Section E.1

Moira, the Athena Service Management System

Peter Levine
Michael R. Gretzinger
Mark A. Rosenstein
Jean Marie Diaz
Bill Sommerfeld
Ken Raeburn

1. Abstract

The problem of maintaining, managing, and supporting an increasing number of distributed network services with multiple server instances requires development and integration of centralized data management and automated information distribution. This paper presents *Moira*, the Athena Service Management System, focusing on the system components and interface design. The system provides centralized data administration, a protocol for interface to the database, tools for accessing and modifying the database and an automated mechanism of data distribution.

2. Purpose

The primary purpose of Moira is to provide a single point of contact for administrative changes that affect more than one Athena service. The secondary purpose is to provide a single point of contact for authoritative information about Athena administration.

3. Introduction

Currently, many update tasks and routine service issues are managed manually. As the number of users and machines grows, managing the Athena system becomes significantly more difficult and more economically unfeasible. The Athena Service Management System is being developed in direct response to the problem of supporting the management of an increasing number of independent workstations. A network based, centralized data administrator, Moira provides update and maintenance of system servers.

The development of Moira addresses centralized administration, distributed data services, and routine system updates:

- Conceptually, Moira provides mechanisms for managing Athena servers and services. This aspect comprises the fundamental design of Moira.

- Economically, Moira provides a replacement for the labor-intensive, hand-management now associated with maintaining services.

- Technically, Moira consists of a database, an Moira server and protocol interface, a data distribution manager, and tools for accessing and changing Moira data.

Moira provides a single coherent point of contact for the access and update of data. The access of data is performed by means of a standard application interface. Programs designed to update network servers, edit mailing lists, and manage group members all talk to the application interface. The programs which update servers are commonly driven by crontab and act as a server stuffing mechanism. Applications which are used as administrative tools are invoked by users.

Two examples of Moira use:

- One example is for the user accounts administrator to run an application on her workstation which will change the disk quota assigned to a user. She doesn't need to log in to any other machine to do this, and the change will automatically take place on the proper server a short time later.

- Another example is for a user to run an application to add themselves to a public mailing list. Again, the user can run this application on any workstation. Sometime later, the mailing lists file on the central mail hub will be updated to show this change.

This technical plan discusses Moira from a functional standpoint. Its intention is to establish a relationship between the design of Moira and the clients which use Moira.

Note that the system has changed names since work began. Originally it was simply called SMS, the initials of Service Management System. The new name is a slightly anglicized spelling of the Greek term for the fates. According to the mythology, there are three aspects of fate: Clotho, who spins the thread of life; Lachesis, who assigned to each man his destiny in the great tapestry; and Atropos, who cut the thread at death. Similarly, the Athena Service Management System controls the creation of user accounts, the assignment of resources, and the termination of computer access. Since the name change has occured after much code development, the string "sms" still crops up in some of the code.

4. Requirements

The design criteria and requirements are influenced by the following:

- Simplicity and clarity of the design are more important than complexity or speed. A clear, simple design will guarantee that Moira will be a well structured product capable of being integrated with other system resources. Other systems, such as Hesiod, will provide a speedy interface to the data kept by Moira; the purpose of Moira is to be the authoritative keeper of the database, updating slave systems such as Hesiod as needed.

- A simple interface based on existing, tested products. Wherever possible, Moira uses existing products.

- Moira must be independent of individual services. Each server receiving information from Moira requires information with particular data format and structure. However, the Moira database stores data in one coherent format. Through its own knowledge of each server's needs, a data control manager will access Moira data and convert it to server-appropriate structure.

- Clients must not touch the database directly; that is, they should not see the database system actually used by Moira. An application must talk to an application library. This library is a collection of functions allowing access to the database. The application library communicates with the Moira server via a network protocol.

- Maximize local processing in applications. Moira is a centralized information manager. It is not a computing service for local processors. For efficiency, the Moira protocol supports simple methods of requesting information; it is not responsible for processing complex requests. Applications can select pieces of the supplied information, or produce simple requests for change.

- Ability for expansion and routine upgrades. Moira is explicitly responsible for supporting new information requirements; as new services are added, the mechanism which supports those services must be easily added.

- The system must provide no direct services, i.e. none at user level, so that an environment can exist with or without Moira. (Without it, however, the economic consequence of managing system services by hand must be recognized.) Moira should be reasonably easy to install at other sites.

- Moira must be tamper-proof. It should be safe from denial-of-service attacks and malicious network attacks (such as replay of transactions, or arbitrary "deathgrams").

- Moira must be secure. Authentication will be done using Athena's Kerberos [2] private-key authentication system. Once the identity of the user is verified, their right to view or modify data is determined according to the contents of access control lists (acl's) which reside with the data.

- Fail gracefully.

- Moira does not have to be 100% available. Moira provides timely information to other services which are 100% available (Hesiod, Zephyr, NFS). Once again, the purpose is to provide a centralized source of authoritative information.

5. System Model

The model is derived from requirements listed in the previous section. As previously mentioned, Moira is composed of six components.

- The database.
- The Moira server.
- The application library.
- The Moira protocol.
- The Data Control Manager, DCM.
- The server-specific files.

Because Moira has a variety of interfaces, a distinction must be maintained between applications which directly read and write to Moira (i.e., administrative programs) and services which use information distributed from Moira (i.e. name server). In both cases the interface to Moira database is through the Moira server, using the Moira protocol. The significant difference is that server update is handled automatically through a data control manager; administrative programs are executed by users.

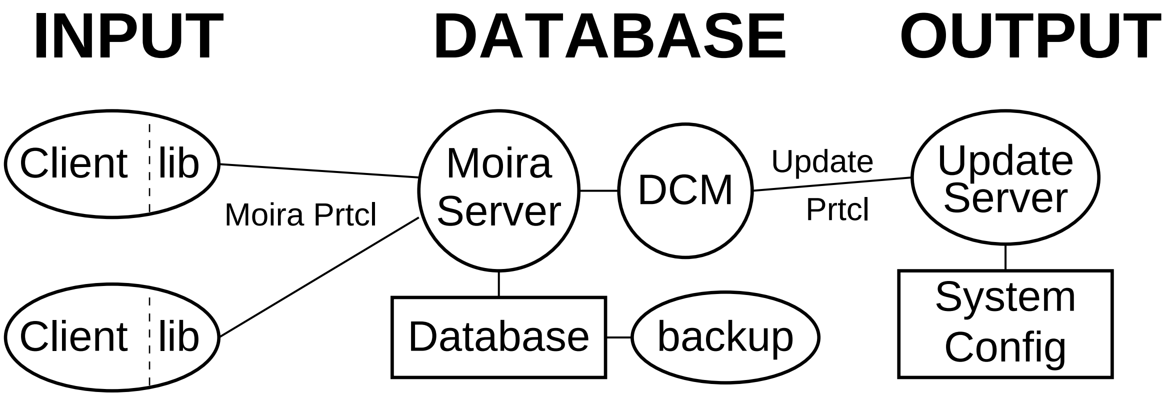

Figure 1: The Moira System Structure

In all cases, a client of Moira uses the application library. The library communicates with the Moira server via a network protocol. The server will process database specific requests to retrieve or update information.

5.1. System Assumptions

The support and function of Moira is derived exclusively in response to the environment which it supports. This section presents factors of the design dealing with considerations such as scalability, size, deployment, and support.

A. The system is designed optimally for 10,000 active users. The database has been designed to delineate between active and non-active status. Active refers to those individuals who have permission to use the system.

B. Moira supports a number of system services. To date there are four system services which are supported. They are:

- Hesiod
- NFS
- Mail Service
- Zephyr - to date, supported but not in use

These services are, however, each supported by a collection of server-specific data files. To date, there are over 20 separate files used to support the above services.

C. Each service is supported by a collection of database fields. Over 100 query handles provide efficient, database independent methods of accessing data. All applications use this method.

D. The system is designed to allow futher expansion of the current database, with the ultimate capability of Moira supporting multiple databases through the same query mechanism. Provision for many more services is recognized through this design.

E. The distribution of server-specific files can occur every 15 minutes, with an optimal time interval being greater than 6 hours. The data control manager is designed to only generate and propagate new files if the database has changed within the previous time interval.

F. The system supports one hesiod server, 20 locker servers running NFS, one /usr/lib/aliases propagation. A hesiod server requires 11 separate files; each hesiod server will receive the same 11 files. Each NFS server requires three files, one of these files will be the same for most NFS servers. /usr/lib/aliases is one file.

G. File Organization:

Service	File	Size	Number	Propagations	Interval
Hesiod	cluster.db	53656	1	1	6 hours
	filesys.db	541482	1	1	6 hours
	gid.db	341012	1	1	6 hours
	group.db	453636	1	1	6 hours
	grplist.db	357662	1	1	6 hours
	passwd.db	712446	1	1	6 hours
	pobox.db	415688	1	1	6 hours
	printcap.db	4318	1	1	6 hours
	service.db	9052	1	1	6 hours
	sloc.db	3734	1	1	6 hours
	uid.db	256381	1	1	6 hours
NFS	*partition*.dirs	2784	20	20	12 hours
	partition.quotas	1205	20	20	12 hours
	credentials	152648	1	20	12 hours
Mail	/usr/lib/aliases	445000	1	1	24 hours
Zephyr	*class*.acl	100	6	18	24 hours
TOTAL			59	90	

NOTE: The above files will only be generated and propagated if the data has changed during the time interval. For example, although the hesiod interval is 6 hours, there is no effect on system resources unless the information relevant to hesiod has changed during the previous 6 hour interval.

H. Application interfaces provide all the mechanisms to change database fields. There will be no need for any Moira updating to be done by directly manipulating the database. For each service, there is at least one application interface. Currently there are twelve interface programs.

5.2. The Database

The database is the core of Moira. It provides the storage mechanism for the complete system.

The database is written using RTI INGRES, a commercially available database toolkit. Its advantage is that it is available and it works. INGRES provides the Athena plant with a complete query system, a C library of routines, and a command interface language. Moira does not depend on any special feature of INGRES. In fact, Moira can easily utilize other relational databases.

A complete description of the INGRES design can be found in RTI's INGRES users' manuals; this paper does not discuss the structure of INGRES. This documentation does, however, describe, in detail the structure of the Moira database.

The database contains the following:

- User information (account, finger)
- Machine information
- Cluster information
- General service information (service location, /etc/services)
- File system information (NFS partitions, logical filesystems)
- Printcap information
- Post office information
- Lists (mail, acl, groups)
- Aliases

The database fields are described in section starting on page 27 of this document.

The database is a completely independent entity from the Moira system. The Ingres query bindings and database specific routines are layered at the lower levels of the Moira server. *All applications are independent of the database specific routines.* This independence is acheived through the use of query handles, Moira specifice functions providing data access and updating. An application passes query handles to the Moira server which then resolves the request. This request is passed to the database via a database specific call. Allowing for additional data and future expansions, Moira can use other databases for information. This mechanism, although not functional at this time, is achieved by having a set of query handles for each accessable database. Then, the application merely passes a query handle to a function, which then resolves the database and query.

The current database supports all activities inherent to operation and data requirements of the previously listed Moira-supported services. No attempt is made to circumvent Moira as the central point of contact. When needed and where applicable, as more services are required, new fields and query handles will be provided for support.

5.2.1. Input Data Checking

Without proper checks on input values, a user could easily enter data of the wrong type or of a nonsensical value for that type into Moira. For example, consider the case of updating a user's mail address. If, instead of typing e40-po (a valid post office server), the user typed in e40-p0 (a nonexistant machine), all the user's mail would be "returned to sender" as undelivereable.

Input checking is done by both the Moira server and by applications using Moira. Each query supported by the server may have a validation routine supplied which checks that the arguments to the query are legitimate. Queries which do not have side effects on the database do not need a validation routine.

Some checks are better done in applications programs; for example, the Moira server is not in a good position to tell if a user's new choice for a login shell exists. However, other checks, such as verifying that a user's home directory is a valid filesystem name, are conducted by the server. An error condition will be returned if the value specified is incorrect. The list of predefined queries (Section 7) defines those fields which require explicit data checking.

5.2.2. Backup

It is not absolutely critical that the Moira database be available 24 hours a day; what is important is that the database remain internally consistant, and that the bulk of the data not be lost. With that in mind, the database backup system for Moira has been set up to maximize recoverability in the event of damage to the database.

Two programs (`mrbackup` and `mrrestore`) are generated automatically (using an `awk` script) from the database description file `db/newdb`[1]. `mrbackup` copies each relation of the current Moira database into an ASCII file. Each row of the database is converted into a single line of text; each line consists of several colon separated fields followed by a newline character (ASCII code 10 decimal). Colons and backslashes inside fields are replaced by \: and \\, respectively. Non-printing characters are replaced by \nnn, where nnn is the octal value of the ASCII code for the non-printing character. The full database dump takes roughly 12 minutes with the current (albeit partially-populated) database; the ascii files take up about 3.2 MB of space. It is intended that mrbackup be invoked by a shell script run periodically by the `cron` daemon; this shell script (currently called `nightly.sh`) maintains the last three backups on line. It is intended that these backups be put on a separate physical disk drive from the drive containing the actual database. Also, they should be dumped to tape using `tar`, or copied to another machine, on a regular basis. Whether they should be dumped to TK50 or reel-to-reel tape is open to discussion at this point.

`mrrestore` does the inverse of `mrbackup`. It requires the existance of a empty database named "smstemp" [2], created as follows:

```
# createdb smstemp
# quel smstemp
* \i /mit/moira/src/db/newdb            load DB definition
* \g                                    execute DB definition
* \q                                    quit
# mrrestore
Do you *REALLY* want to wipe the Moira database? (yes or no): yes
Have you initialized an empty database named smstemp? (yes or no): yes
Opening database...done
Prefix of backup to restore: /site/sms/backup_1/
Are you sure? (yes or no): yes
Working on /site/sms/backup_1/users
        ...
```

This system by itself provides recovery with the loss of no more than roughly a day’s transactions. To improve this, the journal file kept by the Moira server daemon contains a listing of all successful changes to the database.

RTI Ingres provides some checkpointing and journalling facilities. However, past experience with them has shown that they are not particularly reliable. Also, a common failure mode, at least with version 3 of RTI ingres, has been corruption in the binary structure of the database. Since the checkpointing mechanism used is simply a `tar` format copy of the database directory, restoring from the checkpoint will probably not cure the corruption, particularly since they may go for days without being noticed. The only known

---

[1] All pathnames are relative to the root of the Moira source tree

[2] The eventual production version will work on a database named "sms"; however, for test use, "smstemp" is used instead

cure is to dump the entire database to text files, and recreate it from scratch from the text files. Because of this dubious history, it was decided that the RTI checkpoint and journalling mechanism was not sufficiently reliable for use with Moira.

5.3. The Moira Protocol

The Moira protocol is a remote procedure call protocol layered on top of TCP/IP. Clients of Moira make a connection to a well known port (T.B.S.), send requests over that stream, and received replies. Note: the precise byte-level encoding of the protocol is not yet specified here.

Each request consists of a major request number, and several counted strings of bytes. Each reply consists of a single number (an error code) followed by zero or more "tuples" (the result of a query) each of which consists of several counted strings of bytes. Requests and replies also contain a version number, to allow clean handling of version skew.

The following major requests are defined for Moira. It should be noted that each "handle" (named database action) defines its own signature of arguments and results. Also, the server may refuse to perform any of these actions based on the authenticated identity of the user making the request.

Noop	Do nothing. This is useful for testing and profiling of the RPC layer and the server in general.
Authenticate	There is one argument, a Kerberos [2] authenticator. All requests received after this request should be performed on behalf of the principal identified by the authenticator.
Query	There are a variable number of arguments. The first is the name of a pre-defined query (a "query handle"), and the rest are arguments to that query. If the query is allowed, any retrieved data are passed back as several return values, each with an error code of MR_MORE_DATA indicating that there are more tuples coming.
Access	There are a variable number of arguments. The first is the name of a pre-defined query useable for the "query" request, and the rest are query arguments. The server returns a reply with a zero error code if the query would have been allowed, or a reply with a non-zero error code explaining the reason why the query was rejected.
Trigger_DCM	This takes no arguments. It will request the Moira server to immediately spawn a DCM process. Access checking is done by checking permissions for the pseudo-query "trigger_dcm" (tdcm).

5.4. The Moira Server

All remote [3] communication with the Moira database is done through the Moira server, using a remote procedure call protocol built on top of GDB [1]. The Moira server runs as a single UNIX process on the Moira database machine. It listens for TCP/IP connections on a well known service port (T.B.D.), and processes remote procedure call requests on each

[3]The DCM and the Moira backup programs, which run on the host where the Moira database is located, do go through the server, both for performance reasons and to avoid clogging the server

connection it accepts.

One of the major concerns for efficiency in Moira is the time it takes to start up an application's connection to the server. One of the limiting factors for Athenareg, Moira's predecessor, is the time it takes to start up the Ingres back end subprocess which it uses to access the database. This was done for every client connection to the database. As starting up a backend process is a rather heavyweight operation, the Moira server will do this only once, at the start up time of the daemon.

GDB, through the use of BSD UNIX non-blocking I/O, allows the programmer to set up a single process server which handles multiple simultaneous TCP connections. The Moira server will be able to make progress reading new RPC requests and sending old replies simultaneously, which is important if a reasonably large amount of data is to be sent back.

SUN RPC was also considered for use in the RPC layer, but was rejected because it cannot handle large return values, such as might be returned by a complex query.

5.5. Access control

The server performs access control on all queries which might side-effect the database. As most information in the database will be loaded into the nameserver and/or other configuration files, placing access control on read-only queries is unnecessary.

Because one of the requests that the server supports is a request to check access to a particular query, it is expected that many access checks will have to be performed twice: once to allow the client to find out that it should prompt the user for information, and again when the query is actually executed. It is expected that some form of access caching will eventually be worked into the server for performance reasons.

5.6. The Application Library

The Moira application library provides access to Moira through a simple set of remote procedure calls to the Moira server. The library is layered on top of Noah Mendohlson's GDB library, and also uses Ken Raeburn's com_err library to provide a coherant way to return error status codes to applications.

For use by the DCM and other utilities, there exists a version of the library which does direct calls to Ingres, rather than going through the server. Use of this library should result in significantly higher throughput, and will also reduce the load on the server itself. The direct "glue" library provides the exact same interface as the RPC library, except that it does not use Kerberos authentication.

5.6.1. Error Handling

Because of all the possible failure points in a networked application, we decided to use Ken Raeburn's `libcom_err` library. Com_err allows several different sets of error codes to be used in a program simultaneously - every error code is an integer, and each error table reserves a subrange of the integers (based on a hash function of the table name). UNIX system call error codes are included in this system. By convention, zero indicates success, or no error. The following routines may be useful to applications programmers who wish to display the reasons for failure of a routine.

```
char *error_message(code)
int code;
```

Returns the error message string associated with `code`.

```
void com_err(whoami, code, message)
      char *whoami;     /* what routine encountered the error */
      int code;/* An error code */
      char *message; /* printed after the error message */
      int code;
```

By default, prints a message of the form

whoami: *error_message(code) message newline*

If `code` is zero, nothing is printed for the error message.

```
void set_com_err_hook(hook)
      void (*hook)();   /* Function
                 * to call instead of printing to stderr */
```

If this routine is called with a non-NULL argument, it will cause future calls to `com_err` to be directed into the hook function instead. This can be used to, for example, route error messages to `syslog` or to display them using a dialogue box in a window-system environment.

5.6.2. Moira application library calls

The Moira library contains the following routines:

```
int mr_connect();
```

Connects to the Moira server. This returns an error code, or zero on success. This does not attempt to authenticate the user, since for simple read-only queries which may not need authentication, the overhead of authentication can be comparable to that of the query. This can return a number of operational error conditions, such as ECONNREFUSED (Connection refused), ETIMEDOUT (Connection timed out), or MR_ALREADY_CONNECTED if a connection already exists.

```
int mr_auth(clientname);
char *clientname
```

Attempts to authenticate the user to the system. `Clientname` is the name of the program acting on behalf of the user. `mr_auth` can return Kerberos failures, either local or remote (for example, "can't find ticket" or "ticket expired"), MR_NOT_CONNECTED if `mr_connect` was not called or was not successful, or MR_ABORTED if the attempt to send or recieve data failed (and the connection is now closed).

```
int mr_disconnect();
```

This drops the connection to Moira. The only error code it currently can return is MR_NOT_CONNECTED, if no connection was there in the first place.

```
int mr_noop();
```

This attempts to do a handshake with Moira (for testing and performance measurement). It can return MR_NOT_CONNECTED or MR_ABORTED if not successful.

```
int mr_access(name, argc, argv)
      char *name;              /* Name of query */
      int argc;         /* Number of arguments provided */
      char *argv[];            /* Argument vector */
```

This routine checks the user's access to an Moira query named `name`, with arguments argv[0]...argv[argc-1]. It does not actually process the query. This is included to give applications a "hint" as to whether or not the particular query will succeed, so that they won't bother to prompt the user for a large number of arguments if the query is doomed to failure.

```
int mr_query(name, argc, argv, callproc, callarg)
      char *name;              /* Name of query */
      int argc;        /* Number of arguments provided */
      char *argv[];            /* Argument vector */
      int (*callproc)();        Routine to call on each reply */
      caddr_t callarg;  /* Additional argument
                * to callback routine */
```

This runs an Moira query named *name* with arguments argv[0]...argv[argc-1]. For each returned tuple of data, callproc is called with three arguments: the number of elements in the tuple, a pointer to an array of characters (the data), and callarg.

5.6.3. Other provided routines

The Moira library also contains a number of other routines which are used both by the servers and some of the clients. These include

- convert between flags integer and human-readable string
- canonicalize hostname
- string utility routines - trim whitespace, save a copy
- hash table abstraction
- simple queue abstraction
- a menu package used for some of the clients

These routines are documented in the provided manualpage moira.3.

5.7. The Data Control Manager

The data control manager, or DCM, is a program responsible for distributing information to servers. Basically, the DCM is invoked regularly by cron at intervals which become the minimum update time for any service. Whenever the DCM runs, it will determine which services and hosts should be updated now. The update frequency is stored in the Moira database. A server/host relationship is unique to each update time. Through the Moira query mechanism, the DCM extracts Moira data and converts it to server dependent form. The conversion of the Moira data to the server-specific format is done by a sub-program specific to that service.

5.7.1. DCM Operation

On startup, the DCM first checks for the existance of the disable file */etc/nodcm*; if this file exists, it exits quietly. Next it connects to the database and authenticates as **root**. Then it retrieves the value of **dcm_enable** from the values relation of the database; if this value is zero, it will exit, logging this action.

Next, the DCM scans the services table. This table contains:

Name	The name of each service.
Type	The type of service. Currently defined service types are **unique** and **replicat**ed. This type affects some parts of the update algorythm, described below.

ACE type and ACE name
: These specify the access control entity which owns the service. The type may be **list**, **user**, or **none**. The name will then be a list name, a login name, or **none**, respectively. The owner is allowed to manipulate the service or service/host tuples supporting that service.

Interval
: Gives the minimum time between updates of this service, in minutes.

Target
: The name of the target file on the servers. This is where Moira will deposit the new configuration files.

Script
: This is the name of the script file on Moira which will be executed on the server to install the new configuration files.

DFGen
: This is the time that the data files were last generated for this service. It is stored as a unix format time (number of seconds since January 1, 1970 GMT).

DFCheck
: This is the time that the data files were last checked to see if they needed to be regenerated. It is stored as a unix format time.

Enable
: This boolean flag indicates if updates should be performed on this service. It may be set and cleared by the user.

InProgress
: This boolean flag indicates that an update is currently in progress for this service. It is set and cleared by only by the DCM. It is **not** relyed upon for locking.

Harderror
: This field records the error number of any hard errors that occur during an update.

Errmsg
: This is a textual representation of the error reported in *harderror*.

ModTime
: The time this data was created or last modified. This refers only to modification by a user, not by the DCM.

ModBy
: The user name of who last modified this record.

ModWith
: The name of the application that was used for the last modification.

Each time the DCM is invoked, a search through this table will indicate which servers need updating. It will first identify those services which are *enable*d, do not have *hard errors*, have a non-zero *interval*, and do have a generator module. For each of these services, it compares *dfcheck* and the update *interval* against the current time. If it is time for another update, it will obtain an exclusive lock on the service, set the *inprogress* flag, then run the generator.

The generator is a sub-program that does the actual extract. Each generator lives in */u1/sms/bin/**service**.gen*. A generator takes as an argument the name of the output file it should generate. It's exit status will be zero on success, otherwise the number of any error. Note that a common ''error'' for a generator is `MR_NO_CHANGE`, indicating that nothing in the database has changed and the data files were not re-built.

If the generator finishes without error, *dfgen* and *dfcheck* are updated to the current time. If the generator exits indicating that nothing has changed, only *dfcheck* is updated to the current time. If there is a soft error (an expected error that might go away if we try again later) then the *error_message* is updated to reflect this, but *hard error* is not set. If there is a hard error, the *hard_error* and *error_message* are set, and a zephyr message is sent to class **MOIRA** instance **DCM** indicating this error. After this attempt, the lock on the service is released.

For each of the services which passed the initial check above (*enable*d, no *hard errors*, non-zero *interval*, and a generator exists) regardless of the result of attempting to build data files, or even if it was time to build data files, the hosts will be scanned. The serverhost table contains:

Service	The name of the service.
Machine	The name of the machine supporting this service.
Enable	A boolean indicating that this host should be updated.
Override	A boolean indicating that this host should be updated as soon as possible, disregarding the service time interval.
Success	A boolean indicating that the last attempted update was successful.
InProgress	A boolean indicating that this host is currently being udpated.
HostError	This field recrods the error number of any hard errors that occur during an update.
Errmsg	This is a textual representation of the error report in *hosterror*.
LastTry	This is the time that the DCM last attempted to update this host. It is stored as a unix format time.
LastSuccess	This is the time of the last successful DCM update of this host. It is stored as a unix format time.
Value1	This is service specific information, stored as an integer. For POP servers, it is the number of poboxes assigned to this server.
Value2	This is service specific information, stored as an integer. For POP servers, it is the maximum number of poboxes that may be assigned to this server.
Value3	This is service specific information, stored as a string. For NFS servers, it indicates who should be in the credentials file.
ModTime	The time this data was created or last modified. This refers only to modification by a user, not by the DCM.
ModBy	The user name of who last modified this record.
ModWith	The name of the application that was used for the last modification.

During the host scan, the DCM first locks the service. It will lock it exclusively if the service *type* is **replicated**, otherwise it will acquire a shared lock. Then the DCM makes a list of hosts which are *enabled*, do not have *hard errors*, and have not been successfully updated sine the data files were generated for this service (or *override* is set). It will then step through these hosts, updating them. The first part of an update is to obtain an exclusive lock on the host and set the *inprogress* bit. Then it sends the generated file to the *target* file on the host and then executes the *script* on that host. The exit value of the script is reported back, with zero being success and anything else being the error number. If the update is successful, the *last_time_tried* and *last_time_successfull* are updated. If there is a soft fail, then just the *last_time_tried* is updated, and the *error_message* is recorded. If there is a hard failure, then a soft fail is taken, plus *hard_error* is set and a zephyrgram and mail are sent about it. If there is a hard failure and the service is **replicat**ed, then the error code & message are also set in the service record so that no more updates will be attempted to hosts supporting this service. Then the host lock is freed.

When the host scan is complete, the service lock is freed, and the service scan continues with the next service.

5.8. Server Arrangement

Currently, Moira acts to update a variety of servers. Although the data control manager performs this update task, each server requires a different set of update parameters. To date, the DCM uses c programs, not SDFs, to implement the construction of the server specific files. Each c program is checked in via the dcm_maint program. The DCM then calls the appropriate module when the update interval is reached.

For each server file propagated, there is at least one application interface which provides the capability to manipulate the Moira database. Since the Moira database acts as a single point of contact, the changes made to the database are reflected in the contents of recipient servers.

The services which Moira now supports are:

- Hesiod - The athena nameserver.
- NFS - Network file system.
- /usr/lib/aliases - Mail Service.
- Zephyr - The athena notification service.

5.8.1. Server Assumptions

The requirements of each server suggests a level of detail describing the following:

- Service name.
- Service description.
- Propagation interval.
- Data format.
- Target location.
- Generated files.
- File description.
- Queries used to generate the file (including fields queried).
- How the file is modified (application interface).
- Example of file contents.

5.8.2. Server Descriptions

Service: Hesiod

Description: The hesiod server is a primary source of contact for many athena operations. It is reponsible for providing information reliably and quickly. Moira's responsibility to hesiod is to provide authoritative data.

Hesiod uses a BIND data format in all of it's data files. Moira will provide BIND format to hesiod. There are several files which hesiod uses. To date, they are known to include the following:

- cluster.db
- filsys.db
- gid.db
- group.db
- grplist.db
- passwd.db
- pobox.db
- printcap.db
- service.db
- sloc.db
- uid.db

Each of these files are described in detail below. The hesiod server uses these files from virtual memory on the target machine. The server automatically loads the files from disk into memory when it is started. Moira will propagate hesiod files to the target disk and the run a shell script which will kill the running server and then restart it, causing the newly updated files to be read into memory.

With hesiod, all target machines receive identical files. Practically, therefore, the DCM will prepare only one set of files and then will propagate to several target hosts.

For additional technical information on *hesiod*, please refer to the Hesiod technical plan.

Propagation interval:
6 Hours

Data format: tar file of several BIND files

Target locations:
SUOMI.MIT.EDU: /tmp/hesiod.out

Files:

CLUSTER.DB Cluster data

Description: Cluster.db holds the relationships between machines, clusters, and services to service clusters. It must be possible to look up a cluster by name, and find all of the cluster data. It must also be possible to look up a machine by name, and get the union of all of the cluster data for each cluster the machine is a member of.

Queries used:

Client(s): save_cluster_info

Example contents:

```
; lines for per-cluster info (both vs and rt) (type UNSPECA)
; and a line for each machine (CNAME referring to one of the lines
; above)
;
bldge40-vs.cluster HS UNSPECA "zephyr neskaya.mit.edu"
bldge40-rt.cluster HS UNSPECA "zephyr neskaya.mit.edu"
bldge40-vs.cluster HS UNSPECA "lpr e40"
bldge40-rt.cluster HS UNSPECA "lpr e40"
TOTO.cluster       HS CNAME bldge40-vs.cluster
SCARECROW.cluster  HS CNAME bldge40-rt.cluster
```

Note that a machine may be in more than one cluster. In this case, a pseudo-cluster will be made by Moira which has as its cluster data, the union of the data of each of the other clusters this machine is in. Then the machine in question will be CNAME'd into this pseudo-cluster.

FILSYS.DB filesystems

Description These are all of the filesystem entries needed to find and attach NFS lockers and RVDs by name. Each entry contains the name of the filesystem, its name on the fileserver (directory name for NFS filesystems, or packname for RVDs), the name of the server, the default attach mode (r = read-only, w = read/write), and the default client mount point.

Queries used:

Clients(s): attach

Example contents:

```
aab.filsys      HS UNSPECA "NFS /mit/aab charon w /mit/aab"
aabiyaba.filsys HS UNSPECA "NFS /mit/aabiyaba eurydice w /mit/aabiyaba"
ade.filsys      HS UNSPECA "RVD ade helen r /mnt/ade"
```

GID.DB group IDs

Description: This file maps group ID numbers to the group names. There is an entry in this file for each entry in the group.db file, pointing to a corresponding entry in the group.db file.

Queries used:

Clients(s):

Example contents:

```
10914.gid       HS CNAME babette.group
10915.gid       HS CNAME 14.31.group
10916.gid       HS CNAME abarba.group
10917.gid       HS CNAME mga.group
10918.gid       HS CNAME rs1maint.group
10919.gid       HS CNAME pjd.group
```

GROUP.DB unix groups

Description: This file maps group names to their unix group ID numbers. The returned entries are of the same form as lines in an /etc/group file, although none of the members are actually filled in. An entry is only placed in this file if the group is marked *active* in the Moira database.

Queries used:

Clients(s):

Example contents:

```
babette.group   HS UNSPECA "babette:*:10914:"
14.31.group     HS UNSPECA "14.31:*:10915:"
abarba.group    HS UNSPECA "abarba:*:10916:"
mga.group       HS UNSPECA "mga:*:10917:"
rs1maint.group  HS UNSPECA "rs1maint:*:10918:"
pjd.group       HS UNSPECA "pjd:*:10919:"
```

GRPLIST.DB group lists

Description: This file lists the groups that each user is a member of. Each entry consists of a colon-separated list of colon-separated pairs of groupname, group id. No meaning is placed on the order of the groups listed. Only users whose *status* is active will have entries generated, and only groups that are marked *active* in Moira will be output.

Queries used:

Clients(s): login

Example contents:

```
mtalford.grplist  HS UNSPECA "mtalford:5904:3_d0004:689"
mswelsh.grplist   HS UNSPECA "mswelsh:5901:13_461t:867:13_461sa:868:13_012t:800"
mstai.grplist     HS UNSPECA "mstai:5899"
```

PASSWD.DB password entries

Description: This file contains lines similar to those found in /etc/passwd for each active user of Athena. Only users whose *status* is active will have entries generated.

Queries used:

Clients(s): login

Example contents:

```
babette.passwd  HS UNSPECA "babette:*:6530:101:Harmon C Fowler,,,,:/mit/babette
:/bin/csh"
abarba.passwd   HS UNSPECA "abarba:*:6531:101:Angela Barba,,,,:/mit/abarba:/bin
/csh"
mga.passwd      HS UNSPECA "mga:*:6532:101:Gerhard Messmer,,,,:/mit/mga:/bin/cs
h"
kazimi.passwd   HS UNSPECA "kazimi:*:6533:101:Martin Zimmermann,,,,:/mit/kazimi
:/bin/csh"
pjd.passwd      HS UNSPECA "pjd:*:6535:101:Peter J Delaney,,,,:/mit/pjd:/bin/cs
h"
```

POBOX.DB Post Office Box locations

Description: This file locates each user's post office box. Only active users whose pobox type is "POP" will have entries output.

Queries used:

Clients(s): inc, movemail

Example contents:

```
babette.pobox   HS UNSPECA "POP ATHENA-PO-2.MIT.EDU babette"
abarba.pobox    HS UNSPECA "POP ATHENA-PO-1.MIT.EDU abarba"
mga.pobox       HS UNSPECA "POP ATHENA-PO-1.MIT.EDU mga"
kazimi.pobox    HS UNSPECA "POP ATHENA-PO-2.MIT.EDU kazimi"
```

PRINTCAP.DB printer capability entries

Description: This file contains the information from the standard /etc/printcap file.

Queries used:

Clients(s): lpr, lpq, lprm

Example contents:

```
linus.pcap      HS UNSPECA "linus:rp=linus:rm=BLANKET.MIT.EDU:sd=/usr/spool/pri
nter/linus"
la-pika.pcap    HS UNSPECA "la-pika:rp=la-pika:rm=EVE.PIKA.MIT.EDU:sd=/usr/spoo
l/printer/la-pika"
ln03-pika.pcap  HS UNSPECA "ln03-pika:rp=ln03-pika:rm=EVE.PIKA.MIT.EDU:sd=/usr/
spool/printer/ln03-pika"
helios.pcap     HS UNSPECA "helios:rp=helios:rm=M16-034-P.MIT.EDU:sd=/usr/spool
/printer/helios"
```

SERVICE.DB network service map

Description: This file contains the information from the standard /etc/services file. It comes from the services relation of the database, and the aliases.

Queries used:

Clients(s):

Example contents:

```
gdb_test3.service       HS UNSPECA "gdb_test3 tcp 2253"
qotd.service    HS UNSPECA "qotd tcp 17"
rpc_ns.service  HS UNSPECA "rpc_ns udp 32767"
smtp.service    HS UNSPECA "smtp tcp 25"
X1.service      HS UNSPECA "X1 tcp 5801"
```

SLOC.DB service location information

Description: This file identifies which hosts support which services. It is a listing of DCM service/host tuples, indexed by service.

Queries used:

Clients(s): zhm, chpobox, get_message

Example contents:

```
ATHENA_MESSAGE.sloc     HS UNSPECA BITSY.MIT.EDU
FOO.sloc        HS UNSPECA TOTO.MIT.EDU
GMOTD.sloc      HS UNSPECA BITSY.MIT.EDU
HESIOD.sloc     HS UNSPECA KIWI.MIT.EDU
HESIOD.sloc     HS UNSPECA SUOMI.MIT.EDU
LOCAL.sloc      HS UNSPECA HACTAR.MIT.EDU
```

UID.DB user ID mappings

Description: This file maps unix user IDs to the user password entries. There is a corresponding entry in this file for every entry in the passwd.db file.

Queries used:

Clients(s):

Example contents:

```
6530.uid        HS CNAME babette.passwd
6531.uid        HS CNAME abarba.passwd
6532.uid        HS CNAME mga.passwd
6533.uid        HS CNAME kazimi.passwd
6535.uid        HS CNAME pjd.passwd
```

Service: NFS

Description: Moira supports three files which are necessary components of NFS operation. These files are:

- /usr/etc/credentials
- The quotas file
- The directories file

The credentials file determines access permissions to files on the NFS server. It contains mappings from user name to unix UID and a group list. This file can be generated from a list in the Moira database, or may contain all active users. Which credentials file is loaded on a particular server is determined by the value3 field of the serverhost relation. If this field is non-blank, it specifies the list whose membership will appear in the credentials file. If the field is blank, then all active users will appear in the credentials file.

The quotas and directories files are in a private format to Moira, which is understood by the shell script which updates quotas and creates directories. The quotas file contains many tuples matching a unix UID with a quota. The directories file lists directory name, owning user, owning group, and directory type. The type is used to control the directory mode and which init files are loaded into it.

These files reside on the NFS target machine and are used to allocate NFS directories on a per user basis. The mechanism employed is for all programs to communicate to the Moira database, and then for the dcm to handle the propagation and creating of NFS lockers. The best illustration of this process is indicated by the following example:

During new user registration, a person will sit down to a workstation and type 'userreg' for his login name. When validated the user will type a 'real' login name and a password. In addition, the userreg program will allocate, automatically, for the user a post office and an NFS directory. However, the user will not benefit from this allocation for a maximum of six hours. This lag time is due to the operation of Moira and its creation of NFS lockers. During registration, the userreg program communicates exclusively with the Moira database for NFS allocation. Since the NFS file generation is started by the DCM every 6 hours, the real change is not noticed for a period of time. When the 6 hour time is reached the DCM will create the above two files and send them to the appropriate target servers. Once on the target machine, the dcm will invoke a shell script which reads the /mit/quota file and then creates the NFS directory. The basic operation of the script is:

mkdir <username>, chown, chgrp, chmod - using directories file
setquota <quota> - using quotas file

Propagation interval:
12 hours

Data Format: ASCII

Client(s):

NFS server
Moira shell script for creating directories and user quotas.

Files updated:

CREDENTIALS username to uid/gid mapping.

Description: This file is used for both the NFS server information and for the Moira shell script. It provides a username to uid/gid mapping. A master credentials file is generated which contains all active users. In addition, smaller credentials files may be produced if necessary, with their membership taken from an Moira list. Each line is an entry consisting of the username, uid, and the gid of each group the user is in. Each field is separated by a colon.

Queries used:

Contents example:

```
mtarriol:15786:5905
mtalford:14956:5904:689
mswelsh:13764:5901:867:868:800
mstai:9296:5899
```

QUOTAS file containing user to quota mapping.

Description: This file contains the mapping between username and quota. The file is distributed to each filesystem on the recipient machine. Each entry contains a unix uid and a quota. Each of the file's contents is unique to the filesystem which it represents.

Queries used:

Contents example:

```
219 600
567 600
1251 600
1282 600
1312 600
```

DIRECTORIES file containing info for creating NFS lockers

Description: This file contains the info necessary to create lockers. Only lockers with the *autocreate* flag set will be output. The file is distributed to each filesystem on the recipient machine. Each entry contains a directory name, owning uid and gid, and a locker type. If the directory does not already exist, it will be created with the specified ownership. If the type is "HOMEDIR", it will be loaded with the default init files as well.

Queries used:

Contents example:

```
/mit/lockers/test1 6526 10912 HOMEDIR
/mit/lockers/babette 6530 10914 HOMEDIR
/mit/lockers/kazimi 6533 10923 HOMEDIR
/mit/lockers/mastein 14489 10928 HOMEDIR
```

Service: Mail

Description: The /usr/lib/aliases file is created and propagated to athena.mit.edu. Only one file and one propagation is required. This file is not automatically installed on the mailhub because the mail spool must be disabled during the switchover. A second file is also propigated to the mail hub. This is a complete password file so that the finger server on the mailhub will know about everybody.

Data Type: ASCII (sendmail aliases format)

Propagation interval: 24 hours

Target: ATHENA.MIT.EDU

File(s):

/USR/LIB/ALIASES mail forwarding information

Description: This file contains both mailing lists and post office boxes. Mailing lists are output only if the list is marked *active* in the Moira database. Poboxes are only output if the user's account is active.

Queries Used:

Contents example:

```
#  Video Users
owner-video-users: paul
video-users: smyser, paul, mwsmith, davis, rubin@media-lab.mit.edu,
        gid@media-lab.mit.edu, danapple, agarvin

babette: babette@ATHENA-PO-2.LOCAL
yvette: yvette@ATHENA-PO-2.LOCAL
test1: test1@ATHENA-PO-2.LOCAL
```

/ETC/PASSWD user account file

Description: This is a standard format unix password file. It contains an entry for every active account at Athena.

Queries Used:

Contents example:

```
test1:*:6526:101:Test One,,,:/mit/test1:/bin/csh
babette:*:6530:101:Harmon C Fowler,,,:/mit/babette:/bin/csh
abarba:*:6531:101:Angela Barba,,,:/mit/abarba:/bin/csh
mga:*:6532:101:Gerhard Messmer,,,:/mit/mga:/bin/csh
kazimi:*:6533:101:Martin Zimmermann,,,:/mit/kazimi:/bin/csh
pjd:*:6535:101:Peter J Delaney,,,:/mit/pjd:/bin/csh
```

Service: ZEPHYR

Description: The zephyr system has access control lists associated with some actions on some classes of message. Moira updates these access control lists on the zephyr servers from lists stored in Moira.

Data Type: A tar file of ASCII acl files.

Propagation interval:
24 hours

Target:each of the zephyr servers.

File(s):

***.ACL** zephyr ACL file

Description: For each existing ACE (even if it is empty), the membership will be output, one entry per line. Recursive lists will be expanded.

Queries Used:

Contents example:

```
*.*@*
```

5.9. Moira-to-Server Update Protocol

Moira provides a reliable mechanism for updating the servers it manages. The use of an update protocol allows the servers to be reliably managed. The goals of the server update protocol are:

- Completely automatic update for normal cases and expected kinds of failures.
- Survives clean server crashes.
- Survives clean Moira crashes.
- Easy to understand state and recovery by hand.

General approach: perform updates using atomic operations only. All updates should be of a nature such that a reboot will fix an inconsistent database. (For example, the RVD database is sent to the server upon booting, so if the machine crashes between installation of the file and delivery of the information to the server, no harm is done.) Updates not received will be retried at a later point until they succeed. All actions are initiated by the DCM.

Strategy

A. Transfer phase. This step puts all of the files on the server.

1. Connect to the server host and send authentication.

2. Transfer the data files to be installed to the server. These are stored in the *target* as recorded for the service in the Moira database. The file transfer includes a checksum to insure data integrity. Only one file is transferred, although it may be a tar file containing many more.

3. Transfer the installation instruction sequence to the server. This is the *script* as recorded for the service in the Moira database. It will be stored in a temporary file on the server.

4. Flush all data on the server to disk.

B. Execution phase. If all portions of the preparation phase are completed without error, the execution phase is initiated by the DCM. On a single command from the Moira, the server begins execution of the instruction sequence supplied. These can include the following:

1. Extract data files from the tar file. Rather than extract all of the files at once, only the ones that are needed are extracted one at a time.

2. Swap new data files in. This is done using atomic filesystem `rename` operations. The cost of this step is kept to an absolute minimum by keeping both files in the same partition and by not changing the link count on any files during the rename.

3. Revert the file -- identical to swapping in the new data file, but instead puts the old file back. May be useful in the case of an erroneous installation.

4. Send a signal to a specified process. The process_id is assumed to be recorded in a file; the pathname of this file is a parameter to this instruction. The process_id is read out of the file at the time of execution of this instruction.

5. Execute a supplied command.

C. Confirm installation. The server sends back a reply indicating that the installation was successful, or what error occurred. The DCM then records this information in the database.

Trouble Recovery Procedures

A. Server fails to perform action.

If an error is detected in the update procedure, the information is relayed back to the DCM. The 'success' flag is cleared, and the error code and message are recorded in the update table. The error value returned is logged to the appropriate file; zephyr is used to notify the system maintainers a failure occured.

A timeout is used in both sides of the connection during the preparation phase, and during the actual installation on the Moira. If any single operation takes longer than a reasonable amount of time, the connection is closed, and the installation assumed to have failed. This is to prevent network lossage and machine crashes from causing arbitrarily long delays, and instead falls back to the error condition, so that the installation will be attempted again later. (Since the all the data files being prepared are valid, extra installations are not harmful.)

B. Server crashes.

If a server crashes, it may fail to respond to the next attempted Moira update. In this case, it is (generally) tagged for retry at a later time, say ten or fifteen minutes later. This retry interval will be repeated until an attempt to update the server succeeds (or fails due to another error).

If a server crashes while it is receiving an update, either the file will have been installed or it will not have been installed. If it has been installed, normal system startup procedures should take care of any followup operations that would have been performed as part of the update (such as [re]starting the server using the data file). If the file has not been installed, it will be updated again from the Moira, and the existing `filename.moira_update` file will be deleted (as it may be incomplete) when the next update starts.

C. Moira crashes.

Since the Moira update table is driven by absolute times and offsets, crashes of the Moira machine will result in (at worst) delays in updates. If updates were in progress when the Moira crashed, those that did not have the install command sent will have a few extra files on the servers, which will be deleted in the update that will be started the first time the update table is scanned for updates due to be performed. Updates for which the install command had been issued may get repeated if notification of completion was not returned to the Moira.

Considerations

What happens if the Moira broadcasts an invalid data file to servers? In the case of name service, the Moira may not be able to locate the servers again if the name service is lost. Also, if the server machine crashes, it may not be able to come up to full operational capacity if it relies on the databases which have been corrupted; in this case, it is possible that the database may not be easily replaceable. Manual intervention would be required for recovery.

5.9.1. Catastrophic Crashes - Robustness Engineering

In the event of a catostrophic system crash, Moira must have the capability to be brought up with consistent data. There are a list of scenarios which indicate that a complete set of recovery tools are needed to address this isssue. Thought will be given in order that the system reliably is restored. In many cases, the answer to a catastrophic crash will be manual intervention.

5.9.2. Data Transport Security

Kerberos is used to verify the identity of both ends at connection set-up time. The data will not be encrypted. Since a TCP stream is used, the connection should be secure from tampering. This will allow detection of lost or damaged packets, as well as detection of deliberate attempts to damage or change data while it is in transit.

5.10. New User Registration

A new student must be able to get an athena account without any intervention from Athena user accounts staff. This is important, because otherwise, the user accounts people would be faced with having to give out ~1000 accounts or more at the beginning of each term.

With athenareg, a special program (userreg) was run on certain terminals connected to timesharing systems in several of the terminal rooms. It prompted the user for his name and ID number, looked him up in the athenareg database, and gave him an account if he did not have one already. Userreg has been rewritten to work with Moira; in appearance, it is virtually identical to the athenareg version (except in speed).

Athena obtains a copy of the Registrar's list of registered students shortly before registration day each term. Each student on the registrar's tape who has not been registered for an Athena account is added to the "users" relation of the database, and assigned a unique userid; the student is not assigned a login name, and is not known to kerberos. An encrypted form of the student's ID number is stored along with the name; the encryption algorithm is the UNIX C library crypt() function (also used for passwords in /etc/passwd); the last seven characters of the ID number are encrypted using the first letter of the first name and the first letter of the last name as the "salt". No other database resources are allocated at that time.

The Moira database server machine runs a special "registration server" process, which listens on a well known UDP port for user registration requests. There are currently three defined requests:

Verify User	This operation take the user's first name, last name, and authenticator as arguments. The return value will indicate if the user is found in the database, and if so the user's current status.
Grab Login	This operation takes the user's first name, last name, and authenticator with desired login name as arguments. It will attempt to assign the user that login name, and reserve the name in the kerberos database. Expected results are success or login name already taken.
Set Password	This operation takes the user's first name, last name, and authenticator with desired password as arguments. It will attempt to set the user's password in kerberos.

The authenticator used in the protocol is an encrypted form of the ID number and any additional arguments. The simple authenticator is the ID in plaintext with hyphens removed, with the encrypted ID number appended to it, and this whole quantity DES encrypted using the encrypted ID number as a key. This DES encryption is the error propagating cypher-block-chaining mode of DES, as described in the Kerberos document.

$\{\text{IDnumber, hashIDnumber}\}_{\text{hashIDnumber}}$
where
IDnumber is the student's id number (for example: 123456789)
hashIDnumber is the encrypted ID number (for example: lfIenQqC/O/OE)

For the second and third request types, the login or password is also encrypted:

$\{\text{IDnumber, hashIDnumber, login}\}_{\text{hashIDnumber}}$
$\{\text{IDnumber, hashIDnumber, password}\}_{\text{hashIDnumber}}$

The registration server communicates with the kerberos admin_server, and sets up a secure communication channel using "srvtab-srvtab" authentication. In all cases, the server first verifies the request by decrypting the ID number.

When the student decides to register with athena, he walks up to a workstation and logs in using the username of "register", password "athena". This pops up a forms-like interface which prompts him for his first name, middle initial, last name, and student ID number. It calculates the hashed id number using crypt(), and sends a verify_user request to the registration server. The server responds with one of already_registered, not_found, or OK.

If the user has been validated, userreg then prompts him for his choice in login names. It then goes through a two-step process to verify the login name: first, it tries to get initial tickets for the user name from Kerberos; if this fails (indicating that the username is free and may be registered), it then sends a grab_login request. On receiving a grab_login request, the registration server then proceeds to register the login name in the Moira database. If this succeeds, it then reserves the name with kerberos as well.

Userreg then prompts the user for an initial password, and sends a set_password request to the registration server, which decrypts it and forwards it to Kerberos. At this point, pending propagation of information to hesiod, the mail hub, and the user's home fileserver, the user has been established.

6. Data Fields and Relationships

The knowledge base of Moira enables system services and servers to be updated with correct information. The database has the reponsibility of storing information which will be transmitted to the services. The database will not, however, be responsible for knowing the format of data to be sent. This information will be inherent to the Data Control Manager. Specific fields of the database are organized to represent the needs of system. The current Moira database is comprised of the following tables:

Table	Fields and Description	
USERS	User Information. There are two types of user required information: information necessary to identify a user and enable a user to obtain a service (e.g. to login), and personal information about the user (finger).	
	login	a unique username, equivalent to the user's Kerberos principal name.
	users_id	an internal database indentifier for the user record. This is not the same as the Unix uid.
	uid	Unix uid. Temporarily necessary due to NFS client code problems. Ultimately, this field will be removed and uids will be assigned arbitrarily for each client-server connection.
	shell	the user's default shell.
	last, first, middle	The user's full name, broken down for convenient indexing.
	status	contains the user's account status. Currently defined statuses are: 0 - Not registered, but registerable 1 - Active account 2 - Half-registered 3 - Marked for deletion 4 - Not registerable Only accounts in state 1 will have their information show up in database extracts for system services.
	mit_id	the user's encrypted MIT id.
	mit_year	a student's academic year, not modifiable by the student. Used for Athena administrative purposes.
	modtime	the time that this record was last modified (or created).
	modby	the person who modified this record last.
	modwith	the service that modified this record last.
	fullname	the user's full name.
	nickname	the user's nickname.
	home_addr	home address.
	home_phone	home phone.

office_addr	office address.
office_phone	office phone.
mit_dept	MIT address; this is for on-campus students' living addresses.
mit_affil	one of undergraduate, graduate, staff, faculty, other.
fmodtime	time finger record was last modified.
fmodby	person who made this modification.
fmodwith	service that made this modification.
potype	mailbox type: one of POP, SMTP, or NONE.
pop_id	Machine ID of current or last known POP server.
box_id	String ID of box name if type is SMTP.
pmodtime	time pobox record was last modified.
pmodby	person who made this modification.
pmodwith	service that made this modification.
gid	*[unused]* intended for optimized user groups, but never implemented.
uglist_id	*[unused]* intended for optimized user groups, but never implemented.
ugdefault	*[unused]* intended for optimized user groups, but never implemented.

See Section 7.0.1 for the list of queries associated with this table.

There is no entry for a password because it is being subsumed by another service (Kerberos).

MACHINE Machine Information.

name	the canonical hostname.
mach_id	an internal database id for this record.
type	machine type: VAX, RT
modtime	the time that this record was last modified (or created).
modby	the person who modified this record last.
modwith	the service that modified this record last.

See Section 7.0.2 for the list of queries associated with this table.

CLUSTER Cluster Infomation. There are several named clusters throughout Athena that correspond roughly to subnets and/or geographical areas.

name	cluster name.
clu_id	internal database identifier for this record.
desc	cluster description.

	location	cluster location.
	modtime	the time that this record was last modified (or created).
	modby	the person who modified this record last.
	modwith	the service that modified this record last.

See Section 7.0.2 for the list of queries associated with this table.

MCMAP

Machine-Cluster Map. This tables is used to assign machines to clusters.

mach_id	machine id.
clu_id	cluster id.

See Section 7.0.2 for the list of queries associated with this table.

SVC

For each cluster there is a set of services that serve the machines in that cluster. These services are described by an environment variable (to be set on client workstations at login) and a *service cluster* name. Use of the service cluster name is service dependent but in general refers to a group of entities that provide the named service in the particular cluster.

clu_id	references an entry in the cluster table.
serv_label	label of a service cluster type (e.g. "usrlib", "syslib", "zephyr")
serv_cluster	specific service cluster data (e.g. "bldgw20-vssys")

See Section 7.0.2 for the list of queries associated with this table.

LIST

Lists are used as a general purpose means of grouping serveral objects togther. This table contains descriptive information for each list; the MEMBERS table contains the the list of objects that are in the list. The ability to add or delete objects in a list is controlled by an access control entity associated with the list. The access control entity may be a USER, a LIST, or NONE. An access control entity names the user or list of users who have the capability to manipulate the object specifying the access control list.

name	list name.
list_id	internal database id for this list.
active	Indicates this list should be extracted in service updates.
public	Indicates any user may add or delete themselves as members of this list.
hidden	Indicates that neither the list information or membership may be divulged to anyone who is not an administrator of this list.
maillist	Indicates that this list is a maillist.
group	Indicates that this list is a unix group.

gid	Unix GID, if this list is a unix group. This will have value -1 to indicate: assign a unique GID if this list is ever made a unix group.
desc	description of list.
acl_type	Type of access control entity: LIST, USER, or NONE.
acl_id	Access control entity identifier; a list ID, user ID, or ignored if type is NONE.
modtime	the time that this record was last modified (or created).
modby	the person who modified this record last.
modwith	the service that modified this record last.

See Section 7.0.3 for the list of queries associated with this table.

MEMBERS List members. Members are specified by a member type and a member id pair.

list_id	id of a list.
member_type	member type: one of USER, LIST, STRING.
member_id	id of a member (a USERS id, LIST id, or STRING id.)

See Section 7.0.3 for the list of queries associated with this table.

SERVERS Server Information. This table contains information needed by the Data Control Manager or applications for each known service to be updated.

name	name of service.
update_int	server update interval in minutes (for DCM).
target_file	where on the server being updated to put the file generated by the DCM.
script	where on Moira to find the shell script which will install new configuration files on a server.
dfgen	time of last server file generation.
dfcheck	time of last check to see if server files needed to be generated.
type	service type: UNIQUE or REPLICAT(ed).
enable	Enable switch for DCM. This switch controls whether or not the DCM generates files for this service. (0 - Do not Update, 1 - Update)
inprogress	indicator that a DCM is generating new configuration files for this service right now.
harderror	indication that an error has occured while generating files (or while propogating files is service type is replicated). This is not a boolean, but the actual error number.
errmsg	a text description of the *harderror* reported above.

acl_type	type of access control entity for this service.
acl_id	id of access control entity for this service.
modtime	the time that this record was last modified (or created).
modby	the person who modified this record last.
modwith	the service that modified this record last.

See Section 7.0.4 for the list of queries associated with this table.

SERVERHOSTS Server to Host mapping table. Used by the Data Control Manager to map a server to a list of server hosts.

service	name of service.
mach_id	Machine id for a host containing the service.
success	Flag indicating successful completion of most recent server update.
enable	Enable switch for DCM. This switch controls whether or not the DCM updates a server. (0 - Do not Update, 1 - Update)
override	Override flag. Used by DCM and update mechanism to indicate that it should attempt to update this host, even if the necessary time interval has not elapsed.
inprogress	indicator that a DCM is updating this host right now.
hosterror	indication that an error has occured while updating this host. This is not a boolean, but the actual error number.
hosterrmsg	a text description of the *hosterror* reported above.
ltt	Last time tried. Used by the DCM, this field is adjusted each time a service is attemted to be updated, regardless of success or failure.
lts	Last time successful. This records the last time the DCM successfully updated the server.
value1	server-specific integer data used by applications (i.e., number of servers permitted per machine).
value2	additional server-specific integer data.
value3	additional server-specific string data.
modtime	the time that this record was last modified (or created).
modby	the person who modified this record last.
modwith	the service that modified this record last.

See Section 7.0.4 for the list of queries associated with this table.

FILESYS File System Information. This section desribes the file system information necessary for a workstation to attach a file system.

label a name for an attachable file system. This is not necessarily unique.

order an integer used to indicate sort-order for multiple filesystems with the same label. Also, (*label, order*) tuples are unique among filesystems.

filsys_id internal database identifier for the filesystem record.

phys_id internal database identifier for the phsical partition containing the logical filesystem. For filesystems of type NFS this is an nfsphys_id. For other types, it is unused.

type currently one of RVD, NFS, or ERR.

mach_id file server machine.

name name of file system on the server. For type RVD, this is the pack name. For type NFS, this is the directory name on the server.

mount default mount point for file system.

access default access mode for file system.

comments any special notes about the filesystem.

owner this is the users_id of the owner of the filesystem. If the filesystem is automatically created, this user will own it.

owners this is the list_id of the owning group of the filesystem. If the filesystem is automatically created, this group will own it.

createflg indicates that the filesystem should be automatically created if it does not already exist.

lockertype one of HOMEDIR, PROJECT, COURSE, SYSTEM, etc. This may affect what is done when a filesystem is automatically created.

modtime the time that this record was last modified (or created).

modby the person who modified this record last.

modwith the service that modified this record last.

See Section 7.0.5 for the list of queries associated with this table.

NFSPHYS NFS Server Information. This table contains for each nfs server machine a list of the physical device partitions from which directories may be exported. For each such partition an access control list is provided.

nfsphys_id internal database identifier of this parition.

mach_id server machine.

dir	top-level directory of device.
device	partition name.
status	a bit field encoding what the parition is used for. Current assignments are: bit 0 (LSB) - Student lockers bit 1 - Faculty lockers bit 2 - Staff lockers bit 3 - Miscellaneous
allocated	number of quota units allocated to this device.
size	capacity of this device in quota units.
modtime	the time that this record was last modified (or created).
modby	the person who modified this record last.
modwith	the service that modified this record last.

See Section 7.0.5 for the list of queries associated with this table.

NFSQUOTA NFS Server Quota Information. This table contains per user per server quota information.

users_id	user id of account this quota belongs to.
filsys_id	filesys_id of logical filesystem this quota applies to.
phys_id	nfsphys_id of partition that filesystem resides on. This is redundant information, here for performance reasons.
quota	user quota in quota units.
modtime	the time that this record was last modified (or created).
modby	the person who modified this record last.
modwith	the service that modified this record last.

See Section 7.0.5 for the list of queries associated with this table.

ZEPHYR Zephyr class access control list information. This table contains an entry for each controlled class. Each records records an access control entity for each of the four functions of that class.

class	name of the zephyr class.
xmt_type	access control entity type for transmit
xmt_id	access control entity id for transmit
sub_type	access control entity type for subscriptions
sub_id	access control entity id for subscriptions
iws_type	access control entity type for instance wildcard specification

iws_id	access control entity id for instance wildcard specification
iui_type	access control entity type for instance UID identity
iui_id	access control entity id for instance UID identity
modtime	the time that this record was last modified (or created).
modby	the person who modified this record last.
modwith	the service that modified this record last.

See Section 7.0.6 for the list of queries associated with this table.

HOSTACCESS This table contains the necessary information for Moira to be generating /.klogin or /etc/passwd files. It associates an access control entity with a machine.

mach_id	machine.
acl_type	access control entity type
acl_id	access control entity id
modtime	the time that this record was last modified (or created).
modby	the person who modified this record last.
modwith	the service that modified this record last.

See Section 7.0.7 for the list of queries associated with this table.

STRINGS Used for list members of *string* type. An optimization for dealing with foreign mail addresses in poboxes or as list members.

string_id	member id.
string	string.

See Section 7.0.7 for the list of queries associated with this table.

SERVICES TCP/UDP Port Information. This is the information currently in /etc/services. In a workstation environment with Moira and the Hesiod name server, service information will be obtained from the name server.

name	service name.
protocol	protocol: one of TCP, UDP.
port	port number.
desc	description of service.
modtime	the time that this record was last modified (or created).
modby	the person who modified this record last.
modwith	the service that modified this record last.

See Section 7.0.7 for the list of queries associated with this table.

PRINTCAP

Printer capability table. This contains the information currently in /etc/printcap.

name	a unique printer name.
mach_id	server machine.
dir	spooling directory
rp	remote printer name
comments	description of service.
modtime	the time that this record was last modified (or created).
modby	the person who modified this record last.
modwith	the service that modified this record last.

See Section 7.0.7 for the list of queries associated with this table.

CAPACLS

This table associates access control lists with particular capabilities. An important use of this table is for defining the access allowed for executing each of the Moira predefined queries. Each query name appears as a capability name in this list.

capability	a string, usually the full name of a query.
tag	four character tag name for this capability, usually the short name of a query.
list_id	a list id.

See Section 7.0.7 for the list of queries associated with this table.

ALIAS

Aliases are used by several different services to provide alternative names for objects or a mapping one type of object and another. They are also used to record legal values for type-checked fields, and type translations of some type fields.

Some examples of alias usage are file system aliases, service aliases, and printer aliases. Each alias of this form will have the alias as the *name*, a *type* of PRINTER, SERVICE, or FILESYS, and a *trans* of the real name of the object.

All type-checking aliases are of *type* TYPE. The *name* is the name of the type-checked field, and the *trans* is a possible value for that field. Some type-checked fields are "alias", "boolean", "class", "filesys", etc. For example, alias types themselves are typechecked, so that one cannot store an alias of type **aliastype** unless there is an alias entry of the form (alias, TYPE, aliastype).

Type translations are used to record things like the data stored with an SMTP pobox is of type string. These aliases have a *name* which is the type string the user enters, a *type* of TYPEDATA, and a *trans* of the actual data type, i.e. user, list, string, machine, etc.

name	alias name.
type	alias type: currently one of TYPE, PRINTER,

SERVICE, FILESYS, TYPEDATA.

trans alias translation.

See Section 7.0.7 for the list of queries associated with this table.

VALUES Values needed by the server or application programs. These are hints for the next ID number to assign, some state variables such as the dcm enable flag, and values such as the default quota for new users.

name value name.

value value.

See Section 7.0.7 for the list of queries associated with this table.

TBLSTATS Table Statistics. For each table in the Moira database statistics are kept for the number of appends, updates, and deletes performed on the table. In addition, the last modification time of the table is kept.

table table name.

modtime time of last modification (append, update, or delete).

retrieves *obsolete* count of retrievals on this table. This is unused now for performance reasons.

appends count of additions to this table.

updates count of updates to this table.

deletes count of deletions to this table.

See Section 7.0.7 for the list of queries associated with this table.

7. Predefined Queries - List of Database Interfaces

All access to the database is provided through the application library/database server interface. This interface provides a limited set of predefined, named queries, which allows for tightly controlled access to database information. Queries fall into four classes: retrieve, update, delete, and append. An attempt has been made to define a set of queries that provide sufficient flexibility to meet all of the needs of the Data Control Manager and each of the indivual application programs. However, since the database can be modified and extended in the future, the server and application library have been designed to allow for the easy addtion of queries.

Providing a generallized layer of functions affords Moira the capability of being database independent. Today, we are using INGRES; however, in the future, if a different database is required, the application interface will not change. The only change needed at that point will be a new Moira server, linking the pre-defined queries to a new set of data manipulation procedures.

The following list of queries are a predefined list. This list provides the mechanism for reading, writing, updating, and deleting information in the database.

In each query description below there are descriptions of the required arguments, the return values, integrity constraints, possible error codes, and side effects, if any. In addition to the error codes specifically listed for each query, any query may return: MR_PERM "Insufficient permission to perform requested database access" or MR_ARGS "Incorrect number of arguments". Any retrieval query may return MR_NO_MATCH "No records in database match query". Any add or update query may return MR_BAD_CHAR "Illegal character in argument" if a bad character is in an argument that has character restrictions, or MR_EXISTS if the new object to be added or new name of existing object conflicts with another object already in the database. Other errors are listed with each query.

7.0.1. Users, Finger, and Post Office Boxes

get_all_logins

Args: none

Returns: {login, uid, shell, last, first, mi}

Returns info on every account in the database. The returned info is a summary of the account info, not the complete information.

get_all_active_logins

Args: none

Returns: {login, uid, shell, last, first, mi}

Returns info on every account for which the status field is non-zero. The returned info is a summary of the account info, not the complete information.

get_user_by_login

Args: login

Returns: {login, uid, shell, last, first, mi, state, mitid, class, modtime, modby, modwith}

Returns complete account information on the named account. Wildcards may be used in the *login* name specified. If the person executing the query is not on the query ACL, then the query only succeeds if the only retrieved information is about the user making the request.

get_user_by_uid

Args: uid

Returns: {login, uid, shell, last, first, mi, state, mitid, class, modtime, modby, modwith}

Returns complete account information on any account with the specified *uid*. If the person executing the query is not on the query ACL, then the query only succeeds if the only retrieved information is about the user making the request.

get_user_by_name

Args: {first, last}

Returns: {login, uid, shell, last, first, mi, state, mitid, class, modtime, modby, modwith}

Returns complete account information on any account with matching *first* and *last* name fields. Either or both names may contain wildcards, so that this query can do the equivalent of lookup by firstname or lookup by lastname.

get_user_by_class

Args: class

Returns: {login, uid, shell, last, first, mi, state, mitid, class, modtime, modby, modwith}

Returns complete account information on any account with a matching class field. The given class may contain wildcards.

get_user_by_mitid

Args: crypt(id)

Returns: {login, uid, shell, last, first, mi, state, mitid, class, modtime, modby, modwith}

Returns complete account information on any account with a matching MIT ID field. The given id may contain wildcards.

add_user

Args: {login, uid, shell, last, first, mi, state, mitid, class}

Returns: none

Adds a new user to the database. *login* must not match any existing *login*s. *uid* and *state* must be integers. If the given uid is **UNIQUE_UID** as defined in *<moira.h>*, the next unused uid will be assigned. If *login* is **UNIQUE_LOGIN** as defined in *<moira.h>*, the login name will be a "#" followed by the uid. For example, when adding a person so that they may register later, the query ausr(UNIQUE_LOGIN, UNIQUE_UID, /bin/csh, *Last,* *First, M*, 0, [encrypted ID, *class*) is used. The *class* field must contain a value specified as a **TYPE** alias for **class**. This query also initializes the finger record for this user with just their full name, and sets their pobox to **NONE**. It updates the modtime on the user, finger and pobox records. Errors: MR_NOT_UNIQUE "Arguments not unique" if the login name is not unique, or MR_BAD_CLASS "Specified class is not known" if the class is not in the alias database.]

register_user

Args: {uid, login, fstype}

Returns: none

Registers a user. This consists of changing their username, and creating a pobox, a group list, a filesystem, and a quota for them. The user is identified by *uid*, which must match exactly one existing user. Further, this user must currently have a *status* of 0. The user will be left with a *status* of 2. The pobox created will be of type **POP** on the least loaded post office. The group list will have the user as an owner, and a unique GID will be assigned. The filesystem will be allocated on the least loaded fileserver which supports *fstype*, where *fstype* is **MR_FS_STUDENT, MR_FS_FACULTY, MR_FS_STAFF**, or **MR_FS_MISC** as defined in *<moira.h>*. A quota will be assigned to the user on his filesystem with the value taken from **def_quota** in the values table. Errors: MR_NO_MATCH, MR_NOT_UNIQUE "Arguments not unique" if the uid does not specify exactly one user; MR_IN_USE "Object is in use" if the login name is already taken.

update_user

Args: {login, newlogin, uid, shell, last, first, mi, state, mitid, class}

Returns: none

Updates the info in a user entry. *login* specifies the existing login name, the remaining arguments will replace the current values of those fields. This is not equivalent to deleting the user and adding a new one, as all references to this user will still exist, even if the login name is changed. All fields must be specified, even if the value is to remain unchanged. *login* must match exactly one user in the database. *newlogin* must either match the existing login or be unique in the database. The *class* field must contain a value specified as a **TYPE** alias for **class**. *uid* and *state* must be integers. The modtime fields in the user's record will be updated. Errors: MR_USER "No such user" if the login name does not match exactly one user, MR_NOT_UNIQUE "Arguments not unique" if the new login name is not unique, or MR_BAD_CLASS "Specified class is not known" if the class is not in the alias database.

update_user_shell

Args: {login, shell}

Returns: none

Updates a user's shell. *login* must match exactly one user. The modtime fields in the user's record will be updated. This query may be executed by the target user or by someone on the query ACL. Errors: MR_USER "No such user" if the login name does not match exactly one user.

update_user_status

Args: {login, status}

Returns: none

Updates a user's *status*. *login* must match exactly one user. The modtime fields in the user's record will be updated. Errors: MR_USER "No such user" if the login name does not match exactly one user.

delete_user

Args: login

Returns: none

Deletes a user record. *login* must match exactly one user. **The user must have a status of zero, or MR_IN_USE will be returned.** This will only be allowed if the user is not a member of any lists, has any quotas assigned, or is the owner of an object. It will also delete associated finger information, post office box, and any quotas the user has. Errors: MR_USER "No such user" if the login name does not match exactly one user, MR_IN_USE "Object is in use" if the user is a member of a list, has a quota or is an ACE.

delete_user_by_uid

Args: uid

Returns: none

Deletes a user record. *uid* must match exactly one user. This will only be allowed if the user is not a member of any lists or is the owner of an object. It will also delete associated finger information and post office box. Errors: MR_USER "No such user" if the uid does not match exactly one user, MR_IN_USE "Object is in use" if the user is a member of a list or is an ACE.

get_finger_by_login

Args: login

Returns: {login, fullname, nickname, home_addr, home_phone, office_addr, office_phone, department, affiliation, modtime, modby, modwith}

Gets all of the finger information for the specified user. *login* must match exactly one user. The target user may retrieve his information. It is safe to point the query ACL at the list of all users. Errors: MR_USER "No such user" if the login name does not match exactly one user.

update_finger_by_login

Args: {login, fullname, nicname, home_addr, home_phone, office_addr, office_phone, department, affiliation}

Returns: none

Allows any part of the finger information to be changed for a specified account. *login* must match exactly one user. The remaining fields are free-form, and may contain anything. The modtime fields in the finger record will be updated. A user may update his own information. Errors: MR_USER "No such user" if the login name does not match exactly one user.

get_pobox

Args: login

Returns: {login, type, box, modtime, modby, modwith}

Retrieves a user's post office box assignment. The *login* name must match exactly one user. See *set_pobox* for a summary of the returned fields. The owner of the pobox may perform this query. Errors: MR_USER "No such user" if the login name does not match exactly one user.

get_all_poboxes

Args: none

Returns: {login, type, box}

Retrieves all of the post office boxes from the database. See *set_pobox* for a summary of the returned fields.

get_poboxes_pop

Args: none

Returns: {login, type, machine}

Retrieves all of the post office boxes of type **POP** from the database. See *set_pobox* for a summary of the returned fields.

get_poboxes_smtp

Args: none

Returns: {login, type, box}

Retrieves all of the post office boxes of type **SMTP** from the database. See *set_pobox* for a summary of the returned fields.

set_pobox

Args: {login, type, box}

Returns: none

Establishes a user's post office box. The given *login* must match exactly one user. The *type* will be checked against the alias database for valid **pobox** types. Currently allowed types are **POP**, **SMTP**, and **NONE**. If the type is **POP**, then box must name a machine known by Moira. If the type is **SMTP**, then box is the user's mail address with no other interpretation by Moira. A type of **NONE** is the same as not having a pobox. The modtime fields on the pobox record will be set. The owner of the target pobox may perform this query. Errors: MR_USER "No such user" if the login name does not match exactly one user, or MR_TYPE "Invalid type" if the type is not **POP**, **SMTP**, or **NONE**.

set_pobox_pop

Args: login

Returns: none

Forces a user's pobox to be type **POP**. The *login* name must match exactly one user. If the user's pobox is already of type **POP**, nothing will be changed. If the user has previously had a pobox of type **POP**, then the previous post office machine assignment will be restored. If there was no previous post office assignment, the query will fail with MR_MACHINE "Unknown machine" since it will be unable to choose a post office machine. The modtime fields on the pobox record will be set. The owner of the target pobox may perform this query. Errors: MR_USER "No such user" if login does not match exactly one user, or MR_MACHINE.

delete_pobox

Args: login

Returns: none

Effectively deletes a user's pobox, by setting the type to **NONE**. The *login* name must match exactly one user. The modtime fields on the pobox record will be set. The owner of the target pobox may perform this query. Errors: MR_USER "No such user" if login does not match exactly one user.

7.0.2. Machines and Clusters

get_machine

Args: name

Returns: {name, type, modtime, modby, modwith}

Get all the information on the specified machine(s). Wildcarding may be used in the machine *name*. All machine names are case insensitive, and are returned in uppercase. It is safe for the query ACL to be the list containing everybody.

add_machine

Args: {name, type}

Returns: none

Enters a new machine into the database. The given *name* will be converted to uppercase. Then it will be checked for uniqueness in the database. The *type* field will be checked against the aliases database for valid **mach_type**s. Currently defined **mach_types** are **RT** and **VAX**. The modtime fields will be set. Errors: MR_NOT_UNIQUE "Arguments not unique" if a machine with the given *name* already exists, or MR_TYPE "Invalid type" if the given *type* is not in the alias database.

update_machine

Args: {name, newname, type}

Returns: none

Update the information on a machine. The *name* must match exactly one machine. The *newname* must either be the same as the old name, or must be unique among machine names in the database after being converted to uppercase. The type field will be checked against the aliases database for valid **mach_type**s. The modtime fields will be set. Errors: MR_MACHINE "No such machine" if the old *name* does not match exactly one machine, MR_NOT_UNIQUE "Arguments not unique" if the *newname* does not either match the old name or is unique, or MR_TYPE "Invalid type" if the given *type* is not in the alias database.

delete_machine

Args: name

Returns: none

Delete a machine from the database. The given *name* must match exactly one machine. A machine that is in use (post office, file system, printer spooling host, server_host_access, or DCM service update) cannot be deleted. Errors: MR_MACHINE "No such machine" if the *name* does not match exactly one machine, or MR_IN_USE "Object is in use" if the machine is being referenced as a post office, filesystem, spooling host, or server updated by the DCM.

get_cluster

Args: name

Returns: {name, description, location, modtime, modby, modwith}

Returns all the information in the database about one or more clusters. The cluster *name* may contain wildcards. It is safe for the query ACL to be the list containing everybody.

add_cluster

Args: {name, description, location}

Returns: none

Adds a new cluster to the database. The *name* must be unique among the existing cluster names. The names are case sensitive. There are no constraints on the remaining data. The modtime fields will be set. Errors: MR_NOT_UNIQUE "Arguments not unique" if the cluster *name* is not unique.

update_cluster

Args: {name, newname, description, location}

Returns: none

Changes the information about a cluster. The old *name* must match exactly one cluster. The *newname* must either match the old name or be unique among the existing cluster names. The names are case sensitive. There are no constraints on the remaining data. The modtime fields will be set. Errors: MR_CLUSTER "Unknown cluster" if the old cluster name does not match exactly one cluster, or MR_NOT_UNIQUE "Arguments not unique" if the new name does not either match the old name or is unique.

delete_cluster

Args: name

Returns: none

Removes a cluster from the database. The *name* must match exactly one cluster. The cluster must not have any machines assigned to it. Any service cluster information assigned to the cluster will be deleted. Errors: MR_CLUSTER "Unknown cluster" if the old cluster name does not match exactly one cluster, or MR_IN_USE "Object in use" if the cluster has machines assigned to it.

get_machine_to_cluster_map

Args: {machine, cluster}

Returns: {machine, cluster}

Retrieves machine to cluster mappings for the specified *machine(s)* and *cluster(s)*. Either of the fields may contain wildcards. It is safe for the query ACL to be the list containing everybody.

add_machine_to_cluster

Args: {machine, cluster}

Returns: none

Add a machine to a cluster. The *machine* name must match exactly one machine. The *cluster* name must match exactly one cluster. The machine's modtime fields will be updated. Errors: MR_MACHINE "No such machine" or MR_CLUSTER "No such cluster" if one of them does not match exactly one object in the database.

delete_machine_from_cluster

Args: {machine, cluster}

Returns: none

Delete a machine from a cluster. The *machine* name must match exactly one machine. The *cluster* name must match exactly one cluster. The named machine must belong to the named cluster. The machine's modtime fields will be updated. Errors: MR_MACHINE "No such machine" or MR_CLUSTER "No such cluster" if one of them does not match exactly one object in the database.

get_cluster_data

Args: {cluster, label}

Returns: {cluster, label, data}

Retrieve all cluster data matching the named *cluster* and *label*. Either or both may use wildcards. Thus all data for a cluster may be retrieved with gcld(cluster, *), and all data of a particular type may be retrieved with gcld(*, label). It is safe for the query ACL to be the list containing everybody.

add_cluster_data

Args: {cluster, label, data}

Returns: none

Add new data to a cluster. The *cluster* name must match exactly one cluster. The service *label* must be a registered **slabel** in the alias database. The *data* is an arbitrary string. The cluster's modtime fields will be updated. Errors: MR_CLUSTER "No such cluster" if the *cluster* name does not match exactly one cluster, or MR_TYPE "Invalid type" if the *label* is not in the alias database.

delete_cluster_data

Args: {cluster, label, data}

Returns: none

Delete the specified cluster data. The *cluster* name must match exactly one cluster, and the remaining two arguments must exactly match an existing service cluster. The cluster's modtime fields will be updated. Errors: MR_CLUSTER "No such cluster" if the *cluster* name does not match exactly one cluster, or MR_NOT_UNIQUE "Arguments not unique" if the *label* and *data* do not match exactly one existing piece of data for the cluster.

7.0.3. Lists

get_list_info

Args: list

Returns: {list, active, public, hidden, maillist, group, gid, ace_type, ace_name, description, modtime, modby, modwith}

Returns information about the named list. The *list* name may contain wildcards. *active, public, hidden, maillist*, and *group* are booleans returned as integers (0 is false, non-zero is true). The *ace-type* is either **USER**, **LIST**, or **NONE**, and the *ace_name* will be either a login name, a list name, or **NONE**, respectively. This query is allowed if the list is not hidden or the user executing the query is on the ACE of the target list. If the user executing this query is on the query ACL, he may use wildcards in the *list* name, otherwise wildcards are not allowed.

expand_list_names

Args: list

Returns: {list}

Expands wildcards in a *list* name. A name is passed which may contain wildcards, and a set of matching names are returned.

add_list

Args: {list, active, public, hidden, maillist, group, gid, ace_type, ace_name, description}

Returns: none

Creates a new list and adds it to the database. The *list* name must be unique among existing list names. *active, public, hidden, maillist,* and *group* are booleans passed as integers (0 is false, non-zero is true). If *group* is true and *gid* is **UNIQUE_GID** as defined in *<mr.h>*, a new unique group ID will be assigned, otherwise the integer value given for *gid* will be assigned to the GID. The *ace-type* is either **USER**, **LIST**, or **NONE**, and the *ace_name* will be either a login name, a list name, or **NONE**, respectively. The access list may be the list that is being created (self-referential). The list modtime will be set. Errors: MR_EXISTS "Record already exists" if a list already exists by that name, MR_ACE "No such access control entity" if the *ace_type* is not **USER**, **LIST**, or **NONE**, or if the *ace_name* cannot be resolved relative to the *ace_type*.

update_list

Args: {list, newname, active, public, hidden, maillist, group, gid, ace_type, ace_name, description}

Returns: none

Allows the list information and attributes to be changed. This is not equivalent to deleting the list and creating a new one, since references to the old name will still apply to the new name if it is renamed. The *list* name must match exactly one list. The *new name* must either match the old name or be unique among list names in the database. The *active, public, hidden, maillist*, and *group* flags should be 0 if the flag is false, or non-zero if it is true. The *gid* may be **UNIQUE_GID** as defined in *<mr.h>*, in which case a new unique GID will be assigned. The *ace-type* is either **USER**, **LIST**, or **NONE**, and the *ace_name* will be either a login name, a list name, or **NONE**, respectively. The list modtime will be set. This query may be executed by anyone on the ACE of the target list. Errors: MR_LIST "No such list" if the named list does not match exactly one list, MR_NOT_UNIQUE "Arguments not unique" if the new list name doesn't either match the old one or is unique, MR_ACE "No such access control entity" if the *ace_type* is not **USER**, **LIST**, or **NONE**, or if the *ace_name* cannot be resolved relative to the *ace_type*.

delete_list

Args: list

Returns: none

Deletes a list from the database. A list may only be deleted if it is not in use as a member of any other list or as an ACL for an object, and the list itself must be empty. This query may be executed by someone who is on the current ace of the target list. Errors: MR_LIST "No such list" if the named list does not exist, or MR_IN_USE "Object is in use" if the list is still being referenced.

add_member_to_list

Args: {list, type, member}

Returns: none

Adds a member to a list. The specified *list* must match exactly one list. Type must be either **USER**, **LIST**, or **STRING**. *member* is either a login name, a list name, or a text string, respectively. The modtime on the list is updated. This query may be executed by: anyone adding themselves as a *USER* to a list with the *public* bit set or anyone on the ACE of the list being modified. Errors: MR_LIST "No such list" if the named list does not exist, MR_TYPE "Invalid type" if the member *type* is not **USER**, **LIST**, or **STRING**, or MR_NO_MATCH "No records in database match query" if the *member* name cannot be resolved with the member *type*.

delete_member_from_list

Args: {list, type, member}

Returns: none

Deletes a member from a list. The specified *list* must match exactly one list. The specified *type* and *member* must exactly match an existing member of that list. The modtime on the list is updated. This query may be executed by anyone deleting themselves as a **USER** from a list with the *public* bit set or anyone on the ACE of the list being modified. Errors: MR_LIST "No such list" if the named list does not exist, MR_TYPE "Invalid type" if the member type is not **USER**, **LIST**, or **STRING**, or MR_NO_MATCH "No records in database match query" if the *member* name cannot be resolved with the member *type* or if there is no such member in the list.

get_ace_use

Args: {ace_type, ace_name}

Returns: {object_type, object_name}

Finds references to an object as an ACE. Valid *ace_types* are **USER**, **LIST**, **RUSER**, and **RLIST**. For types **USER** and **RUSER**, the *ace_name* must be a login name. If the type is **USER**, then only objects whose ACE is the named user will be found; if it is **RUSER**, it will recursively check down sub-lists of the ACE lists looking to see if the user is a member of that ACE. The types **LIST** and **RLIST** apply to a list name in a similar manner. The returned tuples will be **LIST**, list name; **SERVICE**, service name; **FILESYS** and a filesystem label; **QUERY**, query name; **HOSTACCESS**, machine name; or **ZEPHYR**, zephyr class name. This query may be executed by a user asking about himself or a person on an ACE of a list asking about that list. Errors: MR_TYPE "Invalid type" if the *ace_type* is not one of **LIST**, **RLIST**, **USER**, or **RUSER**; MR_NO_MATCH "No objects in database match query" if the *ace_name* doesn't match a user or list.

qualified_get_lists

Args: {active, public, hidden, maillist, group}

Returns: {list}

Finds the names of any lists that meet the specified criteria. Each of the inputs may be one of **TRUE, FALSE**, or **DONTCARE**. Any user may execute this query with *active* **TRUE** and *hidden* **FALSE**. Errors: MR_TYPE "Invalid type" if one of the arguments is something other than **TRUE**, **FALSE**, or **DONTCARE**.

get_members_of_list

Args: list

Returns: {type, value}

Retrieves all of the members of the named list. The *list* must match exactly one list in the database. The returned pairs consist of the type **USER**, **LIST**, or **STRING**, followed by the login name, list name, or text string respectively. This query may be executed by anyone if the list is not hidden; otherwise by someone on the ACE of the list being modified.

get_lists_of_member

Args: {type, value}

Returns: {list, active, public, hidden, maillist, group}

Retrieves the name and flags of every list containing the named member. The member *type* must be one of **USER**, **LIST**, or **STRING**, and the *value* a login name, list name, or text string respectively. The *type* may also be one of **RUSER, RLIST**, or **RSTRING**, in which case it will also find any lists that contain as sublists a list that the target is a member of. This query may be executed by someone asking about themselves or a person on the ace of a list asking about that list. Errors: MR_TYPE "Invalid type" if the *type* is not **USER, LIST, STRING, RUSER, RLIST**, or **RSTRING**; or MR_NO_MATCH "No records in database match query" if *value* does not match an existing value for the given type.

count_members_of_list

Args: list

Returns: {count}

Determines how many members are on the specified list. The *list* name must match exactly one list. This query may be executed by a anyone on a visible list, or someone on the ACE of the target list. MR_LIST "Invalid list" if the list name does not match exactly one list.

7.0.4. Servers and Serverhosts

get_server_info

Args: name

Returns: {service, interval, target, script, dfgen, dfcheck, type, enable, inprogress, harderror, errmsg, ace_type, ace_name, modtime, modby, modwith}

Retrieves service information from the database. This is the per-service information used by the DCM for updates. The service *name* may contain wildcards. Note that all service names are stored in uppercase, and the passed name will be upper-cased before comparing it. The returned *interval* is in minutes. *dfgen* is the data files were last generated, and *dfcheck* is the date we last checked to see if we needed to generate them. These are passed as integers (unix format date). The *type* must be a **service-type** as stored in the aliases database. *enable, inprogress*, and *harderror* are booleans (0 = false, non-zero = true). *ace_type* is either **USER, LIST**, or **NONE**, and *ace_name* is a login name, a list name, or **NONE**, respectively. This query may be executed by someone on the service ace if only one service is retrieved.

qualified_get_server

Args: {enable, inprogress, harderror}

Returns: service

Finds the names of any services that meet the specified criteria. Each of the inputs may be one of **TRUE, FALSE**, or **DONTCARE**. Errors: MR_TYPE "Invalid type" if any of the flags are not one of the three legal values.

add_server_info

Args: {service, interval, target, script, type, enable, ace_type, ace_name}

Returns: none

Adds a new service to the database. This is the per-service information used by the DCM for updates. Note that only a subset of the information is added in this query, as the remaining fields are only changed by the DCM with the *set_server_internal_flags* query. The *service* name will be converted to uppercase. The *interval* is in minutes. The *type* must be a **service-type** as stored in the aliases database. *Enable* is a boolean (0 = false, non-zero = true). *ace_type* is either **USER, LIST**, or **NONE**, and *ace_name* is a login name, a list name, or **NONE**, respectively. The service modtime will be set. Errors: MR_TYPE "Invalid type" if the type field is not a valid **service-type** in the alias database, or MR_ACE "No such access control entity" if the *ace_type* is not **USER, LIST**, or **NONE** or the *ace_name* cannot be resolved based on the *ace_type*.

update_server_info

Args: {service, interval, target, script, type, enable, ace_type, ace_name}

Returns: none

Updates a service in the database. This is the per-service information used by the DCM for updates. Note that only a subset of the information can be modified with this query, as the remaining fields are only changed by Moira itself. The *service* name must match exactly one existing service after being converted to uppercase. The *interval* is in minutes. The *type* must be a **service-type** as stored in the aliases database. *Enable* is a boolean (0 = false, non-zero = true). *Ace_type* is either **USER, LIST**, or **NONE**, and *ace_name* is a login name, a list name, or **NONE**, respectively. The service modtime will be set. This query may be used by someone on the ACE of the target service. Errors: MR_TYPE "Invalid type" if the *type* field is not a valid **service-type** in the alias database, or MR_ACE "No such access control entity" if the *ace_type* is not **USER, LIST**, or **NONE** or the *ace_name* cannot be resolved based on the *ace_type*.

reset_server_error

Args: service

Returns: none

Updates the specified service by changing the harderror flag from **TRUE** to **FALSE**, and sets *dfcheck* to be the same as *dfgen*. The *service* name must match exactly on existing service after being converted to uppercase. The service modtime will be set. This query may be executed by someone on the ACE of the target service.

set_server_internal_flags

Args: {service, dfgen, dfcheck, inprogress, harderr, errmsg}

Returns: none

Updates the specified service. This is intended to only be used by the DCM, as it changes flags that the user should not have control over. The *service* name must match exactly one existing service after being converted to uppercase. *dfgen* and *dfcheck* are unix format dates (long integers). *inprogress* and *harderr* are booleans (0 = false, non-zero = true). The service modtime will NOT be set.

delete_server_info

Args: service

Returns: none

Deletes a set of service information from the database. The *service* name must match exactly one service in the database after being converted to uppercase. A service may not be deleted if there are any server-hosts assigned to that service, or if the inprogress bit is set for that service. Error: MR_IN_USE "Object is in use" if there are hosts assigned to that service.

get_server_host_info

Args: {service, machine}

Returns: {service, machine, enable, override, success, inprogress, hosterror, errmsg, lasttry, lastsuccess, value1, value2, value3, modtime, modby, modwith}

Retrieves server-host information from the database. This is the per-host information used by the DCM for updates. The given *service* and *machine* names may contain wildcards. *Enable, override, success, inprogress*, and *hosterror* are booleans (0 = false, non-zero = true). *lasttry* and *lastsuccess* are unix format dates (long integers). This query may be executed by someone on the ACE for the target service.

qualified_get_server_host

Args: {service, enable, override, success, inprogress, hosterror}

Returns: {service, machine}

Finds the names of any machine/services pairs that meet the specified criteria. The *service* name may contain wildcards. Each of the remaining inputs may be one of **TRUE, FALSE**, or **DONTCARE**. Errors: MR_TYPE "Invalid type" if any of the flags are not one of the three legal values.

add_server_host_info

Args: {service, machine, enable, value1, value2, value3}

Returns: none

Adds information for a new server-host to the database. This is the per-host information used by the DCM for updates. Note that only a subset of the information is dealt with in this query, as the remaining fields are only changed by the DCM with the *set_server_host_internal* query. *Service* and *machine* must each match exactly one existing service and machine, respectively. *Enable* is a boolean (0 = false, non-zero = true). The 3 values are service specific in function; *value1* and *value2* are integers, *value3* is a string. The server-host's modtime will be set. This query may be used by someone on the ACE for the target service. Errors: MR_SERVICE "Unknown service" if the *service* name does not match exactly one existing service, or MR_MACHINE "Invalid machine" if the *machine* name does not match exactly one machine.

update_server_host_info

Args: {service, machine, enable, value1, value2, value3}

Returns: none

Updates information for a server-host in the database. This is the per-host information used by the DCM for updates. Note that only a subset of the information is dealt with in this query, as the remaining fields are only changed by the DCM with the *set_server_host_internal* query. *Service* and *machine* must each match exactly one existing service and machine, respectively. *Enable* is a boolean (0 = false, non-zero = true). The 3 values are service specific in function; *value1* and *value2* are integers, *value3* is a string. The server-host's modtime will be set. This query may only be executed when the inprogress bit is not currently set for the specified server_host. This query may be used by someone on the ACE for the target service. Errors: MR_SERVICE "Unknown service" if the *service* name does not match exactly one existing service, or MR_MACHINE "Invalid machine" if the *machine* name does not match exactly one machine.

reset_server_host_error

Args: {service, machine}

Returns: none

Resets the hosterr flag for the specified server_host. The *service* and *machine* must each match exactly one service and host. The server_host's modtime will be updated. This query may be used by someone on the ACE for the target service. Errors: MR_SERVICE "Unknown service" if the *service* name does not match exactly one existing service, or MR_MACHINE "Invalid machine" if the *machine* name does not match exactly one machine.

set_server_host_override

Args: {service, machine}

Returns: none

This will set the override flag for a server_host, and start a new DCM running. The *service* and *machine* must each match exactly one service and host. The server_host's modtime will be updated. This query may be used by someone on the ACE for the target service. Errors: MR_SERVICE "Unknown service" if the *service* name does not match exactly one existing service, or MR_MACHINE "Invalid machine" if the *machine* name does not match exactly one machine.

set_server_host_internal

Args: {service, machine, override, success, inprogress, hosterror, errmsg, lasttry, lastsuccess}

Returns: none

Updates the specified service_host. This is intended to only be used by the DCM, as it changes flags that the user should not have control over. The *service* and *host* names name must match exactly one existing service and host each. *override, success, inprogress* and *hosterror* are booleans (0 = false, non-zero = true). *lasttry* and *lastsuccess* are unix format dates (long integers). The service_host modtime will NOT be set. Errors: MR_SERVICE "Unknown service" if the *service* name does not match exactly one existing service, or MR_MACHINE "Invalid machine" if the *machine* name does not match exactly one machine

delete_server_host_info

Args: {service, machine}

Returns: none

Deletes a server-host from the database. The *service* and *machine* names each must match exactly one existing service or host. A server-host may not be deleted if the inprogress bit is set for that server-host. This query may be used by someone on the ACE for the target service. Errors: MR_SERVICE "Unknown service" if the *service* name does not match exactly one existing service, MR_MACH "Invalid machine" if the *machine* name does not match exactly one machine, or MR_IN_USE "Object is in use" if the inprogress bit is set.

get_server_locations

Args: service

Returns: {service, machine}

This query tells which machines support a given service. It does this by listing each of the server-hosts for that service. The *service* name may contain wildcards, and will be converted to uppercase before any comparisons are made. It is safe for this query's ACL to be the list containing everybody.

7.0.5. Filesystems

get_filesys_by_label

Args: name

Returns: {name, fstype, machine, packname, mountpoint, access, comments, owner, owners, create, lockertype, modtime, modby, modwith}

Retrieves all the information about a specific filesystem from the database. The *name* may contain wildcards. *fstype* is one of **NFS** or **RVD**, recorded as aliases of **filesys**. *machine* must match exactly one existing machine. *owner* must match exactly one user, *owners* must match exactly one list. *create* is a boolean (0 = false, non-zero = true) indicating that the locker should be automatically created. *lockertype* is a **lockertype** as recorded in the alias database, currently one of **SYSTEM, HOMEDIR, PROJECT**, or **OTHER**. The *packname, mountpoint*, and *access* vary depending on the filesystem type.

get_filesys_by_machine

Args: machine

Returns: {name, fstype, machine, packname, mountpoint, access, comments, owner, owners, create, lockertype, modtime, modby, modwith}

Retrieves the information about any filesystems on the named machine. The *machine* name must match exactly one machine in the database. The returned information is as specified above for *get_filesys_by_label*. Errors: MR_MACHINE "No such machine" if the named machine does not match an existing machine.

get_filesys_by_nfsphys

Args: {machine, partition}

Returns: {name, fstype, machine, packname, mountpoint, access, comments, owner, owners, create, lockertype, modtime, modby, modwith}

Retrieves the information about all NFS filesystems that reside on the specified NFS server partition. *machine* must match exactly one machine. *partition* is the mount point of the NFS physical partition. Errors: MR_MACHINE "Invalid machine" if the machine name does not match exactly one machine, or MR_NO_MATCH "No records in database match query" if the partition does not match anything.

get_filesys_by_group

Args: list

Returns: {name, fstype, machine, packname, mountpoint, access, comments, owner, owners, create, lockertype, modtime, modby, modwith}

Retrieves the information about all filesystems that have the specified group as the owners list. The *list* must match exactly one existing list. This query may be executed by a member of the target list. Errors: MR_LIST "No such list" if the given list does not match exactly one list in the database.

add_filesys

Args: {name, fstype, machine, packname, mountpoint, access, comments, owner, owners, create, lockertype}

Returns: none

Adds a new filesystem to the database. The *name* must be unique among the existing filesystems. *fstype* is one of **NFS** or **RVD**. *machine* must match exactly one existing machine. *owner* must match exactly one user, *owners* must match exactly one list. *create* is a boolean (0 = false, non-zero = true) indicating that the locker should be automatically created. *lockertype* is a **lockertype** as recorded in the alias database, currently one of **SYSTEM, HOMEDIR, PROJECT**, or **OTHER**. The *packname* and *access* vary depending on the filesystem type. For an RVD filesystem, they may contain anything. For NFS filesystems, the *packname* must match an existing NFS physical filesystem, and *access* must be one of **r** or **w**. The filesystem's modtime will be set. Errors: MR_FSTYPE "Invalid filesys type" if the *fstype* is not a valid **filesys** type, MR_TYPE "Invalid type" if the *lockertype* is not a valid **lockertype**, MR_MACHINE "No such machine" if the *machine* name does not match exactly one machine, MR_USER "No such user" if the *owner* does not match exactly one user, MR_LIST "No such list" if the *owners* does not match exactly one list, MR_NFS "Specified directory not exported" if the *machine* and *packname* do not match an existing NFS physical partition, or MR_FILESYS_ACCESS if the *fstype* is **NFS** and the access mode is not **r** or **w**.

update_filesys

Args: {name, newname, fstype, machine, packname, mountpoint, access, comments, owner, owners, create, lockertype}

Returns: none

Updates the information about a filesystem in the database. The *name* must match exactly one existing filesystem. The *new name* must either match the existing one or be unique among the filesystems. *fstype* is one of **NFS** or **RVD**. *machine* must match exactly one existing machine. *owner* must match exactly one user, *owners* must match exactly one list. *create* is a boolean (0 = false, non-zero = true) indicating that the locker should be automatically created. *lockertype* is a **lockertype** as recorded in the alias database, currently one of **SYSTEM, HOMEDIR, PROJECT**, or **OTHER**. The *packname* and *access* vary depending on the filesystem type. For an **RVD** filesystem, they may contain anything. For **NFS** filesystems, the *packname* must match an existing NFS physical filesystem, and *access* must be one of **r** or **w**. The filesystem's modtime will be updated. Errors: MR_NOT_UNIQUE "Arguments not unique" if the *new name* does not either match the old one or is unique among filesystems, MR_FSTYPE "Invalid filesys type" if the *fstype* is not a valid **filesys** type, MR_TYPE "Invalid type" if the *lockertype* is not a valid **lockertype**, MR_MACHINE "No such machine" if the *machine* name does not match exactly one machine, MR_USER "No such user" if the *owner* does not match exactly one user, MR_LIST "No such list" if the *owners* does not match exactly one list, MR_NFS "Specified directory not exported" if the *machine* and *packname* do not match an existing NFS physical partition, or MR_FILESYS_ACCESS if the *fstype* is **NFS** and the *access* mode is not **r** or **w**.

delete_filesys

Args: name

Returns: none

Deletes a filesystem from the database. The *name* must match exactly one existing filesystem. Any quotas assigned to that filesystem will be deleted, and the allocation count on the nfs physical partition will be decremented accordingly. Errors: MR_FILESYS "No such file system" if the name does not match an existing filesystem.

get_all_nfsphys

Args: none

Returns: {machine, dir, device, status, allocated, size, modtime, modby, modwith}

Retrieves information about NFS physical filesystems. These are the filesystems which are exported by NFS servers.

get_nfsphys

Args: {machine, dir}

Returns: {machine, dir, device, status, allocated, size, modtime, modby, modwith}

Retrieves information about a specific NFS physical filesystem. The *machine* must match exactly one existing machine. The *dir*ectory name may contain wildcards. Errors: MR_MACHINE "No such machine" if the *machine* name does not match exactly one existing machine.

add_nfsphys

Args: {machine, directory, device, status, allocated, size}

Returns: none

Adds a new NFS physical filesystem to the database. The *machine* name must match exactly one existing machine. The *directory* and *device* must be unique among existing NFS physical filesystems for this machine. *status* is an integer, with bit encodings **MR_FS_STUDENT, MR_FS_FACULTY, MR_FS_STAFF**, or **MR_FS_MISC** as defined in *<mr.h>*. *allocated* keeps track of quota allocation, the initial value should be zero unless there is something besides lockers on this filesystem. *size* is the actual size (in blocks) of the filesystem. The modtime will be set for this filesystem. Errors: MR_MACHINE "No such machine" if the *machine* name does not match exactly one existing machine.

update_nfsphys

Args: {machine, directory, device, status, allocated, size}

Returns: none

Changes information about an NFS physical filesystem in the database. The *machine* name must match exactly one existing machine. The *directory* must match an existing NFS physical filesystem on that machine. The remaining arguments will replace the current values of those fields. The modtime will be updated for this filesystem. Errors: "No such machine" if the *machine* name does not match exactly one existing machine.

adjust_nfsphys_allocation

Args: {machine, directory, delta}

Returns: none

Changes the allocation for an NFS physical filesystem. *machine* must match exactly one existing machine. *directory* must match an existing NFS physical filesystem on that machine. The current allocation for this filesystem will have *delta* (which may be positive or negative) added to it. Errors: MR_MACHINE "No such machine" if the *machine* name does not match exactly one existing machine.

delete_nfsphys

Args: {machine, directory}

Returns: none

Deletes an NFS physical filesystem from the database. The *machine* name must match exactly one existing machine. The *directory* name must match exactly one existing NFS physical filesystem on that machine. The physical filesystem must not be in use with logical filesystems. Errors: MR_MACHINE "No such machine" if the *machine* name does not match exactly one existing machine, or MR_IN_USE "Object is in use" if there are any filesystems assigned to this partition.

get_nfs_quota

Args: {filesys, login}

Returns: {filesys,login, quota, directory, machine, modtime, modby, modwith}

Retrieves the quotas assigned to the named filesystems and user. The *filesystem* name may contain wildcards. The *login* name must match exactly one user. This query may be executed by the owner of the target filesystem.

get_nfs_quotas_by_partition

Args: {machine, directory}

Returns: {filesys, login, quota, directory, machine}

Retrieves the quotas assigned to a given device. The *machine* must match exactly one existing machine. The *directory* name may contain wildcards. Errors: MR_MACHINE "No such machine" if the *machine* name does not match exactly one existing machine.

add_nfs_quota

Args: {filesystem, login, quota}

Returns: none

Adds a new quota to the database. The *filesystem* name must match exactly one existing filesystem. The *login* name must match exactly one existing user. The *quota* may be any positive integer. The modtime on the quota record will be set. The allocation count for that NFS physical filesystem will also be updated. Errors: MR_FILESYS "No such file system" if the *filesystem* does not match exactly one existing filesystem, or MR_USER "No such user" if the *login* name does not match exactly one existing user.

update_nfs_quota

Args: {filesystem, login, quota}

Returns: none

Changes a quota in the database. The *filesystem* name must match exactly one existing filesystem. The *login* name must match exactly one existing user, and that user must have a quota assigned on that filesystem. The quota may be any positive integer, and will replace the existing quota. The modtime on the quota record will be set. The allocation count for that NFS physical filesystem will also be updated. Errors: MR_FILESYS "No such file system" if the *filesystem* does not match exactly one existing filesystem, or MR_USER "No such user" if the *login* name does not match exactly one existing user.

delete_nfs_quota

Args: {filesystem, login}

Returns: none

Deletes a quota from the database. The *filesystem* name must match exactly one existing filesystem. The *login* name must match exactly one existing user, and that user must have a quota assigned on that filesystem. The allocation count for that NFS physical filesystem will also be updated. Errors: MR_FILESYS "No such file system" if the *filesystem* does not match exactly one existing filesystem, or MR_USER "No such user" if the *login* name does not match exactly one existing user.

7.0.6. Zephyr

get_zephyr_class

Args: class

Returns: {class, xmttype, xmtname, subtype, subname, iwstype, iwsname, iuitype, iuiname, modtime, modby, modwith}

Retrieves zephyr class information from the database. The *class* name may contain wildcards. There are four pairs of types and names: each type is one of **USER, LIST**, or **NONE**, and each name is a login name, a list name, or **NONE**, respectively.

add_zephyr_class

Args: {class, xmttype, xmtname, subtype, subname, iwstype, iwsname, iuitype, iuiname}

Returns: none

Adds a new zephyr class to the database. The *class* name must be unique among the existing class names. There are four pairs of types and names: each type is one of **USER, LIST**, or **NONE**, and each name is a login name, a list name, or **NONE**, respectively. The class's modtime will be updated.

update_zephyr_class

Args: {class, newclass, xmttype, xmtname, subtype, subname, iwstype, iwsname, iuitype, iuiname}

Returns: none

Change a zephyr class in the database. The *class* name must match exactly one existing class. The *new class* name must either match the old one or be unique among the existing class names. There are four pairs of types and names: each type is one of **USER, LIST**, or **NONE**, and each name is a login name, a list name, or **NONE**, respectively. The class's modtime will be updated.

delete_zephyr_class

Args: class

Returns: none

Deletes a zephyr class from the database. The *class* name must match exactly one existing class.

7.0.7. Miscellaneous

get_server_host_access

Args: machine

Returns: {machine, ace_type, ace_name, modtime, modby, modwith}

Returns information about who has access to a given machine. This will be used to load the */.klogin* file on that machine. The *machine* name may contain wildcards. The *ace_type* is either **USER**, **LIST**, or **NONE**, and the *ace_name* is either a login name, a list name, or **NONE**, respectively.

add_server_host_access

Args: {machine, ace_type, ace_name}

Returns: none

Adds information about who has access to a given machine to the database. The *machine* name must match exactly one existing machine. The *ace_type* is either **USER, LIST**, or **NONE**, and the *ace_name* is either a login name, a list name, or **NONE**, respectively. The modtime on the record will be set. Errors: MR_MACHINE "No such machine" if the *machine* name does not match exactly one existing machine, MR_ACE "Invalid access control entity" if the *ace_type* and *ace_name* together do not specify a valid entity.

update_server_host_access

Args: {machine, ace_type, ace_name}

Returns: none

Updates the information about who has access to a given machine. The *machine* name must match exactly one existing machine. The *ace_type* is either **USER, LIST**, or **NONE**, and the *ace_name* is either a login name, a list name, or **NONE**, respectively. The modtime on the record will be updated. Errors: MR_MACHINE "No such machine" if the *machine* name does not match exactly one existing machine, MR_ACE "Invalid access control entity" if the *ace_type* and *ace_name* together do not specify a valid entity.

delete_server_host_access

Args: machine

Returns: none

Updates the information about who has access to a given machine. The *machine* name must match exactly one existing machine.

add_service

Args: {service, protocol, port, description}

Returns: none

Adds information about a new network service to the database. The service name must not match any existing services. The protocol must be listed as a "protocol" in the aliases database, currently "UDP" and "TCP".

delete_service

Args: service

Returns: none

Deletes information about a network service from the database. The service name must match exactly one existing service.}

get_printcap

Args: printer

Returns: {printer, spool_host, spool_directory, rprinter, comments, modtime, modby, modwith}

Retrieves information about a printer. The *printer* name may contain wildcards. It is safe for this query's ACL to be the list containing everybody.

add_printcap

Args: {printer, spool_host, spool_directory, rprinter, comments}

Returns: none

Adds information about a new printer to the database. The *printer* name must not match any existing printers. *spool_host* must name exactly one existing machine in the database. The printer's modtime will be set. Error: MR_MACHINE if *spool_host* does not match exactly one machine.

delete_printcap

Args: printer

Returns: none

Deletes information about a printer from the database. The *printer* name must match exactly one existing printer.

get_alias

Args: {name, type, translation}

Returns: {name, type, translation}

Looks up an alias in the alias database. This database is used both for user information like alternate names of filesystems, and keyword validation for various queries. Note that type validation entries are of the form (*[type name, usually in lower case]*, **TYPE**, *[type string, always in upper case]*). Some type validation entries are used to further identify another field. These have entries of the form (*[type string in uppercase]*, **TYPEDATA**, *[type, one of:* **none, user, list, string, machine***]*). The *name, type*, and *translation* may contain wildcards. It is safe for this query to be the list containing everybody.

add_alias

Args: {name, type, translation}

Returns: none

Adds a new alias to the alias database. The *type* must be a known type as recorded under **alias** in the alias database. Duplicate translations for a given (name, type) pair are allowed. Note that type validation entries are of the form (*[type name, usually in lower case]*, **TYPE**, *[type string, always in upper case]*). Some type validation entries are used to further identify another field. These have entries of the form (*[type string in uppercase]*, **TYPEDATA**, *[type, one of:* **none, user, list, string, machine***]*). The *name, type*, and *translation* may contain wildcards.

delete_alias

Args: {name, type, translation}

Returns: none

Deletes an alias from the alias database. The combination of all three input arguments must match exactly one alias.

get_value

Args: variable

Returns: value

Look up a value in the values database. This is used for DCM flags and Moira internal ID hints. The *variable* name must match exactly one existing value name in the database. It is safe for this query's ACL to be the list containing everybody. Errors: MR_NO_MATCH "No records in database match query" if the name does not match exactly one variable name.

add_value

Args: {variable, value}

Returns: none

Adds a new value to the values database. The *variable* name must be unique among the variables already in the database. The *value* is an integer.

update_value

Args: {variable, value}

Returns: none

Changes the value of an existing variable in the values database. The *variable* name must match exactly one existing variable. Its *value* will be replaced with the supplied value. Errors: MR_NO_MATCH "No records in database match query" if the name does not match exactly one variable name.

delete_value

Args: variable

Returns: none

Deletes a variable from the values database. The *variable* name must match exactly one existing variable. Errors: MR_NO_MATCH "No records in database match query" if the name does not match exactly one variable name.

get_all_table_stats

Args: none

Returns: {table, retrieves, appends, updates, deletes, modtime}

Retrieves a summary of the table statistics. Each tuple consists of the *table* name, how many *retrieves, appends, updates*, and *deletes* have been performed on that table, and the date of the last change to the table. It is safe for this query's ACL to be the list containing everybody.

7.0.8. Build-in Special Queries

_help

Args: query

Returns: help_message

Returns the short name of the query and a list of arguments and return values. The query name must match an existing query. This query may be executed by anyone.

_list_queries

Args: none

Returns: {long_query_name, short_query_name}

Returns a list of every query name. This query may be executed by anyone.

_list_users

Args: none

Returns: {kerberos_principal, host_address, port_number, connect_time, client_number}

Returns a list of every client currently using the Moira server. This query may be executed by anyone.

7.1. Errors

General errors (may be returned by all queries):

MR_ARG_TOO_LONG - An argument contains too many characters
MR_ARGS - Incorrect number of arguments
MR_DEADLOCK - Database deadlock; try again later
MR_INGRES_ERR - An unexpected error occured in Ingres, the underlying DBMS
MR_INTERNAL - Internal consistency failure
MR_NO_HANDLE - Unknown query specified
MR_NO_MEM - Server ran out of memory
MR_PERM - Insufficient permission to perform requested database access

Any retrieval query may return

MR_NO_MATCH - No records in database match query

Any add or update query may return

MR_BAD_CHAR - Illegal character in argument
MR_EXISTS - new object conflicts with object already in the database
MR_INTEGER - String could not be parsed as an integer
MR_NO_ID - Cannot allocate new ID
MR_NOT_UNIQUE - An attempt to update more than one object at once

Any delete query may return

MR_IN_USE - Object is in use

Query specific errors:

MR_ACE - No such access control entity
MR_BAD_CLASS - Specified class is not known
MR_BAD_GROUP - Invalid group ID
MR_CLUSTER - Unknown cluster
MR_DATE - Invalid date
MR_FILESYS - Named file system does not exist
MR_FILESYS_EXISTS - Named file system already exists
MR_FILESYS_ACCESS - invalid filesys access
MR_FSTYPE - Invalid filesys type
MR_LIST - No such list
MR_MACHINE - Unknown machine
MR_NFS - specified directory not exported
MR_NFSPHYS - Machine/device pair not in nfsphys relation
MR_NO_FILESYS - Cannot find space for filesys
MR_NO_MATCH - Arguments not unique
MR_NO_POBOX - Cannot find space for pobox
MR_NO_QUOTA - No default quota specified
MR_PRINTER - Unknown printer
MR_SERVICE - Unknown service
MR_STRING - Unknown string
MR_TYPE - Invalid type
MR_USER - No such user
MR_WILDCARD - Wildcards not allowed in this case

8. Specialized Management Tools - User Interface

Moira will include a set of specialized management tools to enable system administrators to control system resources. As the system evolves, more management tools will become a part of the Moira's application program library. These tools provide the fundamental administrative use of Moira.

In response to complaints about the user interface of current database maintenance tools such as madm, gadm, and (to a lesser extent) register, the Moira tools will use a slightly different strategy. To accommodate novice and occasional users, a menu interface similar to the interface in register will be the default. For regular users, a command-line switch (such as `-nomenu`) will be provided that will use a line-oriented interface such as those in discuss and kermit. This should provide speed and directness for users familiar with the system, while being reasonably helpful to novices and occasional users. A specialized menu building tool has been developed in order that new application programs can be developed quickly. An X interface is being planned, but is of secondary importance to the functioning of the base system.

Fields in the database will have associated with them lists of legal values. A null list will indicate that any value is possible. This is useful for fields such as user_name, address, and so forth. The application programs will, before attempting to modify anything in the database, will request this information, and compare it with the proposed new value. If an invalid value is discovered, it will be reported to the user, who will be given the opportunity to change the value, or "insist" that it is a new, legal value. (The ability to update data in the database will not necessarily indicate the ability to add new legal values to the database.)

Applications should be aware of the ramifications of their actions, and notify the user if appropriate. For example, an administrator deleting a user should be informed of storage space that is being reclaimed, mailing lists that are being modified. Objects that need to be modified at once (such as the ownership of a mailing list) should present themselves to be dealt with.

The following list of programs will be found on subsequent pages:

- BLANCHE - batch list maintanence tool
- CHFN - change finger information
- CHPOBOX - change forwarding post office
- CHSH - change default shell
- REG_TAPE - Registrar's tape entry program
- MOIRA - administrative client
- MRCHECK - verify that updates have been successful
- USERREG - New user registration.

For clarity, each new program begins on a new page.

PROGRAM NAME: BLANCHE - Batch list operations

DESCRIPTION: This program allows one to examine a list, and to examine or modify the membership of a list. Rather than using menus as the other Moira clients, it takes initial command line arguments, and uses standard input and standard output.

PRE-DEFINED QUERIES USED:

- get_list_info
- count_members_of_list
- get_members_of_list
- add_member_to_list
- delete_member_from_list

Manipulates the following fields:

- (name, active, public, hidden, maillist, group, gid, desc, acl_type, acl_id, modtime, modby, modwith) - LIST relation
- (list_id, member_type, member_id) - MEMBERS relation

SUPPORTED SERVICE(S):

- Mailing lists
- Unix groups
- Moira access control lists

END USER: List maintainers

A SESSION USING BLANCHE:

```
% blanche -info mar
List: mar
Description: User Group
Flags: active, private, and hidden
mar is a maillist and is a group with GID 5271
Owner: LIST mar
Last modified by mar with blanche on 14-sep-1988 14:03:20
% blanche -a carla mar -m
carla
mar
%
```

PROGRAM NAME: CHFN - Finger Information.

DESCRIPTION: This program allows users to change their finger information. This is the information put into the password lines stored in hesiod and the master password file. It should be functionally equivalent to the standard Berkeley Unix chfn program.

PRE-DEFINED QUERIES USED:

- get_finger_by_login
- update_finger_by_login

Manipulates the following fields:

- (fullname, nickname, home_address, home_phone, office_phone, department, year) - FINGER relation

SUPPORTED SERVICE(S):

- User Community - finger

END USER: All.

A SESSION USING CHFN:

```
% chfn

Changing finger information for pjlevine.
Info last changed on 22-mar-1988 15:13:33 by user pjlevine using chfn
Default values are printed inside of of ’[]’.
To accept the default, type <return>.
To have a blank entry, type the word ’none’.

Full name [Peter Levine]:

Nickname [Pete]:
Home address (Ex: Bemis 304) [24 kilsyth rd Brookline]:
Home phone number (Ex: 4660000) [1234567]:
Office address (Exs: 597 Tech Square or 10-256) [E40-342a]:
Office phone (Ex: 3-1300) [0000]:
MIT department (Exs: EECS, Biology, Information Services) []:
MIT year (Exs: 1989, ’91, Faculty, Grad) [staf]:
%
```

PROGRAM NAME: CHPOBOX - View / change home mail host.

DESCRIPTION: The name service and a mail forwarding service need to know where a user's post office is. This program allows the user the capability to forward his mail to a different machine. This program is a command line interface. It will report a user's current mailbox. It can also set a user's mailbox.

PRE-DEFINED QUERIES USED:

- get_pobox
- set_pobox
- set_pobox_pop
- get_server_locations

Manipulates the following fields:

- (potype, pop_id, box_id) - USER relation
- (string_id, string) - STRINGS relation

SUPPORTED SERVICE(S):

- Mail forwarding
- Mail reading

END USERS: All.

A SESSION USING CHPOBOX:

```
% chpobox
User mar, Type POP, Box: mar@E40-PO.MIT.EDU
  Modified by mar on 06-oct-1988 17:43:35 with moira
%
% chpobox -s mar@xx.lcs.mit.edu
User mar, Type SMTP, Box: mar@xx.lcs.mit.edu
  Modified by mar on 18-oct-1988 17:51:32 with chpobox
%
% chpobox -p
User mar, Type POP, Box: mar@E40-PO.MIT.EDU
  Modified by mar on 18-oct-1988 17:52:25 with chpobox
%
```

PROGRAM NAME: CHSH - Change shell.

DESCRIPTION: This program allows users to change their default shell.

PRE-DEFINED QUERIES USED:

- get_user_by_login
- update_user_shell

Manipulates the following fields:

- (shell) - USER relation

SUPPORTED SERVICE(S):

- login

END USERS: All

A SESSION USING CHSH:

```
% chsh

Changing login shell for pjlevine.
Account info last changed on 02-sep-1988 13:56:03 by user pjlevine using moira
Current shell for pjlevine is /bin/csh
New shell: /bin/csh
Changing shell to /bin/csh
%
```

PROGRAM NAME: DBCK - Database consistency checker

DESCRIPTION: This program verifies the internal consistency of the database. It verifies that there are no duplicates of supposedly unique data, that all references to other objects are references which actually exist, there are no unused objects, and that the counts of quotas and poboxes are correct. This is written in the spirit of the unix filesystem checker, *fsck*.

PRE-DEFINED QUERIES USED:
None. It access the database directly.

END USERS: Database administrator

A SESSION USING DBCK:

```
% dbck
Opening database sms...done
Phase 1 - Looking for duplicates
Phase 2 - Checking references
Phase 3 - Finding unused objects
Warning: List saltzer is empty
Warning: List bug-rt is empty
Warning: List athena-bug-scribe is empty
Warning: List ccref is empty
Warning: List rparmelee is empty
Warning: List alens-testers is empty
Unreferenced string mar@xx.lcs.mit.edu id 77
Delete (Y/N/Q)? y
1 entry deleted
Unreferenced string a random string (with * wildcards) id 76
Delete (Y/N/Q)? y
1 entry deleted
Phase 4 - Checking counts
Done.
%
```

PROGRAM NAME: REG_TAPE - Add or remove students from the system using Registrar's tape.

DESCRIPTION: Each term, when the Registrar releases a tape of current students, the system administrator must load the names of new users and delete all old users. This program will automatically use the Registrar's tape as a means of keeping current the Moira database.

The problem of deleting users is a sensitive issue. The removal of a user will reflect this sensitivity. When deleting a user, the expiration date field will be set to the current date, but the user will not be removed. The program db_maint will, among other things, check the expiration stamp of the users. If a stamp is within critical expiration time, the program will notify the administrator that a time-to-live date has been reached. If correct, the administrator will set the user's status field to INACTIVE and set the time to some date in the future. When that date and INACTIVE status are reached, the user is flushed. If incorrect, the administrator will set the date to some time in the future and leave the status field ACTIVE.

PRE-DEFINED QUERIES USED:

- update_user
- update_user_status

Manipulates the following fields:

- (status, expdate) - USERS relation.

SUPPORTED SERVICE(S):

- Moira

END USERS: Administrator.

PROGRAM NAME: Moira - Moira client interface.

DESCRIPTION: This is the primary Moira client. It is capable of examining and modifying all fields in the database. It is menu based, using the curses library. For backwards compatability with older clients, it will look at the name it is invoked with, and start in a sub-menu if it recognizes a menu name.

SUPPORTED SERVICES:

- All

END USER:

- All
- Operations
- System administrators
- User accounts administrator

MENUS:

MENU: Top Level Menu

DESCRIPTION: This is the main menu of the program. It lists sub menus for each of the main types of data that Moira handles.

DISPLAY:

```
                        Moira Database Manipulation
 1. (cluster)      Cluster Menu.
 2. (filesys)      Filesystem Menu.
 3. (list)         Lists and Group Menu.
 4. (machine)      Machine Menu.
 5. (nfs)          NFS Physical Menu.
 6. (user)         User Menu.
 7. (printer)      Printer Menu.
 8. (dcm)          DCM Menu.
 9. (misc)         Miscellaneous Menu.
 t. (toggle)       Toggle logging on and off.
 q. (quit)         Quit.
```

QUERIES USED:

MENU: Cluster Menu

DESCRIPTION: This menu allows the manipulation of clusters. It also has sub-menus to allow the user to examine machine to cluster mappings and cluster data.

DISPLAY:

```
                              Cluster Menu
 1. (show)          Get cluster information.
 2. (add)           Add a new cluster.
 3. (update)        Update cluster information.
 4. (delete)        Delete this cluster.
 5. (mappings)      Machine To Cluster Mappings Menu.
 6. (c_data)        Cluster Data Menu.
 7. (verbose)       Toggle Verbosity of Delete.
 r. (return)        Return to previous menu.
 t. (toggle)        Toggle logging on and off.
 q. (quit)          Quit.
```

QUERIES USED:

show	get_cluster
add	get_cluster, add_cluster
update	get_cluster, update_cluster
delete	get_cluster, delete_cluster

MENU:
Filesystem Menu

DESCRIPTION: This menu allows the manipulation of filesystems. This includes both the filesystem themselves and aliases for filesystems. It also includes a sub-menu for manipulation of quotas.

DISPLAY:

```
                          Filesystem Menu
 1. (get)           Get Filesystem Name Information.
 2. (add)           Add New Filesystem to Database.
 3. (change)        Update Filesystem Information.
 4. (delete)        Delete Filesystem.
 5. (check)         Check An Association.
 6. (alias)         Associate with a Filesystem.
 7. (unalias)       Disassociate from a Filesystem.
 8. (quotas)        Quota Menu.
 9. (verbose)       Toggle Verbosity of Delete.
10. (help)          Help ...
 r. (return)        Return to previous menu.
 t. (toggle)        Toggle logging on and off.
 q. (quit)          Quit.
```

QUERIES USED:

get	get_filesys_by_label
add	get_filesys_by_label, add_filesys
change	get_filesys_by_label, update_filesys
delete	get_filesys_by_label, delete_filesys
check	get_alias
alias	get_alias, add_alias
unalias	get_alias, delete_alias

MENU:
List Menu

DESCRIPTION: This menu allows the manipulation of lists, including retrieval by name, creation, deletion, and updating the characteristics. Note that deleting a list will double-check everything to maintain database consistency, and may prompt the user to take further actions. There is also an option called query_remove which will find all of the lists a member belongs to, and ask the user one at a time which ones the member should be removed from. There are also sub-menus to manipulate the members of a list and to search though lists.

DISPLAY:

```
                                List Menu
 1. (show)          Display information about a list.
 2. (add)           Create new List.
 3. (update)        Update characteristics of a list.
 4. (delete)        Delete a List.
 5. (query_remove) Interactively remove an item from all lists.
 6. (members)       Member Menu - Change/Show Members of a List..
 7. (list_info)     List Info Menu.
 8. (quotas)        Quota Menu.
 9. (verbose)       Toggle Verbosity of Delete.
10. (help)          Print Help.
 r. (return)        Return to previous menu.
 t. (toggle)        Toggle logging on and off.
 q. (quit)          Quit.
```

QUERIES USED:

show	get_list_info
add	get_list_info, add_list
update	get_list_info, update_list
delete	get_list_info, get_ace_use, count_members_of_list, get_lists_of_member, get_members_of_list, delete_list, delete_member_from_list
query_remove	get_lists_of_member, delete_member_from_list

MENU:
Machine Menu

DESCRIPTION: This allows for machines to be manipulated. It includes a sub-menu for machine-to-cluster mapping.

DISPLAY:

```
                              Machine Menu
 1. (show)          Get machine information.
 2. (add)           Add a new machine.
 3. (update)        Update machine information.
 4. (delete)        Delete this machine.
 5. (mappings)      Machine To Cluster Mappings Menu.
 6. (verbose)       Toggle Verbosity of Delete.
 r. (return)        Return to previous menu.
 t. (toggle)        Toggle logging on and off.
 q. (quit)          Quit.
```

QUERIES USED:

show	get_machine
add	get_machine, add_machine
update	get_machine, update_machine
delete	get_machine, delete_machine

MENU:
NFS Physical Menu

DESCRIPTION: This allows for NFS physical filesystems to be manipulated. It includes a submenu for quota manipulation as well.

DISPLAY:

```
                           NFS Physical Menu
 1. (show)          Show an NFS server.
 2. (add)           Add NFS server.
 3. (update)        Update NFS server.
 4. (delete)        Delete NFS server.
 5. (quotas)        Quota Menu.
 6. (verbose)       Toggle Verbosity of Delete.
 r. (return)        Return to previous menu.
 t. (toggle)        Toggle logging on and off.
 q. (quit)          Quit.
```

QUERIES USED:

show	get_nfsphys
add	get_nfsphys, add_nfsphys
update	get_nfsphys, update_nfsphys
delete	get_nfsphys, get_filesys_by_nfsphys, delete_nfsphys

MENU:
User Menu

DESCRIPTION: This allows for user accounts to be manipulated. Lookup may be done by login name, real name, or class. Modifications may be of all fields, or a simple deactivate (changing the account status to indicate ‘‘marked for deletion’’. Registering a user consists of changing the login name and status, and creating a user group, filesystem, and pobox. To expunge a user is to actually delete all record of them from the database (including prompting to delete their filesystem and user group). Sub-menus for manipulation of poboxes and quotas are also provided.

DISPLAY:

```
                                User Menu
 1. (login)         Show user information by login name.
 2. (name)          Show user information by name.
 3. (class)         Show names of users in a given class.
 4. (modify)        Change all user fields.
 5. (adduser)       Add a new user to the database.
 6. (register)      Register a user.
 7. (deactivate)    Deactivate user.
 8. (expunge)       Expunge user.
 9. (pobox)         Post Office Box Menu.
10. (quota)         Quota Menu.
11. (verbose)       Toggle Verbosity of Delete.
 r. (return)        Return to previous menu.
 t. (toggle)        Toggle logging on and off.
 q. (quit)          Quit.
```

QUERIES USED:

login	get_user_by_login
name	get_user_by_name
class	get_user_by_class
modify	get_user_by_login, update_user
adduser	get_user_by_name, add_user
register	get_user_by_name, register_user
deactivate	get_user_by_name, update_user_status
expunge	delete_user, get_filesys_by_label, delete_filesys, get_members-of-list, delete_list, count_members_of_list, get_lists_of_member, delete_member_from_list, get_list_info, get_ace_use

MENU:
Printer Menu

DESCRIPTION: This allows printcap entries to be manipulated.

DISPLAY:

```
                              Printer Menu
1. (get)          Get Printcap Entry Information.
2. (add)          Add New Printcap Entry to Database.
3. (change)       Update Printer Information.
4. (delete)       Delete Printcap Entry.
r. (return)       Return to previous menu.
t. (toggle)       Toggle logging on and off.
q. (quit)         Quit.
```

QUERIES USED:

get	get_printcap
add	get_printcap, add_printcap
change	get_printcap, delete_printcap, add_printcap
delete	get_printcap, delete_printcap

MENU:
DCM Menu

DESCRIPTION: This menu allows for the DCM and it's control information to be manipulated. Options include enabling the DCM, getting the current status of updates, and starting an update immediately. Submenus exist for manipulating service information and host/service tuple information.

DISPLAY:

```
                                DCM Menu
1. (enable)       Enable/disable DCM.
2. (service)      DCM Service Menu.
3. (host)         DCM Host Menu.
4. (active)       Display entries currently being updated.
5. (failed)       Display entries with errors to be reset.
6. (dcm)          Invoke a DCM update now.
r. (return)       Return to previous menu.
t. (toggle)       Toggle logging on and off.
q. (quit)         Quit.
```

QUERIES USED:

enable	get_value, update_value
active	qualified_get_server, qualified_get_server_host
failed	quelified_get_server, qualified_get_server_host
dcm	trigger_dcm

MENU:
Miscellaneous Menu

DESCRIPTION: This menu contains miscellaneous functions which are not necessary, but may be useful in maintaining the Moira system. These include fetching the table use statistics, listing currently active connections to the Moira server, and fetching values and aliases from the database.

DISPLAY:

```
                        Miscellaneous Menu
 1. (statistics)   Show database statistics.
 2. (clients)      Show active Moira clients.
 3. (getval)       Show a database variable value.
 4. (getalias)     Show an alias relation.
 r. (return)       Return to previous menu.
 t. (toggle)       Toggle logging on and off.
 q. (quit)         Quit.
```

QUERIES USED:

statistics	get_all_table_stats
clients	_list_users
getval	get_value
getalias	get_alias

MENU:
Cluster Data Menu

DESCRIPTION: This menu allows the manipulation of data associated with clusters.

DISPLAY:

```
                        Cluster Data Menu
 1. (show)         Show Data on a given Cluster.
 2. (add)          Add Data to a given Cluster.
 3. (delete)       Remove Data to a given Cluster.
 4. (verbose)      Toggle Verbosity of Delete.
 r. (return)       Return to previous menu.
 t. (toggle)       Toggle logging on and off.
 q. (quit)         Quit.
```

QUERIES USED:

show	get_cluster_data
add	add_cluster_data
delete	get_cluster_data, delete_cluster_data

MENU:
Mappings Menu

DESCRIPTION: This cluster allows the machine to cluster mappings to be manipulated.

DISPLAY:

```
                    Machine To Cluster Mappings Menu
1. (map)            Show Machine to cluster mapping.
2. (addcluster)     Add machines to a clusters.
3. (remcluster)     Remove machines from clusters.
4. (verbose)        Toggle Verbosity of Delete.
r. (return)         Return to previous menu.
t. (toggle)         Toggle logging on and off.
q. (quit)           Quit.
```

QUERIES USED:

map	get_machine_to_cluster_map
addcluster	get_machine, get_cluster, add_machine_to_cluster
remcluster	get_machine_to_cluster_map, delete_machine_from_cluster

MENU:
Quota Menu

DESCRIPTION: This menu allows users' quotas and the default quota to be manipulated. The default quota is the quota that new users are assigned when they register for an account.

DISPLAY:

```
                               Quota Menu
1. (shdef)          Show default user quota (in KB).
2. (chdef)          Change default user quota.
3. (shquota)        Show a user's disk quota on a filesytem.
4. (addquota)       Add a new disk quota for user on a filesytem.
5. (chquota)        Change a user's disk quota on a filesytem.
6. (rmquota)        Remove a user's disk quota on a filesytem.
7. (verbose)        Toggle Verbosity of Delete.
r. (return)         Return to previous menu.
t. (toggle)         Toggle logging on and off.
q. (quit)           Quit.
```

QUERIES USED:

shdef	get_value
chdef	get_value, update_value
shquota	get_nfs_quota
addquota	add_nfs_quota
chquota	get_nfs_quota, update_nfs_quota
rmquota	get_nfs_quota, delete_nfs_quota

MENU:
Members Menu

DESCRIPTION: This allows the membership of lists to be manipulated. On entry to this menu, Moira will prompt for the name of the list to be manipulated. Membership may be fetched by a specific type.

DISPLAY:

```
                     Change/Display membership of 'dbadmin'
 1. (add)           Add a member to this list.
 2. (remove)        Remove a member from this list.
 3. (all)           Show the members of this list.
 4. (user)          Show the members of type USER.
 5. (list)          Show the members of type LIST.
 6. (string)        Show the members of type STRING.
 7. (verbose)       Toggle Verbosity of Delete.
 r. (return)        Return to previous menu.
 t. (toggle)        Toggle logging on and off.
 q. (quit)          Quit.
```

QUERIES USED:

add	add_member_to_list
remove	delete_member_from_list
all	get_members_of_list
user	get_members_of_list
list	get_members_of_list
string	get_members_of_list

MENU:
List Information Menu

DESCRIPTION: This menu allows one to get various summaries of lists. They can be retrieved by membership, administration, groups, or maillists.

DISPLAY:

```
                              List Information Menu
 1. (member)        Show all lists to which a given member belongs.
 2. (admin)         Show all items which a given member can administer.
 3. (groups)        Show all lists which are groups.
 4. (public)        Show all public mailing lists.
 5. (maillists)     Show all mailing lists.
 r. (return)        Return to previous menu.
 t. (toggle)        Toggle logging on and off.
 q. (quit)          Quit.
```

QUERIES USED:

member	get_lists_of_member
admin	get_ace_use
groups	qualified_get_lists
public	qualified_get_lists
maillists	qualified_get_lists

MENU:
Post Office Box Menu

DESCRIPTION: This menu allows users' poboxes to be manipulated. The ''set'' option allows the user to return a pobox back from a foreign maildrop to a previously used pobox server, in addition to allowing changes in pobox servers or foriegn addresses.

DISPLAY:

```
                      Post Office Box Menu
1. (show)          Show a user's post office box.
2. (set)           Set (Add or Change) a user's post office box.
3. (remove)        Remove a user's post office box.
4. (verbose)       Toggle Verbosity of Delete.
r. (return)        Return to previous menu.
t. (toggle)        Toggle logging on and off.
q. (quit)          Quit.
```

QUERIES USED:

show	get_pobox
set	get_pobox, set_pobox_pop, get_server_locations, set_pobox
remove	delete_pobox

MENU:
DCM Service Menu

DESCRIPTION: This menu allows DCM services to be manipulated. A service may have an error indicator reset, or its entire state reset, in addition to the usual updates. Note that reseting a service state is a dangerous action that should be used carefully.

DISPLAY:

```
                        DCM Service Menu
1. (showserv)      Show service information.
2. (addserv)       Add a new service.
3. (updateserv)    Update service information.
4. (resetsrverr)   Reset service error.
5. (resetsrvc)     Reset service state.
6. (delserv)       Delete service info.
r. (return)        Return to previous menu.
t. (toggle)        Toggle logging on and off.
q. (quit)          Quit.
```

QUERIES USED:

showserv	get_server_info
addserv	add_server_info
updateserv	get_server_info, update_server_info
resetsrverr	reset_server_error
resetsrvc	set_server_internal_flags
delserv	delete_server_info

MENU:
DCM Host Menu

DESCRIPTION: This menu allows DCM service/host tuples to be manipulated. A tuple may have an error indicator reset, its entire state reset, or an override set in addition to the usual updates. Note that reseting a tuple's state is a dangerous action that should be used carefully.

DISPLAY:

```
                         DCM Host Menu
1. (showhost)     Show service/host tuple information.
2. (addhost)      Add a new service/host tuple.
3. (updatehost)   Update a service/host tuple.
4. (resethosterr) Reset service/host error.
5. (resethost)    Reset service/host state.
6. (override)     Set service/host override.
7. (delhost)      Delete service/host tuple.
r. (return)       Return to previous menu.
t. (toggle)       Toggle logging on and off.
q. (quit)         Quit.
```

QUERIES USED:

showhost	get_server_host_info
addhost	add_server_host_info
updatehost	get_server_host_info, update_server_host_info
resethosterr	reset_server_host_error
resethost	set_server_host_internal
override	set_server_host_override
delhost	delete_server_host_info

MENU:
Zephyr ACL Menu

DESCRIPTION: This menu allows zephyr class access control lists to be manipulated. The list menu can be accessed as a sub-menu.

DISPLAY:

```
                Zephyr Class ACL Menu
1. (show)     Show zephyr class information.
2. (add)      Add a new zephyr class ACL.
3. (update)   Change a zephyr class ACL.
4. (delete)   Delete ACL information for a zephyr class.
5. (list)     List Menu.
r. (return)   Return to previous menu.
t. (toggle)   Toggle logging on and off.
q. (quit)     Quit.
```

QUERIES USED:

show	get_zephyr_class
add	get_zephyr_class, add_zephyr_class
update	get_zephyr_class, update_zephyr_class
delete	get_zehpyr_class, delete_zephyr_class

MENU:
Network Services Menu

DESCRIPTION: This menu allows the network service ports and their aliases to be manipulated.

DISPLAY:

```
                Network Services Menu
1. (show)      Show service information.
2. (add)       Add information about a new service.
3. (update)    Change information about a service.
4. (delete)    Delete information about a service.
5. (alias)     Add an alias for a service.
6. (unalias)   Remove an alias for a service.
r. (return)    Return to previous menu.
t. (toggle)    Toggle logging on and off.
q. (quit)      Quit.
```

QUERIES USED:

show	get_service
add	
	get_service, add_service
update	get_service, delete_service, add_service
delete	get_service, delete_service
alias	get_alias, get_service
unalias	get_alias, delete_alias

PROGRAM NAME: MRCHECK - Check to see if updates have been successful

DESCRIPTION: This program lists any DCM updates which have failed.

PRE-DEFINED QUERIES USED:

- qualified_get_server
- qualified_get_server_host
- get_server_info
- get_server_host_info

SERVICE(S) EXAMINED:

- all

END USERS: Moira system administrator

A SESSION USING MRCHECK:

```
% mrcheck
Service AFS, error 43: Unable to build archive of config files
        last success Jan  4 12:15:13 1989, last try Jan  4 16:30:10 1989
Host HESIOD:KIWI.MIT.EDU, error 44: Unable to open DCM file
        last success Jan 11 14:15:18 1989, last try Jan 24 17:40:16 1989
Host AFS:FOO.MIT.EDU, error 31: Kerberos error: Can't decode authenticator
        last success Dec 12 14:51:06 1988, last try Dec 12 19:00:09 1988
Host NFS:CHIROPTERA.MIT.EDU, error 8: Kerberos principal unknown
        last success Nov 15 23:44:37 1988, last try Nov 18 03:01:52 1988
4 things have failed at this time
%
```

References

[1] Noah Mendelsohn.
A Guide to Using GDB
Version 0.1 (DRAFT) edition, MIT Project Athena, 1987.

[2] S. P. Miller, B. C. Neuman, J. I. Schiller and J. H. Saltzer.
Section E.2.1: Kerberos Authentication and Authorization System
M.I.T. Project Athena, December 21, 1987.