22

Workflow Management in Condor

Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger

22.1 Introduction

The Condor project began in 1988 and has evolved into a feature-rich batch system that targets high-throughput computing; that is, Condor ([262], [414]) focuses on providing reliable access to computing over long periods of time instead of highly tuned, high-performance computing for short periods of time or a small number of applications.

Many Condor users have not only long-running jobs but complex sequences of jobs, or workflows, that they wish to run. In the late 1990s, we began development of DAGMan (Directed Acyclic Graph Manager), which allows users to submit large workflows to Condor. As with Condor, the focus has been on reliability. DAGMan has a simple interface that allows many, but certainly not all, types of workflows to be expressed. We have found through years of experience running production workflows with our users that solving the "simple" problems can be surprisingly complex. In the first half of this chapter, we therefore provide a conceptual (and almost chronological) development of DAGMan to illustrate the complexities that arise in running workflows in production environments.

In the past several years, Condor has expanded its focus from running jobs on local clusters of computers (or *pools* in Condor terminology) to running jobs in distributed Grid environments. Along with the additional complexities in running jobs came greater challenges in transferring data to and from the job execution sites. We have developed Stork [244], which treats data placement with the same concern that Condor treats job execution.

With a combination of DAGMan, Condor, and Stork, users can create large, complex workflows that reliably "get the job done" in a Grid environment. In the rest of this chapter, we explore DAGMan and Stork (Condor has been covered in detail elsewhere).

22.2 DAGMan Design Principles

The goal of DAGMan is to automate the submission and management of complex workflows involving many jobs, with a focus on reliability and fault tolerance in the face of a variety of errors. Workflow management includes not only job submission and monitoring but job preparation, cleanup, throttling, retry, and other actions necessary to ensure the good health of important workflows. Note that DAGMan addresses workflow execution but does not directly address workflow generation or workflow refinement. Instead, DAGMan is an underlying execution environment that can be used by higher-level workflow systems that perform generation and refinement.

DAGMan attempts to overcome or work around as many execution errors as possible, and in the face of errors it cannot overcome, it endeavors to allow the user to resolve the problem manually and then resume the workflow from the point where it last left off. This can be thought of as a "checkpointing" of the workflow, just as some batch systems provide checkpointing of jobs.

Notably, the majority of DAGMan's features—and even some of its specific semantics—were not originally envisioned but rather are the product of years of collaboration with active users. The experience gained from the needs and problems of production science applications has driven most DAGMan development over the past six years.

The fundamental design principles of DAGMan are as follows:

- DAGMan sits as a layer "above" the batch system in the software stack. DAGMan utilizes the batch system's standard API and logs in order to submit, query, and manipulate jobs, and does not directly interact with jobs independently.[1]
- DAGMan reads the logs of the underlying batch system to follow the status of submitted jobs rather than invoking interactive tools or service APIs. Reliance on simpler, file-based I/O allows DAGMan's own implementation to be simpler, more scalable and reliable across many platforms, and therefore more robust. For example, if DAGMan has crashed while the underlying batch system continues to run jobs, DAGMan can recover its state upon restart and there is no concern about missing callbacks or gathering information if the batch system is temporarily unavailable: It is all in the log file.

[1] Note that DAGMan assumes the batch system guarantees that it will not "lose" jobs after they have been successfully submitted. Currently, if the job is lost by the batch system after being successfully submitted by DAGMan, DAGMan will wait indefinitely for the status of the job in the queue to change. An explicit query for the status of submitted jobs (as opposed to waiting for the batch system to record job status changes) may be necessary to address this. Also, if a job languishes in the queue forever, DAGMan currently is not able to "timeout" and remove the job and mark it as failed. When removing jobs, detecting and responding to the failure of a remove operation (leaving a job "stuck" in the queue) is an interesting question.

- DAGMan has no persistent state of its own—its runtime state is built entirely from its input files and from the information gleaned by reading logs provided by the batch system about the history of the jobs it has submitted.

22.3 DAGMan Details

22.3.1 DAGMan Basics

DAGMan allows users to express job dependencies as arbitrary *directed acyclic graphs*, or DAGs. In the simplest case, DAGMan can be used to ensure that two jobs execute sequentially—for example, that job B is not submitted until job A has completed successfully.

Like all graphs, a DAGMan DAG consists of nodes and arcs. Each node represents a single instance of a batch job to be executed, and each arc represents the execution order to be enforced between two nodes. Unlike more complex systems such as those discussed in [476], arcs merely indicate the order in which the jobs must run.

If an arc points from node N_p to N_c, we say that N_p is the *parent* of N_c and N_c is the *child* of N_p (see Figure 22.1). A parent node must complete successfully before any of its child nodes can be started. Note that each node can have any whole number of parents or children (including zero). DAGMan does not require that DAGs be fully connected.

Why does DAGMan require a directed acyclic graph instead of an arbitrary graph? The graph is directed in order to express the sequence in which jobs must run. Likewise, the graph is acyclic to ensure that DAGMan will not run indefinitely. In practice, we find either that most workflows we encounter do not require loops or that the loops can be unrolled into an acyclic graph.

DAGMan seeks to run as many jobs as possible in parallel, given the constraints of their parent/child relationships. For example, in for the DAG in Figure 22.2, DAGMan will initially submit both N_1 and N_5 to Condor, allowing them to execute in parallel if there are sufficient computers available. After N_1 completes successfully, DAGMan will submit both N_2 and N_3 to the batch system, allowing them to execute in parallel with each other, and with N_5 if it has not completed already. When both N_2 and N_3 have finished successfully, DAGMan will submit N_4. If N_5 and N_4 both complete successfully, the DAG will have completed, and DAGMan will exit successfully.

Earlier we defined a node's parent and child relationships. In describing DAGs, it can also be useful to define a node's *sibling* as any node that shares the same set of parent nodes (including the empty set). Although sibling relationships are not represented explicitly inside DAGMan, they are important because sibling nodes become "runable" simultaneously when their parents complete successfully. In Figure 22.2, N_1 and N_5 are siblings with no parents, and N_2 and N_3 are siblings that share N_1 as a parent.

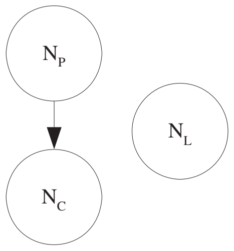

Figure 22.1: The relationship between parents and children. N_p is the parent of N_c. N_L is lonely and has no parents or children.

In practice, however, DAGMan submits individual jobs to the batch scheduler one at a time and makes no guarantees about the precise order in which it will submit the jobs of nodes that are ready to run. In other words, N_1 and N_5 may be submitted to the batch system in any order.

It is also important to remember that, once submitted, the batch system is free to run jobs in its queue in any order it chooses. N_5 may run after N_4, despite being submitted to the queue earlier. Additionally, the jobs may not be run in parallel if there are insufficient computing resources for all parallel jobs.

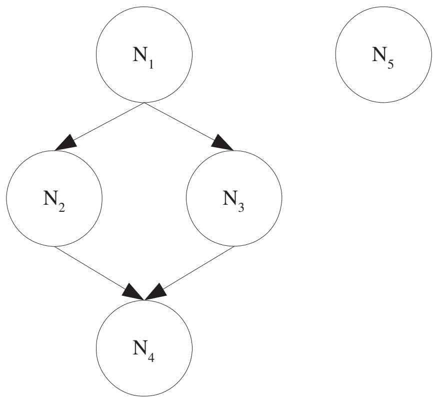

Figure 22.2: This "diamond" DAG illustrates parent and child links. N_1 must complete successfully, then both N_2 and N_3 can execute in parallel. Only when both of them have finished successfully can N_4 begin execution. N_5 is a disconnected node and can execute in parallel with all of the other nodes.

While running, DAGMan keeps a list in its memory of all jobs in the DAG, their parent/child relationships, and their current status. Given this

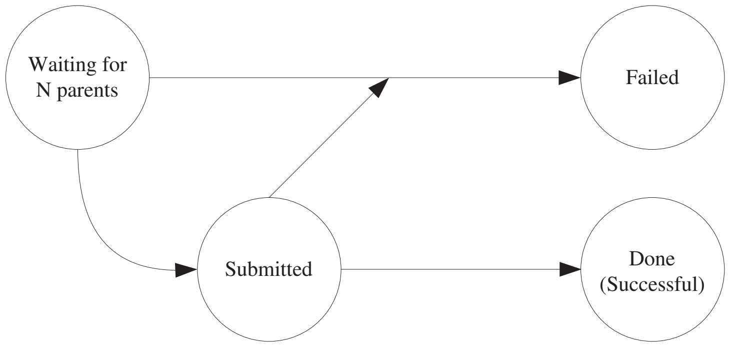

Figure 22.3: The state transition diagram for each node of a DAG. See the text for details.

information, DAGMan submits jobs to the batch system when appropriate, and continues until either the DAG is complete or no more forward progress can be made due to failed jobs. In the latter case, DAGMan creates a list of failed jobs along with the reasons for their failure and produces a *rescue DAG* file.

A rescue DAG is a special DAG that represents the state of a previously partially completed DAG such that the original DAG can be restarted where it left off without repeating any successfully completed work. The rescue DAG is an exact copy of the original input DAG, except that all previous nodes that have successfully completed are marked as done.

When DAGMan is restarted with a rescue DAG, it reconstructs the state of the previous DAG. Internally, DAGMan keeps track of the current status of each node. Figure 22.3 shows the basic state diagram of a DAG node. Note that this is simplified, and we will expand the diagram in the rest of this chapter.

When DAGMan starts, it marks each node as "waiting" and initializes a waiting count (N) for the node equal to its number of parents. In the case of a rescue DAG, DAGMan sets the waiting count equal to the number of parents that are not already marked as "done."

A node's waiting count represents the number of its parents that have yet to complete successfully and are therefore preventing it from being submitted. Only when a node's waiting count reaches zero can DAGMan submit the job associated with the node. If the job is submitted successfully, DAGMan marks the node as "submitted." If the job submission fails for any reason, the node is marked as "failed."

When DAGMan detects that a job has left the batch system queue, it marks the node as "done" if the job exited successfully, otherwise it marks the node as "failed." Success is determined by the exit code of the program:

If it is zero, then the job exited successfully, otherwise it failed. (But see the description of post scripts in Section 22.3.2 for a modification of this.)

When a job is marked “done,” the waiting count of all its children is decremented by one. Any node whose waiting count reaches zero is submitted to the batch scheduler as described earlier.

22.3.2 DAGMan Complications

So far, the description of a DAGMan DAG is not very interesting: We execute jobs and maintain the order in which they must execute, while allowing parallelism when it is possible. Unfortunately, this is insufficient in real environments, which have many complications and sources of errors.

Complication: Setup, Cleanup, or Interpretation of a Node

The first complication occurs when using executables that are not easily modified to run in a distributed computing environment and therefore need a setup or cleanup step to occur before or after the job. For example, before a job is run, data may need to be staged from a tape archive or uncompressed. While this step could be placed in a separate DAG node, this may cause unnecessary overhead because the DAG node will be submitted and scheduled as a separate job by the batch system instead of running immediately on the local computer.

DAGMan provides the ability to run a program before a job is submitted (a *pre script*) or after a job completes (a *post script*). These programs should be lightweight because they are executed on the same computer from which the DAG was submitted and a large DAG may execute many of these scripts. (But see the discussion on throttling for our solution to preventing too many of these scripts from running simultaneously.)

Running these scripts adds complexity to the state diagram in Figure 22.3. The changes needed to support scripts are shown in Figure 22.4. Once a job is allowed to run, it can optionally run a pre script. After the job has run, it can optionally run a post script.

Note that if the scripts fail, the node is considered to have failed, just as if the job itself had failed. There is one interesting case to note that is not easily represented in Figure 22.4: If a node has a post script, it will never go directly into the failed state but will always run the post script. In this way, the post script can decide if a job has really failed or not. It can do some analysis beyond DAGMan’s ability to decide if a node should be considered to have succeeded or failed based on whether the exit code is zero or not, perhaps by examining the output of the job or by accepting alternate exit codes. This significantly enhances the ability of DAGMan to work with existing code.

Some users have discovered an interesting way to use pre scripts. They create pre scripts that rewrite the node’s job description file to change how the job runs. This can be used for at least two purposes. First, it can create

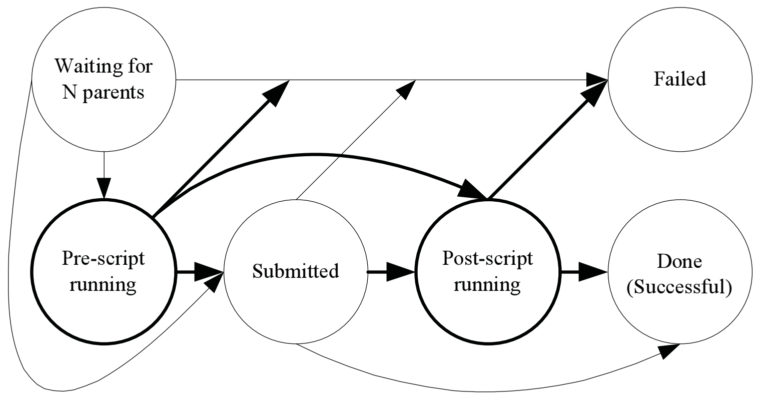

Figure 22.4: A state diagram for executing a single DAG node. Unlike Figure 22.3, this diagram adds the ability to run pre scripts and post scripts. The differences from Figure 22.3 are noted in bold.

conditional DAGs by allowing a runtime decision that changes the DAG. For example, consider Figure 22.5. If N_1 succeeds, then the prescript for N_3 will rewrite N_3 to an empty job—perhaps running the /bin/true command.[1] In this way, only N_2 will run after N_1 succeeds. Similarly, if N_1 fails, then only N_3 will run. While a more generic facility for conditional DAGs may be desirable, it would add complexity, and simple conditional DAGs can be created in this way.

A second use for pre scripts is to do last-minute planning. For example, when submitting jobs to Condor-G (which allows jobs to be submitted to remote grid sites instead of the local batch system), users can specify exactly the Grid site at which they wish their jobs to run. A pre script can decide what Grid site should be used, rewrite the job description, and the job will run there.

Complication: Throttling

All of the mechanisms described so far work very well. Unfortunately, the real world applies additional constraints. Imagine a DAG that can have one thousand jobs running simultaneously, and each of them has a pre script and a post script. When DAGMan can submit the jobs, it will start up one thousand nearly simultaneous pre scripts and then submit one thousand jobs nearly simultaneously. Running that many pre scripts may cause an unacceptable load on the submission machine, and submitting that many jobs to the underlying batch submission system may also strain its capacity.

[1] In recent versions of Condor, the job can be edited to contain "*noop_job* = *true*" which immediately terminates the job successfully.

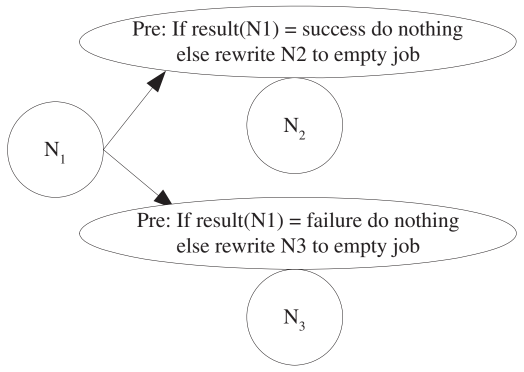
Pre: If result(N1) = success do nothing
else rewrite N2 to empty job
N_1
N_2
Pre: If result(N1) = failure do nothing
else rewrite N3 to empty job
N_3

Figure 22.5: An example conditional DAG.

For this reason, DAGMan can throttle the number of pre scripts, jobs, or post scripts that may run at any time. This results in another modification to our state diagram for running a single node, as shown in Figure 22.6. For example, a node will not leave the pre script ready state unless there are fewer pre scripts running than the throttle allows.

DAGMan can also throttle the number of jobs that it submits to the batch system to avoid submitting more jobs than can be handled by the batch system. This is a good example of a surprising additional constraint: We did not realize that DAGs might be able to submit so many jobs that the number of idle jobs could overwhelm the batch system.

Complication: Unreliable Applications or Subsystems

Some applications are not robust—it is not uncommon to find a program that sometimes fails to run on the first attempt but completes successfully if given another chance. Sometimes it is due to a program error and sometimes due to interactions with the environment, such as a flaky networked file system. Ideally, problems such as this would always be fixed before trying to run the program. Unfortunately, this is not always possible, perhaps because the program is closed-source or because of time constraints.

To cope with unreliable programs or environments, DAGMan provides the ability to retry a node if it fails. Users specify how many times the node should be retried before deciding that it has actually failed. When a node is retried, the node's pre script is also run again. In some cases, a user wants to retry multiple times unless some catastrophic error occurs. DAGMan handles this with the "retry unless-exit" feature, which will retry a job unless it exits with a particular value. One place this might be useful is planning. Imagine a pre script that decides where a job should be run. Retry might be set to 10 to allow the job to be run at ten different sites, but if there is some catastrophic

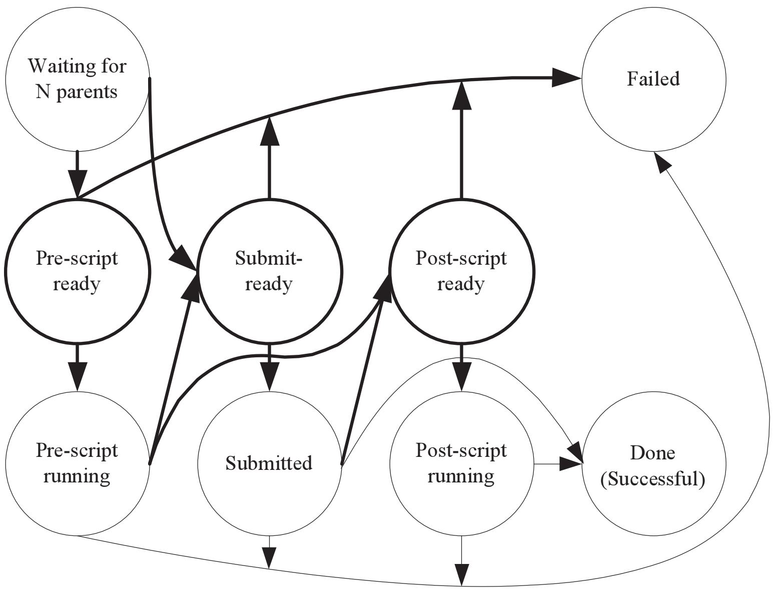

Figure 22.6: A state diagram for executing a single DAG node. In addition to the state in Figure 22.4, this diagram adds DAGMan's ability to throttle pre scripts, jobs, and post scripts. The differences from Figure 22.4 are noted in bold.

error, then the pre script can exit with a specific value that indicates "do not retry." Adding the ability to retry the job results in one final change to our state diagram, as shown in Figure 22.7.

22.3.3 Additional DAGMan Details

Submission Failures

When submitting a job to the underlying batch system, sometimes the job submission itself (not the job execution) will fail for reasons that are not the fault of the user. Usually, this is due to heavy use of the batch system, and it becomes temporarily unavailable to accept submissions. DAGMan will retry the job submission up to six times (by default, but this can be changed), increasing the amount of time between job submissions exponentially in order to increase the chance of a successful submission. If the job continues to fail to submit successfully, Condor will mark the job submission as failed.

Running DAGMan Robustly

What happens if the machine on which DAGMan is running crashes? Although DAGMan would no longer continue to submit jobs, existing jobs

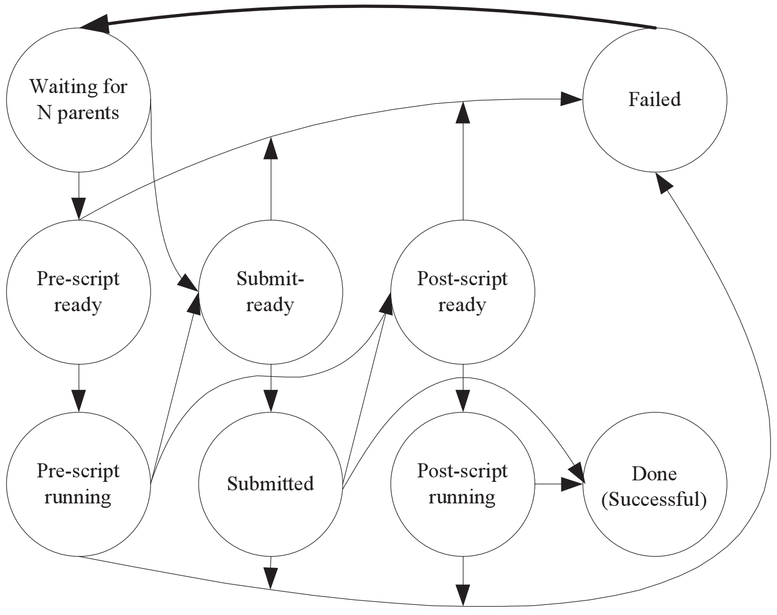

Figure 22.7: The complete state diagram for executing a single DAG node. The single difference from Figure 22.6 is noted in bold.

continue running, but it would be nice if DAGMan could be restarted so that it could continue to make forward progress. Ideally, DAGMan should handle as much as possible for the user, so we run DAGMan as a Condor job. This means that if the machine crashes, when it restarts Condor will restart the DAGMan process, which will recover the state of its execution from persistent log files, and will resume operation. This sort of robustness is essential in allowing users to run large sets of jobs in a "hands-off" fashion.

Recursive DAGs

A DAG node can submit any valid jobs, including submitting another DAG. This allows the creation of DAGs with conditional branches in them. The DAG node can make a choice, then submit an independent DAG based on the result of that choice. This can allow very complex DAGs to be executed. Unfortunately, it also makes it harder to debug a DAG. For an alternative to recursive DAGs, see Section 22.7.

22.3.4 Describing a DAG

It is the user's responsibility to provide DAGMan with a description of each job in the format of the underlying batch scheduler. For Condor, this means associating each node with a "submit file" describing the job to be executed. DAGMan ultimately uses this file to submit the job to the batch scheduler using the standard submission interface.

Users describe a DAG by listing each node and the relationships between nodes. A sample DAG description is shown in Figure 22.8.

```
Job N1 submit-n1
Job N2 submid-n2
Job N3 submid-n3
Job N4 submid-n4
Job N5 submid-n5

Parent N1    Child N2 N3
Parent N2 N3 Child N4

Retry N1 5

Script PRE  N5 uncompress-data
Script POST N5 uncompress-data
```

Figure 22.8: How a user might describe the diamond DAG from Figure 22.2. In this description, node N1 can be retried five times, and none of the other nodes are retried if they fail. Node N5 has both a pre script and a post script

22.3.5 DAGMan Experience

DAGMan has been used extensively with the Condor batch job scheduling system. We have found that our implementation of the DAGMan easily scales to large DAGs of around 1000 nodes without throttling and DAGs of around 100,000 nodes with throttling. We believe it could scale much further than that if necessary. Because DAGMan can manage DAGs of this scale and because we find that the greatest bottleneck is in the underlying batch job submission system capabilities, we have not expended effort to optimize it to work with DAGs larger than 100,000 nodes.

DAGMan has been used in a wide variety of production environments. We will provide two examples here.

Within the Condor Project, we have created a Basic Local Alignment Search Tool (BLAST) [53] analysis service for the Biological Magnetic Resonance Data Bank at the University of Wisconsin–Madison [55]. BLAST finds regions of local similarity between nucleotide or protein sequences. Local researchers do weekly queries against databases that are updated every week. Our service takes a list of sequences to query and creates a pair of DAGs to perform the queries, as illustrated in Figure 22.9. The first DAG performs the setup, creates a second DAG that does the queries (the number of nodes in this DAG varies, so it is dynamically created), and then assembles the results. These DAGs are used differently: The first DAG uses dependencies to order the jobs that are run, while the second DAG has completely independent nodes, and DAGMan is used for reliable execution and throttling. On average, the second DAG has approximately 1000 nodes, but we have experimented with as many as 200,000 nodes. This service has run on a weekly basis for more than two years with little human supervision.

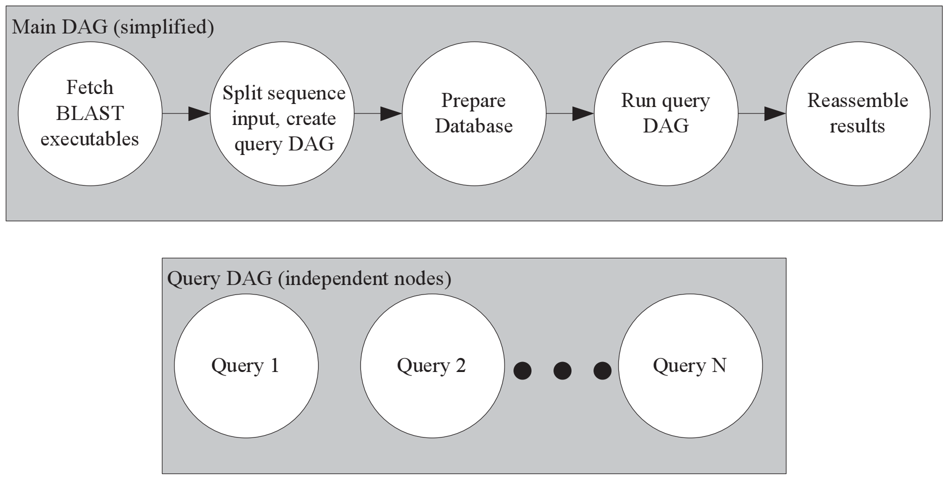

Figure 22.9: The pair of DAGs used to run BLAST jobs. The query DAG is created by the main DAG. See the text for details.

The Virtual Data System (VDS) (see Chapter 23 or [148]) builds on top of DAGMan and Condor-G. Users provide a description of what data are available and how the data can be transformed, then request the data they need. The VDS creates a DAG that fetches and transforms data as needed, while tracking the provenance of the data. As part of the DAG creation and execution, the VDS uses planning to decide which Grid sites should perform the transformations. The VDS has been used for a wide variety of applications, including high-energy physics event simulation, finding galaxy clusters, and genome analysis.

22.4 Implementation Status

DAGMan has been freely distributed as part of the Condor software since 1999. It has been used for numerous large projects and is stable. It is available for a wide variety of Unix platforms, Microsoft Windows, and Mac OS X.

22.5 Interaction with Condor

Condor is a high-throughput batch job scheduler. Because it has been covered in detail elsewhere ([262], [414]), we only briefly review it here.

Condor was originally designed to utilize CPU cycles on computers that would otherwise be idle, such as desktop computers that are unused but turned on overnight. However, Condor has expanded its scope and now works well with dedicated computers and Grid systems. Condor's ability to interact with a Grid system called Condor-G [151] allows Condor to submit jobs to Globus [166] (versions 2, 3, and 4), NorduGrid, Oracle, LSF, PBS, and even remote Condor installations (referred to as Condor-C).

Condor and Condor-G emphasize reliability. If Condor crashes, it will continue running the jobs when it restarts. Condor can provide job checkpointing and migration to facilitate recovery when execution computers fail. Condor-G provides elaborate recovery schemes to deal with network outages and remote Grid site failures.

DAGMan is built to use Condor for job execution, and it can submit jobs to both the local batch system and remote Grid systems with equal ease. We have created many workflows using DAGMan that execute in a Grid environment.

22.6 Integration with Stork

22.6.1 An Introduction to Stork

Just as computation and network resources need to be carefully scheduled and managed, the scheduling of data placement activities across distributed computing systems is crucial, as the access to data is generally the main bottleneck for data-intensive applications. This is especially the case when accessing very large data stores using mechanical tape storage systems.

The most common approaches to solving the problems of data placement are either interactive data movement or the creation of simple scripts. Generally, scripts cannot adapt to a dynamically changing distributed computing environment: They do not have the privileges of a job, they do not get scheduled, and they do not have any automation or fault-tolerance capabilities. Furthermore, most scripts typically require close monitoring throughout the life of the process and frequent manual intervention.

Data placement activities must be first-class citizens in the distributed computing environments, just like computational jobs. Data placement jobs need to be queued, scheduled, monitored, and even checkpointed. Most importantly, users require that their data placement jobs omplete successfully without human monitoring and intervention.

Furthermore, data placement jobs should be treated differently from computational jobs, as they have different semantics and different characteristics. For example, if the transfer of a large file fails, we may not simply want to restart the job and retransfer the whole file. Rather, we may prefer to transfer only the remaining part of the file. Similarly, if a transfer using one protocol fails, we may want to try alternate protocols supported by the source and destination hosts to perform the transfer. We may want to dynamically tune network parameters or decide concurrency levels for unique combinations of the data source, destination, and protocol. A traditional computational job scheduler does not handle these cases. For this reason, data placement jobs and computational jobs should be differentiated, and each should be submitted to specialized schedulers that understand their semantics.

We have designed and implemented the first batch scheduler specialized for data placement: Stork [244]. This scheduler implements techniques specific to queuing, scheduling, and optimization of data placement jobs and provides a level of abstraction between the user applications and the underlying data transfer and storage resources.

A production-quality Stork is bundled with Condor releases. Additionally, research into new features is continuing in parallel.

22.6.2 Data Placement Job Types

Under Stork, data placement jobs are categorized into the following three types:

- *Transfer.* This job type is for transferring a complete or partial file from one physical location to another. This can include a get or put operation or a third-party transfer. Stork supports a variety of data transfer protocols and storage systems, including local file system, GridFTP, FTP, HTTP, NeST, SRB, dCache, CASTOR, and UniTree. Furthermore, sites can create new transfer modules using the Stork modular API.
- *Allocate.* This job type is used for allocating storage space at the destination site, allocating network bandwidth, or establishing a light path on the route from source to destination. It deals with all resource allocations required for the placement of the data.
- *Release.* This job type is used for releasing the corresponding allocated resource.

22.6.3 Flexible Job Representation

Stork uses the ClassAd [367] job description language to represent the data placement jobs. The ClassAd language provides a very flexible and extensible data model that can be used to represent arbitrary services and constraints.

Below are three sample data placement (DaP) requests:

```
[
  dap_type      = "allocate";
  dest_host     = "houdini.example.com";
  size          = "200MB";
  duration      = "60 minutes";
  allocation_id = 1;
]

[
  dap_type = "transfer";
  src_url  = "file:///data/example.dat";
  dest_url = "nest://houdini.example.com/data/example.dat";
]

[
  dap_type      = "release";
  dest_host     = "houdini.example.com";
  allocation_id = 1;
]
```

The first request is to allocate 200 MB of disk space for 1 hour on a NeST server. The second request is to transfer a file from the local file system to the allocated space on the NeST server. The third request is to deallocate the previously allocated space.

22.6.4 Fault Tolerance

Data placement applications must operate in an imperfect environment. Data servers may be unavailable for many reasons. Remote and local networks may encounter outages or congestion. Computers may crash, including the Stork server host itself. Stork is equipped to deal with a variety of data placement faults, which can be configured at both the system and job levels.

For transient environment faults, data placement jobs that fail can be retried after a small delay. The number of retries allowed is configurable.

For longer-term faults associated with a particular data server, Stork can also retry a failed transfer using a list of alternate data protocols. If in the previous example the host houdini.example.com is also running a plain FTP server, the corresponding transfer job could be augmented to retry with plain FTP.

```
[
  dap_type      = "transfer";
  src_url       = "file:///data/example.dat";
  dest_url      = "nest://houdini.example.com/data/example.dat";
  alt_protocols = "file_ftp";
]
```

22.6.5 Interaction with DAGMan

Condor's DAGMan workflow manager has been extended to work with Stork. In addition to specifying computational jobs, data placement jobs can be specified, and DAGMan will submit them to Stork for execution. This allows straightforward execution of workflows that include data transfer.

A simple example of how Stork can be used with DAGMan appears in Figure 22.10. This DAG transfers data to a Grid site using Stork, executes the job at the Grid site using Condor-G, and then transfers the output data back to the submission site using Stork. This DAG could easily be enhanced to allow space allocation before the data transfers, or it could have multiple data transfers.

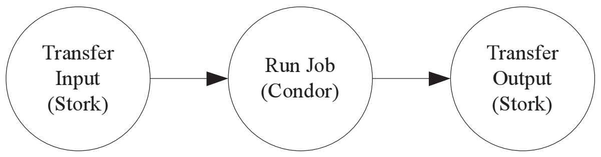

Figure 22.10: A simple DAG that includes Stork.

22.6.6 Interaction with Heterogeneous Resources

Stork acts like an I/O control system between the user applications and the underlying protocols and data storage servers. It provides complete modularity and extendibility. Users can add support for their favorite storage system, data transport protocol, or middleware very easily. This is a crucial feature in a system designed to work in a heterogeneous distributed environment. The users or applications cannot expect all storage systems to support the same interfaces to talk to each other. Furthermore, it is becoming increasingly difficult for applications to talk to all the different storage systems, protocols, and middleware. There needs to be a negotiating system between them that can interact with those systems easily and even perform protocol translation. Stork has been developed with these capabilities as requirements. The modularity of Stork allows users to insert a plug-in to support any storage system, protocol, or middleware easily.

Stork supports several data transfer protocols, including:

- FTP [362]
- GridFTP [9]
- HTTP [141]
- DiskRouter [242]

Stork supports several data storage systems, including:

- SRB [41]
- UniTree [71]

- NeST [45]
- dCache [109]
- CASTOR [82]

Stork maintains a library of plugable "data placement" modules. These modules are executed by data placement job requests. Modules can perform interprotocol translations using either a memory buffer or third-party transfers whenever available (such as GridFTP). For example, if a user requests a data transfer between two remote systems, Stork can often perform the transfer without saving the file to disk. Interprotocol translations are not yet supported between all systems or protocols, but they are available for the major use cases we have encountered so far.

In order to transfer data between systems for which direct interprotocol translation is not supported, two consecutive Stork jobs can be used instead. The first Stork job performs the transfer from the source storage system to the local disk cache of Stork, and the second Stork job performs the transfer from the local disk cache of Stork to the destination storage system.

22.6.7 Modular API

While the Stork server is a single process, the data transfers, allocations, etc., are performed by the separate *modules*. The module application program interface is simple enough for sites to write their own modules as needed. For example, each data transfer module is executed with the following argument list:

src_url dest_url arguments ...

Thus, to write a new module that transfers data from the *foo* protocol to the *bar* protocol, a new module is created with the name `stork.transfer.foo-bar`. Modules need only be executable programs, and may even be written as shell scripts. Furthermore, module binding is performed at runtime, enabling sites to create new modules without restarting the Stork server.

22.6.8 Performance Enhancements

Stork has seen several recent performance enhancements. The first is that multiple data placements may now be specified in a single submission file instead of multiple files. This optimization is significant when transferring many files to Stork because it eliminates extra invocations of the *stork_submit* command, which can be surprisingly time consuming when transferring tens of thousands of files.

The second enhancement was integration of the GridFTP client *globus-url-copy* into Stork server. When doing many simultaneous GridFTP file transfers, this saves considerable time and reduces the total number of processes in the system.

Finally, Stork is now able to execute an arbitrary program when the active job queue size falls below a configurable level. This is envisioned as a simple but high-performance alternative to managing very large data placement workflows with DAGMan: It will allow Stork users to limit the rate at which they submit jobs so that Stork is not overwhelmed, while ensuring that Stork has sufficient work to do at any given time.

22.6.9 Implementation Status

Stork is available, with all features described so far, as part of the Condor distribution. It is available on Linux and will be available for other platforms in the future. Users outside of the Condor Project are just beginning to use Stork in production, and we hope to have more in the near future.

22.6.10 Active Research

Research on Stork is active, and much of it can be found in [243]. Research includes:

- Extending data placement types to include interaction with a metadata catalog.
- Experimentation with scheduling techniques other than first-in, first-out. This includes not only traditional scheduling techniques such as shortest job first or multilevel queue priority scheduling but also scheduling based on management of storage space to ensure that storage space is not exceeded by the data transfers. In addition, scheduling of connection management is important when there are many simultaneous connections to a server.
- Runtime adaptation can be performed to tune the network parameters for a transfer to minimize transfer time. This is discussed in further detail in [241].
- Research has been done to enable Stork to detect problems such as servers that are unreliable in a variety of ways and then base scheduling decisions on this knowledge. More details are in [240].

22.7 Future Directions

There are several promising areas for future work with DAGMan and Stork. For DAGMan, we would like to explore methods (probably utilizing ClassAds [367]) to allow different conditions for deciding when a node should execute. Today, a node executes when all of its parents have finished with an exit code of 0, but allowing more complex conditions would allow conditional execution (equivalent to if-then-else) and partial execution (a DAG that finishes when a certain percentage of nodes have completed).

We also would like to support dymamic DAGs, which are DAGs that can change on-the-fly, based on user input. This has been frequently requested by users and is particularly useful when DAGMan is used by a higher-level scheduling system that may change plans in reaction to current conditions.

For Stork, we are exploring ways to make it more scalable and more reliable. We are also investigating methods to use matchmaking, similar to that in Condor, to select which data transfers should be run and which sites they should transfer data to.

22.8 Conclusions

DAGMan is a reliable workflow management system. Although the workflows it supports are relatively simple, there are many complexities that were discovered as we used DAGMan through the years, such as the need for flexible methods for retrying and throttling. As a result of our experience, DAGMan has found favor with many users in production environments, and software has been created that relies on DAGMan for execution. Used with Condor, Condor-G, and Stork, DAGMan is a powerful tool for workflow execution in Grid environments.