InGrid
Framework for Intelligent Grid Control Software
02/04/02

Drew Meyer
University of Texas at Arlington
ddmeyer@attglobal.net

This document proposes a framework for grid control software that would be useful in any grid-based processing solution. This framework calls for a significant amount of new coding, and expects to use standard grid protocols for basic transport and security services (but not higher-level functions). This is perhaps in contrast with other gridification attempts that to examine existing tools to determine how they can be put to use with mainly configuration effort. A skeleton database and control process description is described to give body to the proposed ideas. Use cases are given at the end, mostly for the D0 experiment, and additional use cases and comments are solicited. If the ideas in this design have resonance and can be improved to the point of general acceptance, then a detail design and prototype coding can follow.

Design Criteria

- Keep processors busy
- Minimize bandwidth requirements, avoiding strategies where requirements scale with the number of researchers times the number of input files
- Optimize cache resources to facilitate future processing needs
- Use scalable control techniques
- Applicable to multiple experiments
- Extend easily to future platforms and grid technologies
- Handle all known types of jobs
- Provide for both sharable and private resources
- Provide guidelines for job design that promote efficient use of a distributed environment, but accept legacy code that does not yet meet those guidelines

Key Features

- Put an intelligent wrapper around each running job segment:
  Monitors the binary itself. Responds to control commands and inquiries. Performs pre- and post-job processing (pulling software, forwarding output).
- Use custom-written versions of specific components:
  Specifically, job creation and job wrapper tasks each have built-in experiment knowledge. Isolating job vagarities will reduce the complexity of the overall system and thus coding time. Custom code allows end users to control many aspects of the job, especially job creation.
- Design to require only the minimum set of grid protocols and tools:
  Namely, file transport and process launching are all that is asked of the grid primitives, and proposed protocols for these functions will be embraced. Thus, heterogeneous grid tools can be simultaneously implemented (e.g., Globus and SAM).
- Separate the functions of file selection and file location:
  File selection is expected to be experiment-dependent and use existing catalogues, accessed by custom-written code. File location is handled by a new database, accessed by the new control software.
- Simplify assumptions about job characteristics:
  Use cases so far encountered appear to need only linear chaining, mainline splitting, and mainline merging (see examples).

Design Approach

- The core scheduling algorithm is centralized, other processes are distributed. . Scalability is achieved by supporting this core logic with multiple discovery and disposition processes as needed.
- A single control cluster and set of databases (per experiment) is used for the bulk of the control processes. The databases will mirror the status of the grid itself, with some latency.
- Part of the control software is custom-written for each experiment, and part would be the same for all experiments that use this approach. The intent is to confine job-dependent processing to custom written modules, and allow the main control software to be unfettered by such considerations.
- Processing daemons play a significant part for tasks such as monitoring, discovery, and mirroring
- A custom package wrapper is used to incorporate intelligence at the running job level
- Grid processing is most efficient when binaries can be run in parallel with small output that can be forwarded, but can still be done even if that is not possible.
- Any routine forwarding of large output files is discouraged in order to conserve bandwidth. At the least, a private cache should be the target of the forwarding so that it happens infrequently.
- Actual grid protocols (such as Globus) are wrapped to provide a measure of independence and flexibility for the control software itself, and only basic services are needed so that there is a high probability that any future grid tool can be used in the grid fabric.
- Software versioning is assumed, to provide unambiguous specification of what the job is
- A typical node in the grid requires communication mainly with the control cluster, which in turn communicates with all other nodes and the researcher's desktop. Occasional access to other nodes is required to pull files for replication (and then the other node's URL is known so there is no discovery process needed).

Terminology

- **Experiment**
  Each experiment (e.g., D0, Atlas) will control its own grid, by providing the priority policies under which its job scheduler operates and the description of its clusters.
- **Grid**
  A grid is simply a set of internet-connected machines that are allocated to the experiment for new tasks, and which have been configured to admit such tasks and provide storage. In other words, one experiment has one job scheduler over one grid.
- **Cluster**
  A cluster is any group of one or more processors with access to a storage cache (could be a single researcher desktop, or a large farm). It is assumed that the processors in the cluster can be assigned tasks that access that storage and essentially soak one processor per task. A cluster profile will supply non-volatile information such as OS type, memory sizes, and number of CPUs allocated. A cluster will be homogeneous in makeup, but multiple clusters can reference a single cache (using an allocation scheme). A cluster will belong to a single experiment, and is flagged as either sharable or private.
- **Cache**
  Any available disk space on a cluster would be considered cache, so long as any processors in that cluster can *open* a file that resides in that disk space (for example, using NFS). A distinction would be made between data caching and program/parameter caching. Data caching and purging would be under the control of the central job distribution logic (not by a local cache algorithm since it does not have global view or adequate knowledge of the future needs of the files now in the cache). Program caching would be under the control of the code authority.
- **Code authority/server**
  Any job to be submitted to the grid must reference a code authority that provides binaries and parameter files for one or more of the platforms. Rules for making phases, chaining binaries, and similar considerations are (in this design) essentially built into the custom programs with little attempt made to standardize or categorize them. This design does not require that the code server encode information about a job's prerequisites or behavior before it can be scheduled. During processing, code and parameter files will be pulled by the jobs themselves from the code server. (It is also possible

that code would be pushed to specific clusters - see use cases). Code and parameter data must be made available to a straightforward download approach, installable without user intervention.

- **Jobs, Phases,** and **Packages**
  A researcher will construct a *job*. Scheduling software will construct one or more job *phases* (where a split or merge of data output will mark a new phase), and will construct one or more *packages* in each phase to complete the job in parallel if possible. Examples of jobs are given in the use cases.
- **Package Characteristics**
  Input-Dependent vs. **No Input Dependence**
  Input-dependent packages are those that take the general form of "run (chain of) binaries on (set of) input files". Job package distribution will follow the location of input files (after replication if necessary). Jobs that have no primary input file are also handled: their distribution will be guided by other means (e.g., running a software-update task on a specific list of machines, or doing a calculation-only job).
  Small Output Phase vs. **No Small Output Phase**
  A package would be considered to have a small output phase if any of its chained binaries produce a highly-reduced output file. Such a phase is an excellent point at which to forward and consolidate the outputs from a large number of distributed job packages onto a single cluster for direct access by the researcher. It is best if this phase is close to the end of the chain with little CPU processing remaining. A concatenation task would likely run on the researcher's local cluster or desktop to produce a final output suitable for inspection. Any routine forwarding of *large* output files is to be discouraged in order to conserve bandwidth. One may distinguish between "good" large-output production, which occurs essentially only once per input file, and "bad" large-output production, which might be routinely ordered by any researcher and thus would scale with the number of researchers times the number of jobs.
  Parallelizable vs. **Non-Parallelizable**
  Obviously, the most efficient use of a grid will occur when the bpackage's binary(s) can be run in parallel on different processors. A package that has the parallelizable characteristic allows the job constructor to break that phase into multiple packages. For input-dependent jobs, this essentially means that the binary must be coded to (a) run on subsets of the full input data, producing subsets of total output, and (b) merge those subsets for final analysis. This allows the packages to be sent to wherever the data now resides now, as opposed to having to bring all the data to one cluster for the benefit of a single task. A job phase that has BOTH the small-output AND is parallelizable characteristics can be arranged to forward those small outputs to a final collection point for ultimate disposition. This is where the most efficient use of the grid can be achieved when input-dependent processing is to be done.
  CPU-Intensive vs. **I/O-Intensive**
  A job that will essentially "soak" a CPU during its lifetime should be scheduled where it does not have to share a CPU to any large extent. Conversely, a job that is I/O bound could be sent to a cluster where CPU sharing can and should be done. Binaries will be tagged with this characteristic to aid scheduling decisions, but if two CPU-intensive tasks wind up on the same CPU then the package wrapper will likely suspend one of them to avoid thrashing swap space.
- **Platform**
  Any machine can be a supported platform for some researcher if the code authority provides binaries for it, and if a minimum set of grid primitives can be implemented on it. This implies adherence to security and connectivity standards. If a researcher wishes to run a custom-written binary, they have two choices. First, they could register it with the code authority so that the standard methods can be used to run it. Second, they must construct jobs that simply send input files to their private cluster where they have preloaded the customer binaries. (In the future, more complex implementations of this control software might allow foreign binaries to be included in the job package).
- **Selection Catalogues**
  An experiment will likely provide catalogues (metadata databases) to assist its researchers with input file selection for input-dependent jobs. It is required that files be given unique names within an experiment. Any experiment that cannot provide a good selection catalogue will require much more use of the grid, because jobs may not then be able to specify a priori what input files will be needed. Hence, any experiment with jobs that dynamically determine what input they need should consider

catalogue improvement. Otherwise, in the design presented herein, such a job would have to be given access to every input file in the experiment

- **Location Catalogues**
  Input files can exist in some master location (e.g., a tape robot) and/or in some cache in the grid. A new database is constructed in this design to handle both cases, due to the internal needs of the scheduler to have complete control over this database.
- **Shared resources**
  Each experiment will likely provide funding for processing, storage, and bandwidth resources that are available to any researcher in that experiment.
- **Private resources**
  An individual researcher or group in an experiment should be allowed to fund and implement private processing, storage, and even bandwidth resources that are reserved for their tasks only, and still be able to use the experiment's software for job management.

Communication Methods

For the grid to function, processes must be able to send and receive files between nodes (FTP-like capability) and launch tasks on other nodes (RPC-like capability). All control and application functions that are about to be described can be implemented with just these two capabilities. In order to meet the stated criteria of extensiblity to future grid platforms, this design requires that there be a software wrapper around each *grid primitive* that may be actually used to provide the FTP and RPC services. For example, if Globus is used for the frabric on a set of nodes on the grid, then instead of invoking Globus tools and functions directly, all control software described in this design will instead invoke a *wrapper* function that in turn implements a *standard protocol* to perform FTP and RPC tasks. There are several implications of this approach:

- We do not require that the entire grid be a Globus grid, or any other specific fabric:
  The grid may now, or in the future, have clusters that use different fundamental protocols for transport and security. This is nothing more than a good OO practice: implementing a new grid protocol would simply require writing and testing the proper wrapper functions. (If the grid were to consist of mixed protocols, it would likely be possible to implement gateway machines in a manner that is transparent to the control software).
- It is in the design stage (now) that this decision must be made, so that control software can be written to use *just* an FTP and an RPC capability:
  This design essentially requires a *lowest common denominator* approach to actual grid access, so that the large amount of control software now being written can be easily extended in the future even if a different grid protocol is to be incorporated. Any reliance at this point on a "special feature" of any particular grid tool would reduce the chance of easy future extension.
- This will make it easier to interface with existing data transport functions such as SAM:
  SAM would be invoked whenever a master copy of a file must be extracted from the tape robot (but only to place the copy onto a local disk cache). This is simply a case where an FTP-like service is needed, but the fundamental protocol is not Globus. The control software would simply request the file transport, and wrapper functions would handle the details. To implement, one would place Globus on the SAM clusters to forward the disk file. Although difficult in implementation, it appears to the control software to be a simple FTP service.
- See also the discussion about transport-latency under the section *Distribution Daemons*.

Using this approach requires that a simplified method be employed to communicate between running tasks, since we do not count on having any specific grid or OS signaling mechanism. Sending small command files to a *drop box* on a target node is one way to accomplish this. For example, when a package wrapper process (on a typical node) wishes to notify the job monitor process (on the control cluster) that an intermediate input file has just been created, it would simply send an appropriate message file to a pre-determined directory in the control cache. The job monitor process would pick it up and act on it (e.g., make a change to the location database).

Integration with Experiment's Software

As mentioned above, it is necessary for the experiment to provide the Selection Catalogue. In this design, the only software component that accesses this catalogue is the custom-written Job Creation program. Thus, integration would be complete and seamless for this aspect of grid control, and would take advantage of any and all capabilities that the experiment could provide for file selection (now and in the future).

It could be argued that another design criteria might be to use an experiment's existing Location Catalogue as well, such as SAM now offers. This cannot be done while still meeting the stated criteria of being applicable to different experiments, because use of the Location Catalogue cannot be encapsulated into just the custom components. If this criterion were to be dropped so that SAM capabilities were exploited, then naturally this would become a D0-experiment solution instead of a general solution. SAM would need to be extended in a couple of ways to support the needs of the present design. First, location catalogue needs to record not only the existing location of files in caches, but also the *planned* location from the scheduler. Also, the basic premise that the package wrapper, in this design, is responsible for pulling any needed software assumes that that software can be loaded without human intervention. (The only exception to this so far is the grid primitive itself, which this design considers to be part of the OS profile of the machine).

The present design approach would use SAM only to place tape files into a disk cache that is local to the tape robot, and then the package wrapper (using the standard FTP-like function) would pull the file to the target node. If this design does in fact lead to code that can be shared amongst experiments, the software development effort is both improved and less expensive.

Control Databases

Each experiment will have a single **Control Cluster** that contains control databases and control processes, as shown on the next page, and described as follows:

- Researcher Database
  Contains one entry for every researcher in the experiment. Includes:

  Researcher ID
  Email address
  List of private clusters authorized for this researcher's jobs

- Platform Database
  Contains one entry for every possible platform supported by binaries in the experiment. Includes:

  Platform ID
  Description of OS

- Cluster Database
  Contains one entry for every cluster in the experiment's domain. Includes:

  Cluster ID
  Platform ID
  Sharable or Private
  Total cache space allocated to this experiment
  Percentage of allocated cache reserved for software and parameter files
  List of Software ID's present in software cache (software ID and version)
  List of all input files present in data cache (name and size)

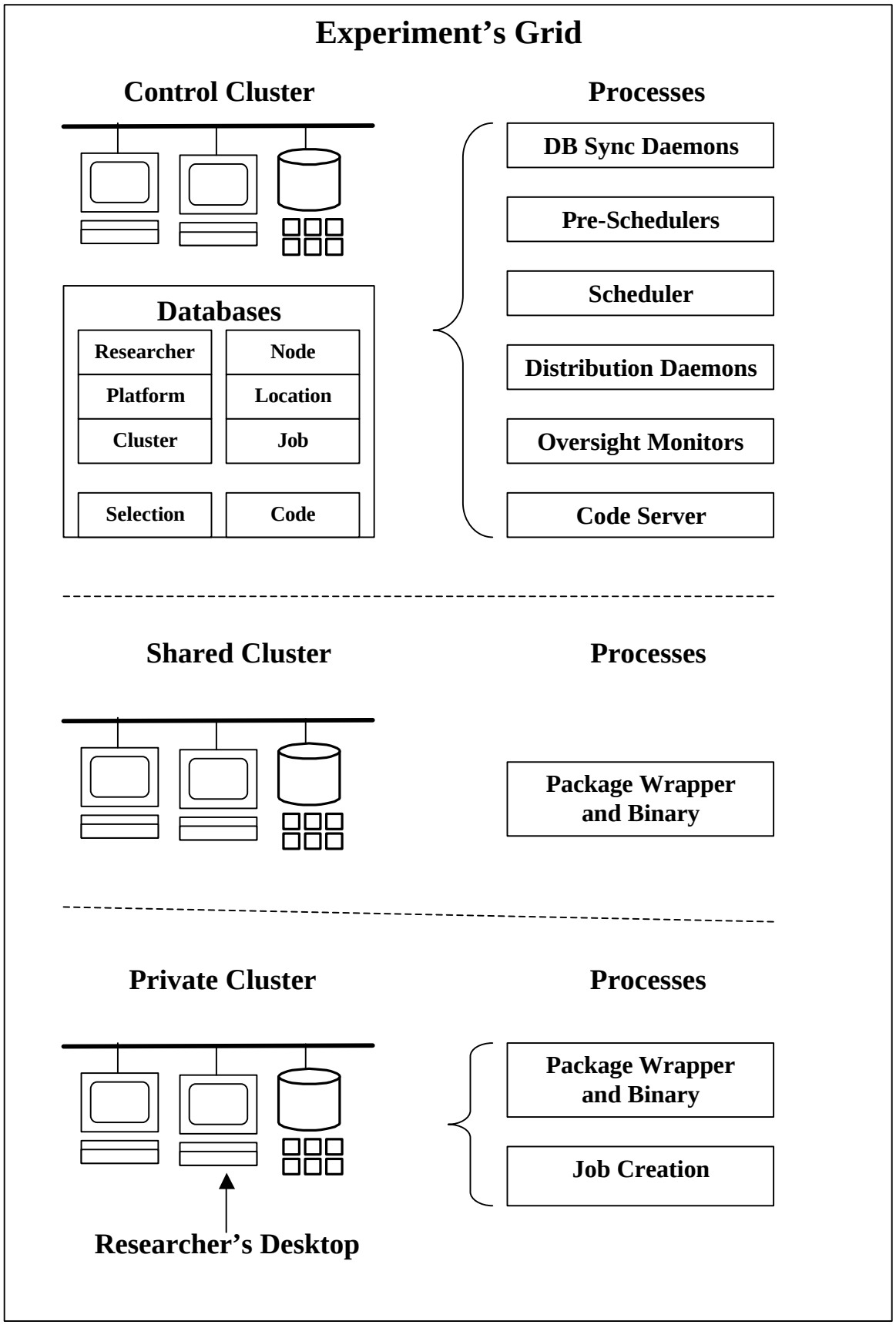
- Node Database
  Contains one entry for every node available to the experiment. Includes:

  Node URL
  Cluster ID
  Number of CPU's allocated to this experiment
  CPU speed for a CPU-intensive task (some universal measure)
  Physical memory allocated to this experiment

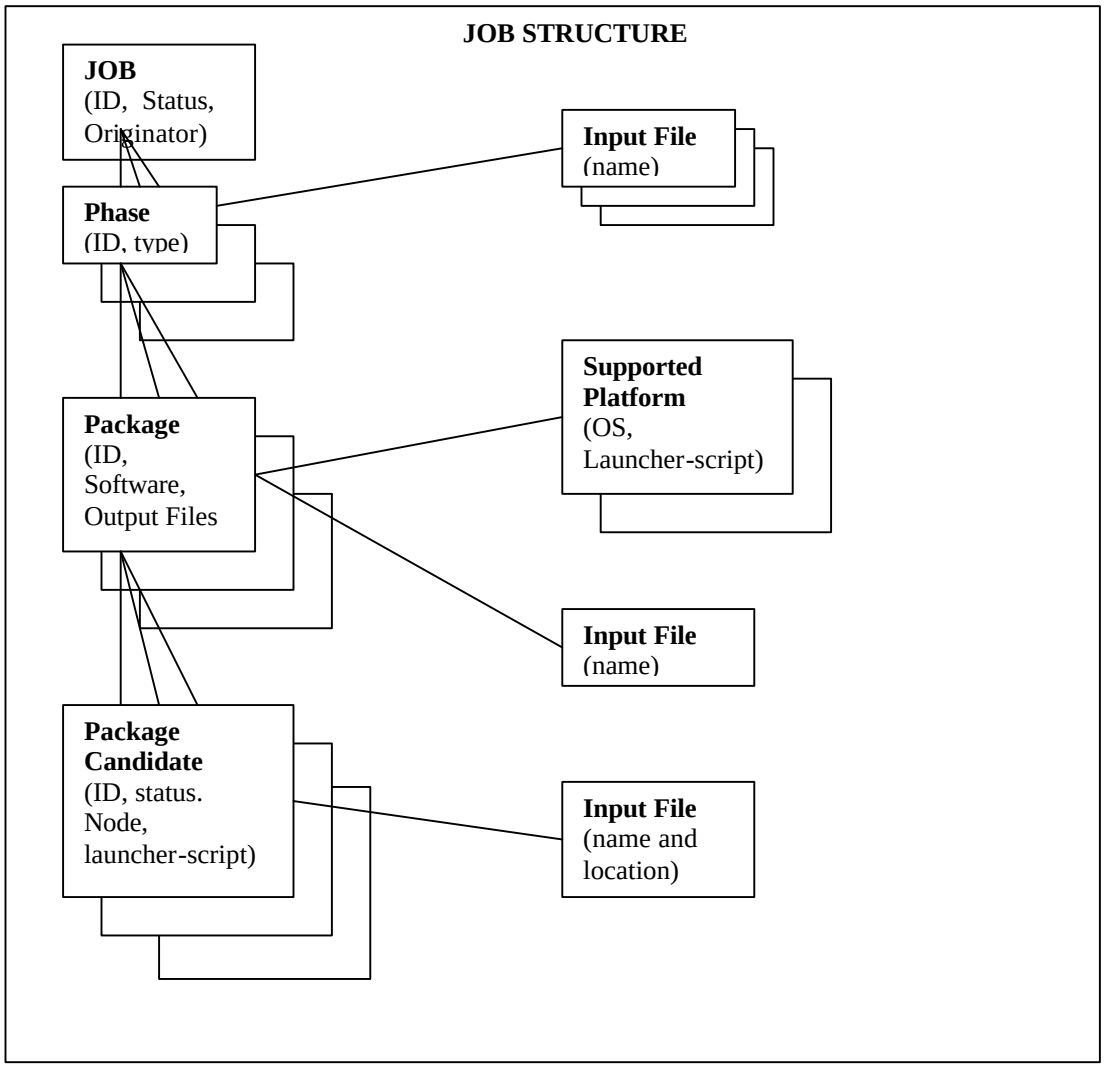
- Selection Database
  This may or may not reside on the control cluster, and is assumed to already exist for each experiment (e.g. SAM). It is accessed only by the job construction process (a custom-written program) to help select input files for a new job. It does not specify where the files now exist.

- Location Database
  Contains one entry for each input file that has ever been or will ever be needed by any job submitted to the grid. It is initially constructed by a database-sync process (see below), and is updated by that and other control software. Input files can exist in both a master location (e.g., the experiment's tape robot) and on some cache on the grid.

  Filename (must be unique)
  List of locations, each entry consisting of:
    Location type (master/cache)
    Cluster ID (if it resides in a cache)
    Sticky flag - set by job constructor to explicitly retain this file in this cache
    Experiment-dependent data (for master locations)


Experiment's Grid
Control Cluster
Processes
DB Sync Daemons
Pre-Schedulers
Scheduler
Distribution Daemons
Oversight Monitors
Code Server
Databases
Researcher
Node
Platform
Location
Cluster
Job
Selection
Code
Shared Cluster
Processes
Package Wrapper and Binary
Private Cluster
Processes
Package Wrapper and Binary
Job Creation
Researcher's Desktop

- Job Database
  Contains an entry for each submitted job. Jobs are added to this database by a job construction program, updated by the job pre-scheduler, job scheduler, and various progress monitors, and deleted by a job monitor when the job is completed. Schematically, a job would look like this:



Information for each job includes:

Job ID
Current Status ("to be pre-scheduled", "to be final-scheduled", "scheduled")
Originator ID (researcher)
Priority
List of Phases (any split or merge of processes requires a new phase), each entry contains:
  Phase ID
  Phase type: first, merge, or split
  Early-start (yes/no) – if yes, then package(s) in this phase will be started without waiting for all packages in the prior phase to complete
  Require 100% success of prior phase (yes/no) – if this phase did not have early-start capability, then it might require all output of prior phases.

List of Packages, each entry contains:
Package ID
Software ID (this identifies the operation to be performed by this job, e.g. "reco")
Software Version
Minimum number of CPUs required
Input Parameter Set (experiment-dependent)
Input FileNames (required if this job is input-dependent).
Early-start (yes/no) – if yes, then this package will be started without waiting for all input files to be present.
Output FileNames
Estimated max output
List of platforms supported, each entry contains:
Platform ID
Package wrapper Script ID (identifies the script to be used to launch the package, which is software-dependent, version-dependent, and platform-dependent)
List of candidates to instantiate this package (only one is chosen), each entry contains:
Candidate ID (unique identifier for a candidate choice for where to run this package)
Current Status ("to be chosen", "to be distributed", "rejected", "running", "completed"). All candidates but one will be "rejected" by scheduler.
Node ID (where this candidate would run or is running)
Bandwidth Cost (how much bandwidth is needed to run this candidate)
Processing Cost (CPU cycles needed to run this candidate)
Script ID (identifies the script to be used to launch the package, which is software-dependent, version-dependent, and platform-dependent)
List of Input Files (if any), each entry contains:
Input FileName
Source Location (if replication is necessary) – can be master or cache location
Estimated or actual start time
Estimated or actual end time
CPU-intensive or not
Profile of disk space requirements (timed)

Control Software

The following processes will be use to create, schedule, monitor, and clean up the jobs and packages.

Job Construction

- Runs on the originator's desktop, and accesses control databases using intermediary processes on the control cluster, and accesses the selection catalogue.
- A custom job construction program would be written for each experiment to handle each type of job. This is the first place where experiment-dependent logic is handled by a custom program (the other is the package wrapper). The jobs will be *normalized* before submission to the job scheduler, in the sense that phases will be constructed and packages and chaining identified.
- It is interactive whenever an individual researcher is using it (e.g., construct an Analysis job), and scripted whenever mass job construction is required (e.g., Monte Carlo production).
- Some job construction will center around the listing of the input files, while others would center around the specification of clusters on which to run some specific task.
- It has access to the selection-catalogue whenever jobs are input-dependent. It also has access to the location catalogues, but generally speaking this is not the time to find the best location to run this job. Access is provided in the event that this program wishes to restrict processing to specific nodes or pre-existing files.
- It can if necessary force a job to run on a specific cluster, or it may defer that decision to the job scheduler. Cluster information is obtained by a query to the cluster database.

- Job construction should be painless. One should be able to clone a new job using a prior job as a model. Macros and templates should be easily available. A GUI interface would be appropriate for interactive access. Lists of input files should be easy to introduce, since a researcher may already know from other sources what files to use.
- The final output of the job construction process is an entry to be added to the job database, flagged as "to be pre-scheduled", and containing all information except for candidates.

Package Wrapper

- Runs on each node where some package of the job is to be processed
- A package is the smallest processing component of a job. Packages start with a custom script that launches the chain of binaries: this package wrapper is the instantiation of a job package. For input-dependent, parallelizable jobs, there would be one package per input file in a given phase. For jobs with no input dependence or that are don't have the parallelizable characteristic, there would be one package per target cluster.
- The custom written package wrapper will *normalize* the job package so that all packages on the grid can have the same apparent capabilities and characteristics, as seen from the other control software, regardless of what the binary actually does.
- Each package wrapper is custom written for an experiment and type of job to do the following:
    1. It will initially consist only of a small machine-independent script (e.g., python) that is sent to the target cluster and executes on the target CPU. This script simply invokes the full-sized package wrapper for this type of job (if not already in place on the target cluster then it pulls from the code server). It executes this full package wrapper supplying the necessary job-specific parameters. This implies that each target platform have the ability to run python as a prerequisite.
    2. The full package wrapper (e.g., a C++ program with better run-time behavior than python) will pull any additional software and parameter files from the code server down to this cluster. This is an optional capability: an experiment could have opted to push all software to the clusters beforehand.
    3. It will investigate the possibility that it could start at some binary other than the first one in its planned chain. This could be done if some intermediate file was still present in the cache. The job scheduler would have known about this file also, but this package would have been built to perform the entire chain in case it has to be restarted elsewhere later.
    4. If the job is input-dependent but the input file is missing, then it will pull the input from whatever node the scheduler told it to use. The scheduler would not have placed the package here unless the input file was already here, or available from some other source. If the file is not where it was supposed to be, then the package wrapper would terminate with a suitable error and the job scheduler would try again to locate the input file for it. The acquisition of the input file could take quite a lot of time.
    5. It will launch each binary in the chain (at a lower priority level than itself), after it ensures that the CPU is mostly available for this binary. After each binary finishes, the package wrapper will verify that it completed successfully before proceeding to the next. This verification clearly requires significant knowledge about the binary's characteristics
    6. While each binary is executing, the package wrapper will perform periodic monitoring tests to ensure that it is not looping, that it is not about to run out of disk space, and that the swap space is not being overwhelmed. In particular, it will be aware of other grid tasks that are presently on the same CPU, and will attempt to prevent thrashing by suspending its binary
    7. The package wrapper will accept requests from external nodes to provide job status information such as percentage completed and estimated completion time.
    8. The package wrapper would allow external tasks to request that it terminate the binary early, possibly saving the output produced thus far. This requires the binary to have been coded to accept a certain level of communication (see below).
    9. Upon completion of the last binary in the chain, the package wrapper would if necessary forward the output to a designated node. It might also now purge any intermediate files left in the cache (if they were designated to NOT be re-useable by the job originator). Any files that were left in the cache might be purged later by the job scheduler (not by any local cache controller).

10. A final status message would be sent to the experiment's monitoring process before the package wrapper finally cleans up after the binary and ends. The status message, successful or not, also notifies the job distributor that a CPU has been freed up

- The binaries themselves (from the code authority) should be re-coded as necessary to support certain features. Note that none of this requires that the binary be "grid aware" in any way, or to be recompiled with any special API or library, and this does not have to be done on day one.
    1. The binary should periodically output a line to a progress file that shows in some way what is has processed so far (e.g., what event it is on). This file should be opened and closed each time to avoid buffering problems. The package wrapper will inspect this file, and its time stamp, to assess current progress and predict the time of the next progress message. If the binary does not operate on discrete chunks of an input file, or has no input file at all, it should still produce some periodic output, to show whether or not it is still running.
    2. The binary should periodically inspect a pre-defined *command file* to see if the package wrapper has asked it to pause or to terminate. Termination should be done, if possible, only at points where the output so far produced is useable (for example, between events). Pausing can be done at any point. The binary should append a compliance message to the end of the command file, with the time that it complied, so the package wrapper can tell it has complied. Pausing should be done by some simple method such as sleeping for a few seconds (e.g., not a "spin loop", and something low-tech that is supported by all OS platforms). It can resume when the command-file disappears (when the package wrapper removes the pause injunction).
    3. If the binary detects any error that it cannot overcome, an appropriate message should appear in a log file. The package wrapper can be coded to look for specific messages, interpret them, and relay them to the job scheduler. The scheduler has to make a determination as to whether or not this chain is to be restarted, and specific messages can assist it in making that determination.
    4. In the event that a researcher might be interested in just a part of an input file (e.g., certain events) the binary should have a mechanism that can be used to direct it to process just the interesting segments of the file, skipping over the rest. A special parameter file would serve.
    5. A standard tool to concatenate each type of file would be helpful.

Database-Sync Daemons

Some control databases (the location database and parts of the job and cluster databases) contain information that mirrors the state of the grid, which is constantly changing. A variety of techniques will be used to keep these databases accurate. First, any grid process that alters the state of a node or cache will immediately effect a change in the appropriate mirror database. For example, the package wrapper would notify a sync daemon process (running on the control cluster) that a new file had been created, and that sync daemon would update the appropriate database. Second, the sync daemons will themselves periodically discover changes to the state of a node or cache that may have occurred outside of grid control (e.g., a file being manually deleted from a cache). This sanity check that can be done frequently as bandwidth allows.

Job Submission/PreScheduler Daemons

A set of processes on the control cluster handle the chore of picking up a new job that is being submitted by a job creation program, editing it, and performing the pre-scheduling duties. They will reply to the job constructors with an accept/reject message (and quickly, because they are waiting). Only clean data gets into the job database, and this is a useful point of control for things like priority assignment. Rejection could be due to badly formed requests or referencing input files or nodes that do not exist anywhere.

Jobs that are not input-dependent would already know the target nodes because they were determined during job creation. For input-dependent jobs, pre-scheduling includes a discovery of every location of every needed input file (accessing the Location Catalogue) which provides a list of clusters that already have the input file present. The pre-scheduler attempts to apply as many scheduling rules as it can without having to know anything about the present global state of the grid. For example, a job might have been forced by policy to run on a specific private cluster, hence the pre-scheduler will have

to put all packages onto the designated cluster. Certainly, only those clusters with the proper OS profile (Platform ID) are to be considered. Basically, the pre-scheduler attempts to eliminate as many cluster options as it can, so that whatever is left is essentially an optimization decision that is best left to a single process with a global view. Thus, it will prepare a list of *candidate* packages that could be sent out to accomplish the processing of a given package, and assign a cost to each. *Cost* might consist of a certain amount of data transport, a certain amount of re-processing to create the needed input file at some cluster, and so forth. The pre-scheduler passes the job onto the scheduler by changing the job's status to "to be scheduled".

Job Scheduler Daemon

There is a single core job scheduling process for the entire experiment. Having a global view, it examines the jobs needing scheduling in priority sequence and makes optimization decisions from the candidates that the pre-scheduler processes provided. Then the actual launch of each package is left to the job distribution processes. The design calls for the load on the core scheduler itself to be light enough that it will not become a bottleneck.

There are many worthy criteria that could guide optimization decisions, some of which are mutually exclusive:

- No CPU should be neglected
- Cache purging should be minimized, because a purged file might be needed again, but there must be room for the expected output file(s)
- Commonly used input files should be replicated in more than one place to promote more options for future processing (other clusters). At the same time, a file might be a good candidate to purge if it is replicated already.
- One package might have a lower cost than another, although the weighting of various resources (bandwidth vs. storage space, for example) is likely to fluctuate with the state of the grid.
- If most of a job can be scheduled in the near future, then any remaining stragglers should be scheduled as soon as possible regardless of cost so that job completion does not drag on.
- It will in general schedule only one phase of a job at a time. Packages in a phase that follows a split can be started as soon as their input file is available (the output file of the prior phase). Packages in a phase that follows a merge may either be started early (if they have the capability to wait for input to arrive) or may have to wait until all output is available (otherwise).

The scheduler needs accurate information about the current status of each cluster. In this design it can quickly access this information from the various mirror databases:

- What packages are running there, when they will finish, how much disk space they will use at maximum
- What other packages have already been scheduled for a cluster, but have not yet been initiated
- How much space is left in the cache (allocated to this experiment)
- How many processors are available (allocated to this experiment) and what types of jobs are running on them (are they soaked or not)

It is premature to attempt a detail design on the job scheduler at this point. Most likely, it will undergo repeated revisions before it performs satisfactorily. However, even early versions should be able to load up the processors with *some* set of packages. It should be noted that it can theoretically make very good decisions about where to run packages, because it has sole responsibility for those decisions and access to all pertinent information.

Also note that it is important that the pre-scheduler and scheduler not get too far ahead of the current state of the grid. On the one hand, an empty CPU somewhere in the grid should trigger the scheduler to attempt to find a candidate package to send there (and *empty* wold include a CPU that has a package running but which is currently waiting for input to become available). On the other hand, any

scheduling that is based on predicted end times of running packages could become stale as the grid state changes in unexpected ways. To address this issue, it is likely that jobs will need to be re-scheduled several times before they finally complete. (As a specific case, consider the need to reschedule a failed package). It is likely that it will simply reschedule all uncompleted packages periodically and routinely. There are some advantages to scheduling a job as soon as it comes into the system, even if the start times are only estimates based on running jobs. First, the originator can be given an estimate of the completion time soon after the job is submitted. Second, by looking ahead as far as possible, cache purging decisions can be made more intelligently.

Package Distribution Daemons

Once the job scheduler has made the packaging decisions and targeted them to specific clusters, it is necessary to launch each package on the target node. A set of distribution daemons (as many as necessary to handle all clusters for this experiment) will run on the control cluster and perform this task. Each will do the following:

- Wait for one of the nodes that it services to have a free CPU (or, for I/O bound packages, for a CPU that still has some capacity remaining). Identification of a free CPU will initially come from the Cluster database, and will be known immediately when any job packet ends.
- Take the next package that is flagged as "to be distributed" to that cluster and verify that it is not awaiting some prior input or phase completion. If it is ready to roll, start its package wrapper on that node and flag that package as "running" in the job database. If it is not ready, flag it as :to be re-scheduled".
- The distribution daemon knows whether or not the launched package will have to first pull an input file before it can occupy the target CPU. It if does, then the package would be launched even if the target CPU was already soaked with a prior package. However, this new package would be configured to perform the pull processes only (at a slightly higher priority than the binary runs), and not to launch its binary until the prior package was finished. There could be multiple packages on a CPU doing their software pull phase only (an I/O bound process), and one that is actually running a binary (the package wrappers will coordinate with each other). This is intended to handle to possibility that the replication of the input file could take significant time, (if, for example, it had to come from a tape robot).

Package Oversight Daemons

A set of processes will run on the control cluster to monitor running packages. They will do the following:

- Accept and handle error messages from the package wrappers. Some error conditions could imply a restart, and others could imply no restart.
- Accept completion messages from the package wrappers
- Periodically query the packages that are running to obtain updated estimates of completion time and cache usage, and place this information into the Job Databnase.
- Accept and handle job-status requests from the originator and grid managers, using informatin form the job database.
- Accept and handle job-termination requests from the originator and grid managers. As mentioned above, this would be effected by a communication with the package wrapper that in turn communicates with the binaries. Terminations would be of two forms: retain partial output or not.

Upon any form of completion (success or failure), the daemon will flag the package as completed in the job database. If this results in the job itself now being completed, then the originator is notified by email (or for mass-production tasks, some automated method), and the job is completely removed from the job queue database (and likely placed into an archive database for post-processing analysis).

A failed package that is to be restarted will be re-injected into the pre-scheduling queue. This would result in that job's status changing to "to be pre-scheduled", so the pre-scheduler and scheduler will pick it up again (but of course they will not make changes to completed or running packages). A failed package that is not to be restarted would not be re-injected, but the (eventual) completion message to the originator would show it as an uncompleted segment, and why.

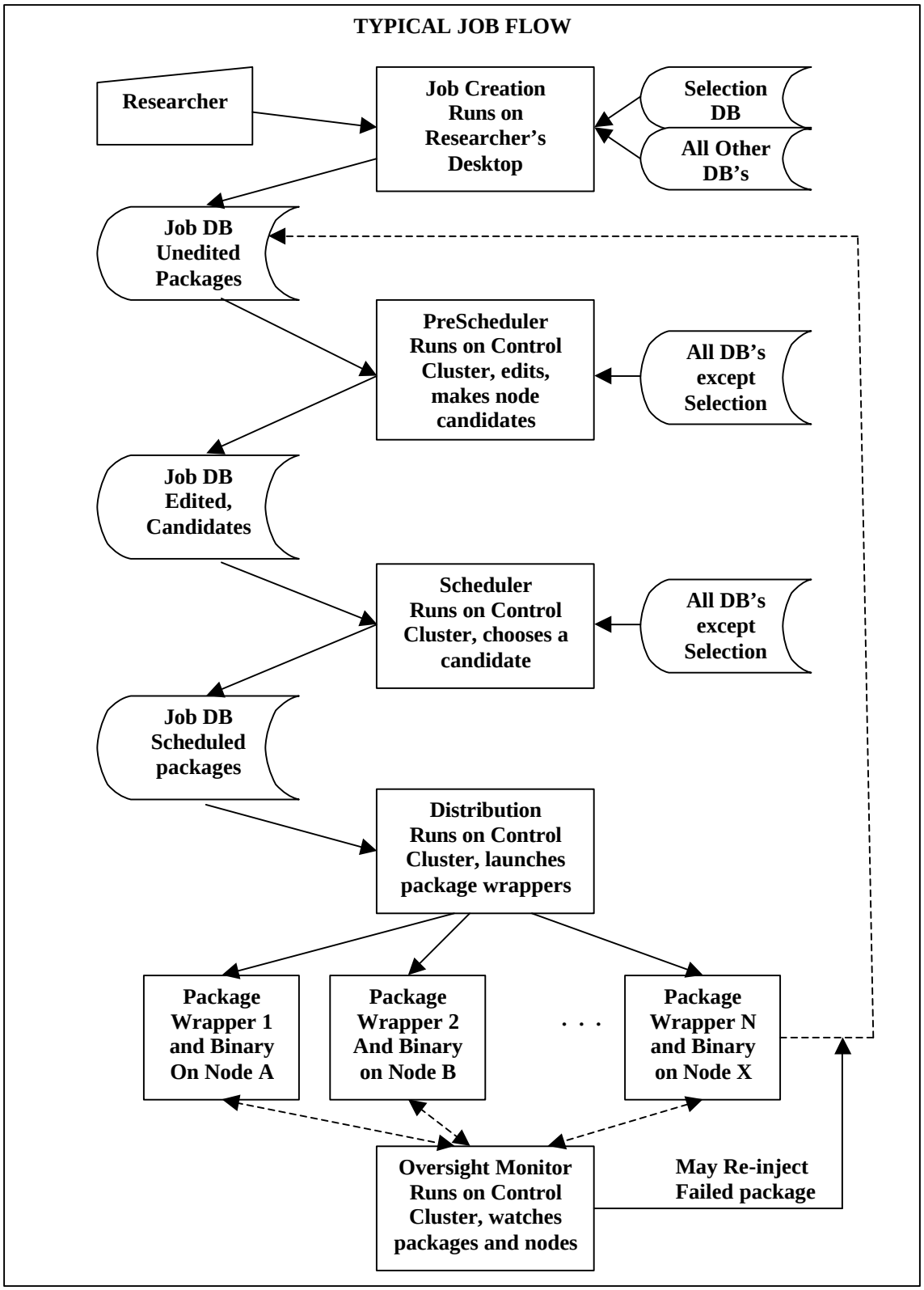
Response to Failures

Control software must automatically handle failures of job packages, nodes, and caches. Job packages can fail for reasons having nothing to do with the job itself, and thus might not be expected to occur again if the job were restarted. Examples include running out of disk space or a failure of the machine's OS that requires a reboot. Other job failures would be likely to happen again (e.g., a segmentation fault to due to input data flaws). The binaries being run in a package might be able to detect some failures, and if so some attention should be given to categorizing them so that the package wrapper can decide whether or not to request a restart.

Some failures, such as a machine losing power or a binary in a loop, can only be detected by an external monitoring process. This design calls for the package wrapper to monitor for loops (using progress files and knowledge of the binary's characteristics). The design calls for an oversight monitor on the control cluster to periodically discover downed CPUs (power loss) and packages that are on a node but no longer running (power loss followed by a reboot). Looping conditions would likely be treated as non-restartable, while power failures would of course be considered restartable.

Caches can fail by becoming full. The job scheduler has the initial responsibility for clearing out cache files for any job that it submits, using the estimate of maximum output for a package. In the event that the cache still starts getting full during actual processing, the package wrapper would be coded to suspend the binary, query the control cluster (Package Oversight Daemon) to obtain a list of files that can safely be purged, purge one or more of them, then resume the suspended binary.

Job packages that are not to be restarted are simply ignored for the rest of the life of the job, and the researcher is notified that they failed and why. In many cases, the researcher will not need 100% of the job packages to succeed, so output from other packages (or even partial output from this one) would likely still be presented to her.

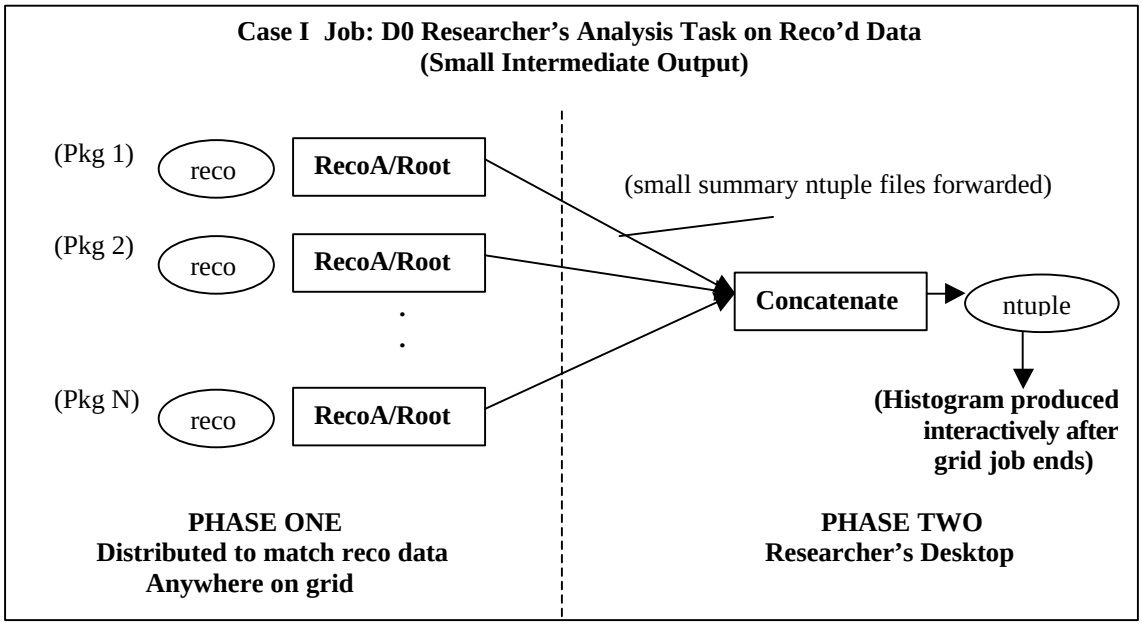


Not shown: DB sync-daemons keep databases in sync with state of the grid, and researcher or manager status queries are submitted to the DB's and oversight monitors.

Use Cases

Case I. D0 Researcher's Analysis Task on Reco'd Data, Final Output is Histogram. Researcher configures root to produce small output at the end of each reco/root chain.

Job characteristics: input-dependent, small-output phase, parallelizable, CPU-intensive



For the first phase of the job, a list of, say, 100 reco files is submitted to a job constructor by the researcher, and a chain of two binaries (reco-analyze and root) is to be run on each file. The root output is structured to be highly-reduced (small output). For the second and final phase, a concatenation task is required, designated as I/O-intensive, and constrained to be performed on the researcher's local cluster or desktop. This task is coded to have the intelligence to wait for input files as they appear, so it can be started before 100% of the output from phase one is available. The pre-scheduler locates the 100 input files wherever they reside (possibly finding recoanalyze files that might still be in a cache from a prior job), and makes up 100 packages to perform each recoanalyze/root pair. The scheduler plans for one chain per CPU since this phase is CPU-intensive, plus a second phase with one concatenation task that starts immediately and waits for input (the researcher could inspect partial results).

Each of these 100 packages, which each consists only of a small script, is launched on a target node and begins execution by locating the necessary binaries and parameter files in the local cache. Each will pull any missing files from the code server, and in fact may have to pull its primary input file from another location, before launching the chain of binaries. (That last step would not be necessary if the job scheduler was able to place this segment on a cluster that already had the input file). This script should be smart enough to determine where in the chain it has to start (e.g., perhaps its recoanalyze file already exists here).

During execution, the package wrapper implements two important services. First, it periodically monitors the binary's progress to detect a loop and to verify successful completion. Second, it responds to queries from external sources for job status and also handles requests to terminate early (perhaps from the original researcher). It is custom-written for this type of job on this experiment.

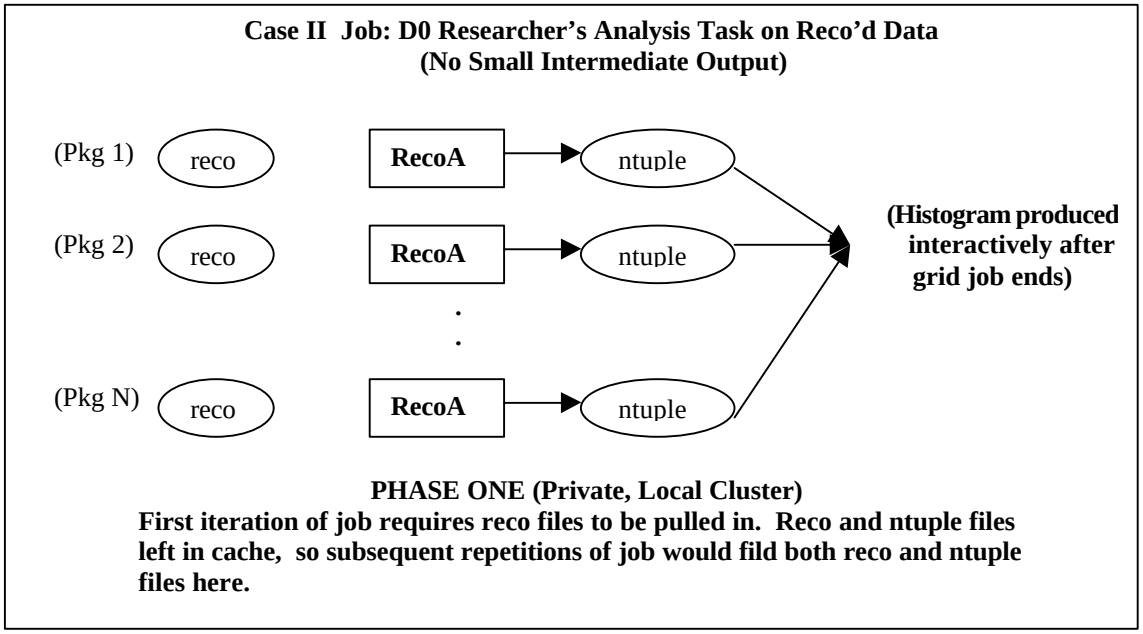
Each of these packages ends by forwarding its (small) output to the user's desktop and purging them from that cluster's cache. The scheduler has launched the final concatenation task already at the user's desktop. It will share a CPU if necessary, since this task is not CPU-intensive. It simply awaits the arrival of 100 output segments, which it concatenates as they arrive. When all 100 have been received, the user is

notified (email). The histogram is created using a local task (outside grid pervue), with the concatenated output file to which she has local access.

Any intermediate files that had to be created are probably left in the caches, and are subject to being purged since they are replications. The job constructor could have asked the researcher if she expects to need the same input files in a future job, to facilitate better cache control by purging them now if they will not be re-used.

Case II. **D0 Researcher's Private Analysis Task on Reco'd Data, Final Output is Histogram**. Researcher does NOT employ a small-output approach

Job characteristics: input-dependent, no small-output phase, parallelizable, CPU-intensive

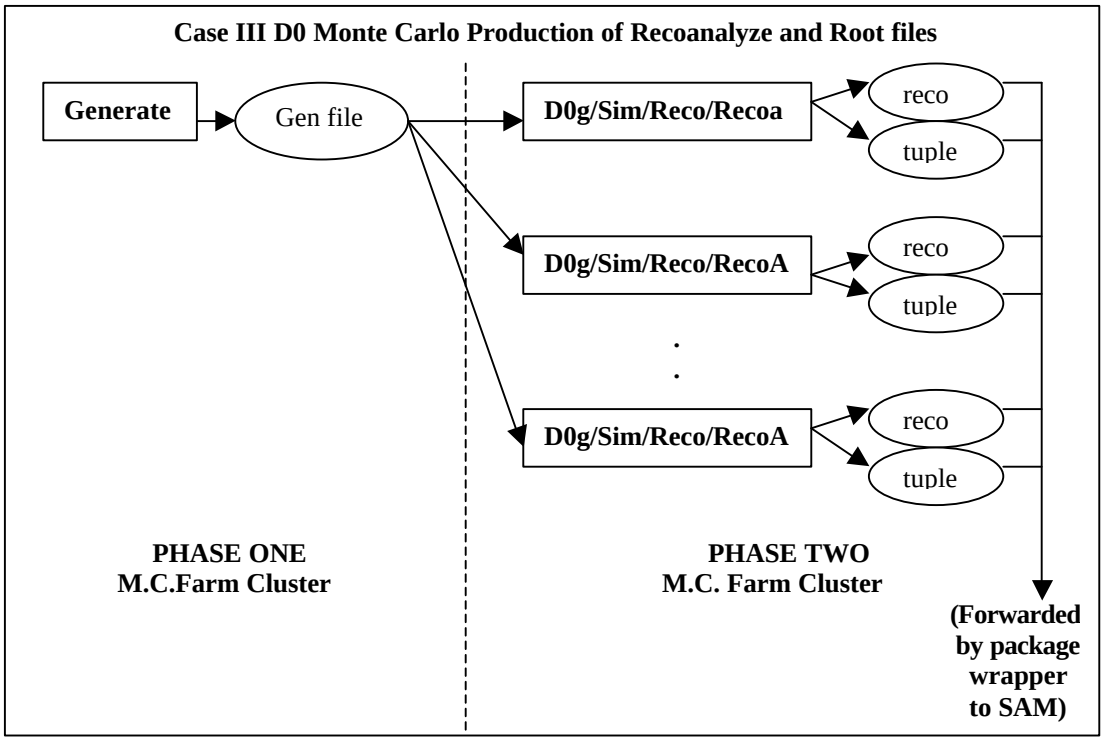


The same list of 100 input files is supplied to the job constructor, but it is not allowed to employ forwarding of highly-reduced output from which a histogram can be produced. The best remaining approach is to replicate the original reco files on a cluster that is local to the researcher (probably a private cluster) since this can (hopefully) be done once and then re-used for subsequent jobs (see Case IV). It would then force the 100 job packages to run on the researcher's local cluster (in parallel to the limit of local CPUs). There, they would initially copy-in any missing reco input file (but only once) and then run recoanalyze. The final histogram production would then be run interactively by the researcher after the ntuple files were present, but here the large (and now local) root files are used as input. Assuming the local research group reuses these locally resident reco, recoanalyze, and/or root files over and over, this would be an example of "good" large-output jobs because hence the bandwidth needs do NOT scale to the number of jobs.

An alternative approach for this case would be to run the chain of binaries on non-local clusters (wherever the reco files presently exist or on any shared cluster), and then forward the (large) root files to the local cluster for histogram production. Note that the reco and reco-analyze files do not exists locally after this is done, so any future job that uses the same reco input cannot avoid having to send the large root files to the researcher yet again. This is an example of "bad" large output, where the amount of data being forwarded is scaling to the researcher and her number of iterations of this job. The lesson is that binaries and jobs must be designed (or re-designed) to employ a small-output intermediate step, or that locally-controlled clusters must be implemented and input files re-used.

Case III. **D0 Monte Carlo Production of Recoanalyze and Root files**

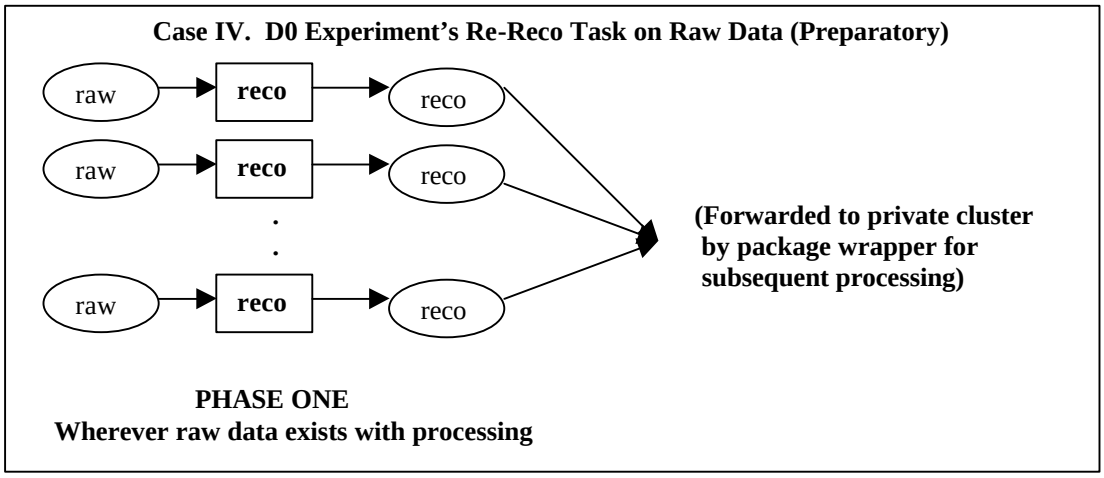
Job characteristics: no input file dependence, no small-output phase, mostly parallelizable, CPU-intensive



Assuming one starts with a generator binary then there is no primary input, so this job has no input dependence. There will be a large amount of secondary input in the form of minbias-files for the D0Sim binary, but minbias files should probably be treated as parameter data instead of a input data, and hence they will be present along with the software. The most efficient way to handle this is to push minbias data to the cache of some special cluster (a "Monte Carlo farm"), then during job construction, specify that cluster as the only allowable cluster for the new jobs. The number of CPUs dictates the amount of Monte Carlo production allowed. The large output is "good" in that it occurs only once per job, when it is forwarded to the central storage and then purged, and bandwidth requirements will be predictable. One could construct a number of Monte Carlo jobs using a modified version of mc_runjob and submit them to the job scheduler, which would queue them for this farm only. Each job would have two phases: a (non-parallelizable) generate phase that produces a single large "gen" file, then a split is structured and many d0gstar / sim / reco / recoanalyze chains packages would constitute the second phase, each with a unique parameter set. Being parallelizable, CPU-intensive jobs, they would be distributed as CPU's became available. The job package would purge that package's output files, after forwarding as required.

Case IV. **D0 Experiment's Re-Reco Task on Raw Data (to be followed by multiple recoanalyze jobs)**

Job characteristics: input-dependent, no small-output phase, parallelizable, CPU-intensive



This is similar to Case II because the output is large, but differs in that it is the large final output itself that is desired by a researcher, so there is no final concatenation phase. The reco tasks can be run anywhere that the raw data exists (or can be replicated), including but not limited to the local cluster, and the reco files are forwarded to a local cluster. (It is suggested that a local cluster should be involved so that the reco output need be forwarded only once). This makes it a case of "good" large-output. Additional jobs can now be constructed to analyze the local reco files (see Case II).

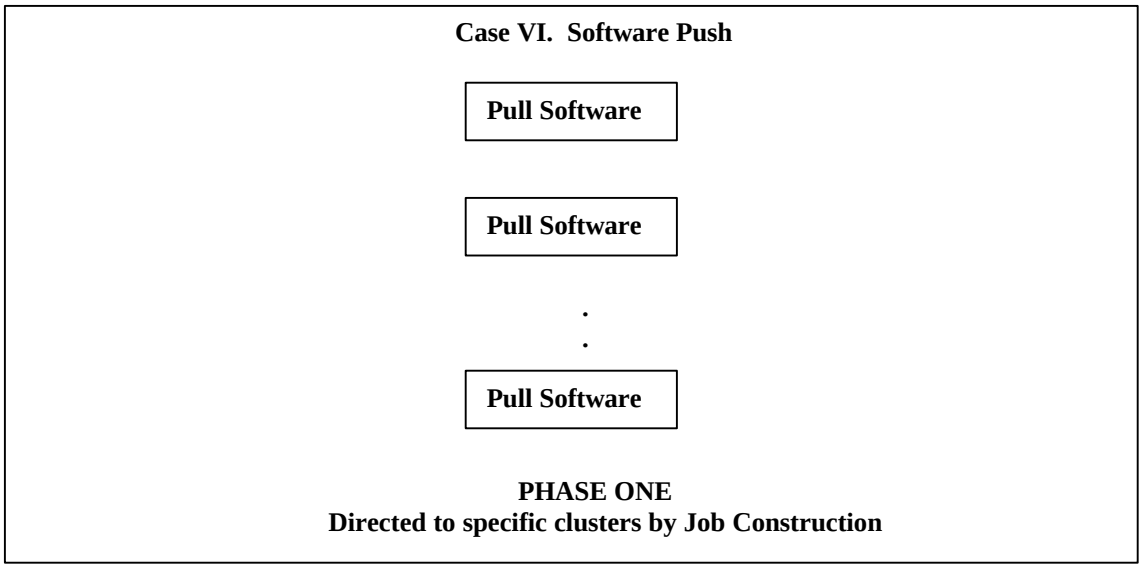
Case V. D0 Experiment's Re-Reco Task on Raw Data, to be done multiple times

Job characteristics: input-dependent, no small-output phase, parallelizable, CPU-intensive

If it is the reco phase itself that is to be run multiple times, then the raw data should be replicated at a local cluster. This case is thus the same as Case IV, except that the reco jobs are forced by job construction to run on the local cluster only.

Case VI. Software Distribution

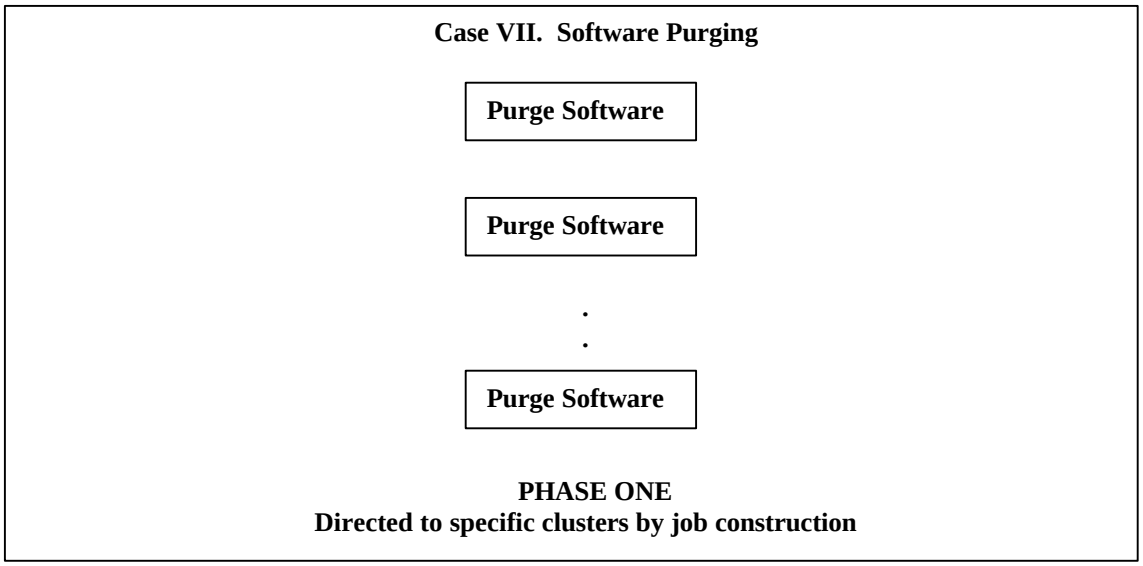
Job characteristics: no input dependence, has small output phase, parallelizable, not CPU intensive



Two methods can be used to put software and related parameter files to grid caches. First, the job packages can be coded to automatically pull any software that they require just prior to launching. Second, software can be explicitly pushed to specific nodes (as shown above, where packages are explicitly sent to targeted clusters where they pull in software). Both methods can be implemented for the same experiment, so long as the job scheduler knows whether or not a job's package has built-in pull logic (if it does not, then it may distribute only to clusters that have the proper software already). The pull method has several desirable characteristics (naturally distributed, no manual component to fail) but both methods have their place. If the code authority wishes to push a new software version to specific set of clusters, then a custom job-construction program would be coded to allow them to specify the necessary information, including a specific list of clusters and the desired software version. The job scheduler could then distribute these pull-packages to those clusters (without waiting for a free CPU, since these packages are not CPU-intensive). Each pull package would simply pull the specified versions to its cluster's program cache. The final output is small – simply a report of success or failure to the originator.

Case VII. Software Purging

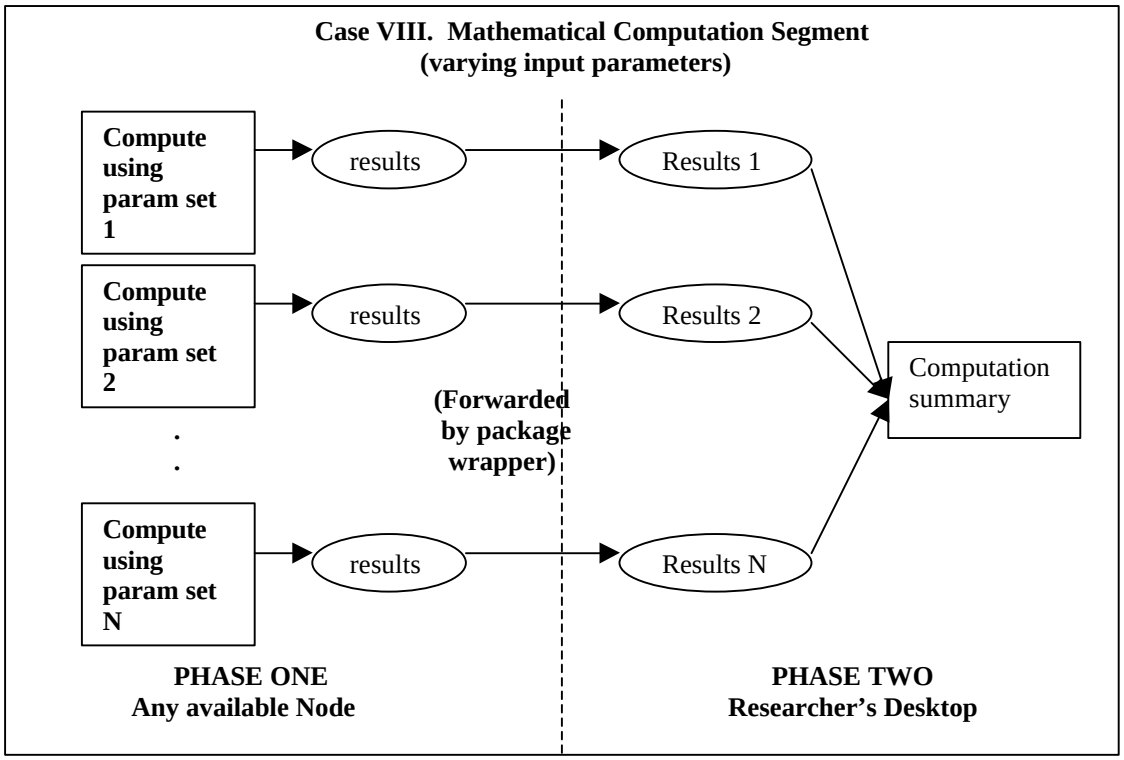
Job characteristics: no input dependence, has small output phase, parallelizable, not CPU intensive



When a software version becomes obsolete, it should be purged from all program caches throughout the grid. It is probably best to make the job construction program itself discover the locations of the obsolete software and then submit the list to the job scheduler. This allows the code authority to tweak the list, it they wish to retain the software somewhere. The job scheduler would then be asked to create job packages for each listed cluster that simply purged the obsolete software. Again, this could be done without waiting for a free CPU.

Case VIII. Mathematical Computation

Job characteristics: no input dependence, has small output phase, parallelizable, CPU intensive



Assume that some computational task can be divided up into multiple tasks, each given a unique parametyer set to do a portion of the computation independently, and that the small output of each can be later consolidated to make a final result. Here, there are no restrictions on what nodes can be used (assuming the platform is supported) – any free node will do. The job construction program would therefore omit the specification of a target node when submitting the job, so the pre-scheduler would be allowed to choose any suitable and free platform. Note also that here, each package would likely have a different input parameter set (as contrasted with the earlier cases, where each package had the same input parameter set). The first phase would thus consist of multiple uniquelyparameterized packages, and the second and last phase would be the consolidation task (likely not started until all prior results were available to it).

Next

- Review by the user community, and ensure that a complete set of use cases is included:
  Additional use cases are requested at this time, as are any and all comments from the user community. Use cases from experiments other than D0 are encouraged as well, as this will help mature the design.
- Construct a detail design and estimate programming time:
  A detail design would describe the control databases, interface protocols, and software component functionality to the point that they could be coded. Programming estimates could be made for the prototype step and (roughly) for the coding step.
- Prototype extensively to refine the design and prove the concepts:
  Basically, shortcuts and simplifications are employed to focus on the unknowns in the design, such as how the grid primitives are best implemented. A working set of software would be coded using a language such as python. Databases might be implemented with simplified engines since heavy volume is not yet encountered. Aspects of the use of grid primitives would be fully investigated and handled. Job creation might be scripted instead of GUI. At the end of this phase, one or two use cases would be demonstrated to work, and the detail design would have been refined. Some actual coding could be ported to the next phase.
- Code and implement first version:
  This will be a high-volume, 24/7, mission-critical application. C++ would be used for most logic. A high-capacity database engine would be implemented on the control cluster for control databases. Any modifications to the experiment's binaries would be performed at this time. Nearly all work could proceed in parallel since the design and prototype are now proven. Some of the team that codes this phase would subsequently handle support and documentation.

Additional Issues

- This design has not addressed the issue of directory names within a cluster's cache, or the use of functions such as NFS to provide access to a cache. Clearly, the job package must be able to locate both software and input data on whatever cluster it lands. Logic for this would be built into the custom package wrapper.
- The minimum composition of a platform has not been stated, nor has potential issues with using multiple platforms, but this low-tech approach should be as easy to implement as anything.
- It has been assumed that the control cluster does not itself need failure-recovery. Restarting any cluster has been lightly treated, ignoring the problem of cleaning up after failed jobs.
- Replicating the experiment's binaries and parameter files on many clusters can lead to issues of omitted or garbled files. This can be handled by additional monitoring tasks and checksum techniques. The package wrapper should not have to verify the entire software version each time it launches a binary, but in the event of failures such verification would be a useful explicit diagnostic function.
- Security aspects have been ignored. Information that is required for secure transport and task launching will have to be added to the control databases to properly invoke the grid primitives.
- The job/phase/package approach assumes a simplified job flow having at most only mainline merges and mainline splits. Use cases that cannot be handled by this approach are solicited.
- Monitoring of grid performance for tuning purposes has been neglected. Probably, jobs that complete will be moved off to an archive for analysis, and other programs can be written to examine the changing state of the mirror databases. Notification of problems has not been fully addressed: who will deal with a downed node, who will decide if a job is really stalled, etc.
- A measure of checkpoint/restart is offered for most of the processing scenarios – those that allow distributed processing – because failed packages would be restarted by the monitors, and because built-in package logic could resume after the last fully-completed binary in a chain (if the package wrapper is intelligent enough). Mid-binary restart is not handled, nor is the restart for a single binary running on multiple input files, one after the other, on one CPU.
- It should be expected that this application will undergo continual refinement from this point on. Custom coding is a two-edged sword: it can give you exactly what you need, but then you must retain your own team of experts to implement your needs.