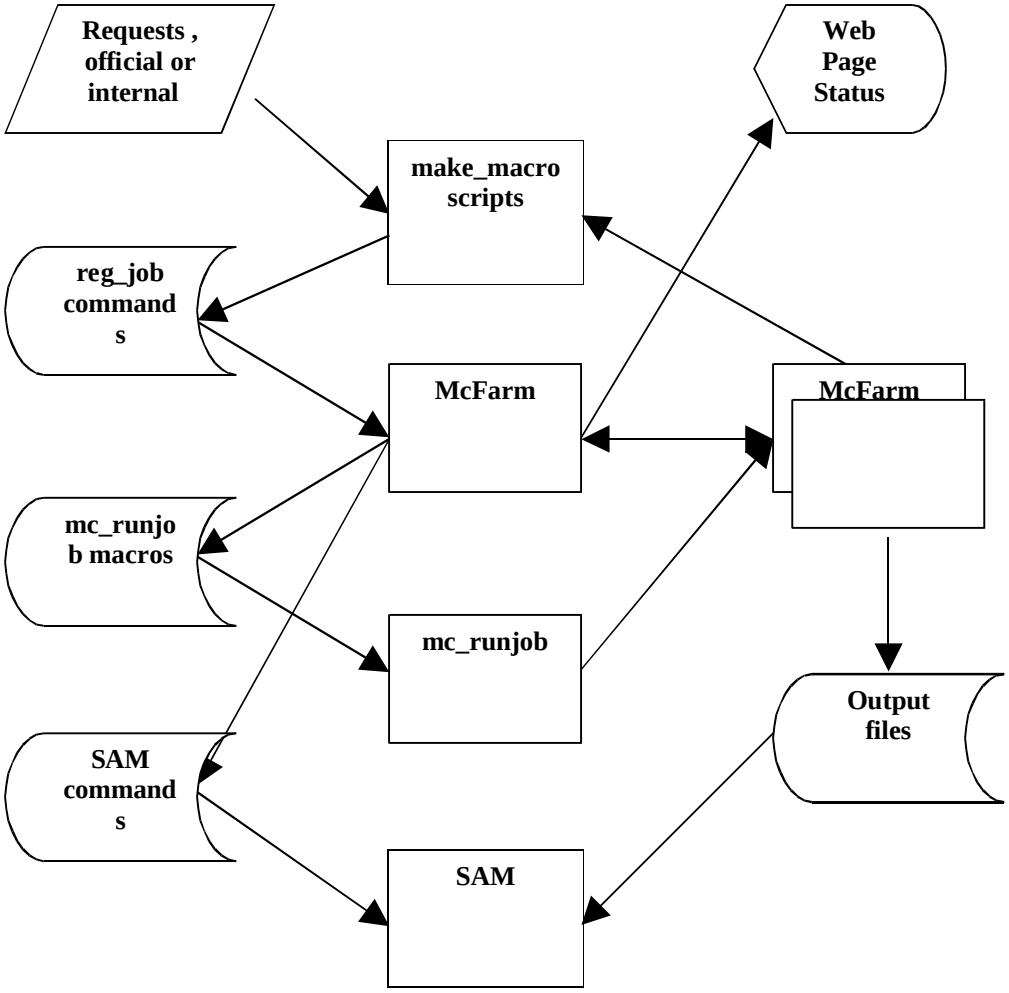
McFarm Guide

Version 9.0 April 2003

This document provides detail information about the operation of the McFarm software for controlling a production Monte Carlo farm. Other documents are available from d0race.fnal.gov to describe implementation and configuration. McFarm is integrated with other software as shown here:



Contents

- Operating hints and troubleshooting

FARM PREREQUISITES

A Monte Carlo farm is constructed of one or more loosely coupled Linux-based processing nodes. Nodes need not be homogeneous in terms of hardware, and can contain multiple CPUs so long as sufficient disk and memory is available for each cpu (approximately 128MB of memory and approximately 2-3GB of local disk space per cpu). With careful configuration the nodes can use different versions of Linux.

The farm software consists mainly of python scripts and modules, and was coded using an OO approach. No farm command or daemon requires the x-windows server (although an extra-wide window is helpful when using some commands). The farm's default shell in the present version must be bash. Many farm processes are multi-threaded, partly to improve performance when querying many nodes but also to provide robust operation in the face of a sudden failure of a node (in which case any hanging thread is simply terminated while mainstream processing continues). The software requires the use of NFS and SSH commands, and NIS/YP is also needed for NFS access without passwords.

An important locality assumption is made about the nature of farm processing tasks: each task can in general be performed on a production node using only local disk space and resources. This allows the farm to be scalable, and alleviates concerns about moving large amounts of data across the network. The current version of the farm runs tasks like pythia, d0gstar, d0sim, d0reco, and recoanalyze and this locality requirement is easily met for most trigger-input files by clever scheduling. However, a d0sim job that must access d0gstar files for minbias input can severely violate the locality assumption, thus raising scalability issues. Solutions include multiple file-servers, multiple NICs, multiple switches, and possibly redundant copies of data.

File caching is the practice of placing files from a previous job onto a **file server** for input to subsequent jobs. An example is the creation of fixed background events from d0gstar for input to later d0sim jobs. In this version of the McFarm software, file serving is performed as follows. One or more file servers can be defined in the farm and their cache directories can be accessed by subsequent jobs on any other node for input. Also, every node has its own local cache directory. The farmer can place files into these caches directly, or by specifying *cache* or *cachelocal* as a job *disposition*. Subsequent jobs that will read those cached files can be set to delete their input, so that the file is removed automatically from the farm cache when (all) jobs that need it are done.

If a significant amount of file caching is to be done, a potential network bottleneck at the file-server NIC card can limit the farm's scalability. Solutions include the same ideas as mentioned above for minbias input. Generally, d0sim operation is only about 20% of any job, so file-serving issues have minimal impact.

It is suggested that the nodes be connected via an Ethernet switch, preferably one that is capable of allowing multiple node-to-node conversations at speeds of 100 MBS. Operation is possible with less expensive network hardware, especially if the locality requirement is met. Exceptions to the locality constraint occur in two areas: gathering and file caching. Gathering is typically done by writing output files to SAM. NFS is used for the transfer of data from a production node to the SAM station node, and it should be tuned with large transfer blocksize to facilitate these large-file transfers. If multiple SAM nodes are required (or if file serving is a large part of daily operation), it becomes more important to have high-speed switch or switches.

Network host names

A node can have any nodename and still be attached to the farm. If you have a choice in the nodename, then make it as follows: First, decide upon a *name* for the network (e.g., hepfm) and then assign a hostname to each node of the form *name*NNN (e.g., hepfm000, hepfm001, etc.). The last 3 characters must be numeric and unique. If you cannot assign that name to a node, then (in the *distribute.conf* file) you must supply the actual nodename on the line where the node is introduced to the farm (see comments in that file).

MC Farm Account

A single non-root account, preferably **mcfarm**, must be established for all farm processing. NIS and slogin/ssh must be configured so that this account has free access (no password required) to any other node. The "ssh" command is used internally by the farm software.

The default shell for mcfarm must be **bash**, and it is suggested that the ".bashrc" file contain an automatic invocation of the farm script to define all farm variables (e.g., a line with ". ~/bin/setup_farm").

Typically, the home directory for mcfarm will contain all farm executables and most of the central control files, queues, and archives. Other configurations are possible so long as the directories are visible to production nodes (see NFS below).

Job Server

One of the nodes must be designated as the **job server**. Typically it is node 000 (e.g., hostname hepfm000). It can be a production node as well (if it has 2 CPUs), so it starts with the basic amount of disk and memory that every node needs. Above that it needs disk space to hold all the d0 software (to be shared by all nodes), plus 1-2 GB to hold archived jobs. This **server** will also be running the locking daemon, the distribute daemon, the monitor daemon, and (likely) SAM and cache gathering daemons, as well as all manual commands like jobstat. These tasks are mostly I/O intensive, so unless you plan to run non-farm functions on this server, it does not have to be the most powerful machine.

Gather Server

Once a job has created an output file, a gather daemon takes over to forward it to the desired disposition (cache, SAM, FTP, etc.). SAM station must be implemented one each node that is to do SAM gathering. On the job server, one generally does all these gathering operations:

- **cache** gathering (copying specific output files to local file servers, such as root files, for local consumption)
- **metadata** gathering (extracting and retaining only the metadata for a binary)
- **sammetadata** gathering (declaring parent file metadata to SAM even if they are not actually being sent to SAM).
- **merge** gathering (merging small files from a request then storing them)

Also, if you will be re-processing files that are presently stored in SAM, you will need an **acquire** daemon on one of the gather servers. It responds to the need for Monte Carlo input files, as specified in your reg_job commands, and automatically acquires them from SAM and places them into the cache.

A gather server operates as a pull process to gather one job's output at a time, and thus its NIC card is generally not a scalability bottleneck. Gather processes issue email to alert the farmer when problems arise.

File server

The current farm version allows for multiple file-servers to handle minbias data, local-consumption caching, job archiving (for farm statistics), and possibly for files acquired from SAM. McFarm will also use cache for D0 input files (such as *gen* files that need to be visible to all d0gstar jobs).

Each file-server is visible to the rest of the nodes via NFS through a link name, such as "/cachennn_A" (where nnn is the node number of the file-server, and "_A" means it is the first cache partition on that node). Farm software will automatically search all file-servers (and its own local cache) for input files using these links, so the farmer does not have to tell a job WHERE the input file will be, only its base name.

The use of a file-server violates the locality requirement of farm processing, so bandwidth bottlenecks at the file-server NIC card may have to be dealt with. Also, if a gather-server and a file-server are located on the same node, consideration has to be given to the sharing of the NIC card.

A file-server can have up to 26 cache directories, each on a different partition than its regular scratch directory. Each cache directory is uniquely named.
Note: It is suggested that the job-server always be given all the access of a file-server as well. This is because the distribute daemon must be allowed to move cached files around if necessary.

Simple Production Node

If a node is neither a job server, gather server, nor file server, then it is a simple production node. Job servers and gather servers can also do job processing if cpu/memory/disk resources permit, but file servers probably should not. A simple production node runs only one daemon (the execute daemon), and one processing job per cpu. A job's output files are always local, and its input files (if any) are located on a local cache or a file-server. When a job is completed, it self-checks by running a final audit for acceptability (e.g., all events handled) and places itself into either a *gather queue* or an error queue (with email notification). In either case, the node is done with a job when the last job phase has finished writing its output file. Some gathering daemon will take up the task of pulling the output to SAM (etc.), and the farmer must handle any error jobs.

NFS implementation

Farm software, control files, caches, and scratch areas are shared among nodes using NFS. The job server contains the executables and control files, and thus must have a sharable partition for them (e.g., /home). Similarly each file-server must share its cache (e.g., /cachennn_A). Each simple production node must share its scratch (/scratch) with the job-server and with each gather server, and (for faster gathering) a special NFS link to the gather-queue is recommended (e.g., /gathernnn). Conversely, the job-server and each gather-server must "see" every simple node's scratch area, cache, and gather-queue, and each file-server's cache.

In McFarm, the mounting of disks can be automated and linked to the start_farm command itself, but this requires the implementation of a "root daemon" (since mounting is a root-level operation), It also involves a

set of additional scripts (attach_node, detach_node, attach_nodes_to_js, etc.) that must (in this version) be separately maintained when nodes are added to the farm. Instructions for this auto-mounting is included in the farm implementation guide.

NFS Example

Here is an example for a 3-node network where hepfm000 is the job-server (and file-server), hepfm001 is a combination gather-server and file-server, and hepfm002 is a simple production node.

Job-server (hepfm000) "/etc/exports" file, to allow visibility of the main farm partition to all other nodes, the local scratch partition to the gather-server, and the local cache partition to all other nodes:

```
/home       hepfm001(rw,no-root-squash)
/home       hepfm002(rw,no-root-squash)
/scratch    hepfm001(rw,no-root-squash)
/cache000_A hepfm001(rw,no-root-squash)
/cache000_A hepfm002(rw,no-root-squash)
```

File/Gather-server (hepfm001) "/etc/exports" file, to allow visibility of the local scratch partition to the job-server, and the local cache partition to all other nodes:

```
/scratch    hepfm000(rw,no-root-squash)
/cache001_A hepfm000(rw,no-root-squash)
/cache001_A hepfm002(rw,no-root-squash)
```

Simple production node (hepfm002) "/etc/exports" file, to allow visibility of the local scratch partition to the job-server and to the gather-server, and the local cache partition to the file-servers:

```
/scratch    hepfm000(rw,no-root-squash)
/scratch    hepfm001(rw,no-root-squash)
/cache000_A hepfm000(rw,no-root-squash)
/cache002_A hepfm001(rw,no-root-squash)
```

Job-server (hepfm000) "/etc/fstab" file, to mount all other node's scratch partitions (because its a job-server) and cache-partitions (because its a file-server). (These auto-mounts are NOT present in the /etc/fstab file if the root daemon is implement to do the mounts later):

```
hepfm001:/scratch     /mnt/hepfm001/scratch nfs rw,intr,rsize=8192,wsize=8192,actimeo=0
hepfm002:/scratch     /mnt/hepfm002/scratch nfs rw,intr,rsize=8192,wsize=8192,actimeo=0
hepfm001:/cache001_A /mnt/hepfm001/cache_A nfs rw,intr,rsize=8192,wsize=8192,actimeo=5
hepfm002:/cache002_A /mnt/hepfm002/cache_A nfs rw,intr,rsize=8192,wsize=8192,actimeo=5
hepfm001:/gather001 /mnt/hepfm001/gath_queue nfs rw,intr,rsize=8192,wsize=8192,actimeo=5
hepfm002:/gather002 /mnt/hepfm002/gath_queue nfs rw,intr,rsize=8192,wsize=8192,actimeo=5
```

(Note that the "actimeo" parameter must be 0 for access to the scratch areas, so that changes to directories can be seen immediately. This parameter can be non-zero for faster NFS processing on cache and gather activities).

To complete the hepfm000 example, one must create the “/mnt” directories referenced above, and then one must create a link to those mount directories. The names of these links must be:

/scr001 -> /mnt/hepfm001/scratch
/scr002 -> /mnt/hepfm002/scratch
/cache001_A -> /mnt/hepfm001/cache_A
/cache002_A -> /mnt/hepfm002/cache_A
/gather001 -> /mnt/hepfm001/gath_queue
/gather002 -> /mnt/hepfm002/gath_queue

(where “/scr”, “/cache”, and “/gather” are farm variables). Also, there must be a link to local scratch and cache for standardization purposes:

/scr000 -> /scratch
/cache000_A -> /scratch/cache_A (or diff mount name)

File/Gather-server (hepfm001) “/etc/fstab” file, to mount the job-server’s home partition and all other node’s scratch partitions (because its a gather-server) and cache partitions (because its a file-server). (Note: only the/home paretition is automounted if the root-dameon is implemented to do the mounts later):

hepfm000:/home /mnt/hepfm000/home nfs rw,intr,rsize=8192,wsize=8192,actimeo=0
hepfm000:/scratch /mnt/hepfm000/scratch nfs rw,intr,rsize=8192,wsize=8192,actimeo=0
hepfm002:/scratch /mnt/hepfm002/scratch nfs rw,intr,rsize=8192,wsize=8192,actimeo=0
hepfm000:/cache000_A /mnt/hepfm000/cache_A nfs rw,intr,rsize=8192,wsize=8192,actimeo=5
hepfm002:/cache002_A /mnt/hepfm002/cache_A nfs rw,intr,rsize=8192,wsize=8192,actimeo=5
hepfm000:/gather000 /mnt/hepfm000/gath_queue nfs rw,intr,rsize=8192,wsize=8192,actimeo=5
hepfm002:/gather002 /mnt/hepfm002/gath_queue nfs rw,intr,rsize=8192,wsize=8192,actimeo=5

To complete the hepfm001 example, one must create the “/mnt” directories referenced above, and then one must create a link to those mount directories. The names of these links must be:

/home -> /mnt/hepfm000/home
/scr000 -> /mnt/hepfm000/scratch
/scr002 -> /mnt/hepfm002/scratch
/cache000_A -> /mnt/hepfm000/cache_A
/cache002_A -> /mnt/hepfm002/cache_A
/gather000 -> /mnt/hepfm000/gath_queue
/gather002 -> /mnt/hepfm002/gath_queue

Also, there must be a link to local cache for standardization purposes:

/cache001_A->/scratch/data/cache_A

Simple node (hepfm002) “/etc/fstab” file, to mount the job-server’s home partition and both file-server’s cache partition:

hepfm000:/home /mnt/hepfm000/home nfs rw,intr,rsize=8192,wsize=8192,actimeo=0
hepfm000:/cache000_A /mnt/hepfm000/cache_A nfs rw,intr,rsize=8192,wsize=8192,actimeo=5
hepfm001:/cache001_A /mnt/hepfm001/cache_A nfs rw,intr,rsize=8192,wsize=8192,actimeo=5

To complete the hepfm002 example, one must create the "/mnt" directories referenced above, and then one must create a link to those mount directories. The names of these links must be:

/home -> /mnt/hepfm000/home
/cache000_A -> /mnt/hepfm000/cache_A
/cache001_A -> /mnt/hepfm001/cache_A

Also, there must be a link to local cache for standardization purposes:

/cache002_A -> /scratch/data/cache_A

Also, if you choose to place segments of the archive queue on nodes other than the server, you must create a mounted link to each such segment so the job-server can see them:

On the node, create an archive directory and a link such as:

/cache001_B/archive/jobs
ln -s /cache001_B /archive001_B

and make sure that that partition is exported, then on the job server create a mount point and link:

/mnt/hepfm001/archive_B
ln -s /mnt/hepfm001/archive_B /archive001_B/archive/jobs

and mount it in /etc/fstab using acttime=0

Describe the link in setup_farm:

export FARM_ARCHIVE=/archive
export FARM_ARCHIVE_JOBS_QUEUE_01=$FARM_ARCHIVE'001_B/archive/jobs'
export FARM_ARCHIVE_MINIMUM_01=1000 # leave 1GB

To summarize, the job-server and the gather-server can now access all scratch areas via:

/scr000
/scr001
/scr002

and also access all the gather-queue's directly (quickly) via:

/gather000
/gather001
/gather002

and (if additional archive segments):

/archive001_B/archive/jobs

Every node can see the job-server's executables and the file-servers' caches, and its own scratch and cache:

/home
/cache000_A
/cache001_A
/scratch
/cachennn_A

When the network is booted up, the suggested sequence is: boot the job-server first since its /home directory is auto-mounted by the production nodes and thus must be ready when they boot. Boot any file-servers next, since their disks are also auto-mounted by the production nodes. Boot each slave node last after giving all servers time to boot. Then, when all nodes are up, issue the "start_farm" command which will use the root-daemon to issue all the mounts that were NOT auto-mounted in an /etc/fstab file. (Note: if you do not implement the root daemon, then at this point you will have to issue the mounts yourself as root from the job server and all gather servers. Then and only then do you issue the start_farm command).

To make dynamic changes to the export and fstab files, you can make use of the "mount -a" command (as root) in order to re-load anything mentioned in a node's /etc/fstab file. You might have to "umount" something first. If you make a change to a node's /etc/exports files, you can refresh the export daemon by doing "/usr/sbin/exportfs -ra" as root.

To verify that NFS and NIS are all properly working, one can do the following manual commands when logged in as mcfarm:

- From the job-server and from each gather-server, one should be able to issue:

ls /scrnnn (nnn = 000, 001, ... for every node)
ls /gathernnn (nnn = 000, 001, ... for every node)
ls /cachennn_A (nnn = 000, 001, ... for every node, or _B etc.)

and "see" every node's scratch, gather, and cache directories. If you cannot, there is an NSF problem. (If the node is "down", omit this test, since the farm will operate robustly even in the face of a downed node). Also issue:

ssh hepfmnnn ps (nnn = 000, 001, ... for every node EXCEPT itself).

One should see the results of a "ps" command as run on the target node, WITHOUT having to type in a password. The McFarm software uses "ssh" extensively. If there is a problem, review NIS and SSH implementation.

2. Login to every node as mcfarm and issue:

ls /home

One should see the /home directory of the job-server. Farm processing is “semi-robust” in the face of a downed job-server. Jobs that are presently in a processing phase will continue until that phase ends, then they will wait for the server to be up before starting the next phase (due to the need to read the next executable from the job-server).

- Also, from every node issue:

ls /cachennn_A (nnn = any file-server node, A or B etc)

One should see the cache directory of each file-server. Naturally, a downed file-server prevents the execution of any job that needs an input file from it.

LIFE CYCLE OF A JOB

A typical **job** in McFarm has the following stages in its *life cycle*.

1. The farmer uses reg_job (indirectly through the Bookkeeper software, or more directly though the Request interface) to create pone or more jobs from the request, and define their phases.

   As described in the section on reg_job, this step defines a job and its phases and parameters, and registers a job into the farm system. The job's status is now **ready to distribute**.

2. The distribute daemon gives the job to an available processor.

   As described in the section on the distribute daemon, this step will dispatch a job as a unit to some production node, based on priorities as defined when the job was registered, its order in the gather.conf file and the contents of the dist_queue. At most one job per cpu is sent to a node by Mcfarm. A job is not dispatched until its initial input file, if any, is present in some cache (and it will dispatch to a specific node if the job is present in a *local* cache). (If necessary, the acquisition of this input file from SAM will be done by the acquire daemon automatically). The job's status is now **ready to execute**.

3. The execute daemon on the target node launches the job.

   As described in the sections on the execute daemon and on the *process_run* script, the job is tailored to fit that node, input files, and event-constraints and then spawned as a child process on that node. The job's status is now **in process** with a phase-status of "initiating". If a batch queue is in place on that node, the job is submitted to it instead of being spawned as a child task (see detail info later in this document).

4. The job is processed to completion using *process_run* script.

   Each phase is run, with error checking. All output is written locally. The job's status is now **in process** with a phase-status of "generate", "d0gstar", etc.

5a. The job completes all phases successfully.

   If it passes all audits from all phases, process_run ends by moving the job to the gather queue on that target node. Its status is now **ready to gather**, and this node is now ready to accept a new job.

5b. The job fails due to a D0-software problem.

   A job can fail for a variety of reasons, mostly thought to be due to latent bugs in the executables themselves. These failures are generally detected by the process_run audit logic, and the job is terminated and a suitable email is sent. (Looping errors are detected by the farm monitor - and the farmer is alerted to abort the job). Its status is now **errored out**, and this node is now ready to accept a new job.

5c. The job fails due to an McFarm script problem or looping problem.

If the process_run script itself fails, or if the binary loops, the job cannot detect its own failure. The execute-daemon on each node checks its running jobs right after initiation, and again every 30 minutes, for such failures and errors-out the job with a suitable email. Its status is now **errored out**, and this node is now ready to accept a new job.

5d. The job fails because the target node goes down.

The farmer will be alerted to the fact that a node is off-line (or that it is on-line but a job has stalled) via an email from the farm monitor (checked every 4 hours). Previously running jobs will now be considered "in process" but "stalled". The farmer will have to take actions such as restarting the execute daemon, and performing the command "check_job —fix" on the aborted jobs. Previous output will be lost. The job may be distributed to a different node when next distributed.

5e. Error jobs are killed or restarted by the farmer.

Generally, a job that looped or seg-faulted has to be discarded entirely. If it is thought that the job can run if tried again, then it can be restarted. The farmer can use "check_job —kill" or "check_job —fix" for these actions. If fixed, the job's status may again be "ready to distribute". If it had completed some output and metadata in its last operation, it may be altered to resume at the NEXT binary instead of at the very beginning.

- Each output is gathered to the designated disposition, and metadata is saved.

Individually, each gather daemon will be signaled that a successful job's output is ready to gather and will then handle its own type of gathering (e.g., SAM). The job stays in **ready to gather** status (or perhaps **now gathering**) until all output from all phases is handled. As each output file is gathered, the (pair of) metadata files for that output is copied to a pre-determined save directory (~/metadata). For SAM output, the metadata goes to Fermi along with the output. For CACHE, METADATA, and MERGE output, the metadata goes to the metadata' directory in mcfarm. (All such metadata is later pushed to Fermi from the '~/metadata' directory by the SAMMETADATA daemon). See the section on gathering for more information. Any files sent to MERGE are actually staged in a special directory, from which the MERGE_B daemon can later merge and store them.

7. The job is archived

The gathering daemon that handles the last output from a job will also then **archive** the job. Archiving consists of making a copy of the original job with its parameters, selected metadata files, and selected processing log files and placing them into the archive queue (e.g., "/home/mcfarm/archive/jobs" or one of the segments on some file server). All remnants of the job are removed from other queues (distribute, execute, gather, etc.) and from the main priority file (gather.conf) and all signal and lock directories. The job's status is now **archived**.

8. The job is purged.

According to configurable time parameters, the farm monitor can be set to periodically purge job entries themselves, or just the bulky log files. One keeps the archive information for farm statistics (efficiency reports) and in the event that a question arises on how a particular job was run.

The detail statistics of any job in any state can be captured to an ASCII file (see *prodstat*). Summarizing statistics are also available to provide a continuing metric on farm efficiency (see *prodsumm*). Another form

of historical information is to produce an “estimates” file to use for estimating resource needs of new jobs (see *prodest*).

FARM EMAIL

The McFarm software issues error messages and alerts to the farm using standard text email. There are two classes of email: routine and alert. Each class is associated (via the setup_farm script) with a text file of email addresses. Alerts consist of problems and warnings.

Here is a summary of the messages than can be issued by the farm, sorted by subject, and the events that trigger them. These are presented to provide a perspective on the tasks done by the farm and expected of the farmer.

MCFARM_ALERT

Node(s) have gone off-line
Disk space projected to (nearly) fill up within 24 hours
Normal farm daemon(s) not running - execute, distribute, and each (needed) gather daemon.
Job(s) in error-out status (run check_job)
Job(s) ready to gather but gather files out of sync (run check_job). Node(s) have large discrepancy in time-of-day, which was corrected (IF the root-daemon is up and responding)
Distribution queue empty (or about to be)
Daemon logs too big
Archived jobs take too much room
These cached files are present in multiple caches

MCFARM_BAD_CONTROL_FILE

Improper syntax in the distribute.conf file.

MCFARM_DEADLOCK

The monitor detected a deadlock condition. Notify programming support and abort one of the processes.

MCFARM_DISTRIB_FAILURE

Distributable job status corrupted
Failure to be able to copy a file from a local cache to a file-server cache
Failure to copy a tar file to a node, or node off-line during a copy.
Unable to flag a job as distributable

MCFARM_EXECUTE_PROBLEM

A tar file could not be un-tarred
The job's phase-file is missing (bad registration)
Failure to copy the process_run script into the job directory
A process_run script did not report itself to be started
A running job disappeared from the "ps" list
A running job could be looping.

MCFARM_FATAL_ERROR

Programming error has surfaced
User configuration error has surfaced

MCFARM_FILE_ACCESS_CONFLICT

Access to the gather.conf file impaired

MCFARM_GATHER_PROBLEM

A failure to FTP all desired files
A failure to archive cleanly
SAM failed or timed-out

MCFARM_ORPHANED_LOCKS

The monitor detected a lock whose owner is no longer running. If you know of no reason for this (e.g., aborted program) then notify programming support.

MCFARM_PURGE_JOB_PROBLEM

The monitor could not automatically execute the *purge_job* command.

OPERATION OF FARM DAEMONS

Most McFarm control is achieved through the use of various **daemons**, which are python scripts run as background Linux tasks, and which examine various queues (disk directories) on a regular basis to find work. Once these daemons are invoked (either by the *setup_farm* command or by an individual command below), they handle all routine processing needs. They can be stopped and started as needed (except that *lock manager* must always be running if any other farm software is running).

start_lockman / stop_lockman

The lock manager runs on the job server and is responsible for granting locks requested by any McFarm process on any node. It examines the *~/lock* directory for this purpose. When any process wishes to lock a resource (such as, say, the distribute queue) then the requesting process will place a file into the lock directory starting with "req_lock". If the lock-manager can grant this request, it will place a file in the lock directory with the name of the resource (e.g., "distribute_queue") containing the requesting process ID as the first line in the file. If the resource is already owned, then the requester becomes the second entry in the resource file (or third, etc.). The requester checks the result by examining the resource file himself, and simply waits until he is the owner of the resource. Similar activities occur when a process unlocks its owned resources.

Without the lock manager running, no other McFarm process can be granted a lock, therefore no McFarm script will run unless the lock manager is in-place. It should always be the first daemon started and the last daemon stopped.

There are actually two daemons running to constitute the lock manager. One (lockman) handles requests from other McFarm processes to grant or release a lock on a resource. The other (lockmonitor) checks for orphaned locks and deadlock conditions. If necessary, emails will be issued and/or purging of orphaned locks will be done.

start_distribute / stop_distribute

Stopping arguments...

[--wait]

The "—wait" argument tells the stop command to wait indefinitely until the stop is confirmed. Since the distribute daemon could be working on a job at the current time, it might take a while before it was in a position to stop (e.g., between jobs). Without this argument, the stop command will cease by itself after a few seconds (if the gather daemon is in process).

Distribute daemon logic...

The distribute daemon runs on the job server.
When a new job is registered into the farm, an entry for it (directory with its name) is placed into the distribute queue (the directory ~/dist_queue on the job server). Also at that time, a line is placed into the standard priority file *gather.conf*. It is the task of the distribute daemon to periodically examine the gather.conf file (and occassionally, the distribution queue itself) and look for any job that has not yet been assigned to a processor (distributed). Once a job has been distributed to, say, node XXX, then its directory will be moved form ~/dist_queue to ~/dist_queue/XXX. The daemon issues messages via email, and also write to a log file.

The daemon looks for jobs in priority sequence, according to the following rules:

- The "gather.conf" file is read, looking for all "job=" lines, **or** the dist_queue is searched for undistributed jobs.

  Typically, all active jobs are in the gather.conf file in the order in which they were originally registered. The farmer may re-arrange them to re-order their distribution sequence (*within* the same priority – see next paragraph). If this file is empty, then every 10 minutes the dist_queue is directly searched. (It is normally an error to have a job in the distribution queue but not in the gather.conf file, but this is a fail-safe to distribute jobs even if the gather.conf file is bad). Either way, a list of job candidates is obtained.

If a group of jobs are found to be distributable as described above, they are prioritized as follows. Within each of the following priority "groups", the farmer's job-priority controls the order of selection of jobs. In order words, the priority that is assigned to the job by reg_job (which can be 0 thru 10 and which defaults to 5), is used to select the next job from *inside* a group, but can never make a job be advanced to a "higher" group. A priority-5 generate-only job will still be done before a priority-1 d0gstar-job, but given two generate-only jobs, the one with the lowest priority number is distributed first. (For identical priorities, the one whose name comes first in the alphabet is distributed first).

- Generate-only jobs are handled first, in the theory that they are short and the farmer is possibly waiting to see if they run before registering more jobs.
- Any job that was explicitly forced onto a specific node (using the "—node=xxx" argument in reg_job) is handled, if that node needs work.
- If a job requires an input file that is presently residing on a LOCAL cache, the daemon will distribute that job if the node needs work.

- The daemon will distribute jobs whose input is in a fileserver cache (thus accessible to any node) if there is an exact memory match up (e.g., the node has 256MB allocated per cpu and the jobs requires 256MB).
- The daemon will distribute those same jobs even if the cpu memory more than the job requires. (Note that the daemon will not distribute a job to a node that has too-little memory per cpu, because it assumes that some node will have enough and will thus wait for such a node to become available. The only exception to this is if you force a job onto a node).

Of course, no D0Sim jobs will be started unless all of their minbias files are present in the fileserver cache(s).
The daemon discovers that a node needs work by looking in the signal directory (on the job server) for an entry that starts with “execute_needs_work”. Such an entry will tell the distribute daemon which nodes have fewer jobs running than they have been allowed to run (per the “max=n” argument in the distribute.conf file). If the daemon fails to find any node that has an open slot, then there is nothing to be done at this time, and it will retry the entire process after a brief sleep.

To effect the distribution, the daemon will make and copy a “tar” file that contains the job parameters to the target node’s *exec_queue*. This is a small file sent across the network. It then moves the job entry in the distribute queue to a subdirectory NNN to flag it as distributed.

start_monitor / stop_monitor

The monitor daemon runs on the job server. It maintain the health of the system and alerts the farmer regarding problems. Generally speaking, its purpose is to periodically review the status of all (non-archived) jobs, all defined nodes, all gathering daemons, all log files, signal files, and lock files, and either issue an email “alert” to the farmer or repair farm files itself. This monitor is similar to a “cron” facility, and their frequency and retention-periods are configurable.

Here are the individual tasks that it performs

FARM STATUS REVIEW

Periodically (e.g., every 4 hours) during the day (e.g., 7AM 11PM) the monitor emails alerts for any of the following conditions:

Node off-line

Node’s time-of-day different than job-server’s

Disk space filling up on any node
The monitor looks through a time period equal to the end of the first job on each node, plus 1 hour for gathering, to see if the disk-usage pattern exceeds 95% of capacity. It assumes the first job will be gathered/deleted within an hour of completion.

Daemons needed but not running
Normal farm operation requires an execute daemon on each node, a distribute daemon on the job-server, and a gather-daemon for each type of job-disposition (e.g., SAM, cache, ftp) that is needed by any job currently registered.

Remind about any errored-out jobs
The farmer can use check_job to repair a job in a gather-queue but not in the main job-priority file (gather.conf). Such a condition prevents the job from being gathered. The farmer can use check_job to repair jobs.

An insufficient number of undistributed jobs.
Within 24 hours of exhausting the distribution queue, and alter is issued to register new jobs to avoid idle CPUs.

A file in a cache directory that is not needed.

A file in a cache directory that is duplicated in some other cache directory.

FRIDAY SCAN

Each Friday morning, these tasks are done.

Current daemon log files too big. Continuous long-term daemon operation can create large log files, so stop and restart can be done periodically.

Archived job log-files purged. The “purge_job” command is run to affect part of the job archive information (the large d0gstar log file), leaving the rest of the job still archived. Job-server archives

exceed 20% of server disk Archives include jobs and metadata files. Even after log files are dropped, the archives can still grow without bound in the present version of the software.

purges old daemon log-files. Each daemon creates new log files each time it is invoked, renaming prior log files using a numbering scheme. Periodically, the prior log files are silently purged.

PURGE TEMP FILES

Any file or directory in the farm's "tmp" directory that is older than a certain number of days is silently purged. Files can be left in this directory if farm processes are aborted or miscoded.

PURGE ORPHANED FILES

If input files in any cache have been processed by jobs using the "delete_input=YES" argument, and if there are no more jobs in the distribution queue that need this input file, then the monitor will purge that cache file. Purging is examined every 30 minutes (see farm configuration).

PURGE ORPHANED SIGNALS

The farm has a "signal" directory used to facilitate inter-task communication. The monitor reviews signals that appear to be too old to ensure that they are still viable, and silently purges them if not. Files can be left in this directory if farm processes are aborted or miscoded.

REGENERATE THE ESTIMATES FILE

The jobstat, nodestat, and reg_job commands all rely on being able to accurately estimate the amount of time and disk space that a new job will require. An "estimates" file is created from the prodest program that contains such average numbers taken from archived jobs, so it is only necessary to update this file periodically to have good working estimates. The monitor will perform the command

prodest —estimates_output

every Friday morning.

Gather Server Daemons

Any node
that has a SAM station will be a gather server and/or an acquire server, and the job server is also used for gathering other types of output dispositions. A gather daemon and an acquire daemon require access to all nodes' scratch areas.

An individual gather daemon is typically invoked to handle one type of output disposition.

TAPE - write the output files to an enstore tape then purge. This method is no longer documented.

SAM - send the output files and specific metadata using SAM, according to the FARM_SAM_* parameters, then purge. Note: there can be significant latency involved in a SAM export, since it requires SAM to be operational at FNAL. The SAM gather logic will terminate storage attempts for files that are not succeeding and try other files, cycling between them and retrying until they are sent (or the gather daemon is terminated). The mcfarm account must be able to do *setup_sam* in order for this daemon to be used. Also note: if you export a file to SAM, the daemon will automatically try to "declare" all its parent metadata (at least any parents that belong to the same job).

FTP - send the output files and specific metadata using FTP to a specified directory, then purge. Note: FTP is currently disallowed by Fermi. If you wish to restrict the time of day when FTP is to be attempted, use the FARM_FTP_WINDOW_START and FARM_FTP_WINDOW_END settings in setup_farm. FTP will be attempted only during your window (for better network behavior).

METADATA – At the present time, Fermi does not want Monte Carlo production to store generate, d0gstar or d0sim files, but the metadata for them must still be declared to SAM since they are parent files to the reco and recoanalyze output that IS stored. In McFarm, these parent outputs are set to have the disposition "metadata" to indicate that the metadata is to be gathered but the output is to be discarded. This daemon simply copies such metadata to the ~/metadata directory, and a subsequent daemon (SAMMETADATA) forwards it from that central location to SAM.
.

DISCARD - Want to retain neither the metadata nor the output for the phase, so delete the output file and do not save the metadata. This disposition would be used if you were producing reco files for local consumption only, and wanted to discard the gen, d0gstar, and d0sim files completely.

CACHE - copy the output file to the node's "preferred" fileserver cache directory (or, if no room, to its local cache). If the job used the "cachelocal" disposition, then caching FIRST attempts to move the file to the node's local cache, and only if that fails will the file is cached to a file server. This daemon also handle "cacheinternal" dispositions.

SAMMETADATA - Unlike the other gather daemons, this daemon handles no output files. Instead, it takes the metadata description files (import*.py) that are associated with each finished (and gathered) phase, and pushes them to a SAM declaration. After successful declaration, the pairs of files are purged from the ~/metadata dir (they still exist in the job archives). This parent-file declaration is partly redundant with the ones that the SAM gather daemon does, but it also handles the metadata for the gen files (that are typically not part of each reco job). This daemon requires the ability to "setup sam".

Once all of a job's output has been gathered, the gather daemon that handles the last such disposition will also archive the job. Note that each phase of a job has an output file, and each output file can be set to have any or all of the above dispositions done for it (using the *reg_job* command).

MERGE – copy the output file to a special staging directory (FARM_MERGE_STAGE_DIR) and declare the file immediately. Later, part B of the process (daemon MERGE_B) will examine the stage directory for completed jobs (or sufficient small files to make up a large output file), merge small files into large one and store the result. The merged data remains archived in the archive-queue under its request-ID code. NOTE: Specifying MERGE actually causes two daemons, MERGE_A and MERGE_B, to be launched. You can also launch each daemon individually. MERGE_A is responsible for copying files to the staging directory, and must complete its actions before the job can be archived. MERGE_B is responsible for doing the merging on staged files, and subsequent storing in SAM (by handing off the file to a SAM gather daemon).

start_gather / stop_gather

You must be logged into the gather server to issue the start command (or else use the *start_farm* command from the job-server). You can use the stop_gather command from the job-server or the gather-server.

Starting arguments...

--

method=XXX

At least one method is required (SAM, FTP, METADATA, DISCARD, CACHE, MERGE, or SAMMETADATA). Multiple methods are permitted, but the daemon is not multi-threaded. The CACHE, FTP, DISCARD, METADATA, MERGE, and SAMMETADATA arguments are allowed once per farm, on any node that has full access to all scratch areas. The SAM gatherer may be started once per node that has a SAM station on it.

Prior to invoking a FTP gather daemon, you must have set the variable FERMIUSER to the account name to be used to login at the Fermi node. The start-up process will then request the password to be used to authorize access for that account. FTP is non-secure access.

--

accept_files=xxx-yyy

This optional argument is available if performing SAM-gathering or FTP-gathering. The "xxx" and "yyy" arguments are file-types (such as "gen", "d0g", "sim", "reco", "rAtpl", etc.). This argument allows only files of the specified type(s) to be gathered by this daemon. For example, say that you want to isolate gen and d0g files from other files. The argument:

--
m=ftp --accept_files=gen-d0g

would send only gen and d0g files.

Stopping arguments...

--

method=XXX --all [--wait] node#[s]

From the job server or the gather server, one can stop daemons by referencing (each) method. For example, to stop a combination SAM and cache daemon, one would have to include both methods in the stop command. Omit the method argument to stop all daemons on that node.

The “—all” argument from the job-server causes daemons on all nodes to be stopped. Specifying one or more specific node numbers causes daemons on those nodes to be stopped.

The “—wait” argument tells the stop command to wait indefinitely until the stop is confirmed. Since the gather daemon could be working on a file at the current time, it might take several minutes before it was in a position to stop (e.g., between outputs). Without this argument, the stop command will cease by itself after a few seconds (if the gather daemon is in-process).

Gather Daemon Operation

A gather daemon (other than SAMMETADATA and MERGE_B) will proceed according to these rules:

It will determine if any jobs have output that is ready to be gathered. Normally (for efficiency purposes) it scans the signal directory for this information (signal files are placed there when a job ends). Every 2 hours it bypasses the signal directory and scans every node's gather-queue directly. It is looking for a job that was successfully completed, and which has an output file whose disposition matches at least one of the gathering methods that this daemon was invoked to handle.

Next, the daemon examines the priority file (e.g., "gather.conf") to see if the candidate job appears in an early "group". Normally, all jobs in the file belong to a single group, so this review amounts to simply finding the job in the gather file where it should always be.
(The farmer is allowed to impose "groups" on the files in this list, however, if he wishes to have all the jobs gathered from a given group before any job is gathered from any subsequent group. The grouping rules apply only to jobs that actually have output that applies to this gatherer).

If those hurdles are passed, the job's output(s) are gathered to their respective destinations. Possible errors such as missing files, FTP transmission errors, SAM timeout, or the node going off-line during a tape-write may prevent gathering, and the daemon will fail-safe and issue alert messages.

Having gathered an output file or files for a job, the daemon will then update the job's file of pending dispositions (the job.phase file) as well as the information in the signal directory and in the gather.conf file. If it turns out that this daemon successfully gathered the last pending output file for that job, then it will proceed to **archive** the job. Archiving was previously described. After archiving, no trace of the job exists anywhere in the farm except in the archive queue, and possibly some of its output files in a cache directory.

The SAMMETADATA gather daemon is simpler. It examines the ~/metadata dir (and volume-name subdirectories, if any) for pairs of description files, which are placed these when output phases are gathered. It tries to declare them to SAM and get a positive response. Once it does, it purges them from this directory. If it fails for a specified period of time, it will send email notifications. See the "SAM_INTERVALMINS_nn" environment variables for a way to specify the retry interval(s).

The SAMMETADATA daemon basically examines the ~/metadata directory for import files to be declared, then does so parent-first (e.g., d0gstar before sim). Errors and problems are retried for awhile before email alerts are issued.

The MERGE_B daemon handles the second part of the MERGE disposition. (The MERGE_A daemon is a "normal" daemon that staged the output files and their metadata files into a stage directory). The MERGE_B daemon examines the stage directory periodically and groups files by their request-ID. It then determines if either (a) the job is finished and all merged files are present in the stage directory now, or (b) more than 50,000 events are present even if the job is not yet fully staged. In each case it will "make up a group" of files, merge them, and create an import file for them. During this process, a directory under stage called 'req_XXXX_TTT_NNNNN' is used, where XXXX is the request ID, TTT id the file-type (e./g., tmb for

thumbnail), and NNN is a unique group number (the first event in the group). Once the file and metadata are created, a SAM daemon will do the actual store in the near future. After a successful store, the metadata and logs are moved to the archive queue under this same group name.

start_acquire / stop_acquire

You must be logged into a gather server to issue the start command (or else use the *start_farm* command from the job-server). You can use the stop_acquire command from the job-server or the gather-server.

Starting arguments...

--

method=XXX

At least one method is required (SAM is currently the only supported method). The SAM acquire daemon may be started once per node that has a SAM station on it.

Stopping arguments...

--

method=XXX --all [--wait] node#[s]

From the job server or the gather server, one can stop daemons by referencing (each) method. For example, to stop a combination SAM and cache daemon, one would have to include both methods in the stop command. Omit the method argument to stop all daemons on that node.

The "—all" argument from the job-server causes acquire daemons on all nodes to be stopped. Specifying one or more specific node numbers causes daemons on those nodes to be stopped.

The "—wait" argument tells the stop command to wait indefinitely until the stop is confirmed. Since the acquire daemon could be monitoring multiple projects at the current time, it might take several minutes before it was in a position to stop. Without this argument, the stop command will cease by itself after a few seconds (if the acquire daemon is in-process).

Note: The acquisition of each file is done with an individual SAM project, and if such a project has been submitted to SAM, but not completed, then stopping the daemon will cause each such project to be terminated.

Acquire Daemon Operation

Presently the acquire daemon handles the acquisition of input files from SAM at fnal. In normal operation, it periodically scans the undistributed jobs, looking for those with initial input files where the reg_job command contained "—input_source=SAM". The distribute daemon will not dispatch that job until the input file is present in (some) cache, so the acquire daemon first checks all the cache contents to see if the file is already present, and if not then it creates and submits a SAM project to retrieve the file.

The farmer controls the number of simultaneous download-projects that can be launched by this daemon, through the FARM_SAM_MAX_ACQUIRE_FILES environment variable. It is suggested that multiple files be allowed, but not too many to swamp the bandwidth. The acquire daemon will maintain this number of open projects, until the need for input files is exhausted.

The acquire daemon will request input files in groups that correspond to the tape-volume on which they are stored in SAM. If the files in SAM are not in a tape volume (instead in some cache), those will be requested first.

By default, the acquire daemon will select one of the processing node's local cache as the target destination for a given file. The selection algorithm considers space availability, and also attempts to spread out the files amongst the various nodes. The reason for this is so that when processing of that file commences, it will be a local-I/O process. The downside of this method is that the job that uses this input file may have to wait until another job finishes before getting dispatched. Thus, there is another option available to the farmer to place an input file in a file-server cache (where it can be seen by any node) instead of a local cache. To do this, the job must be created with the argument "—input_placement=CACHE" (instead of the default CACHELOCAL).
Suggestion: whenever a job is to read an input file that is presently in SAM, and assuming that this is the only purpose of having the file, the reg_job command should also contain the argument "—delete_input=YES" so that after processing is done, McFarm will clean up the cache by deleting that file. Naturally this will not be done if there are future needs for the input file.

File Server Daemons

There are presently no daemons directly associated with file-serving or caching. The gather-daemon can cache a job's output to the node's preferred fileserver or to a local cache, and the distribute daemon can possibly copy a file from one node's local cache to the preferred file-server of a different target node.

It is the farmer's task to ensure that other needed input files are placed into some file-server's cache (using *cache* or *cachelocal* as a job disposition), and to remove them when no longer needed. The removal is typically accomplished by using "delete_input=YES" on all subsequent jobs. It can also be manually running farmstat periodically to see unneeded files and deleting them.

The "preferred file server" concept is implemented as follows. If there are N nodes and M file servers, then the first (N/M) nodes will prefer to use file server 0, then next (N/M) nodes will prefer to use file server 1, and so forth. Ordinality is determined by node number. At present, the "preference" is invoked when a gather to cache is done. All nodes can "see" all file servers' cache directories, so spreading the work is a performance issue. The farmer is responsible for spreading the work when multiple file servers are implemented, so that one NIC card is not a bottleneck.

Simple Production-Node Daemons

Every node that actually does Monte Carlo processing must have an execute daemon started or else work will not be distributed to them.

start_execute / stop_execute

You can issue start and stop commands from the job-server for any node, or from the node itself for its own daemon. No arguments are required to start an execute daemon.

Both commands, if run from the job-server, can have the optional arguments "—all" and/or node-numbers, to specify that all nodes' daemons (or specific node's daemons) are to be started or stopped. By default, only the current node's daemon is affected.

Execute Daemon Operation

T
he daemon will determine how many jobs are presently on its node, and verify that it is indeed running (using the job's pid from its "job.pid" file). If it has fewer jobs than it has CPUs, it will place an "execute_needs_work" file in the signal directory and begin scanning for new jobs in its execute queue.

The job-server's distribute daemon will send work in response to this signal, and jobs will then appear in this node's execute queue in "tar" form.
When the daemon has a new job to execute, it will do the following:

Un-tar the job file into its in-process ("run") directory.

Flag the job as "executed" (read-flag on tar file is set to not-readable)
Copy the "process_run" script into that directory, so that any future changes to that script will not affect this running job.
IF there is NO batch queue associated with this node (e.g., pbs, Condor, etc.), then the execute daemon next spawns a background task to execute process_run from the job directory in the run queue. It waits up to 2 minutes for the job to show a "job.pid" file, indicating that process_run has successfully started processing the phases of the file. If no file appears, an email message is issue and the job is removed from the execute queue. The execute daemon also (every 30 minutes) re-checks all its jobs to be still running and not looping. This is done in case there was a late script error in process_run, or process_run itself was aborted by the farmer. In either of those cases, process_run cannot clean up after itself, so the execute daemon does the cleanup (error-out the job) and email notification. Some D0 binaries can go into a loop, and if that is detected, then then process is killed and the job is moved to the error_queue (where you would likely use check_job –k to kill it).

If there IS a batch-queue associated with this node, the process_run script is handed-off to that batch queue at this point. The execute daemon still checks for continued running (or at least

continued presence in the batch-queue). See the section on batch-queue integration later in this document.

When a job completes, successfully or otherwise, the daemon removes its tar-file entry from the execute queue. This action triggers the execute daemon to request additional work from the job server.

Root Daemon Operation

In addition to the execute daemon, each node that actually does Monte Carlo processing may optionally also have a “root daemon”. It is started (at boot time) by the rc.local file, or (any time) from the root account, using the *start_rootman* script that is provided with McFarm.

Having a root daemon in place on each node will allow root-level commands to be issued from the farm. For example, see the “reboot_node” command.

installing a root daemon

The root daemon must use all-local (non NFS) programs and directories, so that if NFS is having a problem it can still be communicated with (via ssh). Thus, make the directories:

/scratch/localbin
/scratch/root
/scratch/logs

and place copies of rootman and rootreq into the localbin directory. For best results, append the command to start the daemon at the end of the /etc/rc.d/rc.local file, and also make it an executable script in the /root account (e.g., start_rootman).

The root daemon operates by regularly examining a local directory(“/scratch/root” for command files that have been placed thereby some McFarm operation (such as reboot_node). The daemon will execute the commands inside the file and return the results in a second file. The first line is encrypted so that the root daemon can verify that the command came from an McFarm program or a root_command.

To get a request file into that directory, the program uses ssh to invoke a special program /scratch/localbin/rootreq, whose task it is to construct the request file, get the reply from the root manager, and return it the caller. One does not have to be root to run this request program. Special argument-encryption is used to verify that requests comes from an McFarm program or from root_command.

Invoking the root daemon for general commands:

root_command [--script=xxx|--command=xxx|--restart|--stop|--reboot] nodename[s]

In addition to the root daemon being used by McFarm programs, you can send it general commands usig the root_command program. This program can be executed by any account, not just be mcfarm. You supply either a command or a scriptname (or a special “—“ argument), and one or more node names (using the name by which the node is accessed via ssh).

Use the '—command=xxx' argument if you have a single binary or script that you wish to execute. The target node must have access to it by that name, and it must be executable. For example:

```
root_command hepfm003 --command=date
root_command hepfm003 --command="ps axef"
root_command hepfm003 --command=/home/mcfarm/bin/someexe
```

Use the '—script=xxx' argument if you have constructed a script that you wish to execute, but the script itself does not have to be visible to the target node nor flagged as executable. Its contents will be made available to the target node in the command itself, and then it will be executed. For example (to access all mounted nodes):

```
root_command /mnt/hepfm* --script=/home/mcfarm/somescript
```

Use the '—restart' argument to make a root daemon restart itself (presumably after you have copied a new version of its code into the node's /scratch/localbin directory).

Use the '—stop' argument to make a root daemon simply terminate.

Use the '—reboot' argument to make a root daemon issue a /sbin/shutdown -r 1 command.

Naturally, the presence of the root_command script and available root daemons is a security issue. You should consider restricting the command to the mcfarm account (which has ssh access to all the nodes without a password), and hence the mcfarm password is to be guarded just like the root password.

process_run

The python script “process_run” is invoked once for each job, using a private copy and with the current-directory set to the job-directory in some node’s in-process queue (in “run”). It performs the following tasks:

> Ensures that all rcp files have a unique signature, and all “dump” periods are set to 1 event and turned on (so that the farm software can observe job progress). Progress messages will be sent to files named “progress.XXX”, where “XXX” is GENERATE, D0GSTAR, D0SIM, D0RECO, RECOA, etc.
>
> Reads the “job.phase” file (created by reg_job) which lists the phases to be done. For each phase it will...
>
> - Show a start line in the farm.log file.
> - Convert the make script to bash format, and include an invocation of the “source /fnal...” command to allow setup to be done for bash.
> - Execute the appropriate make file (e.g., gen.make). The output from each phase is always placed in the job directory itself.
> - Wait for the server to be on-line (in case it went off-line during the long processing period)
> - Audit that phase to have the correct number of output events, existing output files, no termination or seg-fault messages, and an output file that is comfortably below 2GB in size. Any problems cause immediate termination and erroring-out of the job.
> - Record the processing statistics for the phase (cpu cycles, page swaps, duration).

Having successfully completed all phases, the job’s metadata files are updated to reflect the size and time-stamp of the output files. The job’s tar-file entry is removed from the node’s execute queue, thus allowing a new process to start (via the execute daemon). This job-directory is moved to either the gather queue or the error queue. If successful, an entry is placed into the signal directory for (each) output disposition needed for this job, which triggers a gather daemon. If unsuccessful, an appropriate email is sent and it is now “errored-out”. process_run exits

FARM COMMANDS

If the farm ran perfectly smoothly, the only commands the farmer would need would be “start_farm”, and a periodic “reg_job” to register new jobs.
Anticipating that things will not run perfectly smoothly, several commands have been provided to get status information, and to fix and purge jobs.

check_job [-hdurgalvcefk] [jobname(s)] [nodenumber(s)]

--help | -h - print this help message
--dist | -d - distributed jobs (running, OK-to-gather, errored-out)
--undist | -u - undistributed jobs
--run | -r - running jobs
--gath | -g - ready-to-gather jobs
--act | -a - jobs active in farm (excludes archive)
--all | -l - jobs anywhere in farm (includes archive)
--arch | -v - archived jobs
--err | -e - jobs with any problem
--fix - -f - fix the job (if omitted, check_job will suggest only)
--kill | -k - kill and completely remove the job from all queues, (except archives) regardless of its condition

Purpose: Diagnose and/or repair jobs. Repair usually means restart.

Notes:

- Run from job-server only

- If neither “—fix” nor “—kill” are specified, jobs are diagnosed only and suggested actions are displayed. The “—fix” argument will cause error conditions to be “repaired” (see below). The “—kill” argument will completely remove the job from the farm (INCLUDING the distribute-queue).

- Archived jobs are never diagnosed or affected.

Jobs that are supposed to be running are checked to be actually running on the proper node.

“Repairs” can include the following actions (“KILL” included if the job is now running):

KILL, DELETE, AND RESTART-FROM-BEGINNING
Can apply to a job that is in some other invalid state, but which would be expected to work right the next time (e.g., a job that failed when a node went down). New job status will be “undistributed” and would re-enter the farm normally.

KILL, DELETE, AND RESTART-FROM-PHASE-XXXX

Same as the restart-from-beginning, except that the output from one or more chained binaries is completed so the restart will be at a later point in the chained job. The input to this later point has been placed into some production node's local cache, so this job will resume on that same node.

KILL, DELETE, AND FORWARD-TO-GATHERING
The job did not do all its final cleanup steps, but all output and metadata was produced (or can be easily produced now by check_job), so the job will simply be moved to the gather queue (after generating any missing metadata).

KILL, DELETE FROM ALL QUEUES, INCLUDING DIST QUEUE
Can apply to a job that is looping or whose data file exceeds 2GB. Nothing will be left.

KILL, DELETE FROM ALL QUEUES EXCEPT ARCHIVE
Can be done to a job that is in the archive queue and also in some other queue. Only the archive entry will be left.

RECREATE EXEC_QUEUE ENTRY
If an otherwise-OK job is simply missing an entry in the node's execute queue, that entry is re-copied from the distribution queue. The job is not otherwise disturbed.

GENERATE GATHER FILE ENTRY
If an otherwise-OK job is simply missing an entry in the main priority file (gather.conf), an entry is created there for it. The job is not otherwise disturbed.

GENERATE/CORRECT SIGNAL QUEUE ENTRY
If an otherwise-OK job is simply missing an entry in the signal directory for gathering, an entry is created there for it. The job is not otherwise disturbed. This can happen if you manually change the job.phase file specification for disposition.

farmstat [--help] [--brief]'

--help | -h - print this help message'
--brief | -b - omit low-priority actions'

Purpose: Review the status of the entire farm and see alert messages, (essentially the same as provided by the monitor daemon but available on-demand).

Notes:

- Run from job-server only

- See the discussion of the monitor daemon for a summary of the information provided. Basically, you will be alerted to anything that is wrong now or potentially a problem in the next 24 hours.

ALSO: farmstat will provide a list of files in any cache directory that are not needed for any known job. Normally, you would have used the "delete_input=YES" argument on trailing jobs, so the farm will be cleaning up these files itself.

jobstat [-hbndurgalvce] [-sxx] [jobname(s)] [nodenumber(s)]

--help | -h - print this help message
--brief | -b - omit times and completion-estimates
--output_format=x – switch output format to either ‘d’ (display) or ‘c’ (comma-delimited)
--narrow | -n - limit display to 80 columns
--dist | -d - distributed jobs (running, OK-to-gather, errored-out)
--undist | -u - undistributed jobs
--run | -r - running jobs
--gath | -g - ready-to-gather jobs
--act | -a - jobs active in farm (excludes archive)
--all | -l - jobs anywhere in farm (includes archive)
--arch | -v - archived jobs
--err | -e - jobs with any problem
--output_format=c | -ofc dirname - output format, x=d for display, or c for comma-delimited (and then dirname must be supplied)
--sort=jobname | -sjn - sort by job name
--sort=nodename | -snn - sort by node name
--sort=starttime | -sst - sort by start time
--sort=endtime | -set - sort by end time
--sort=totaltime | -stt - sort by total time
--sort=donepercent | -sdp - sort by done percent
--sort=currentspace | -scs - sort by current space
--sort=addspace | -sas - sort by additional space
--sort=peakspace | -sps - sort by peak space

Purpose: Display status and some alerts for specific job(s).

Notes:

Run on any node (but only the job-server and gather-server nodes can see all jobs).

Jobs are selected for display by jobname(s), and/or nodename(s), and/or queue-switches (e.g., -g means all ready-to-gather jobs). If no selections are specified, it defaults to all distributed jobs (which includes in process, ready to gather, and errored-out jobs). Archived jobs are omitted unless specifically included.

Display can be sorted by various individual parameters. Default to by-name.

The display varies according to the job status:

Jobname

Status - shows the job’s state:
READY TO DISTRIBUTE - in distribute queue, not yet distributed to any node for processing
WAITING FOR INPUT - in distribute queue, but distribute daemon has failed to distribute it at least once due to a missing input file. Examine the ~/signal queue for the missing file (or examine the job’s job.phase file for the first input file).

WAITING FOR SAM - in distribute queue, but the first input file, which is supposed to be acquired by the SAM acquire daemon, has not yet arrived in any cache.
READY TO EXECUTE - in a node's execute queue, but not yet picked up by its execute daemon. Short-lived status.
BEGIN_GEN - job is starting a generate phase, but has not yet loaded the executable.
GENERATE - job is running a generate executable.
GEN_LOOP - job is running generate, but has not updated output file on schedule. Check again, and use check_job to completely destroy job.
GEN_PAUSED - job is running generate, but has been paused.
BEGIN_D0G - job is starting a d0gstar phase, but has not yet loaded the executable.
D0GSTAR - job is running a d0gstar executable.
D0G_LOOP - job is running d0gstar, but has not updated output file on schedule. Check again, and use check_job to completely destroy job.
D0G_PAUSED - job is running d0gstar, but has been paused.
BEGIN_SIM - job is starting a d0sim phase, but has not yet loaded the executable.
D0SIM - job is running a d0sim executable.
SIM_LOOP - job is running d0sim, but has not updated output file on schedule. Check again, and use check_job to completely destroy job.
SIM_PAUSED - job is running d0sim, but has been paused.
BEGIN_RECO - job is starting a d0reco phase, but has not yet loaded the executable.
D0RECO - job is running a d0reco executable.
RECO_LOOP - job is running d0reco, but has not updated output file on schedule. Check again, and use check_job to completely destroy job.
RECO_PAUSED - job is running reco, but has been paused.
BEGIN_RECOA - job is starting a recoanalyze phase, but has not yet loaded the executable.
RECOA - job is running a recoanalyze executable.
RECOA_LOOP - job is running recoanalyze, but has not updated output file on schedule. Check again, and use check_job to completely destroy job.
RECOA_PAUSED - job is running recoanalyze, but has been paused.
FINISHING - job has completed all phases and is updating metadata. Short-lived status.
READY TO GATHER - job is in gather queue, successful.
GATHERING NOW - some output of the job is now being gathered.
ARCHIVING NOW - a gather daemon has finished gathering output and is now making the archive entry. Short-lived status.
ARCH:xxx - job is in archive queue, and its output was sent to 'xxx' (could be more than one).
INVALID NAME - job whose status was requested does not have a valid name format.
NONEXISTENT - job whose status was requested does not exist.
INVALID - job should be in a run directory, but info such as the job.pid file cannot be accessed. Error state. Repeat jobstat, and use check_job to repair/restart.
RUN ORPHAN - job is running but has no entry in that node's execute queue. An error state - check_job will repair.
ERRORED OUT - job is in the error queue for some reason (see recent emails, see also job's "farm.log" file). Error state, check_job will restart. gathered.
INCONSISTENT - job is in an unexpected combination of queues. For example, if it is distributed then it must be in the execution queue or the gather queue or the error queue. check_job will kill and restart. If you know which queues it does not belong in, remove it from these first and check_job may not have to kill it.

Status may also be suffixed with an entry such as (PBS_NN:RUNNING) or (CONDOR_NN:XXXXX), meaning that the job has been handed off to a batch-queue (where it is now job NN), and its status in the batch-queue is then displayed. If UNKNOWN is displayed, McFarm cannot find the job in the batch queue.

Node - the node on which the job resides if applicable

Started - the start day and time of the job

Est.End - For in-process jobs, the predicted end time, based either on the events done so far (if at least 5 done), or on a recent average of similar jobs (see prodest).

For archived jobs, the number of days since archiving.

THrs - total estimated hours of an in-process job

Events - events done / target events for the current phase of an in process job.

CurrMB - total MB used by the job at this time.

AddtMB - additional estimated MB needed for this job, at the most. (See note under “Peak”).

PeakMB - the most MB it is expected to need at one time for output. This is based either on the events done so far (if at least 5 done), or on a recent average of similar jobs (see prodest). Note that certain types of jobs can have their output deleted “early”, IF being disposed to METADATA or DISCARD, and after read by the next phase of the same job, hence this figure (and PeakMB) may decline partway through the job.
Remaining Dispositions - this is a list of all the output dispositions still to be handled for a completed job.

nodestat [-hbnale] [-sxx] nodenumber(s)

--help | -h - print this help message
--brief | -b - omit specific error messages and jobs
--narrow | -n - limit disply to 80 columns
--err | -e - nodes with any problem
--act | -a - nodes active in farm
--all | -l - nodes defined for the farm
--sort=nodename | -snn - sort by node name
--sort=usedspace | -sus - sort by used space
--sort=usedpercent | -sup - sort by used percent
--sort=addspace | -sas - sort by additional space
--sort=addperceent | -sap - sort by additional percent
--sort=peakspace | -sps - sort by peak space
--sort=peakperceent | -spp - sort by peak percent
--sort=peaktime | -spt - sort by peak time

Purpose: Display status and alerts for one or more nodes.

Notes:

- Run on any node (but only the job-server and gather-server nodes can see all nodes).
- Nodes are selected for display by nodename(s) or number(s), and/or queue-switches (e.g., -e means any node with a problem). If no selections are specified, it defaults to all active nodes.
- Display can be sorted by various individual parameters. Default to by-name.
- The display contains:

Node - The node number

UsedMB - Total MB now in-use in the scratch partition.

% Cap - What percent UsedMB is of the scratch size.

Add.MB - Additional projected MB needed on this node through the ending of the earliest running job, assuming its output is gathered within 1 hour.

PeakMB - Total of Used and Additional MB

% Cap - What percent full the scratch area will be at peak,

Running - the number of jobs and total MB for running jobs.

Done - the number of jobs and total MB for ready-to-gather jobs.

Error - the number of jobs and total MB for errored-out jobs.

- Alerts - the program will issues alerts of this type:

Node off-line
Time-of-day different
Disk projected to get over 95% full
Daemons should be but are not running
Daemon log files getting large

pause_job [-haf] jobname(s)

--help | -h - print this help message
--all | -a - all running jobs
--force | -f - issue command even if job previously paused

Purpose: Issue a Linux STOP command to a jobs that has a running phase,which will make the phase cease activity until resumed. This is useful if you are taking a server down for a brief time.

Notes:

Works only to jobs that have a phase running (gen,d0gstar, etc.)

Pausing multiple times is harmless but unnecessary. The—force argument is used if you think the farm's internal record of which jobs have been paused is incorrect.

prodest [-h] --estimates_output[=x] --from:x --to:x --farm=xx

--help | -h - print this help message
--estimates_output[=x] = produce a revised "estimates" file containing the average time & size of events by phase/process. The optional "x" allows you to specify the output file, if "=x" is omitted then the standard farm estimates files is refreshed. If this argument is omitted, output is to std-out.
--from:YYYY/MM/DD/HH/MM/SS - accept output that was gathered on or after this point in time. If omitted, no limit.If the HH/MM/SS is omitted, 00/00/00 is assumed.
--to:YYYY/MM/DD/HH/MM/SS - accept output that was gathered before this point in time. If omitted, no limit. If the HH/MM/SS is omitted, 23/59/59 is assumed.
--farm=xxx if you wish to filter on the farm where the job was run, specify its network-name here. This can be repeated. If omitted, all farms are shown.

Purpose: The "estimates" output creates a file by process that is used by subsequent farm tasks whenever an estimate of job resources is needed. If should be refreshed regularly.

Notes:

- Can be run on any node, but is most efficient on the job-server.
- Only archived jobs are accessed.
- If the "from" or "to" filters are used, you are restricting the output(s) to data that was gathered during the period.
- Each detail line contains the following fields separated by blanks:

phase production decay MinBi AvgMB/Event AvgSecs/Event

- The MinBi field will be 0.000-None if no minbi is applicable. There is a line for the overall phase (e.g., D0GSTAR) after all detail lines for that phase, and it has None for the production, decay, and MinBi field.
- If various CPUs in the farm have different MHz ratings, their times on this output have been normalized to the standard as defined in setup_farm.
- The "estimates" output should be produced on a regular basis (the monitor will run it weekly). Perhaps more frequently is better if you are introducing new types of jobs into the farm). It will by default produce a file named ~/archives/estimates that has one line per phase/production/decay/minbi process, and records the average MB/Event and Secs/Event experience when producing that type of output on the farm (normalized to the "standard" MHz rating on the farm). Each time you produce this output (which should probably be done for as large a time period as you have that is "homogeneous" to the current farm), these averages are re-computed. Subsequent farm jobs (such as jobstat) will use this information to estimate the resources needed by a job.

prodstat [-h] --detail_output=x --from:x --to:x --farm=x

--help | -h - print this help message
--detail_output=x = output the detail info to this file. If omitted, output if to std-out. Detail output contains one line per output file.
--from:YYYY/MM/DD/HH/MM/SS - accept output that was gathered on or after this point in time. If omitted, no limit.If the HH/MM/SS is omitted, 00/00/00 is assumed.
--to:YYYY/MM/DD/HH/MM/SS - accept output that was gathered before this point in time. If omitted, no limit.If the HH/MM/SS is omitted, 23/59/59 is assumed.
--farm=xxx if you wish to filter on the farm where the job was run, specify its network-name here. This can be repeated. If omitted, all farms are shown.

Purpose: The “detail” is a single print line for each output file that has been produced by the farm (or is in-process now), in a comma-delimited ASCII format, with all pertinent information as shown below.

Notes:

Can be run on any node, but is most efficient on the job-server.

If the “from” or “to” filters are used, you are restricting the output(s) to data that was gathered during the period. Since in-process jobs do not have a gather time, using either of these filters means all in-process jobs will be excluded.

- Each detail line contains the following fields, comma-delimited, with text field surrounded by “ characters:

Data Item	Description
output_file	The output filename
file_type	gen,d0g,sim,reco,rtpl
primary_input_file	The primary input filename
jobname	Job name that produced output
output_status	NOT YET STARTED (in dist. queue) IN PROCESS (now being run, or awaiting gather) COMPLETED (in our archives, not in SAM) IN SAM (has been placed into SAM)
num_events	number of events in output
iteration	this number is no longer used: a 0 will be here as a placeholder
skip_events	if any events were skipped in the input
file_size	the output file size in KB
phase_duration	the number of second of wall-clock time
cpu_time	the number of second of cpu time
swaps	the number of page swaps
done_date	the date done, YYYY/MM/DD, if completed
run_nodename	the node on which it was produced
speed	Speed rating for the node, where ‘1’ is a par node (e.g., a 866Mhz cpu with 200Mhz memory bus)
channel	ttbar, Z, qcd, etc.
decay	Zee, incl, etc.
ptlt	PtLt paramater value if known
ptgt	PtGt parameter if known
etalt	EtaLt paramater value if known

etagt	EtaGt paramater value if known
run_number	The run number of the parent gen file, propagated thru to all child outputs
gen_method	pythia or isajet
minbi_method	blank, or Fixed, or Poisson
minbi_average_events	If applicable, the average minbi bg events per trigger event
dispositions	All output dispositions, with a '-' character joining them, from: CACHE,CACHEINTERNAL,SAM,FTP,METADATA,DISCARD,MERGE (METADATA, CACHEINTERNAL, or DISCARD will be by themselves if present.
volume	Blank: tape writing is no longer supported
d0release	The release of the executable
farm_version	The version of the farm code
gather_date	The date output gathered YYYY/MM/DD
gather_moment	A number representing the exact second that this output was gathered
request_id	The D0 request ID under which the job was run, or blanks if none

prodsumm [-h] --summary_output=x—from:x --to:x --farm=xx

--help | -h - print this help message
--summary_output=x = output the summary info to this file. If omitted, output is to std-out. Summary output is a free-form file showing gathered output by phase/production/decay/minbi tag.
--from:YYYY/MM/DD/HH/MM/SS - accept output that was gathered on or after this point in time. If omitted, no limit. If the HH/MM/SS is omitted, 00/00/00 is assumed.
--to:YYYY/MM/DD/HH/MM/SS - accept output that was gathered before this point in time. If omitted, no limit. If the HH/MM/SS is omitted, 23/59/59 is assumed.
--farm=xxx if you wish to filter on the farm where the job was run, specify its network-name here. This can
be repeated. If omitted, only this farm is shown.

Purpose: The summary output shows statistics by physics process on how long the output takes to produce, and shows farm CPU utilization with breakdowns of various problems.

Notes:

- Can be run on any node, but is most efficient on the job-server.

- Assumes it knows how many CPU's can be brought to bear, hence it defaults to showing statistics just for the nodes on this farm and ignoring nodes from satellite farms.

- If the "from" or "to" filters are used, you are restricting the output(s) to data that was gathered during the period.

- Only archived jobs are accessed.

- The "summary" output is hopefully self-explanatory. It is suitable for simple email format if desired. It can be used on a regular basis (say, weekly) to obtain a metric on farm efficiency. It also shows interesting statistics for long-range planning. If various cpus in the farm have different MHz ratings, their times on this report have been "normalized" to the "standard" as defined in setup_farm.

- In order for the metrics concerning CPU utilization to be accurate, you must always use the "check_job" program to kill or restart any problem job (as opposed to performing manual queue & file manipulation that leaves no traces). Error-job residual information is placed into a special queue (~/archive/failed_jobs) so it will be available for such categorization later.

purge_job [-hcptfs] [--retaindays=nnn] [jobname(s)]

--help | -h - print this help message
--complete | -c - purge all of the job, not just the log file
--partial | -p - partial purge: just the large log file
--trial | -t - trial mode - just list the files/jobs to be purged
--final | -f - final mode - actually do the purge
--summary | -s - print summary totals only
--retaindays=nnn | -r=n - retain unless older in days than this

Purpose: Remove all or part of archived jobs, based on their age.

Notes:

Runs on the job-server only.

Presently, the monitor daemon runs this program automatically every Friday morning with a “partial” purge of jobs, based on a configuration age.

The trial/final option controls whether the jobs are just listed, or are actually purged. Default to trial.

The partial/complete option specifies whether just the job’s large log file is purged, or the entire job. Default to partial.

Detail job names are shown unless summary is specified.

The “—retain_days=nnn” argument specifies the number of days old a job must be before being purged. This defaults to the value in the configuration variable FARM_RETAIN_ARCHIVE_JOB_DAYS.

reg_job [-h]

--macro=xxx

--request_id=nnnn

--script=xxx —cardfile=xxx

--d0sim_minbi_input_list=xxx
--d0sim_minbi_percent=nn
--d0sim_minbi_method=xxx
--d0sim_minbi_average_events=nn.n

--mode=xxx
--priority=nn
--node=xxx
--memory=xxx
--input=xxx
--input_source=xxx
--input_placement=xxx
--delete_input=xxx
--
num_events=xxx
--
skip_events=xxx

--generate_disposition=xxx
--d0gstar_disposition=xxx
--d0sim_disposition=xxx
--d0reco_disposition=xxx
--thumbnail_disposition=xxx
--recoa_disposition=xxx (disabled)
--root_disposition=xxx (root no longer being used)

Purpose: Create a new job (using mc_runjob and a macro) and register it into the farm (make it distributable), with the ability to specify input files, event numbers, and varying output dispositions.

Notes:

- Run on the job server only, after you have made **either** a “macro” file (from the Request interface) **or** a “script” file for mc_runjob. One file name must be supplied, making one job. The script can be in any directory (if no directory is specified, reg_job looks in ~/conf_files). The macro should be given the full pathname.

- If you submit a **macro**, then you do not specify the cardfile argument or any of the d0sim arguments. The cardfile must be present in the appropriate “mcc-dist/packages/cardfiles” directory, according to the *cardfileversion, groupname, production*, and *decay* arguments in the request. The minbias arguments are taken from the request itself.

- Your script file is merged with the lines in the “*.basic” file found in the conf_files directory, to make a complete config-file for mc_runjob. The script expects mc_runjob to have been properly prepared. See “Preparing mc_runjob for McFarm integration”).

- If the initial phase is a generate phase (isajet or pythia) then you should also specify the full pathname to the actual cardfile to be used, and this must have the name format production-decay.pythia or production-decay.isa. Furthermore, the production and decay taken from this name will be substituted into the script. If you omit this argument, the cardfile name will be derived from the script and must already be present in the mc_runjob/templates/cardfiles dir.

- In the present version, one can construct a job that contains any or all of the following **phases**: generate, d0gstar, d0sim, d0reco (by asking for either/both of d0reco or thumbnail dispositions), or recoanalyze (by asking for root dispositions). If one creates a job that has multiple phases, it is always assumed the 2nd, 3rd, etc. phases use the output from the prior phase as their primary input. (D0Sim is allowed one set of secondary inputs - for minbias data). In order to have an McFarm job use a cached input file as primary input to some phase, then the first phase of the job must be that phase. For example, to run D0Sim on the output files of a previous d0gstar job, one must make a job whose first phase is D0Sim.

- Each output disposition can be as follows:

cache - place the output data file into its preferred file-server.
cachelocal - place the output data file into the local cache of the node on which the job runs. (This method is suggested when you are about to run subsequent jobs on that output).
cacheinternal - place the output file into the node's preferred file server, but do not create metadata for Fermi.
ftp - send the output file to Fermi via FTP. Note: FTP is currently disallowed by Fermi.
metadata - just delete the output, but retain the metadata for SAM. This is useful for the first phase(s) of a multi-phase job, when the output of those beginning phases is not to be saved at this time.
discard - delete the output and do not retain any metadata for Fermi. This would be used, for example, for the gen/d0g/sim/reco output in a job where the reco-analyze output was being cached internally.

sam - send the output and its metadata to SAM.
merge – will also store in SAM, but all such files are first staged, then when enough are available, they are merged together and the larger file is actually stored.
tape - no longer documented for McFarm.

- A phase can have multiple dispositions. Either repeat the disposition argument or use the format XXX-XXX (e.g., d0gstar_disposition=sam-cache). Note that the d0reco phase can have either or both "d0reco_disposition" and "thumbnail_disposition". Note that the recoanalyze phase can have only the "root_disposition" at present (no recoA file is presently produced by recoanalyze)..

- If the first phase is not generate, then you must specify the name of the primary input file for the first phase. This is done using the "—input=xxx" argument, where "xxx" is the name of the file (does not have to contain any directories). "reg_job" will issue a warning if that file cannot now be found in any file-server. The job will hit a "hard" stop when being subsequently considered for distribution, if this input file can still not be found at that time.

- If you do specify an input file, then you can also specify the input source: "create" or "sam". The default is "create", which means that this farm (or the farmer) is supposed to create or

upload the file to some cache. If you specify SAM, then a sam project will be run (by the SAM acquire daemon) to acquire the file and place it onto some node's local cache (or the file-server cache). You might then also specify where to place the file (--input_placement) as CACHE or CACHELOCAL. See the "sam acquire daemon" for more information.

- If you do specify an input file, then you can also specify the number of events to skip over in that file and/or the number of events to process out of it. The burden is presently on the farmer to ensure that valid and non-overlapping events are used from a previous file for subsequent processing. By default all events in that file are processed, skipping over none. Anytime you specify a non-zero skip_events, then you MUST also specify an iteration number of 2 or higher (to state which "iteration" this job represents on that single input file).

- If you do specify an input file, then you can also specify whether or not to "delete" that input after this job is done. You would normally do this for ALL jobs that need this same input file, and be sure to have them all present in the distribution queue, so that the farm can tell when the LAST one finishes successfully and then can safely purge the input.

- You can force a job to be distributed to a specific node using the '—node=xxx' argument, where xxx is a node number or name. If the specified node disagrees with the node on which the job's input has been locally cached, then the local-cache node will override your node assignment (and a warning will be issued by the daemon).

- You can override the minimum memory needed for this job by specifying "—memory=xxx", where xxx is a non-zero number in MB. If you omit this, then farm will determine the minimum memory needed by any phase in this job, and possibly restrict the job to specific production nodes. You would use this to force a job to a "large memory" node when you know that it subsequent jobs with more memory required

will follow on the same node.

- You can override the priority of the job, which defaults to 5 and can be set to 0 thru 10 (0 is considered a more important job than 1, etc.). This priority will affect up to three dispatch rules. First, the distribute daemon will dispatch the most important job first, assuming all other considerations for dispatch are the same (in other words, it will sort the entries in the gather.conf file by priority before analyzing them). Second, when a D0 binary is actually launched by the farm "process_run" script, the Linux "nice" command will be used to make the job run below default CPU priority. The reg_job priority of 1 thru 10 will be mapped to a nice number of 1 thru 19 (scaling by 2), so that the cpu-intensive binaries are positioned to soak up excess CPU cycles on the node but not interfere with normal operations. You should avoid using priority 0 for this reason. Third, if a batch queue is used to submit the actual job on a node, and if the batch queue has a priority scheme, then this priority number will be mapped accordingly. For PBS, the numbers 0 thru 10 will map to 1000 thru –1000 respectively, scaling by –200. For Condor, the numbers 0 thru 10 will map to 20 thru –20 respectively, scaling by –4. Also for Condor, farm priorities of 5 thru 10 will cause the "nice_user" option to be set True, while priorities from 0 thru 4 will cause it to be set False. Setting it to be True means Condor will pause it whenever othre work is being done on the target node (a nice behavior to have for a computation-intensive process). You can take direct control of the "nice-user" value (and other Condor parameters) using a configuration file – see Integration with Batch Queues later in this document.

- If D0SIM is one of the phases of the job (and you are not using a macro input), you can specify a set of secondary input file(s) to be used for background events, via the "d0sim_minbi_input_list=xxx" argument. The "xxx" argument must be an existing text file with one line in it for each minbias input file you wish to provide to d0sim, that line having just the simple name of the file (but should appear in SOME cache directory now). This argument can be repeated if you have multiple files that each specify multiple d0gstar input files. If you omit all the "d0sim_minbi_*" arguments, then no background will be used ("FIXED" with zero rate is parameterized).

- Along with the list of minbias files, you would normally also specify the d0sim_minbi_method to be "fixed" or "poisson", and the d0sim_average_events (how many minbias events per trigger event, per bunch). Also, the specify d0sim_minbi_percent to be the max percent of events to be extracted from any one input file before moving on to the next input file. (Suggestion: set this to 0, or omit it here, and reg_job will calculate it based on the number of events, average_events, and the farm parameter FARM_MINBI_EVENTS so that all minbi files are evenly used).

- The "mode=" argument can be set to either SETUP or MINITAR, to specify whether the binaries of the release being used originally came from a standard D0 release (e.g., you can do a "setup D0RunII") or from a farm-production minitar (e.g., untarred into /mcc-dist). The default is minitar mode.

- The program will issue a warning if any phase's expected output exceeds 2GB (according to recent history is similar jobs). You should probably cancel (check_job -k jobname) and re-size it into multiple jobs.

- As a result of a successful job-registration, the job's status becomes "ready to distribute". Also, an entry is placed into the gather.conf file (at the end of the last group), containing the job name and a summary of each type of gathering that it will eventually require. By default, then, the jobs will be distributed and processed in the order that they are registered (see distribute daemon for exceptions).

resume_job [-haf] jobname(s)

--help | -h - print this help message
--all | -a - all paused jobs
--force | -f - issue command even if job not paused

Purpose: Issue a Linux CONT command to a jobs that has a running phase, which will make the phase resume activity after a pause_job.

Notes:

Works only to jobs that have a phase running (gen,d0gstar, etc.) and that have been previously paused.

Resuming can be forced using the—force argument, if you wish to override the farm's internal record of which job's have been paused.

show_active_nodes [--number_only | -n]'

--help | -h - print this help message
--number_only | -n - show the node numbers only
--full_hostname | -f - show the url name including domain
--server_only | -s - show only the job server

Purpose: Display node names (or numbers) that are presently on-line.

Notes:

- If run from a non-job-server, will show only that node as active.

- Information could be a few seconds out-of-date.

show_server_url

--help | -h - print this help message

Purpose: Display the full job-server URL.

THE “setup_farm” ENVIRONMENT VARIABLES

Farm operation is controlled by a set of environment variables, presently available in a bash script called “setup_farm”. These variables must be established prior to attempting any farm command or daemon, by issuing the command:

. ~/bin/setup_farm (note that the leading ‘.’ is required to make the settings persist)

(The location of D0 executables and rcp-files is handled by the appropriate “setup” commands issued by the actual scripts themselves).

Please see the current McFarm release material for a template of the setup_farm script.

Notes:

- The intervals at which daemons check for more “work” can and should be brief to make them responsive. They are coded to use only small amounts of overhead for these checks. The lock daemon is normally on a very short interval since a node’s processing will stop until this daemon grants locks requests.

- The software is designed to know about the locations of node-specific resources through the existence of certain environment variables. For example, this scheme is used to identify file-servers:

FARM_FILESERVER_CACHE_nnn_A=$FARM_CACHE’nnn_A’

- Thus, every node knows what nodes have tapes and do file-serving.

- If you alter any setup_farm variables, you must stop all daemons (stop_farm), re-issue the “. setup_farm”, command, then restart all daemons (start_farm) before the variables would be recognized. Some variable-changes would also require that in-process jobs be finished first. Daemons on all nodes should be stopped and restarted.

- The “start_farm” and “stop_farm” scripts are bash scripts that should be customized to the way your farm is configured (e.g., what gather daemons you need and where).

INTEGRATING MC_RUNJOB WITH MCFARM

The mc_runjob package is invoked to create new jobs. If you integrate it into McFarm as described here, then you can use the "reg_job" command to make jobs.

1. Prepare your "~/conf_files" directory to contain a file called "xxx.basic", where "xxx" is anything such as your farm name, and have it contain all the mc_runjob parameters that are NOT particle-specific. Here's an example:

```
#  Basic script for starting a farm job for UTA
#
MiniDB=0         # ignore his tracking of status info on jobs
SaveOnMake       # allow makejob to save the info to the create queue
StandardD0=0     # require to be turned off when using minitar files
#
#  Attach and initialize configurators
#
#  Global defines the output directory
#
attach samglobal
cfg samglobal define string DestinationDir /home/mcfarm/tmp
cfg samglobal define string CurrentDir /home/mcfarm/createqueue
cfg samglobal define string Phase mcp11
cfg samglobal define string OriginName UTA
cfg samglobal define string FacilityName hepfm000.uta.edu
cfg samglobal define string ProducedForName Ian Bertram
cfg samglobal define string ProducedByName Tomasz Wlodek
cfg samglobal define string GroupName MCC99

cfg pythia define string D0Release p11.13.00
cfg pythia define int UseMaxopt 0

cfg isajet define string D0Release p11.13.00
cfg isajet define int UseMaxopt 0

cfg d0gstar define string D0Release p11.13.00
cfg d0gstar define int UseMaxopt 0

cfg d0sim define string D0Release p11.13.00
cfg d0sim define int UseMaxopt 0

cfg d0reco define string D0Release p11.13.00
cfg d0reco define int UseMaxopt 0

cfg recoanalyze define string D0Release p11.13.00
cfg recoanalyze define int UseMaxopt 0
```

- The next series of steps is dependent on whether or not you have an "official request" to be processed. First assume that you do. You will obtain the Request_nnnn.py file from the MC Production web page (after following instructions thereon to know which request nnnn to get). To obtain this file, you can either paste it in from the web-page (click on the request ID) or you can use sam to get it for you (preferred):

sam get request details --dictfmt > Request_XXXX.py

- Place this file in a directory such as "~/job_submit/req_nnnn", then from that directory run

make_macros nnnn

which will create a pair of files Request_nnnn_gen.macro and Request_nnnn_dsra.macro. Now you have a macro with all the request info necessary to make a generator file for all events. Submit this to McFarm using a command such as:

**reg_job --macro=~/job_submit/req_nnnn/Request_nnnn_gen.macro –num_events=xxxxx **
--generate_disposition=cache –request_id=xxxx

Note the output file name (gen-xxx) and allow McFarm to run, cache, and archive this job. If you have problems at this point, it is likely a cardfile naming or content problem. Don't proceed until the gen file is present in some /cache* directory.

It is suggested that you next create a series of 500-event jobs to each handle some segment of the gen file for d0gstar/d0sim/reco/recoanalyze processing (DSRA processing). Issue a reg_job command similar to this one as many times as necessary to cover the gen file: each time you increment the "skip_events" by 500.

**reg_job --macro=~/job_submit/req_nnnn/Request_nnnn_dsra.macro –num_events=500 **
**--request_id=xxxx **
**--skip_events=xxxx **
**--input=genfilename --delete_input=yes **
**--d0gstar_disposition=sam **
**--d0sim_disposition=metadata **
**--doreco_disposition=sam **
**--thumbnail_disposition=merge **
--root_disposition=sam

(Note that the gen file will not actually be deleted until all the DSRA jobs have been archived).

(Note that you may have to produce multiple gen files, if there are so many events that one gen file would exceed 2GB in size).

There are additional MC Production steps you should take to denote the request as finished, but you would do them only when all the output files have been stored in SAM (or some jobs killed and dropped due to errors).. Please see MC Production documentation for those steps.

3. If you are **not** starting with an official request, then this (and the next) step are necessary to supply "scripts" from which reg_job can construct a macro. For each particle-type you wish to investigate, create a suitably named script file with just the particle-specific information. For example, here is a file from which sim and reco jobs are created:

```
#  Script for starting a d0sim and d0reco job
#
#  Attach a list of files in place of a d0gstar phase
#
attach filestream
cfg filestream setinfo simulator None filestream
```

```
cfg filestream define string FilesToProcess fileproc.list
cfg filestream define int MergeFiles 0

#  d0sim is a configurator for the d0sim.x program


attach d0sim
cfg d0sim define string OutputFileName dummy
cfg d0sim define int NumRecords 0
cfg d0sim define int SkipRecords 0
cfg d0sim define string D0Release psim01.02.00
cfg d0sim define string MinBiOpt Poisson
cfg d0sim define float NumMinBi 2.5
cfg d0sim define string MinBiFileName dummy

#  d0reco is a configurator for the d0reco.x program


attach d0reco
cfg d0reco define string OutputFileName dummy
cfg d0reco define int NumRecords 0
cfg d0reco define int SkipRecords 0
cfg d0reco define string D0Release preco04.00.04

#
#  Read in list of filenames
#
cfg filestream make filelist
repeat
reset chain
make job
end
return
```

4. You can now use the command “reg_job ~/xxx/.../sr” from the farm account which will do the following:

- Merge the basic and the particle-specific files into the mc_runjob macro that will be used to create the job, and placed into the new job directory under the name mc_runjob.macro.
- Invoke the mc_runjob linker to create a skeleton job in the createqueue.
- Patch the key files in that new job to conform to your other reg_job arguments, then tar the resulting job into the distribute queue.

INTEGRATING BATCH-QUEUES WITH MCFARM

Presently McFarm will *hand-off* the actual processing task to certain types of batch queues. McFarm is still responsible for determining *where* a job is to be run, because such decisions are highly dependent on cache contents (and also somewhat dependent on other resource considerations known to McFarm). Once the job is submitted to a batch-queue, it is up to the batch-queue to determine *when* that job will actually be launched, and at what priority. McFarm will never submit more jobs to a batch queue than it would normally launch itself on that node (e.g., more than the "max=n" figure in the distribute.conf file).

It is possible that the batch queue will delay the launch of the job. In this case, the job's status in McFarm will be "INITIATING" with a sub-status showing the particular queue's status for that job (e.g., IDLE, HELD, etc.). McFarm will not report the job as STALLED during a lengthy INITIATING phase (as it normally would for a job it launched itself), since it assume the batch queue is responsible for any delay.

The main purpose for integrating McFarm with various batch queues is so that the resources can be shared between Monte Carlo jobs and other jobs. Thus, the point of sharing is the batch queue itself, which is assumed to be operated by the cluster manager. The use of the **–priority=n** argument when creating a McFarm job (see **reg_job**) will allow McFarm jobs to be meshed with other jobs given to that batch queue. A note of caution: each D0 binary that is used in Monte Carlo production will completely *soak* one cpu on the node where it runs. There is little if nothing to be gained by trying to run a Monte Carlo binary and some other high-cpu-user job on the same CPU simultaneously (except perhaps for short jobs, or under Condor's "nice user" paradigm). In fact, it can be destructive to attempt to do so, since you have to allow both extra memory and possibly extra swap space to accommodate such simultaneous tasking, and if you under-estimate such needs, the node can perform *very* poorly due to **thrashing** of the swap space. It can crash completely under extreme conditions. Hence, the batch-queue should be configured to throttle the number of simultaneous jobs when M.C. jobs are accepted into it. This consideration would be just as important even if the job did not come from McFarm originally – it is a reality of high-demand processing. Also note that McFarm jobs usually run with a positive "nice" number, meaning they put themselves at lower-than-normal priority (see **reg_job** and the discussion of priority).

The rest of this discussion assumes you have already implemented a batch-queue – see our web site if you need guidelines for some common batch queues.

Extra Status Information

If a job has been handed-off to a batch queue, its **jobstat** and **check_job** status information will include the results of a query to that batch queue for that job's queue status. For example, such a display might show (PBS_29:RUNNING) meaning that PBS is the designated batch queue, it uses 29 as its number for the job, and its status for the job is RUNNING.

You can query your batch queue directly if you know that queue-number. For example, if the McFarm job name shows CONDOR_123.RUNNING, then the condor_q command could be used to see more details about its job 123. Once a job has been actually submitted to the batch queue, additional queue-specific information can be found in the job directory itself under the file name id.xxx (where "xxx" is pbs, condor, etc.). For PBS, the command used to submit the job to PBS can be found in the /scrNNN/logs/exec.log file. For Condor, the actual "submit" file can be found in the job directory under the name "condor.submit".

Once the batch queue is running on one or more of the nodes that McFarm is allowed to service, the **distribute.conf** file should have one of the following arguments appear on the line that describes that node to McFarm (depending on what type of batch-queue is being used). The queue "master" does not have to be on the job-server.

PBS

For PBS batch queues, the argument must be of the form

batchqueue=pbs:dque@hepfm007.uta.edu

where “dque” is the name of the queue server residing on the designated URL, to which McFarm should submit jobs.

CONDOR

For Condor batch queues, the argument must be of the form

batchqueue=condor:hepfm007.uta.edu/jobmanager

Where the URL has the condor master running, and the name of the desired queue-manager ion that node s given.

Also for Condor, you are allowed to override configuration parameters for job submission. By default, McFarm will set the following parameter values:

Universe=vanilla

Priority = nn (where NN is calculated using reg_job priority PP, which defaults to 5, as follows:

NN = (PP – 5) * -4

(so it maps McFarm priority of 0 thru 10 to Condor priority 20 thru –20)

Nice_user = XXXX (defaults to True if Priority is –20 thru 0, else defaults to False)

Notification = never

Your overrides for these and other values can be given to McFarm in a text file, which must be named in the setup_farm script using environment variable FARM_CONDOR_USER_PARAM_FILE. You may set other parameters in that file, but do not attempt to override any of the following:

Executable

Globusscheduler

Output

Error

Log

Requirements

Initialdir

Getenv

Arguments

Queue

OPERATING HINTS AND TROUBLESHOOTING

Resources for resolving problems include this document, the Bookkeeper document, the various installation guides (all of which are available from d0race.fnal.gov), and the McFarm list server d0-gem@fnal.gov.
Here is some guidance on diagnosing problems:

Make sure you have configured properly (see "Farm Prerequisites" for a list of test-commands that you can issue). Remember that if you change an environment variable, the daemons must be stopped, the account logged out and back in, and the daemons restarted.

Useful information and error message can appear in the following places:

- email - to the "notify" account (per contents of ~/notify_alert)

- Inside the job directory itself:

job.pid - the pid of process_run (if and while it was running)
job.phase - a list of the phases to be run, any input files (other than the last phase's output), and any output dispositions that have not yet been gathered.
farm.log - step by step progress messages as the phases are executes.
error.log - certain python-script errors may show up here.
gather.log - some results of gathering.
gen.log, sim.log, d0sim.log - the standard-output and error-output of the d0 executables themselves, present in subdirectories
progress.XXX (where XXX is GENERATE, D0GSTAR, etc) - shows what event it was on. Available in subdirectories.
Id.pbs, id.condor (etc.) – if a batch queue is involved, this file contains queue info.

- In the "log" directory (there is one on each node: probably named /scratch/log), you can see these files:

lockman.log - from the farm's single lock_manager daemon
lockmonitor,.log – from the farm's sngle lock monitor daemon
dist.log - from the job-server's single distribute daemon.
exec.log - from each node's execute daemon.
gather-XXX.log - for a gather-server's daemon(s), where XXX is the method(s) handled by each daemon, such as SAM or CACHE-SAM).
Acquire-XXX.log – for the acquire daemon

Note that log files that end with a number (e.g., dist.log.23) represent the log of a prior session. The higher the number the more recent the session. They have to be consulted if the error occurred in a prior session (you have since stopped/started the daemon).

- In the /home/mcfarm directory (and later in the job-server's /scratch/log directory):

diagnostic.log – this contains entries that reflect failures of the ssh command, and other internal McFarm commands, and usually reflect communication-type problems between nodes. Once each

Friday, the current diagnostic.log file is moved to the server's /scratch/logs directory and assigned a numbered-sequence for archiving.

- Attempt to isolate problems reported by the D0 executables themselves from farm processing. Once a job is errored-out on, say, node xxx, it will be in the error_queue on that node. You can log into that node as mcfarm, change to the job directory itself, and try to execute the make script directly (e.g., ./sim.make). You might have to squeeze off (one of) the normal processing jobs on that node first, in order to have memory and swap space to run a large D0 executable. (This can be done by changing downward the "max=" parameter in the distribution.conf file, and waiting for a job to end).

If you can reproduce the problem this way, then McFarm software is not involved in the error. Perhaps your links into the "d0dist" and/or "fnal" directories are missing (they must be present for each node).

- Next, try reproducing it on d0mino to see if your network or Linux machine is faulty or inadequate. You can tar up this job directory to encapsulate the job. If it fails there, contact the appropriate d0 support group.

- Any "fatal errors", "orphaned locks", and "deadlocks" may be latent program bugs. If you can eliminate a configuration problem as the cause, contact programming support.

- Be sure to have the lock manager and the monitor running: they are responsible for keeping lock clean, and for various other error conditions that might otherwise silently impede progress.

- When jobs fail for known reasons, such as a node power-off or the farmer interrupting it, the check_job command should be used to restart it instead of any attempt to manipulate directories manually.

- If the farm is having problems, consider doing a "check_job -a" to review all jobs (except archives). If it turns up any problems, re-run to fix the error jobs.

- Is it suggested that you have one gather daemon for each SAM station node, and that one each designated to gather to cache, metadata, merge, and sammetadata. You will also need a "discard" daemon if you do any work where you simply discard some of the output. You may find use for the ftp daemon if you have multiple loosely-connected farms.