Child B C
PARENT  B C Child D
```



The figure above demonstrates the declaration language accepted by DAGMan. A JOB statement associates an abstract name (A) with a file that describes complete Condor job (job-A.condor). Each one of these pairs represents a node in the DAG. A PARENT/CHILD statement describes the relationship between two or more nodes. In this script, nodes B and C may not run until A has completed, while node D may not run until C has completed. Nodes that are independent of each other may run in any order, and possibly simultaneously.

In this script, node B and C are associated with a RETRY value. This value indicates how many times DAGMan should retry a failed node before giving up. Node D is associated with a PRE and a POST program. These commands indicate optional programs to be run before and after a node executes. A job, combined with its optional PRE and POST programs, constitutes a single "node" in the DAG.

PRE and POST programs are not submitted as Condor jobs, but are simply run by DAGMan on the submitting machine. PRE programs are generally used to prepare for execution (e.g., transferring or uncompressing input files), while POST programs are generally used to clean up after, or evaluate the output of the job to determine whether it succeeded or not.

Each node in the DAG is atomic with respect to recovery. In other words, either all the components of a node (PRE, job, and POST) succeed, in which case the node succeeds, or else the node fails and can be retried in its entirety.

DAGMan presents an excellent opportunity to study the problem of multi-level error processing. In a complex system that ranges from the high-level view of DAGs all the way down to the minutiae of remote procedure calls, it is essential to tease out the source of an error in order to avoid unnecessarily burdening the user with error messages.

Jobs may fail because of the nature of the distributed system. Network outages and reclaimed resources may cause Condor to lose contact with a running job. Such failures are not indications that the job itself has failed, but rather that the system has failed the job. Such situations are detected and retried by Condor in its responsibility to execute jobs reliably. DAGMan is never aware of such failures.

Jobs may also fail of their own accord, however. A job may produce an ordinary error result if the user forgot to provide a necessary argument or input file. In this case, DAGMan is notified that the job has completed and detects a program result indicating an error. It responds by noting the error, and continuing to run any other jobs which are not dependant on the failed one. When DAGMan can no longer make forward progress, it writes out a "rescue" DAG and itself exits with an error code. The rescue DAG is simply a copy of the original DAG, indicating which nodes succeeded and which did not. To continue, the user may examine the rescue DAG, fix any mistakes in submission, and resubmit it as a normal DAG.

Internally, DAGMan represents the DAG with a Dag object, which contains Job objects. Each Job object contains references to the Job objects of its parents and children. The

Dag object also contains a number of internal data structures which DAGMan uses to manage its work. The first is the readyQ, which contains all of the uncompleted nodes which are ready to run given the DAG's dependencies. When DAGMan starts, the readyQ is populated with all of the nodes with no parents (i.e., no dependencies), and is updated each time a node completes. The submitQ keeps track of all the jobs DAGMan has submitted to Condor, but has not yet received confirmation of.

DAGMan receives all of its information on the state of its jobs by monitoring the Condor user log. The user log is a file into which Condor writes records for every event in a job's lifetime: submission, execution, preemption, termination, etc. DAGMan does not assume that a job has been successfully submitted or terminated until it sees the corresponding event in the user log.

The preScriptQ and postScriptQ contain all of the PRE and POST programs that are ready to run given the state of the DAG. Although in many cases these queues are small, because PRE and POST scripts execute quickly, the queues are needed because DAGMan allows the user to throttle the number of local programs it runs simultaneously to avoid overwhelming the local system resources.

Likewise, DAGMan allows the user to throttle the number of jobs it submits to Condor at one time, to avoid resource over-utilization for jobs that consume resources (e.g., disk) even when idle in the queue.

Technical Manual

A directed acyclic graph (DAG) can be used to represent a set of programs where the input, output, or execution of one or more programs is dependent on one or more other programs. The programs are nodes (vertices) in the graph, and the edges (arcs) identify the dependencies. Condor finds machines for the execution of programs, but it does not schedule programs (jobs) based on dependencies. The Directed Acyclic Graph Manager (DAGMan) is a meta-scheduler for Condor jobs. DAGMan submits jobs to Condor in an order represented by a DAG and processes the results. An input file defined prior to submission describes the DAG, and a Condor submit description file for each program in the DAG is used by Condor.

Each node (program) in the DAG needs its own Condor submit description file. As DAGMan submits jobs to Condor, it uses a single Condor log file to enforce the ordering required for the DAG. The DAG itself is defined by the contents of a DAGMan input file. DAGMan is responsible for scheduling, recovery, and reporting for the set of programs submitted to Condor.

The following sections specify the use of DAGMan.

Input File describing the DAG

The input file used by DAGMan specifies four items:

1. A list of the programs in the DAG. This serves to name each program and specify each program's Condor submit description file.
2. Processing that takes place before submission of any program in the DAG to Condor or after Condor has completed execution of any program in the DAG.
3. Description of the dependencies in the DAG.
4. Number of times to retry if a node within the DAG fails.

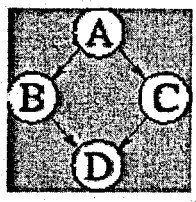
Comments may be placed in the input file that describes the DAG. The pound character (#) as the first character on a line identifies the line as a comment. Comments do not span lines.

An example input file for DAGMan is

```
# Filename: diamond.dag
#
Job  A  A.condor
Job  B  B.condor
Job  C  C.condor
Job  D  D.condor
Script PRE  A top_pre.csh
Script PRE  B mid_pre.perl  $JOB
Script POST B mid_post.perl $JOB $RETURN
Script PRE  C mid_pre.perl  $JOB
Script POST C mid_post.perl $JOB $RETURN
Script PRE  D bot_pre.csh
PARENT A CHILD B C
PARENT B C CHILD D
Retry  C 3
```

This input file describes the DAG shown in Figure 2.2.

Figure 2.2: Diamond DAG



The first section of the input file lists all the programs that appear in the DAG. Each program is described by a single line called a Job Entry. The syntax used for each Job Entry is

JOB JobName CondorSubmitDescriptionFile [*DONE*]

A Job Entry maps a *JobName* to a Condor submit description file. The *JobName* uniquely identifies nodes within the DAGMan input file and within output messages.

The keyword *JOB* and the *JobName* are not case sensitive. A *JobName* of *joba* is equivalent to *JobA*. The *CondorSubmitDescriptionFile* is case sensitive, since the UNIX file system is case sensitive. The *JobName* can be any string that contains no white space.

The optional *DONE* identifies a job as being already completed. This is useful in situations where the user wishes to verify results, but does not need all programs within the dependency graph to be executed. The *DONE* feature is also utilized when an error occurs causing the DAG to not be completed. DAGMan generates a Rescue DAG, a DAGMan input file that can be used to restart and complete a DAG without re-executing completed programs.

The second type of item in a DAGMan input file enumerates processing that is done either before a program within the DAG is submitted to Condor for execution or after a program within the DAG completes its execution. Processing done before a program is submitted to Condor is called a *PRE* script. Processing done after a program successfully completes its execution under Condor is called a *POST* script. A node in the DAG is comprised of the program together with *PRE* and/or *POST* scripts. The dependencies in the DAG are enforced based on nodes.

Syntax for *PRE* and *POST* script lines within the input file

SCRIPT PRE JobName ExecutableName [*arguments*]

SCRIPT POST JobNameExecutableName [*arguments*]

The *SCRIPT* keyword identifies the type of line within the DAG input file. The *PRE* or *POST* keyword specifies the relative timing of when the script is to be run. The *JobName* specifies the node to which the script is attached. The *ExecutableName* specifies the script to be executed, and it may be followed by any command line arguments to that script. The *ExecutableName* and optional *arguments* have their case preserved.

Scripts are optional for each job, and any scripts are executed on the machine to which the DAG is submitted.

The PRE and POST scripts are commonly used when files must be placed into a staging area for the job to use, and files are cleaned up or removed once the job is finished running. An example using PRE/POST scripts involves staging files that are stored on

tape. The PRE script reads compressed input files from the tape drive, and it uncompresses them, placing the input files in the current directory. The program within the DAG node is submitted to Condor, and it reads these input files. The program produces output files. The POST script compresses the output files, writes them out to the tape, and then deletes the staged input and output files.

DAGMan takes note of the exit value of the scripts as well as the program. If the PRE script fails (exit value != 0), then neither the program nor the POST script runs, and the node is marked as failed.

If the PRE script succeeds, the program is submitted to Condor. If the program fails and there is no POST script, the DAG node is marked as failed. An exit value not equal to 0 indicates program failure. It is therefore important that the program returns the exit value 0 to indicate the program did not fail.

If the program fails and there is a POST script, node failure is determined by the exit value of the POST script. A failing value from the POST script marks the node as failed. A succeeding value from the POST script (even with a failed program) marks the node as successful. Therefore, the POST script may need to consider the return value from the program.

By default, the POST script is run regardless of the program's return value. To prevent POST scripts from running after failed jobs, pass the *-NoPostFail* argument to *condor_submit_dag*.

A node not marked as failed at any point is successful.

Two variables are available to ease script writing. The `$JOB` variable evaluates to *JobName*. For POST scripts, the `$RETURN` variable evaluates to the return value of the program. The variables may be placed anywhere within the arguments.

As an example, suppose the *PRE* script expands a compressed file named `JobName.gz`. The *SCRIPT* entry for jobs A, B, and C are

```
SCRIPT PRE  A  pre.csh $JOB .gz
SCRIPT PRE  B  pre.csh $JOB .gz
SCRIPT PRE  C  pre.csh $JOB .gz
```

The script `pre.csh` may use these arguments

```
#!/bin/csh
gunzip $argv[1]$argv[2]
```

The third type of item in the DAG input file describes the dependencies within the DAG. Nodes are parents and/or children within the DAG. A parent node must be completed successfully before any child node may be started. A child node is started once all its parents have successfully completed.

The syntax of a dependency line within the DAG input file:

PARENT ParentJobName... CHILD ChildJobName...

The *PARENT* keyword is followed by one or more *ParentJobNames*. The *CHILD* keyword is followed by one or more *ChildJobNames*. Each child job depends on every parent job on the line. A single line in the input file can specify the dependencies from one or more parents to one or more children. As an example, the line

```
PARENT p1 p2 CHILD c1 c2
```

produces four dependencies:

1. `p1 to c1`
2. `p1 to c2`
3. `p2 to c1`
4. `p2 to c2`

The fourth type of item in the DAG input file provides a way (optional) to retry failed nodes. The syntax for retry is

Retry JobName NumberOfRetries

where the *JobName* is the same as the name given in a Job Entry line, and *NumberOfRetries* is an integer, the number of times to retry the node after failure. The default number of retries for any node is 0, the same as not having a retry line in the file.

In the event of retry, all parts of a node within the DAG are redone, following the same rules regarding node failure as given above. The PRE script is executed first, followed by submitting the program to Condor upon success of the PRE script. Failure of the node is then determined by the return value of the program, the existence and return value of a POST script.

Condor Submit Description File

Each node in a DAG may be a unique executable, each with a unique Condor submit description file. Each program may be submitted to a different universe within Condor, for example standard, vanilla, or DAGMan.

Two limitations exist. First, each Condor submit description file must submit only one job. There may not be multiple `queue` lines, or DAGMan will fail. The second limitation is that the submit description file for all jobs within the DAG must specify the same log. DAGMan enforces the dependencies within a DAG using the events recorded in the log file produced by job submission to Condor.

Here is an example Condor submit description file to go with the diamond-shaped DAG example.

```
# Filename: diamond_job.condor
#
executable   = /path/diamond.exe
output       = diamond.out.$(cluster)
error        = diamond.err.$(cluster)
log          = diamond_condor.log
universe     = vanilla
notification = NEVER
queue
```

This example uses the same Condor submit description file for all the jobs in the DAG. This implies that each node within the DAG runs the same program. The `$(cluster)` macro is used to produce unique file names for each program's output. Each job is submitted separately, into its own cluster, so this provides unique names for the output files.

The notification is set to NEVER in this example. This tells Condor not to send e-mail about the completion of a program submitted to Condor. For DAGs with many nodes, this is recommended to reduce or eliminate excessive numbers of e-mails.

Job Submission

A DAG is submitted using the program *condor_submit_dag*. See the manual page for complete details. A simple submission has the syntax

condor_submit_dag DAGInputFileName

The example may be submitted with

```
condor_submit_dag diamond.dag
```

In order to guarantee recoverability, the DAGMan program itself is run as a Condor job. As such, it needs a submit description file. *condor_submit_dag* produces the needed file, naming it by appending the *DAGInputFileName* with `.condor.sub`. This submit description file may be editted if the DAG is submitted with

```
condor_submit_dag -no_submit diamond.dag
```

causing *condor_submit_dag* to generate the submit description file, but not submit DAGMan to Condor. To submit the DAG, once the submit description file is editted, use

```
condor_submit diamond.dag.condor.sub
```

An optional argument to *condor_submit_dag*, *-maxjobs*, is used to specify the maximum number of Condor jobs that DAGMan may submit to Condor at one time. It is commonly used when there is a limited amount of input file staging capacity. As a specific example,

consider a case where each job will require 4 Mbytes of input files, and the jobs will run in a directory with a volume of 100 Mbytes of free space. Using the argument *-maxjobs 25* guarantees that a maximum of 25 jobs, using a maximum of 100 Mbytes of space, will be submitted to Condor at one time.

While the *-maxjobs* argument is used to limit the number of Condor jobs submitted at one time, it may be desirable to limit the number of scripts running at one time. The optional *-maxpre* argument limits the number of PRE scripts that may be running at one time, while the optional *-maxpost* argument limits the number of POST scripts that may be running at one time.

Job Monitoring

After submission, the progress of the DAG can be monitored by looking at the common log file, observing the e-mail that program submission to Condor causes, or by using *condor_q -dag*.

Job Failure and Job Removal

condor_submit_dag attempts to check the DAG input file to verify that all the nodes in the DAG specify the same log file. If a problem is detected, *condor_submit_dag* prints out an error message and aborts.

To omit the check that all nodes use the same log file, as may be desired in the case where there are thousands of nodes, submit the job with the *-log* option. An example of this submission:

```
condor_submit_dag -log diamond_condor.log
```

This option tells *condor_submit_dag* to omit the verification step and use the given file as the log file.

To remove an entire DAG, consisting of DAGMan plus any jobs submitted to Condor, remove the DAGMan job running under Condor. *condor_q* will list the job number. Use the job number to remove the job, for example

```
% condor_q
-- Submitter: turunmaa.cs.wisc.edu : <128.105.175.125:36165> :
turunmaa.cs.wisc.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
    9.0   smoler          10/12 11:47   0+00:01:32 R  0   8.7
condor_dagman -f -
   11.0   smoler          10/12 11:48   0+00:00:00 I  0   3.6  B.out
   12.0   smoler          10/12 11:48   0+00:00:00 I  0   3.6  C.out

         3 jobs; 2 idle, 1 running, 0 held

% condor_rm 9.0
```

Before the DAGMan job stops running, it uses *condor_rm* to remove any Condor jobs within the DAG that are running.

In the case where a machine is scheduled to go down, DAGMan will clean up memory and exit. However, it will leave any submitted jobs in Condor's queue.

Job Recovery: The Rescue DAG

DAGMan can help with the resubmission of uncompleted portions of a DAG when one or more nodes resulted in failure. If any node in the DAG fails, the remainder of the DAG is continued until no more forward progress can be made based on the DAG's dependencies. At this point, DAGMan produces a file called a Rescue DAG.

The Rescue DAG is a DAG input file, functionally the same as the original DAG file. It additionally contains indication of successfully completed nodes using the *DONE* option in the input description file. If the DAG is resubmitted using this Rescue DAG input file, the nodes marked as completed will not be reexecuted.

The Rescue DAG is automatically generated by DAGMan when a node within the DAG fails. The file is named using the *DAGInputFileName*, and appending the suffix `.rescue` to it. Statistics about the failed DAG execution are presented as comments at the beginning of the Rescue DAG input file.

If the Rescue DAG file is generated before all retries of a node are completed, then the Rescue DAG file will also contain Retry entries. The number of retries will be set to the appropriate remaining number of retries.