

# Section E.1

# Service Management System

Peter Levine
Michael R. Gretzinger
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 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 SMS 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, SMS provides update and maintenance of system servers.

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

- Conceptually, SMS provides mechanisms for managing Athena servers and services. This aspect comprises the fundamental design of SMS.
- Economically, SMS provides a replacement for the labor-intensive, hand-management now associated with maintaining services.

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

SMS 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 SMS use:

- The simplest example is to run a database administration program on the host running SMS. The program will use the SMS application library as its interface to the database. At a later time, a different application is executed to retrieve the SMS information and distribute it to the specific servers.

- Another example is to run an application (i.e., a mailing list administration program) on a workstation. In this case, the program transmit requests to the SMS application library using the SMS network protocol. A server running on the SMS host interprets the application's request. The database is queried or updated (depending on the client's request), and database information is sent back to the application.

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

## 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 SMS 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 SMS; the purpose of SMS 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, SMS uses existing products.

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

- Maximize local processing in applications. SMS is a centralized information

manager. It is not a computing service for local processors. For efficiency, the SMS 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. SMS 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 SMS. (Without it, however, the economic consequence of managing system services by hand must be recognized.) SMS should be reasonably easy to install at other sites.

- SMS 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").

- SMS 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.

- SMS does not have to be 100% available. SMS 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, SMS is composed of six components.

- The database.

- The SMS server.

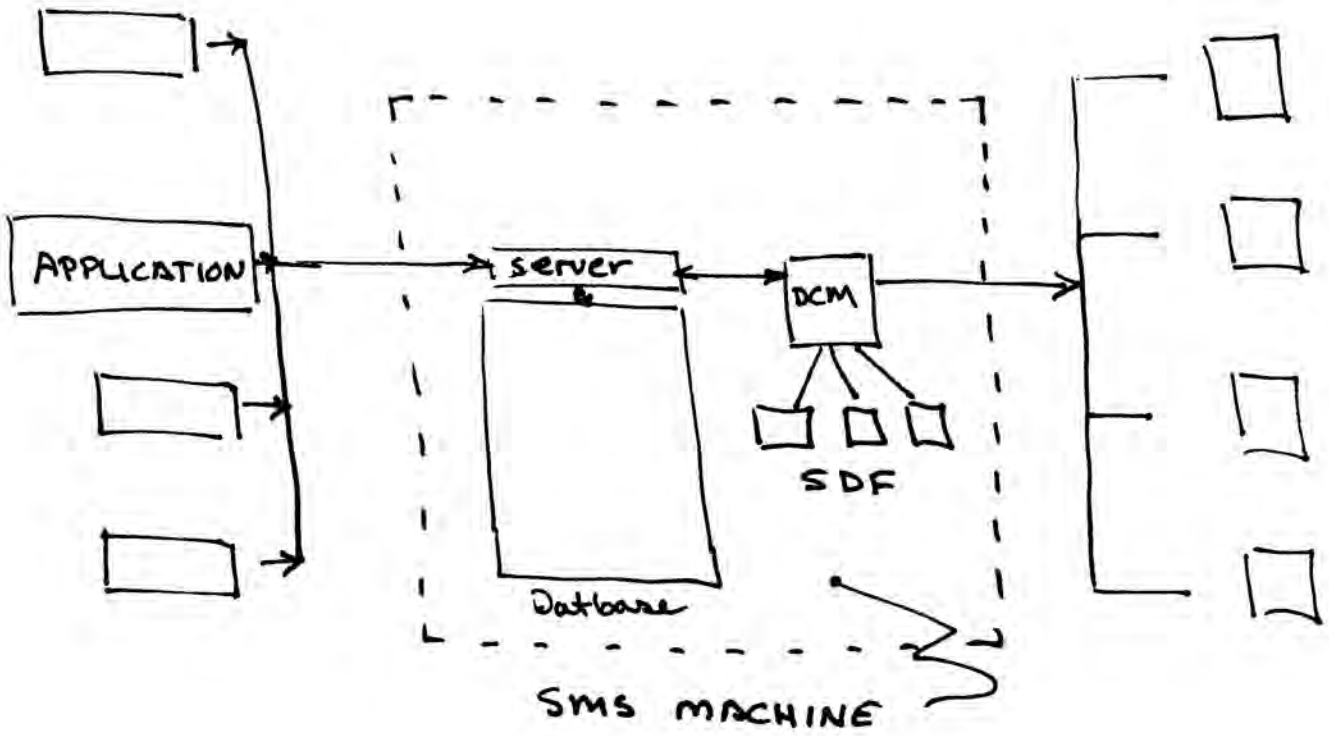
- The application library.

- The SMS protocol.

- The Data Control Manager, DCM.

- The server-specific files.

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




THE SMS SYSTEM STRUCTURE

In all cases, a client of SMS uses the application library. The library communicates with the SMS 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 sms 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. SMS supports a number of system services. To date there are six system services which are supported. They are:

- Hesiod
- RVD
- NFS
- Mail Service
- MDQS - not fully supported

- Zephyr - to date, not supported

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 sms 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 three hesiod servers, 10 library servers running RVD, 20 locker servers running NFS, five class libraries running NFS, one /usr/lib/aliases propagation. A hesiod server requires 9 separate files; Each hesiod server will receive the same 9 files. Each RVD server requires one file, each file being different. Each NFS server requires two files, one identical file to all NFS servers, one different file to each filesystem on each server. Usr/lib/aliases is one file.

G.

File Organization

| Service | File | Size | N | Propagati | Interval |
|---|---|---|---|---|---|
| Hesiod | cluster.db | 22300 | 1 | 3 | 6 hours |
| | service.db | 10100 | 1 | 3 | 6 hours |
| | printers.db | 3833 | 1 | 3 | 6 hours |
| | lpr.db | 3250 | 1 | 3 | 6 hours |
| | printcap.db | 9800 | 1 | 3 | 6 hours |
| | pobox.db | 325000 | 1 | 3 | 6 hours |
| | sloc.db | 300 | 1 | 3 | 6 hours |
| | filesys.db | 36000 | 1 | 3 | 6 hours |
| | passwd.db | 880000 | 1 | 3 | 6 hours |
| RVD | /site/rvddb | 2000(90%) - 20000 | 10 | 10 | 15 minutes |
| | /site/rvd/acl/OP.acl | small | 10 | 10 | 15 min |
| | /site/rvd/acl/AD.acl | small | 10 | 10 | 15 min |
| | /site/rvd/acl/SH.acl | small | 10 | 10 | 15 min |
| | /site/rvd/acl/file.acl | small | 10 | 10 | 15 min |
| NFS | /site/nfsid | 880000 | 1 | 25 | 6 hours |
| | /mit/quota | 35000 | 25 | 25 | 6 hours |

| | | | | | |
|---|---|---|---|---|---|
| Mail | /usr/lib/aliases | 445000 | 1 | 1 | 24 hours |
| TOTAL | | | 86 | 110 | |

H. NOTE: The above files will only be generated and propagated if the data has changed during the time interval. For example, although the RVD interval is 15 minutes, there is no effect on system resources unless the information relevant to RVD's changed during the previous 15 minute interval.

I. Application interfaces provide all the mechanisms to change database fields. There will be no need for any sms 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 SMS. 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. SMS does not depend on any special feature of INGRES. In fact, SMS 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 SMS database.

The database contains the following:

- User information
- Machine information
- General service information (service location, /etc/services)
- File system information (RVD, NFS)
- Printer information (Berkeley, MDQS)
- Post office information
- Lists (mail, acl, groups)
- Aliases

The database field are described in the section *Database Fields*, described later in this document.

The database is a completely independent entity from the sms system. The Ingres query

bindings and database specific routines are layered at the lower levels of the sms server. *All applications are independent of the database specific routines.* This independence is acheived through the use of query handles, sms specifice functions providing data access and updating. An application passes query handles to the sms 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, sms 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 sms-supported services. No attempt is made to circumvent sms 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 SMS. 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 undeliverable.

Input checking is done by both the SMS server and by applications using SMS. 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 SMS 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 9) defines those fields which require explicit data checking.

### *5.2.2. Backup*

It is not absolutely critical that the SMS 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 SMS has been set up to maximize recoverability in the event of damage to the database.

Two programs (`smsbackup` and `smsrestore`) are generated automatically (using an `awk` script) from the database description file `sms/db/dbbuild`[1]. `smsbackup` copies each relation of the current SMS 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)

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

database; the ascii files take up about 3.2 MB of space. It is intended that smsbackup 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[2]. 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.

`smsrestore` does the inverse of `smsbackup`. It requires the existance of a empty database named "smstemp"[3], created as follows:

```
# createdb smstemp
# quel smstemp
* \i /ul/sms/db/dbbuild                    load DB definition
* \g                                       execute DB definition
* \q                                       quit
# smsrestore
Do you *REALLY* want to wipe the SMS 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 log file kept by the SMS server daemon contains a listing of all transactions with the server[4].

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 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 SMS.

### 5.3. The SMS Protocol

The SMS protocol is a remote procedure call protocol layered on top of TCP/IP. Clients of SMS 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

---

[2]This is not running automatically yet

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

[4]A small change to the server would cause only transactions which side-effected the database to be logged in a separate journal, in a format readable by a recovery program; this would reduce the number lost transactions to virtually none in the event of a disk head crash

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 SMS. 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 SMS_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. |
| Shut down server | Requests that the server cleanly shut down. This gets one argument, a string, which is entered into the server log before the server shuts down as an explanation for the shutdown. |

### 5.4. The SMS Server

All remote[5] communication with the SMS database is done through the SMS server, using a remote procedure call protocol built on top of GDB [1]. The SMS server runs as a single UNIX processes on the SMS 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 connection it accepts.

One of the major concerns for efficiency in SMS is the time it takes to start up an application's connection to the server. One of the limiting factors for Athenareg, SMS'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 SMS server will do this only once, at the start up time of the daemon.

---

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

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 SMS 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 SMS application library provides access to SMS through a simple set of remote procedure calls to the SMS server. The library is layered on top of Noah Mendelsohn'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. SMS application library calls

The SMS library contains the following routines:

```
int sms_connect();
```

Connects to the SMS 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 SMS_ALREADY_CONNECTED if a connection already exists.

```
int sms_auth();
```

Attempts to authenticate the user to the system. It can return Kerberos failures, either local or remote (for example, "can't find ticket" or "ticket expired"), SMS_NOT_CONNECTED if sms_connect was not called or was not successful, or SMS_ABORTED if the attempt to send or recieve data failed (and the connection is now closed).

```
int sms_disconnect();
```

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

```
int sms_noop();
```

This attempts to do a handshake with SMS (for testing and performance measurement). It can return SMS_NOT_CONNECTED or SMS_ABORTED if not successful.

```
int sms_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 SMS 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 sms_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 SMS 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`.

Comment[ $Source: /u2/sms/doc/RCS/dcm.mss,v $ $Author: pjlevine $ $Header: dcm.mss,v 1.14 87/08/05 18:02:04 pjlevine Locked $

$Log: dcm.mss,v $ Revision 1.14 87/08/05 18:02:04 pjlevine changed dcm definition.

Revision 1.13 87/08/05 17:19:52 pjlevine *** empty log message ***

Revision 1.12 87/06/19 16:20:55 spook Removed update stuff (moving to update.mss).

Revision 1.11 87/06/19 11:27:34 pjlevine pjlevine adds words of wisdom

Revision 1.10 87/06/02 15:42:45 ambar spelling fixes.

Revision 1.9 87/06/01 16:27:39 ambar consistency checks.

Revision 1.8 87/06/01 10:51:22 spook Merged update.mss into dcm.mss where it belongs.

Revision 1.7 87/05/29 18:26:53 ambar fixed scribe error.

Revision 1.6 87/05/29 17:57:42 ambar added to the section on security of datagrams.

Revision 1.5 87/05/29 17:47:48 ambar replaced "dcm" with "DCM" for consistency.

Revision 1.4 87/05/29 14:29:29 ambar more changes from Peter.

Revision 1.3 87/05/29 03:27:37 ambar fixed scribe error

Revision 1.2 87/05/29 03:14:20 ambar Added in Peter's changes.

Revision 1.1 87/05/20 14:42:38 wesommer Initial revision

]

SCRIBE ERROR (NOT INCLUDED IN TEXT]

## 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 by cron at times which are relevant to the data update needs of each server. The update frequency is stored in the SMS database. A server/host relationship is unique to each update time. Through the SMS query mechanism, the DCM extracts SMS data and converts it to server dependent form. The conversion of database specific information to site specific information is done through a server description file, a SMS-unique language which is used to describe the personality of a target service.

When invoked the DCM will perform some preliminary operations to establish the relationship between the SMS data and each server. The very first time the DCM is called, a table is constructed (as a disk file) describing the relationship between servers and update frequency. The table will be the primary mechanism used by the DCM for recognizing the servers which need updating at given times. As a note here, crontab will invoke the DCM at a pre-established time interval, say every 15 minutes. Obviously, the maximum update time will be limited to the time interval the DCM is being invoked at. Every interval, the DCM will search the constructed table and determine whether or not a server's update time is within the current time range. The table has the following components:

| Last time tried update | Success | Time interval | Server | Hostname | Target Pathname | Override | Enable |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

A description of each field follows:

- *Last Time of Update* - This field holds the time when a last successful update occurred. This time will be used against the current time to determine if the interval criteria has been met.
- *Success* - Flag for indicating whether or not the last time tried was successful. 0-fail, 1-success
- *Time interval* - Derived from the SMS database. Gives the interval update time for each server's information needs.
- *Server* - This is the server name. Derived from SMS database.
- *Hostname* - This is the host name where the server resides. Derived from the SMS database.
- *Target Pathname* - Gives the location of the file which needs to be updated on the target or server end. Derived from the SMS database.
- *Override* - Provides an automatic facility for the authorized user to invoke a used-once mechanism for overriding the established time interval. The facility will be very useful when a server has received bogus data and needs updating immediately. After the DCM uses this value, the field is reset to -1. Value: -1 - Use established time interval. 0 and greater - New once-used interval.
- *Enable* - This switch allows the authorized user to turn the update facility on or off. Value: 0 - Off, Non-zero - On.

Each time the DCM is invoked, a search through this table will indicate which servers need updating. Once located, the DCM will use the server/hostname combination to identify the server description files to process. Of course, if the enable switch is off, the update will not occur.

#### *5.7.1. DCM Operation*

The data control manager acts as an interpreter on the SDF's, or as an initiator of executable programs. The basic mechanism is for the DCM to read the above entry table, determine which servers need updating and then locate the appropriate SDF for interpretation. The breakdown of the SDF is a procedure based primarily on the associated query handle and it's associated input and output structure. The output of the DCM is a file stored on the SMS host which is exactly the same format of the server-based file. The update mechanism takes this localized data and ships it over the net.

The DCM is capable of accepting either SDF or executable format. Currently all

programs the DCM calls are written in C.

The DCM, therefore, is an application program designed to orchestrate the distribution and generation of server-specific files.

### *5.7.2. Server Description Files*

The server description files, or SDF, are files which contains a unique description of each server SMS updates. The SDF description has been developed to make support of SMS and the addition of new services a reasonably easy process. The demand of SMS support is clearly rising. Without an easy method of adding new servers, SMS is circumvented in search for easier methods of service support. To date, there are over 15 different files which need generation and propagation. With the support of zephyr this number will likely double (propagation of at least 20 more different acls).

Hand created, these files hold information, in English-like syntax, which the DCM uses for manipulating generic data into server specific form. Accommodating additional servers of primary server type requires, simply, adding a new SDF to the system. The purpose of the server description files is to provide a parseable, readable text files for determining the structure and personality of a given server. The three reasons for SDF:

- To provide a local, uniform and expandable method of providing server information to the Data Control Manager.
- To maintain simplicity and readability.
- To present a regular way of describing many models of servers.

Each SDF will be accessed by a hostname/server combination. This combination will, in fact, be the search path on the source machine. When the DCM needs a SDF to process, it will find a unique server description file in the directory: /hostname/server

Part of the DCM is an interpreter which will parse the SDF and run a generalized syntax and logic check. Assuming the file is syntactically correct, the DCM will use the format of the SDF to generate a server specific file.

The SDF is comprised of a generalized syntax which allows the user to create the information needs of any system server. The limitations are that the data must be in character format.

The Server Description Language consists of key words and commands which the DCM interprets. The format of the files is line oriented and parseable. An example of an SDF is:

```
#one sharp sign is for this file only
##two sharp signs puts data in the target file too

#variables to be used

var name, attribute

begin header

        <"this is verbatim information">
        <"this is the header">

end header
        ##query 1, all print clusters
        begin query
```

```
                #get all print clusters

                handle = 1 #this is the associated query handle
                input : NULL
                         #no input
                output : name
                        #name is a declared variable having the
                        #structure of the query's associated
                        #output structure.
                <"cluster = ", name.prcluster>
                        #verbatim info with passed in
                        #output variable
        end query
                #the above query will continue to query until
                #all fields are found.
        ##
        ##query 2, printers by cluster
        begin query #another query
                handle = 1  #get all print clusters
                input : NULL
                output : name
                begin query
                        handle = 2 #get printers by cluster
                        input : prcluster = name.prcluster
                        #input the output of the previous query
                        output : attribute

                <"print cluster = ", name.prcluster
                 "printer name = ", attribute.printer
                 "printer type = ", attribute.type>
                end query
        end query
```

The target file would look like the following:

```
#two sharp signs puts data in the target file too
#query 1, all print clusters
cluster = e40
cluster = sloan
cluster = admin

#query 2, printers by cluster
print cluster = e40
printer name = ln03-bldge40-3
printer type = laser
printer name = ln03-bldge40-2
printer type = laser
printer name = ln03-bldge40-4
printer type = laser
print cluster = sloan
printer name = ln03-sloan-1
printer type = laser
print cluster admin
printer name = juki-admin-1
printer type = daisy
```

From a generic database the DCM will take the information, process it into localized form,

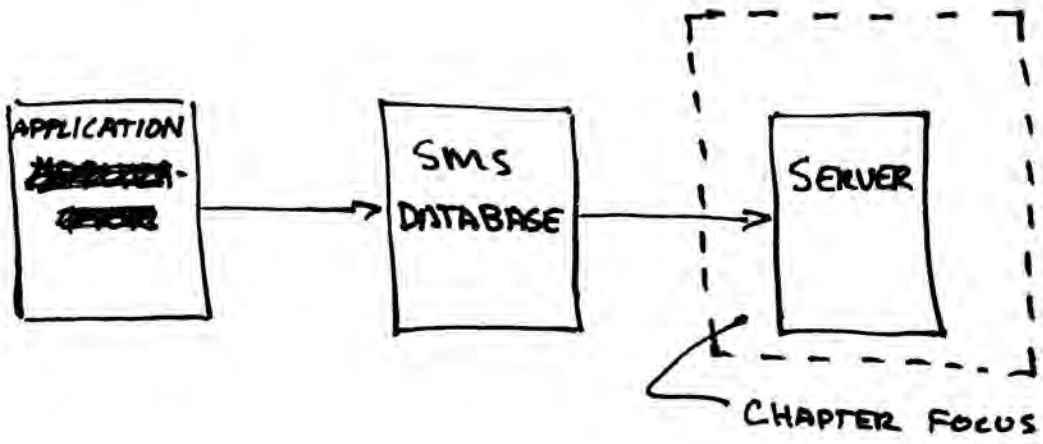
and cache it locally. From here, the server update mechanism grabs the file and serves it over the net.

### 5.8. Server Arrangement

Currently, sms 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, ther is at least one application interface which provides the capability to manipulate the sms database. Since the sms database acts as a single point of contact, the changes made to the database are reflected in the contents of recipient servers.

The following diagram:




This section focusses on the above component contained within the dotted-lined box. This section of the system is the server side, receiving propagated data. Reference, however, is made to each application interface which ultimately effects the server contents.

The services which sms now supports are:

- Hesiod - The athena nameserver.
- RVD - Remote virtual disk.
- NFS - Network file system.
- /usr/lib/aliases - Mail Service.
- MDQS - Multiple device queueing system. (currently not avaliable)
- Zephyr - The athena notification service. (currently not available)

### *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.
- ~~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. SMS's responsibility to hesiod is to provide authoritative data. Hesiod uses a BIND data format in all of it's data files. SMS will provide BIND format to hesiod. There are several files which hesiod uses. ~~To date, they are known to~~ include the following:
  - cluster.db
  - service.db
  - printers.db
  - lpr.db
  - pobox.db
  - sloc.db
  - rvdtab.db
  - passwd.db
  - printcap.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. SMS will propagate hesiod files to the target disk and then 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, 0:00, 6:00, 12:00, 18:00
- Data format : BIND
- Target locations :

  JASON.MIT.EDU: /etc/athena/nameserver
  ZEUS.MIT.EDU: /etc/athena/nameserver
  MENELAUS.MIT.EDU: /etc/athena/nameserver

- Files:

  HESIOD.DB - Hesiod data

  Description:
  Contains hesiod specific data.

  Queries used:
  NOT CREATED FROM SMS QUERY

  How modified:
  EDITOR by system administrator.

  Client(s):
  Hesiod

  Example contents:

```
; Hesiod-specific cache data (for ATHENA.MIT.EDU)
;
;       $Source: /u2/sms/doc/RCS/server_arrang.mss,v $
;       $Header: server_arrang.mss,v 1.8 87/08/05 23:14:05 mike Locked $
; pointers to Hesiod name servers
NS.ATHENA.MIT.EDU. 99999999 HS  NS      JASON.MIT.EDU.
NS.ATHENA.MIT.EDU. 99999999 HS  NS      ZEUS.MIT.EDU.
NS.ATHENA.MIT.EDU. 99999999 HS  NS      MENELAUS.MIT.EDU.
; Hesiod address records (simply duplicates of IN address records)
JASON.MIT.EDU.  99999999    HS    A       18.71.0.7
ZEUS.MIT.EDU.   99999999    HS    A       18.58.0.2
MENELAUS.MIT.EDU. 99999999  HS    A       18.72.0.7
; Internet address records for the same Hesiod servers
; required because of implementations of gethostbyname() which use
; C_ANY/T_A queries.
JASON.MIT.EDU.  99999999    IN    A       18.71.0.7
ZEUS.MIT.EDU.   99999999    IN    A       18.58.0.2
MENELAUS.MIT.EDU. 99999999  IN    A       18.72.0.7
;
```

CLUSTER.DB - Cluster data

Description:
Cluster.db holds the relationships between machines, clusters, and services to service clusters.

Queries used:
get_all_service_clusers() ->
(cluster, service_label, service_cluster)
get_machine_to_cluster_map(*,*) -> (machine, cluster)

How modified:
cluster_maint

Client(s):

Example contents:

```
; Cluster info for timesharing machines and workstations
; format is:
; lines for per-cluster info (both vs and rt) (type UNSPECA)
; followed by line for each machine (CNAME referring to one
; of the lines above)
;
; E40 cluster
bldge40-vstesters.cluster HS UNSPECA "zephyr neskaya.mit.edu"
bldge40-rttesters.cluster HS UNSPECA "zephyr neskaya.mit.edu"
bldge40-vstesters.cluster HS UNSPECA "lpr e40"
bldge40-rttesters.cluster HS UNSPECA "lpr e40"
;
```

SERVICE.DB - services

Description:
Holds the relationship between a canonical service name and its physical protocol, port, and tranlation.

Queries used:
get_all_services -> (service, protocol, port, description)
get_alias(*, service) -> (name, type, translation)

How modified:
service_maint

Client(s):

Example contents:

```
;
; Network services, Internet style
;
echo.service    HS     UNSPECA "echo tcp 7"
echo.service    HS     UNSPECA "echo udp 7"
discard.service HS     UNSPECA "discard tcp 9 sink null"
sink.service    HS     CNAME   discard.service
null.service    HS     CNAME   discard.service
discard.service HS     UNSPECA "discard udp 9 sink null"
systat.service  HS     UNSPECA "systat tcp 11 users"
users.service   HS     CNAME   systat.service
daytime.service HS     UNSPECA "daytime tcp 13"
```

PASSWD.DB - username and group information

Description:
This file is used as a template for toehold. Its contents are username, uid, gid (all users get gid = 101), fullname, home filesys (now limited to /mit/<username>), and shell.

Queries used:
get_all_passwds() -> returns all active logins.

How modified:
ueser_maint
userreg -> initial info
attach_maint

Client(s):
Toehold

Example contents:
```
pjlevine.passwd  HS     UNSPECA "pjlevine:*:1:101:
                    Peter J. Levine,,,,:/mit/pjlevine:/bin/csh"
```

PRINTERS.DB - MDQS printer info

Description:
Maps printer clusters to physical locations.

Queries used:
get_all_printer_clusters() -> (cluster)
get_printers_of_cluster(cluster) ->
(pname, qname, serverhost, ability, hwtype)

How modified:
printer_maint

Client(s):
MDQS

Example contents:
```
MDQS Hesiod printer info
;
; prclusterlist returns all print clusters
;
*.prclusterlist HS UNSPECA bldge40
*.prclusterlist HS UNSPECA bldg1
*.prclusterlist HS UNSPECA bldgw20
```

LPR.DB - lpr printer info

Description:
Line printer information.

Queries used:
NOT GENERATED BY SMS

How modified:
HAND

Example contents:

PRINTCAP.DB - line printer information

Description:
Line printer info, derived from /etc/printcap

Queries used:
NOT GENERATED BY SMS

How modified:
HAND

Example contents:

[INTENTIONALLY LEFT BLANK]

POBOX.DB - post office info

Description:
Contains a username to post office mapping.

Queries used:
get_poboxes_pop(*)
get_poboxes_local(*)
get_poboxes_foreign(*) -> (login, type, machine, box)

How modified:
userreg
usermaint
chpobox

Client(s):

Example contents:

```
abbate.pobox    HS    UNSPECA "POP E40-PO.MIT.EDU abbate"
ackerman.pobox  HS    UNSPECA "POP E40-PO.MIT.EDU ackerman"
ajericks.pobox  HS    UNSPECA "POP E40-PO.MIT.EDU ajericks"
ambar.pobox     HS    UNSPECA "POP E40-PO.MIT.EDU ambar"
andrew.pobox    HS    UNSPECA "POP E40-PO.MIT.EDU andrew"
annette.pobox   HS    UNSPECA "POP E40-PO.MIT.EDU annette"
austin.pobox    HS    UNSPECA "POP E40-PO.MIT.EDU austin"
```

SLOC.DB - service location

Description:
This file maps a service name to a machine name.

Queries used:
get_server_location(*) -> (server, location)

How modified:
dcm_maint

Client(s):

Example contents:

```
lcprimaryhost.sloc   HS    UNSPECA matisse.MIT.EDU
olctesthost.sloc     HS    UNSPECA castor.MIT.EDU
kerberos.sloc  HS    UNSPECA kerberos.MIT.EDU

zephyr.sloc    HS    UNSPECA NESKAYA.MIT.EDU
zephyr.sloc    HS    UNSPECA ARILINN.MIT.EDU
zephyr.sloc    HS    UNSPECA HOBBES.MIT.EDU
zephyr.sloc    HS    UNSPECA ORPHEUS.MIT.EDU
```

FILESYS.DB - Filesystem info

Description:
This file contains all the filesystems and their related information. The information presented in this file is a filesystem name relating to the following information: filesystem type, server name, filesystem name, default mount point, and access mode.

Queries used:
get_all_filesys() -> (label, type, machine, name, mount, access)
get_alias(*, FILESYS) -> (name, type, trans)

How modified:
attach_maint
user_reg -> associates user to a new filesys.

Client(s):
attach

Example contents:

```
NewrtStaffTool.filsys   HS     UNSPECA
   "RVD NewrtStaffTool helen r /mit/StaffTools"
NewvsStaffTool.filsys   HS     UNSPECA
   "RVD NewvsStaffTool helen r /mit/StaffTools"
Saltzer.filsys  HS     UNSPECA "RVD Saltzer helen r /mnt"
athena-backup.filsys    HS     UNSPECA
   "RVD athena-backup castor r /mnt"
```

- Update mechanism: Updating hesiod is a relatively simple process. Every six hours the DCM will initiate a build on each of the above files(assuming the informtion has changed). Once a file is constructed, the update mechanism will transport the file to each of the above machine.
- Service : RVD
- Description: The nature of RVD servers recognizes a very different approach from that of the hesiod discussion. The RVD mechanism is updated through two different means. The first method is for RVD_MAINT (an application interface) to talk to the RVD server directly. This program is described in detail in the section Specialized Management Interfaces. The important note here is that the RVD is updated by feeding the server directly with specific information, not complete files. The current program vdbdb performs the updating process to each RVD server. RVD_MAINT will use the same protocol. This process affords instantaneous changes to RVD's.

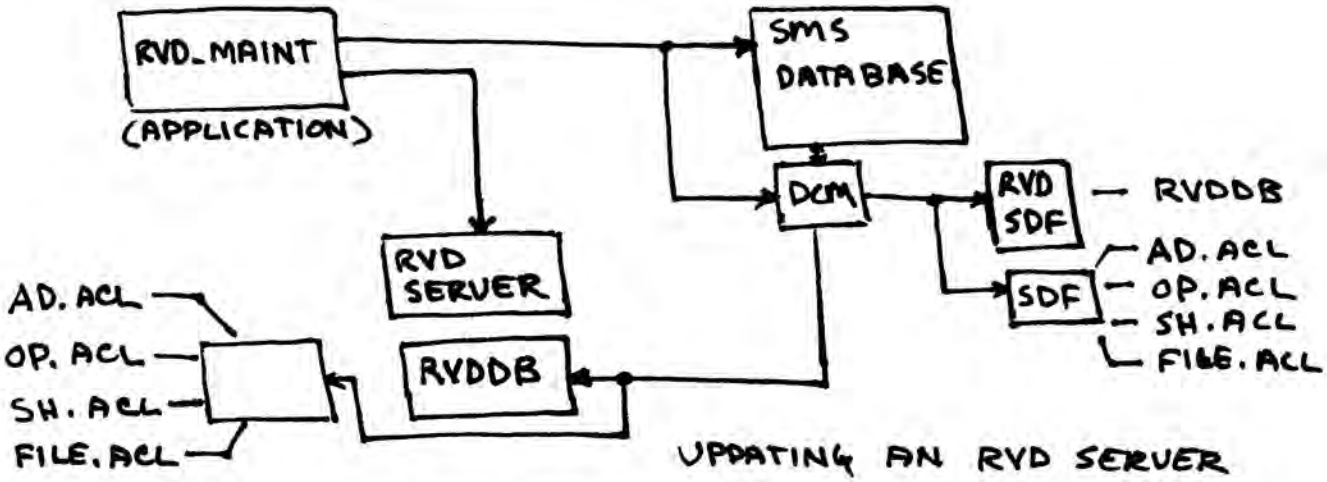
Secondly, when invoked, RVD_MAINT will also communicate with the sms database. This communication path will allow the updating of all the fields necessary to create rvddb, a RVD server specific file. The generation of this file is inherent to the DCM. If information has changed (via RVD_MAINT), the dcm will invoke a module which creates an rvddb file. This file is then propagated to the relevant RVD server. This file resides on

the target disk and is used in the event of server failure.

During a session with RVD_MAINT an administrator may make several changes to the RVD server. These changes will go into effect immediately. In addition to rvddb, a few acls must be present with the propagation. These files are:

- /site/rvd/acl/AD.acl /site/rvd/acl/OP.acl /site/rvd/acl/SH.acl /site/rvd/acl/file.acl - where file is host-packname.

RVD support can best be illustrated by the following diagram:




The discussion which follows describes the generation and contents of the rvddb file.

- Propagation interval: 15 minutes, hour aligned
- Data Format : RVD specific, ASCII
- Target Machines:

  andromache
  gaea
  hactar
  helen
  jinx
  m4-035-s
  m4-035-v
  m4-035-w
  oath
  persephone
  prak
  slartibartfast
  socrates
  zarquon
  calliope
  polyhymnia

- Target Path: /site/rvd/rvddb

• File(s):

RVDDB - RVD specific file

Description:
RVDDB is athe rvd specific file which is used by an rvd server. This file is only used in the event of a catastrophic failure with the rvd server. Nonetheless, this file represents all of the information integral to rvd servers.

Queries used:
get_rvd_servers(machine) -> (oper, admin, shutdown)
get_rvd_physical(machine) -> (device, size, created, modified)
get_all_rvd_virtual(machine) -> (name, device, packid, owner, rocap excap, shcap, modes, offset, size, created, modified, ownhost)
get_members_of_list(list) -> (member_type, member_name)

How modified:
rvd_maint

Client(s):
RVD server.

Content example:

```
operation = add_physical |
filename = /dev/         |
blocks=                  | - This is the header
created=                 |   unique to each
modified=                |   physical disk on
                        a machine

operation=add_virtual    |
physical=                |
name=                    |
owner=                   |
rocap=                   | - This is the information
excap=                   |   unique to each virtual
shcap=                   |   disk.
modes=                   |
offset=                  |   Block gets repeated n
blocks=                  |   times.
created=                 |   Where n is the number
modified=                |   of allocated RVDs on a
ownhost=                 |   physical disk.
uid=                     |
```

•

• Service: NFS

- Description: Sms supports two files which are necessary components of NFS operation. These files are:
  - /site/nfsid
  - /mit/quota

- 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 sms 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 sms and its creation of NFS lockers. During registration, the userreg program communicates exclusively with the sms 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> - using /mit/quota file
chown <UID> - using /site/nfsid
setquota <quota> - using /mit/quota
```

-
  - Propagation interval : 6 hours, 0:00, 6:00, 12:00, 18:00
  - Data Format : ASCII
  - Client(s):

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

  - Files updated:

/SITE/NFSID - username to uid/gid mapping.

Description:
This file is used for both the nfs server information and for the sms shell script. It provides a username to uid/gid mapping. The file is distributed to every NFS server and is identical on all.

Queries used:
get_all_logins() ->
(login, uid, shell, home, last, first, middle)

How updated:
created at registation time with userreg.
maintained with user_maint.

Contents example:

<username> <UID> <GID1, GID2,...GID32>

where: username is the user's login name (Ex: pjlevine)
UID is the users id number (Ex: 123456)
GIDn are the groups in which the user is a member (max 32)

/MIT/QUOTA - file containing username to quota mapping.

Description:
This file contains the mapping between username and quota. The file is distributed to each filesystem on the recipient machine. The contents of this file is used to create the NFS directory on the target machine. Each of the file's contents is unique to the filesystem which it represents.

Queries used:
get_all_nfsphys() -> (machine, dir, status, allocated, size)
get_nfs_quotas(machine, device) -> (login, quota)

How updated:
created at registation time with userreg.
maintained with user_maint.

Contents example:

<username> <quota>

where: username is the user's login name (Ex. pjlevine)
quota is the per user allocation (in Mbytes)

•

• Service: Mail (/usr/lib/aliases)

- Description: The generation of /usr/lib/aliases is a process which makes use of a currently existing program aliasbld.c. This program is called by the dcm every 24 hours. The hooks into the sms database are the significant changes made to aliasbld.c. The /usr/lib/aliases file is created and propagated to athena.mit.edu. Only one file and one propagation is required. The use of /usr/lib/alaises file is done manually by executing a shell script extract_aliases. The use, however, is not an sms-related function or responsibility.

- Data Type: ASCII

- Propagation interval: 24 hours, 3:00

- Target: ATHENA.MIT.EDU

- File(s):

  /USR/LIB/ALIASES - mail forwarding information

  Description:

  Queries Used:
  get_all_mail_lists() -> (list)
  get_members_of_list(list) -> (member_type, member_name)
  get_all_poboxes() -> (login, type, machine, box)

  How updated:
  listmaint -> for all mail list info.
  user_maint -> for pobox info.
  userreg -> initial info for poboxes.

  Contents example:

### 5.9. SMS-to-Server Update Protocol

SMS 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 SMS 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 SMS.

***Strategy***

A. Preparation phase. This phase is initiated by the SMS when it determines that an update should be performed. This can be triggered by a change in data, by an administrator, by a timer (run by cron), or by a restart of the SMS machine.

1. Check whether a data file is being constructed for transmission. If so, do nothing and report the fact. Otherwise, build a data file from the SMS database. (Building the data file is handled with a locking strategy that ensures that "the" data file available for distribution is not an incomplete one. The new data file is placed in position for transfer once it is complete using the **rename** system call.)

2. Extract from SMS the list of server machines to update, and the instructions for installing the file. Perform the remaining steps independently for each host.

3. Connect to the server host and send authentication.

4. Transfer the files to be installed to the server. These are stored in **filename.sms_update** until the update is actually performed. At the same time, the existing file is linked to **filename.sms_backup** for later deinstallations, and to minimize the overhead required in the actual installation (freeing disk pages). (The locking strategy employed throughout also ensures that this will not occur twice simultaneously.) A checksum is also transmitted to insure integrity of the data.

5. Transfer the installation instruction sequence to the server.

6. 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 SMS.

On a single command from the SMS, the server begins execution of the instruction sequence supplied. These can include the following:

1. 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 directory and by retaining the **filename.sms_backup** link to the file.

2. Revert the file -- identical to swapping in the new data file, but instead uses **filename.sms_backup**. May be useful in the case of an erroneous installation.

3. 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.

4. Execute a supplied command.

C. Confirm installation. The server sends back a reply indicating that the installation was successful. The SMS then updates the last-update-tried field of the update table, clears the override value, and sets the 'success' flag.

***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 SMS. The last-update-tried flag is set, and the 'success' flag is cleared, in the update table. The override value may be set, depending on the error condition and the default update interval.

The error value returned is logged to the appropriate file; at some point it may be desirable to use Zephyr to notify the system maintainers when failures occur.

A timeout is used in both sides of the connection during the preparation phase, and during the actual installation on the SMS. 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 SMS 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 SMS, and the existing **filename.sms_update** file will be deleted (as it may be incomplete) when the next update starts.

C. SMS crashes.

Since the SMS update table is driven by absolute times and offsets, crashes of the SMS machine will result in (at worst) delays in updates. If updates were in progress when the SMS 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 SMS.

***Considerations***

What happens if the SMS broadcasts an invalid data file to servers? In the case of name service, the SMS 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, SMS 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. For worst case scenario preparation, subsection 11.2 presents guidlines and mechanisms for catasrophic recovery procedure.

#### *5.9.2. Data Transport Security*

Each datagram which is transmitted over the network is secure using Kerberos, the Athena authentication system. Encyphered information headers will proceed each datagram. The servers decrypt the header information and use the packet accordingly.

Data going over the net will be checksummed before it is sent. This checksum will be put in a small encrypted "header", which will be decoded on the receiving side. 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.

Encryption of the data itself is an option that can be invoked depending on the sensitivity of the data involved. For instance, files such as /etc/rvdtab are not particularly secret, but the MIT id numbers of users are.

### 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 SMS; 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 algorhythm 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 SMS 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, First Last, {IDnumber, hashIDnumber}$_{hashIDnumber}$
grab_login, First Last, {IDnumber, hashIDnumber, login}$_{hashIDnumber}$
set_password, First Last, {IDnumber, hashIDnumber, password}$_{hashIDnumber}$

where
First Last is the student's name,
IDnumber is the student's id number (for example: 123456789)
hashIDnumber is the encrypted ID number (for example: lflenQqC/O/OE)

({ a, b }$_K$ means that 'a, b' is encrypted using the error propagating cypher-block-chaining mode of DES, as described in the Kerberos document).

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 retrieves each

When the student decides to register with athena, he walks up to a workstation and logs in using the username of "userreg" (no password is necessary). 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 with kerberos; if the login name is already in use, it returns a failure code to userreg. Otherwise, it allocates a home directory for the user on the least-loaded fileserver, builds a post office entry for the user, sets an initial quota for the user, and returns a success code to userreg.

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. Deployment, Integration, and Scheduling

The integration of SMS into the existing environment is as important as the design of SMS itself. The factors involved in deploying SMS into the current system revolve around the system's changing environment and increased demand for system resources. For SMS to be a functional tool in the Athena environment, it must allow the system administrator, from the start, a method of accessing and controlling data.

SMS's deployment will occur in conjunction with the existing administrative process. The current database will be used as SMS begins to take on responsibiliy. Basically, SMS will take on more responsibility as the current system relinquishes responsibility. Administrative changes, therefore, must be carefully monitored so that the SMS fields which are "on-line" receive the most relevant data.

Because Athena will not support time sharing systems in the future, SMS will not support time sharing. This then assumes that the old database will support the time sharing environment as long as it still exists.

### *6.0.1. Deployment*

SMS is a system which does not operate in peices. For every service, there is a corresponding application interface. For every service, there is a field in the database; and, for every service, there is a consumer. The deployment and testing of SMS is a difficult and sensitive process. Basically, once SMS is committed to support Athena, everything must be in place. There is no redundancy. For example, the support of hesiod alone requires seven application programs and the generation and propagation of nine different files. Tetsing the system components individually is a relatively straight forward process; testing the complete system and simulating real usage patterns is much more difficult. At this time, a plan is being drafted which will present the testing and deployment schedule for SMS.

A suitable plan, under consideration, is to provide a dummy database which will be a test bed for SMS. Staff members at Athena will be notified of this database. For a week's time this database will be tested as a dummy, allowing staff time to flush out bugs. ALL CHANGES MADE TO THIS DATABASE WILL NOT BE PERMANENT. This means that if I change my 'finger' information in this dummy database, it will not be affected in the real world. Testing must first occur in a closed-cell environment, where potential snags will not disrupt the day-to-day operations of Athena. Each application program will be tested in as realistic of an environment as possible. Userreg, the new student registration program, is especially important. This program will see great demand during the first week of school and therefore must work without a problem. There will be no time for debugging userreg after school starts. Therefore, userreg will be stressed by having many staff members try to use the program all at the same time. This will attempt to simulate a worst case scenario of operation.

The deployment plan will be flushed out forthcoming.

## 7. Long Term Support Strategy

The eventual users of SMS will be the operations component of the Athena organization. The support of SMS will be left to the operations and informations systems groups.

Both vendor has indicated an interest - it will be difficult to choose a possible future.

## 8. Data Fields and Relationships

The knowledge base of SMS 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 SMS 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. |
| | *last, first, middle* | The user's full name, broken down for convenient indexing. |
| | *shell* | the user's default shell. |
| | *home* | name of the users home file system. |
| | *status* | contains flags for the user. The only currently defined flag is bit 0, which when set indicates that the user is active. (An active user is one who has been assigned system resources such as a mailbox, a home directory, and who may be a member of one or more lists.) |
| | *mit_id* | the user's encrypted MIT id. |
| | *mit_year* | a student's academic year, not modifiable by the student. Used for Athean administrative purposes. |
| | *expdate* | the expiration date of the user and the user's resources (the user becomes inactive.) |
| | *modtime* | the time that the user record was last modified (or created). |
| | See Section 9.0.1 for the list of queries associated with this table. | |
| | There are no entries for password and primary gid because these are being subsumed by other services (Kerberos, ACLS). | |
| FINGER | This table contains the "finger" information for users. | |
| | *users_id* | corresponds to the users_id in the users table. |
| | *fullname* | the user's full name. |

| | | |
|---|---|---|
| | *nickname* | the user's nickname. |
| | *home_address* | home address. |
| | *home_phone* | home phone. |
| | *mit_address* | MIT address; this is for on-campus students' living addresses. |
| | *mit_phone* | MIT phone. |
| | *office* | office address. |
| | *affiliation* | one of undergraduate, graduate, staff, faculty, other. |
| | *department* | student's major or employee's department. |
| | *year* | student's year or "G". |
| | *modtime* | time finger record was last modified. |

See Section 9.0.2 for the list of queries associated with this table.

MACHINE Machine Information.

| | | |
|---|---|---|
| | *name* | the canonical hostname. |
| | *machine_id* | an internal database id for this record. |
| | *type* | machine type: VS, RTPC, VAX |
| | *model* | machine model: VS2, VS2000, RTFLOOR, RTDESK, 750, 785. |
| | *status* | machine status: PUBLIC, PRIVATE, SERVER, TIMESHARE. |
| | *serial* | serial number. |
| | *sys_type* | system type (for use by release engineering and operations). |

See Section 9.0.3 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. |
| | *description* | cluster description. |
| | *location* | cluster location. |
| | *cluster_id* | internal database identifier for this record. |

See Section 9.0.4 for the list of queries associated with this table.

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

| | | |
|---|---|---|
| | *cluster_id* | cluster id. |
| | *machine_id* | machine id. |

See Section 9.0.4 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.

| | |
|---|---|
| *cluster_id* | references an entry in the cluster table. |
| *serv_label* | label of a service cluster type (e.g. "prcluster", "usrlib", "syslib") |
| *serv_cluster* | specific service cluster name (e.g. "e40-prcluster") |

See Section 9.0.5 for the list of queries associated with this table.

PRCLUMAP

This table provides a mapping between printer service cluster names and printer names.

| | |
|---|---|
| *prcluster* | printer cluster name |
| *p_id* | printer id. |

See Section 9.0.6 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 server.

| | |
|---|---|
| *service_name* | name of service. |
| *update_int* | server update interval in minutes (for DCM). |
| *target_file* | target file on server for DCM generated server files. |
| *dfgen* | time of server file generation |
| *script* | shell script used by servers for particular use. |

See Section 9.0.7 for the list of queries associated with this table.

HOSTS

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

| | |
|---|---|
| *service_name* | name of service. |
| *mach_id* | Machine id for a host containing the service. |
| *enable* | Enable switch for DCM. This switch controls whether or not the DCM updates a server. (0 - Do not Update, 1 - Update) |
| *override* | Override time (minutes). Used by DCM and update mechanism to indicate that an update has failed. This time is used to update services at a different time from the default update interval time. (-1 - Use the default interval time, 0 or greater - Use the override interval). |
| *ltt* | Last time tried. Used by dcm, this field is adjusted each time a service is attemted to be updated, regardless of success or failure. |

| | | |
|---|---|---|
| | *success* | Flag indicating successful completion of server update. |
| | *value1* | server-specific data used by applications (i.e., number of servers permitted per machine). |
| | *value2* | additional server-specific data. |

See Section 9.0.7 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 SMS and the Hesiod name server, service information will be obtained from the name server. |
| | *service* | service name. |
| | *protocol* | protocol: one of TCP, UDP. |
| | *port* | port number. |
| | *description* | description of service. |

See Section 9.0.8 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 unique name for an attachable file system. |
| | *type* | currently one of RVD, NFS, or RFS. |
| | *machine_id* | file server machine. |
| | *name* | name of file system on the server. |
| | *mount* | default mount point for file system. |
| | *access* | default access mode for file system. |

See Section 9.0.9 for the list of queries associated with this table.

| | | |
|---|---|---|
| RVDSRV | | RVD Server Information. This table contains the top level access control lists for each rvd server. Any other per-server information should be added here. |
| | *machine_id* | server machine. |
| | *operations_acl* | operations access control list. |
| | *admin_acl* | administrative access control list. |
| | *shutdown_acl* | shutdown access control list. |

See Section 9.0.10 for the list of queries associated with this table.

| | | |
|---|---|---|
| RVDPHYS | | Physical device partition table. |
| | *machine_id* | server machine. |
| | *device* | rvd physical device. |
| | *size* | size in 512-byte blocks. |
| | *created* | creation time. |

*modified* modification time.

See Section 9.0.10 for the list of queries associated with this table.

RVDVIRT Virtual device table. This table contains the list of virtual devices for each rvd server machine. Information per device includes the rvd physical device it resides on, its name, unique pack id, owner (*users* id), access control lists for the device (*rocap, excap, shcap*), allowable access modes, the offset within the physical device, its size, a *machine* id for a host that has default access to the device, and the creation and modifications dates for the device.

*machine_id* server machine.

*device* rvd physical device.

*name* virtual device name.

*packid* unique pack id.

*owner* owner (currently an arbitrary string, should be a USERS entry id.)

*rocap, excap, shcap* currently three passwords providing read-only, exclusive, and shared accesss to the file system. These should be condensed into one access control list id. The access control list would indicate read and write access (shared access has never been implemented).

*modes* allowable access modes for device.

*offset* offset of virtual device into physical device (in blocks).

*blocks* size of virtual device.

*ownhost* name of host from which device may be mounted without a password (not used with acl's?).

*created* creation time.

*modified* modification time.

See Section 9.0.10 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.

*machine_id* server machine.

*device* file system name.

*dir* top-level directory of device.

*allocated* number of quota units allocated to this device.

*size* capacity of this device in quota units.

See Section 9.0.11 for the list of queries associated with this table.

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

| | |
|---|---|
| *machine_id* | nfs server machine. |
| *device* | nfs server file system. |
| *users_id* | user id. |
| *quota* | user quota in quota units. |

See Section 9.0.11 for the list of queries associated with this table.

PRINTER — Printer Information.

| | |
|---|---|
| *name* | a unique printer name. |
| *printer_id* | internal database identifier for this record. |
| *type* | printer hardware type: one of LPS40, 3812, LN03, LN01, etc. |
| *description* | description of this printer. |

See Section 9.0.12 for the list of queries associated with this table.

QUEUE — Printer queues. This table contains a list of unique queue names and the attributes of each queue. Attributes of queue are its *ability*, a status string (used by MDQS servers), and an access control list.

| | |
|---|---|
| *name* | unique queue name. |
| *queue_id* | internal database id for this queue. |
| *machine_id* | server machine. |
| *abilities* | printer abilities associated with this queue (stored as an integer bitmask). |
| *default* | flag indicating whether this printer should be treated as a default printer. |
| *status* | queue status string (used by MDQS). |

See Section 9.0.12 for the list of queries associated with this table.

PQM — Printer to queue mapping. This table provides the mapping between printers and queues.

| | |
|---|---|
| *printer_id* | printer id. |
| *queue_id* | queue id. |

See Section 9.0.12 for the list of queries associated with this table.

QDEV — MDQS device information. Each MDQS server has a device table that assigns a logical name and status information to each known physical printer device.

| | |
|---|---|
| *machine_id* | MDQS server machine. |
| *qdev_id* | internal database id for this device. |
| *name* | logical name of device. |

| | | |
|---|---|---|
| | *device* | physcial device name (e.g.: /dev/lp0) |
| | *status* | status string kept by MDQS. |

See Section 9.0.12 for the list of queries associated with this table.

QDM

MDQS queue to device mapping. This table ties together the queue and device information for each MDQS server. Printer names are not actually known to MDQS; printer to queue mapping is handled by Hesiod using the PQM table information described above.

| | |
|---|---|
| *machine_id* | MDQS server machine. |
| *queue_id* | queue id. |
| *device_id* | device id. |
| *server* | name of a server program to invoke for jobs using this queue and device. |

See Section 9.0.12 for the list of queries associated with this table.

POBOX

Post Office Information. This list matches users with one or more post office boxes. A post office box is identified by its type (POP, LOCAL), the machine on which the box resides, and the box name on that machine.

| | |
|---|---|
| *users_id* | id for a USERS entry. |
| *type* | mailbox type: one of POP, LOCAL, or FOREIGN. |
| *machine_id* | post office server machine (or string_id if type is FOREIGN). |
| *box* | mailbox name on server. |

See Section 9.0.13 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 list associated with the list. An access control list, which is itself a list, contains as members a set 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. |
| *flags* | currently one or more of ACTIVE, PUBLIC, HIDDEN. (These flags are used for mailing lists.) |
| *description* | description of list. |
| *acl_id* | a list id for the administrators' list. |
| *creator* | users id of the creator of this list. |
| *expdate* | expiration date of list. |
| *modtime* | time list was last modified (LIST entry or MEMBERS entry). |

See Section 9.0.14 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 9.0.14 for the list of queries associated with this table.

STRINGS — Used for list members of *string* type. An optimization for dealing with (usually long) foreign mail addresses.

| | |
|---|---|
| *string_id* | member id. |
| *string* | string. |
| *refc* | Reference count. A single string can be a member of multiple lists. When the reference count goes to zero, the string is deleted. |

See Section 9.0.14 for the list of queries associated with this table.

MAILLISTS — This table contains the set of list ids for the lists which are to be used as mailing lists.

| | |
|---|---|
| *list_id* | a list id. |

See Section 9.0.14 for the list of queries associated with this table.

GROUPS — This table contains the set of list ids for the lists which are to be used as groups.

| | |
|---|---|
| *list_id* | a list id. |
| *gid* | unix gid. |

See Section 9.0.14 for the list of queries associated with this table.

ACLS — This table contains a set of service, list id pairs which define the access control lists that are needed for each service.

| | |
|---|---|
| *service* | service name. |
| *list_id* | a list id. |

See Section 9.0.14 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 SMS predefined queries. Each query name appears as a capability name in this list.

| | |
|---|---|
| *capability* | a string. |
| *tag* | four character tag name for this capability. |
| *list_id* | a list id. |

See Section 9.0.14 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. Some examples of alias usage are printer aliases, service aliases, cluster aliases, file system aliases, and print cluster to printer maps. As an integrity constraint it is required that all aliases be of a known type. The list of known alias types is actually stored in the database as the set of translations of aliases with name "alias" and type "type". Therefore, it is also quite easy to add new alias types. Another use of the "type" alias type is for storing known field values for validated table fields.

| | |
|---|---|
| *name* | alias name. |
| *type* | alias type: currently one of TYPE, PRINTER, SERVICE, CLUSTER, FILESYS, PRCLUSTER, MACH-CLU-MAP. |
| *trans* | alias translation. |

See Section 9.0.15 for the list of queries associated with this table.

VALUES — Values needed by the server or application programs.

| | |
|---|---|
| *name* | value name. |
| *value* | value. |

See Section 9.0.16 for the list of queries associated with this table.

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

| | |
|---|---|
| *table* | table name. |
| *retrieves* | count of retrievals on this table. |
| *appends* | count of additions to this table. |
| *updates* | count of updates to this table. |
| *deletes* | count of deletions to this table. |
| *modtime* | time of last modification (append, update, or delete). |

See Section 9.0.17 for the list of queries associated with this table.

#### *8.0.1. Predefined Database Queries*

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 SMS 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 SMS server, linking the pre-defined queries to a new set of data manipulation procedures.

See Section 9 for a complete list and description of the predefined queries.

## 9. Predefined Queries - List of Database Interfaces

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, the following two error codes may be returned by any query: SMS_SUCCESS for successful completion of the query, and SMS_PERM indicating that permission was denied for the query.

### *9.0.1. Users*

**get_all_logins**
```
Args: none
Returns: {login, uid, shell, home, last, first, middle}
```

**get_all_active_logins**
```
Args: none
Returns: {login, uid, shell, home, last, first, middle}
```

**get_all_active_users**
```
Args: none
Returns: {login, uid}
```

**get_user_by_login**
```
Args: (login(*))
Returns: {login, uid, shell, home, last, first, middle, status,
          mit_id, mit_year, expdate, modtime}
Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**get_user_by_firstname**
```
Args: (firstname(*))
Returns: {login, uid, shell, home, last, first, middle, status,
          mit_id, mit_year, expdate, modtime}
Errors: SMS_NO_MATCH
```

**get_user_by_lastname**
```
Args: (lastname(*))
Returns: {login, uid, shell, home, last, first, middle, status,
          mit_id, mit_year, expdate, modtime}
Errors: SMS_NO_MATCH
```

**get_user_by_first_and_last**
```
Args: (firstname(*), lastname(*))
Returns: {login, uid, shell, home, last, first, middle, status,
          mit_id, mit_year, expdate, modtime}
Errors: SMS_NO_MATCH
```

**get_user_by_mitid**
```
Args: (mit_id)
Returns: {login, uid, shell, home, last, first, middle, status,
          mit_id, mit_year, expdate, modtime}
```

**get_user_by_year**
```
Args: (year)
```

```
  Returns: (login, uid, shell, home, last, first, middle, status,
            mit_id, mit_year, expdate, modtime)
  Errors: SMS_NO_MATCH
```

**update_user_shell**

```
  Args: (login, shell)
  Returns: none
  Integrity: application should check for valid shell program
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**update_user_status**

```
  Args: (login, status)
  Returns: none
  Integrity: application should check for valid status value
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**update_user_home**

```
  Args: (login, home)
  Returns: none
  Integrity: home must be a known filesys entry
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE, SMS_FILESYS
```

**add_user**

```
  Args: (login, uid, shell, home, last, first, middle, status,
         mit_id, mit_year, expdate)
  Returns: none
  Integrity: application must check for valid shell and status
             values; home must be a valid filesys entry; expdate must
             be reasonable; modtime is set by the server.
  Errors: SMS_EXISTS, SMS_FILESYS, SMS_DATE
  Side Effects: blank finger entry created
```

**update_user**

```
  Args: (login, newlogin, uid, shell, home, last, first, middle,
         status, mit_id, mit_year, expdate)
  Returns: none
  Integrity: application must check for valid shell and status
             values; home must be a valid filesys entry; expdate must
             be reasonable; modtime is set by the server.
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE, SMS_FILESYS, SMS_DATE
```

**delete_user**

```
  Args: (login)
  Returns: none
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
  Side Effects: finger entry deleted
```

### *9.0.2. Finger*

**get_finger_by_login**

```
  Args: (login(*))
  Returns: (login, fullname, nickname, home_addr, home_phone,
            office_addr, office_phone, mit_dept, mit_year, modtime)
  Errors: SMS_NO_MATCH, SMS_USER
```

**get_finger_by_first_last**

```
  Args: (first(*), last(*))
  Returns: {login, fullname, nickname, home_addr, home_phone,
            office_addr, office_phone, mit_dept, mit_year, modtime}
  Errors: SMS_NO_MATCH
```

**update_finger_by_login**

```
  Args: (login, fullname, nickname, home_addr, home_phone,
         office_addr, office_phone, mit_dept, mit_year, *modtime)
  Returns: none
  Integrity: modtime set by server, all other fields validated by
             application. (Perhaps there will be a table specifying
             valid values for the mit_year field.)
  Errors: SMS_NO_MATCH, SMS_USER
```

### *9.0.3. Machine*

**get_machine_by_name**

```
  Args: (name(*))
  Returns: {name, type, model, status, serial, sys_type}
  Errors: SMS_NO_MATCH
```

**add_machine**

```
  Args: (name, type, model, status, serial, sys_type)
  Returns: none
  Integrity: type, model, and sys_type are checked against valid
             values in the database
  Errors: SMS_EXISTS, SMS_TYPE
```

**update_machine**

```
  Args: (name, newname, type, model, status, serial, sys_type)
  Returns: none
  Integrity: type, model, and sys_type are checked against valid
             values in the database
  Errors: SMS_MACHINE, SMS_TYPE
```

**delete_machine**

```
  Args: (name)
  Returns: none
  Errors: SMS_MACHINE
```

### *9.0.4. Cluster*

**get_cluster_info**

```
  Args: (name(*))
  Returns: {name, desc, location}
  Errors: SMS_NO_MATCH
```

**add_cluster**

```
  Args: (name, desc, location)
  Returns: none
  Errors: SMS_EXISTS
```

**update_cluster**

```
  Args: (name, newname, desc, location)
```

```
  Returns: none
  Errors: SMS_CLUSTER
```

**delete_cluster**

```
  Args: (name)
  Returns: none
  Errors: SMS_CLUSTER
```

**get_machine_to_cluster_map**

```
  Args: none
  Returns: (machine, cluster)
```

**add_machine_to_cluster**

```
  Args: (machine, cluster)
  Returns: none
  Integrity: machine and cluster must exist is machine and cluster
             tables.
  Errors: SMS_MACHINE, SMS_CLUSTER, SMS_EXISTS
```

**delete_machine_from_cluster**

```
  Args: (machine, cluster)
  Returns: none
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

### *9.0.5. Service Clusters*

**get_all_service_clusters**

```
  Args: none
  Returns: (cluster, service-label, service-cluster)
```

**add_service_cluster**

```
  Args: (cluster, service-label, service-cluster)
  Returns: none
  Integrity: cluster must exist in cluster table;
             cluster/service-label must be unique.
  Errors: SMS_CLUSTER, SMS_EXISTS
```

**update_service_cluster**

```
  Args: (cluster, service-label, service-cluster)
  Returns: none
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**delete_service_cluster**

```
  Args: (cluster, service-label)
  Returns: none
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

### *9.0.6. Printer Clusters*

**get_all_printer_clusters**

```
  Args: none
  Returns: (prcluster)
```

**get_printers_of_cluster**

```
  Args: (prcluster)
  Returns: {printer, queue, machine, printer-type, abilities,
            default}
  Errors: SMS_CLUSTER
```

### *9.0.7. Servers*

**get_server_info**

```
  Args: (service(*))
  Returns: {service, update_int, target_file, script, dfgen}
  Errors: SMS_NO_MATCH
```

**add_server_info**

```
  Args: (service, update_int, target_file, script, dfgen)
  Returns: none
  Integrity: application must verify that target_dir and script
             exist; dfgen must be a valid date
  Errors: SMS_EXISTS, SMS_DATE
```

**update_server_info**

```
  Args: (service, update_int, target_file, script, dfgen)
  Returns: none
  Integrity: application must verify that target_dir and script
             exist; dfgen must be a valid date
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE, SMS_DATE
```

**delete_server_info**

```
  Args: (service)
  Returns: none
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**get_server_host_info**

```
  Args: (service(*), machine(*))
  Returns: {service, machine, enable, overide, ltt, success, value1,
            value2}
  Errors: SMS_NO_MATCH
```

**add_server_host**

```
  Args: (service, machine, enable, overide, ltt, success, value1,
         value2)
  Returns: none
  Integrity: machine must exist; last must be a valid date or null.
  Errors: SMS_EXISTS, SMS_MACHINE, SMS_DATE
```

**update_server_host**

```
  Args: (service, machine, enable, overide, ltt, success, value1,
         value2)
  Returns: none
  Integrity: last must be a valid date or null.
  Errors: SMS_MACHINE, SMS_NO_MATCH, SMS_NOT_UNIQUE, SMS_DATE
```

**delete_server_host**

```
  Args: (service, machine)
  Returns: none
  Errors: SMS_MACHINE, SMS_NO_MATCH, SMS_NOT_UNIQUE
```

```
get_server_locations
  Args: (service)
  Returns: (service, machine)
  Errors: SMS_SERVICE
```

### 9.0.8. Services

```
get_all_services
  Args: none
  Returns: (service, protocol, port, description)

add_service
  Args: (service, protocol, port, description)
  Returns: none
  Integrity: application should validate all fields
  Errors: SMS_EXISTS

delete_service
  Args: (service, protocol(*))
  Returns: none
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE

update_service
  Args: (service, protocol, port, description)
  Returns: none
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

### 9.0.9. File Systems

```
get_all_filesys
  Args: none
  Returns: (label, type, machine, name, mount, access)

get_filesys_by_label
  Args: (label(*))
  Returns: (label, type, machine, name, mount, access)
  Errors: SMS_NO_MATCH

get_filesys_by_machine
  Args: (machine(*))
  Returns: (label, type, machine, name, mount, access)
  Errors: SMS_MACHINE, SMS_NO_MATCH

add_filesys
  Args: (label, type, machine, name, mount, access)
  Returns: none
  Integrity: type must be a known type; machine must exist in machine
             table.
  Errors: SMS_EXISTS, SMS_TYPE, SMS_MACHINE

update_filesys
  Args: (label, type, machine, name, mount, access)
  Returns: none
  Integrity: type must be a known type; machine must exist in machine
```

```
            table.
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE, SMS_TYPE, SMS_MACHINE

delete_filesys
  Args: (label)
  Returns: none
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

### *9.0.10. RVD*

```
get_rvd_server
  Args: (machine)
  Returns: (machine, oper_acl, admin_acl, shutdown_acl)
  Errors: SMS_MACHINE, SMS_NO_MATCH

add_rvd_server
  Args: (machine, oper_acl, admin_acl, shutdown_acl)
  Returns: none
  Integrity: oper, admin, and shutdown must be valid list names.
  Errors: SMS_EXISTS, SMS_MACHINE, SMS_LIST

update_rvd_server
  Args: (machine, oper_acl, admin_acl, shutdown_acl)
  Returns: none
  Integrity: oper, admin, and shutdown must be valid list names.
  Errors: SMS_LIST

delete_rvd_server
  Args: (machine)
  Returns: none
  Errors: SMS_MACHINE, SMS_NO_MATCH
  Side Effects: deletes all rvd_physical and rvd_virtual entries
                associated with the server.

get_all_rvd_physical
  Args: (machine)
  Returns: (device, size, created, modified)
  Errors: SMS_MACHINE, SMS_NO_MATCH

get_rvd_physical
  Args: (machine, device)
  Returns: (size, created, modified)
  Errors: SMS_MACHINE, SMS_NO_MATCH

add_rvd_physical
  Args: (machine, device, size, created, modified)
  Returns: none
  Integrity: machine must exist in machine table
  Errors: SMS_MACHINE, SMS_EXISTS

delete_rvd_physical
  Args: (machine, device)
  Returns: none
  Errors: SMS_MACHINE, SMS_NO_MATCH

get_all_rvd_virtual
```

```
  Args: (machine)
  Returns: {name, device, packid, owner, rocap, excap, shcap, modes,
            offset, size, created, modified, ownhost}
  Errors: SMS_MACHINE, SMS_NO_MATCH
```

**get_rvd_virtual**

```
  Args: (machine, name)
  Returns: {device, packid, owner, rocap, excap, shcap, modes,
            offset, size, created, modified, ownhost}
  Errors: SMS_NO_MATCH
```

**add_rvd_virtual**

```
  Args: (machine, name, device, packid, owner, rocap, excap, shcap,
         modes, offset, size, created, modified, ownhost)
  Returns: none
  Integrity: machine, ownhost must exist in machine table;
             machine/device must exist in rvdphys table.
  Errors: SMS_EXISTS, SMS_MACHINE, SMS_DEVICE
```

**update_rvd_virtual**

```
  Args: (machine, name, newname, device, packid, owner, rocap, excap,
         shcap, modes, offset, size, created, modified, ownhost)
  Returns: none
  Integrity: ownhost must exist in machine table.
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE, SMS_MACHINE
```

**delete_rvd_virtual**

```
  Args: (machine, device, name)
  Returns: none
  Errors: SMS_MACHINE, SMS_NO_MATCH, SMS_NOT_UNIQUE
```

### *9.0.11. NFS*

**get_all_nfsphys**

```
  Args: (machine)
  Returns: {device, dir, status, allocated, size}
  Errors: SMS_MACHINE, SMS_NO_MATCH
```

**get_nfsphys**

```
  Args: (machine, device)
  Returns: {dir, status, allocated, size}
  Errors: SMS_MACHINE, SMS_NO_MATCH
```

**add_nfsphys**

```
  Args: (machine, device, dir, status, allocated, size)
  Returns: none
  Errors: SMS_MACHINE, SMS_EXISTS
```

**delete_nfsphys**

```
  Args: (machine, device)
  Returns: none
  Errors: SMS_MACHINE, SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**get_nfs_quotas**

```
  Args: (machine, device)
  Returns: {login, quota}
```

```
  Errors: SMS_NO_MATCH, SMS_MACHINE

get_nfs_quotas_by_user
  Args: (login)
  Returns: {machine, device, quota}
  Errors: SMS_USER

add_nfs_quota
  Args: (machine, device, login, quota)
  Returns: none
  Integrity: machine must exist, user must exist
  Errors: SMS_EXISTS, SMS_MACHINE, SMS_USER

update_nfs_quota
  Args: (machine, device, login, quota)
  Returns: none
  Integrity: machine must exist, user must exist
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE, SMS_MACHINE, SMS_USER

delete_nfs_quota
  Args: (machine, device, login)
  Returns: none
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE, SMS_MACHINE, SMS_USER
```

#### *9.0.12. Printers*

```
get_all_printers
  Args: none
  Returns: {printer, type, desc}

get_printer_info
  Args: (printer)
  Returns: {printer, type, desc}
  Errors: SMS_NO_MATCH

add_printer
  Args: (printer, type, desc)
  Returns: none
  Integrity: type must be known
  Errors: SMS_EXISTS

update_printer
  Args: (printer, type, desc)
  Returns: none
  Integrity: type must be known
  Errors: SMS_PRINTER, SMS_NO_MATCH

delete_printer
  Args: (printer)
  Returns: none
  Errors: SMS_PRINTER, SMS_NO_MATCH

get_all_queues
  Args: none
  Returns: {queue, machine, abilities, default, status}
```

**get_queue_info**
```
Args: (queue)
Returns: {queue, machine, abilities, default, status}
Errors: SMS_QUEUE, SMS_NO_MATCH
```

**add_queue**
```
Args: (queue, machine, abilities, default, status)
Returns: none
Integrity: machine must exist; abilty is validated by application
Errors: SMS_EXISTS, SMS_MACHINE
```

**update_queue**
```
Args: (queue, machine, abilities, default, status)
Returns: none
Integrity: machine must exist; abilty is validated by application
Errors: SMS_QUEUE, SMS_NO_MATCH, SMS_MACHINE
```

**delete_queue**
```
Args: (queue)
Returns: none
Errors: SMS_QUEUE, SMS_NO_MATCH
```

**add_printer_to_queue**
```
Args: (printer, queue)
Returns: none
Integrity: printer and queue must exist
Errors: SMS_PRINTER, SMS_QUEUE
```

**delete_printer_from_queue**
```
Args: (printer, queue)
Returns: none
Errors: SMS_PRINTER, SMS_QUEUE
```

**get_qdev**
```
Args: (machine)
Returns: {device, physical, machine, status}
Errors: SMS_MACHINE
```

**add_qdev**
```
Args: (device, physical, machine, status)
Returns: none
Integrity: machine must exist; application must verify that
           physical device exists.
Errors: SMS_EXISTS, SMS_MACHINE
```

**update_qdev**
```
Args: (device, physical, machine, status)
Returns: none
Integrity: only status may be updated
Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**delete_qdev**
```
Args: (device, machine)
Returns: none
Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**get_queue_device_map**

```
  Args: (machine)
  Returns: {queue, device, machine, server}
  Errors: SMS_MACHINE
```

**add_queue_device_map**

```
  Args: (queue, device, machine, server)
  Returns: none
  Integrity: queue, device, machine must exist; application must
             verify that server program exists.
  Errors: SMS_QUEUE, SMS_DEVICE, SMS_MACHINE
```

**update_queue_device_map**

```
  Args: (queue, device, machine, server)
  Returns: none
  Integrity: only server may be updated; application must verify
             existence of server program.
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**delete_queue_device_map**

```
  Args: (queue, device, machine)
  Returns: none
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

#### *9.0.13. Post Office Boxes*

**get_pobox**

```
  Args: (login)
  Returns: {login, type, machine, box}
  Errors: SMS_USER, SMS_NO_MATCH
```

**add_pobox**

```
  Args: (login, type, machine, box)
  Returns: none
  Integrity: type, user, machine must exist
  Errors: SMS_EXISTS, SMS_TYPE, SMS_USER, SMS_MACHINE
```

**delete_pobox**

```
  Args: (login, type, machine, box)
  Returns: none
  Errors: SMS_USER, SMS_MACHINE, SMS_NO_MATCH
```

#### *9.0.14. Lists*

**get_list_info**

```
  Args: (list_name)
  Returns: {list_name, description, flags, admin_acl, expdate,
            modtime}
  Errors: SMS_LIST
```

**add_list**

```
  Args: (list_name, description, flags, admin_acl, expdate)
  Returns: none
  Integrity: expdate must be reasonable; application is responsible
             for
```

```
            flags; modtime set by server.
  Errors: SMS_EXISTS, SMS_ACL, SMS_DATE
```

**update_list**

```
  Args: (list_name, description, flags, admin_acl, expdate)
  Returns: none
  Integrity: expdate must be reasonable; application is responsible
             for flags.
  Errors: SMS_LIST, SMS_ACL, SMS_DATE
```

**update_list_admin**

```
  Args: (list_name, admin_acl)
  Returns: none
  Integrity: admin_acl must be a known list.
  Errors: SMS_LIST, SMS_ACL
```

**delete_list**

```
  Args: (list_name)
  Returns: none
  Errors: SMS_LIST
  Side Effects: All members of list are deleted.
```

**add_member_to_list**

```
  Args: (list_name, member_type, member_name)
  Returns: none
  Integrity: member_type must be known; list_name and member_name
             must be unique; if member_type is "user" or "list",
             then corresponding user or list must exist.
  Side Effects: if member_type = "string", entry added to strings
                table; list modtime updated.
  Errors: SMS_EXISTS, SMS_TYPE, SMS_LIST, SMS_USER, SMS_NO_MATCH,
          SMS_NOT_UNIQUE
```

**delete_member_from_list**

```
  Args: (list_name, member_type, member_name)
  Returns: none
  Integrity: member_type must be known; list_name and member_name
             must be unique.
  Side Effects: if member_type is "string", then corresponding string
                entry is deleted; list modtime updated.
  Errors: SMS_TYPE, SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**get_members_of_list**

```
  Args: (list_name)
  Returns: {member_type, member_name}
  Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

### *Mailing Lists*

**get_all_maillists**

```
  Args: none
  Returns: {list}
```

**get_all_visible_maillists**

```
  Args: none
  Returns: {list}
```

**add_maillist**
```
Args: (list)
Returns: none
Errors: SMS_EXISTS, SMS_LIST
```

**delete_maillist**
```
Args: (list)
Returns: none
Errors: SMS_LIST, SMS_NO_MATCH
```

### *Groups*

**get_all_groups**
```
Args: none
Returns: {list}
```

**add_group**
```
Args: (list)
Returns: none
Integrity: list must exist
Errors: SMS_EXISTS, SMS_LIST
```

**add_user_group**
```
Args: (login)
Returns: none
Errors: SMS_USER, SMS_LIST
Description: Optimized query for creating a user group (list) and
             adding the user as a member.  Returns SMS_LIST if a
             list already exists with the user's name.
```

**delete_group**
```
Args: (list)
Returns: none
Errors: SMS_NO_MATCH, SMS_LIST
```

### *Access Control Lists*

**get_acls**
```
Args: (service)
Returns: {list}
Errors: SMS_SERVICE
```

**add_acls**
```
Args: (service, list)
Returns: none
Errors: SMS_EXISTS, SMS_SERVICE, SMS_LIST
```

**delete_acls**
```
Args: (service, list)
Returns: none
Errors: SMS_SERVICE, SMS_LIST, SMS_NO_MATCH
```

### *9.0.15. Aliases*

**get_alias**
```
Args: (name(*), type)
Returns: {name, type, trans}
Errors: SMS_TYPE, SMS_NO_MATCH
```

**add_alias**
```
Args: (name, type, trans)
Returns: none
Integrity: type must exist as a translation of
           get_alias("alias", "TYPE").
Errors: SMS_EXISTS, SMS_TYPE
```

**delete_alias**
```
Args: (name, type)
Returns: none
Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

### *9.0.16. Values*

This section contains values that are needed by the server or application programs for updating the database. Some examples are:

- next users_id
- next list_id
- default user disk quota

**get_value**
```
Args: (name)
Returns: {value}
Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**add_value**
```
Args: (name, value)
Returns: none
Errors: SMS_EXISTS
```

**update_value**
```
Args: (name, value)
Returns: none
Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

**delete_value**
```
Args: (name)
Returns: none
Errors: SMS_NO_MATCH, SMS_NOT_UNIQUE
```

### *9.0.17. Table Statistics*

**get_all_table_stats**
```
Args: none
Returns: {table, retrieves, appends, updates, deletes, modtime}
```

**qet_query_need**
```
Args: (query, last_get_time)
Returns: {true | false}
```

### 9.1. Errors

General errors (may be returned by all queries):

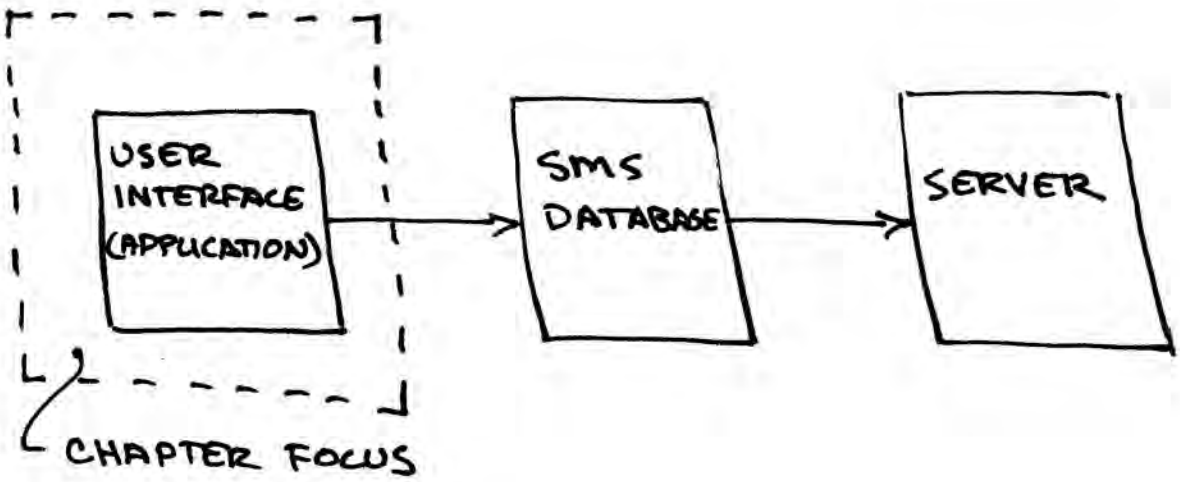
SMS_SUCCESS - Query completed successfully
SMS_PERM - Insufficient permission to perform requested database access

Query specific errors:

SMS_ACL - No such access control list
SMS_ARGS - Insufficient number of arguments
SMS_CLUSTER - Unknown cluster
SMS_DATE - Invalid date
SMS_DEVICE - No such device
SMS_EXISTS - Record already exists
SMS_FILESYS - Named file system does not exist
SMS_FILESYS_ACCESS - invalid filesys access
SMS_FILESYS_EXISTS - Named file system already exists
SMS_LIST - No such list
SMS_MACHINE - Unknown machine
SMS_NFS - specified directory not exported
SMS_NFSPHYS - Machine/device pair not in nfsphys
SMS_NOT_UNIQUE - Arguments not unique
SMS_PRINTER - Unknown printer
SMS_QUEUE - Unknown queue
SMS_RVD - no such rvd
SMS_SERVICE - Unknown service
SMS_STRING - Unknown string
SMS_TYPE - Invalid type
SMS_USER - No such user

## 10. Specialized Management Tools - User Interface

SMS 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 SMS's application program library. These tools provide the fundamental administrative use of SMS. *For each system service there is an administrative interface.* In this section, each interface discussed will provide information on the services it effects. The following diagram:




The user interface is indicated by the left hand side of this diagram, the component surrounded by the dotted line.

In response to complaints about the user interface of current database maintenance tools such as madm, gadm, and (to a lesser extent) register, the SMS 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:

- ATTACH_MAINT - Associate information to filesystems
- CHFN - change finger information
- CHPOBOX - change forwarding post office
- CHSH - change default shell

- CLUSTER_MAINT - machine and cluster management
- DB_MAINT - Database integrity check.
- DCM_MAINT - Update DCM table entries, including service / machine mapping.
- LIST_MAINT - List administration (madm & gadm)
- PRINTER_MAINT - MDQS printer maintenance
- REG_TAPE - Registrar's tape entry program
- RVD_MAINT - Create/update RVD server
- SERVICE_MAINT - Services management
- USER_MAINT - User information including NFS and PO information
- USERREG - New user registration.

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

PROGRAM NAME: ATTACH_MAINT - Associate information to filesystems.

DESCRIPTION: This program will allow the administrator to associate a user, a project, or a course to a filesystem, whether it is an RVD pack, or an NFS-exported filesystem. Right now, each workstation has the file /etc/rvdtab which is manually updated by the operations staff. By associating a course to a filesystem in the SMS database, Hesiod, the Athena name server, will be able to find arbitrary filesystem information, and the system will no longer require /etc/rvdtab.

This program will maintain the database tables ufs (fields userid, filesys), and filesys (label, type, machine_id, name, mount, access)

PRE-DEFINED QUERIES USED:

- update_user_home - for user to filesys mapping
- add_alias - user, project, or course

manipulates the following fields: (home) - USERS relation. (name, type, translation) ALIAS relation.

- add_filesys
- update_filesys
- delete_filesys

manipulates the following fields: (label, type, machine_id, name, mount, access) FILESYS relation

SUPPORTED SERVICE(S):

- Hesiod - filesys.db

END USERS: Administrators.

A SESSION USING ATTACH MAINT:

```
%attachmaint

                    Attach/Filesystem Maintenance
 1. (filesystem  ) Filesystem Work.
 2. (update      ) Update User's Home.
 3. (+           ) Associate with a Filesystem.
 4. (-           ) Disassociate from a Filesystem.
 5. (check       ) Check An Association.
 6. (toggle      ) Toggle Verbosity of Delete.
 7. (help        ) Help ...
 r. (return      ) Return to previous menu.
 q. (quit        ) Quit.
Command:1

                      Filesystem Maintenance
 1. (get         ) Get Filesystem Name.
 2. (add         ) Add Filesystem.
 3. (change      ) Update Filesystem.
 4. (delete      ) Delete Filesystem.
 r. (return      ) Return to previous menu.
 q. (quit        ) Quit.
Command:

Command:2
Update User's Home
Login name:
Home Filesys:

Command: 3
Associate with a Filesystem
name (user/course/project):
Filesystem Name:

Command: 4
Disassociate from a Filesystem
name (user/course/project):
Filesystem Name:

Command: 5
Check An Association
name (user/group/course):

Command: 6
Toggle Verbosity of Delete
Delete functions will first confirm

Command:q
%
```

PROGRAM NAME: CHFN - Finger Information.

DESCRIPTION: This program allows users to change their finger information.

The functionality of the old finger should not be changed. A new program (athenafinger) should be provided that will ask Hesiod for a user's information, including the location of their home directory. If the .plan and .project files appear there, and are world-accessible, they will be printed out. So far, there are no changes in functionality -- but the difference is that no machine need be specified. The user no longer has to know (or guess at) which of the many possible machines his target might be logged into.

PRE-DEFINED QUERIES USED:

- get_finger_by_login
- get_finger_by_first_last
- update_finger_by_login

manipulates the following fields: (fullname, nickname, home_address, home_phone, office_phone, department, year)

SUPPORTED SERVICE(S):

- User Community - finger

END USER: All.

A SESSION USING CHFN:

```
% chfn

Changing finger information for pjlevine.
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 - Add / change home mail host. (This program is the new chhome.)

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. Basically there are two options:

Usage: chpobox [-d | a address] [-u user]

where:
-d deletes a currently used mail address
-a adds a mail address

Chpobox without any option will return the current state of the user's mail addresses (see below).

PRE-DEFINED QUERIES USED:

- get_po_box
- add_po_box
- delete_po_box

manipulates the following fields: (login, type, machine, box) POBOX relation

SUPPORTED SERVICE(S):

- User Community - forward mail

END USERS: All.

A SESSION USING CHPOBOX:

```
% chpobox
Current mail address(es) for pjlevine is/are:
type: LOCAL
address: pjlevine@menelaus.mit.edu
%
%chpobox -a pjlevine@menelaus.mit.edu  ;adds a new mail address
%chpobox -d pjlevine@menelaus.mit.edu  ;deletes a mail address
%
```

PROGRAM NAME: CHSH - Default shell.

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

PRE-DEFINED QUERIES USED:

SUPPORTED SERVICE(S):

- Hesiod - passwd.db

END USERS: All

A SESSION USING CHSH:

```
% chsh

Changing login shell for pjlevine.
Current shell for pjlevine is /bin/csh
New shell: /bin/csh
Changing shell to /bin/csh
%
```

PROGRAM NAME: CLUSTER_MAINT - This program manages machines and clusters.

DESCRIPTION: Handles the relationships of various machines and clusters.

PRE-DEFINED QUERIES USED:

MACHINE:

- get_machine_by_name
- add_machine
- update_machine
- delete_machine

manipulates the following fields:

(name, machine_id, type, model, status, serial, ethernet, sys_type) MACHINE relation

CLUSTERS:

- get_cluster_info
- add_cluster
- update_cluster
- delete_cluster
- get_machine_to_cluster_map
- add_machine_to_cluster
- delete_machine_from_cluster

manipulates the following fields:

(name, description, location, cluster_id) CLUSTER relation

SERVICE CLUSTERS

- get_all_service_clusters
- add_service_clusters
- delete_service_clusters

manipulates the following fields:

(cluster_id, serv_label, service_cluster) .

SUPPORTED SERVICE(S):

- Hesiod - cluster.db

END USERS: Administrator. Staff.

A SESSION WITH CLUSTER_MAINT:

[See next page]

```
                    Cluster Maintenence
 1. (machine    ) Work on Machines.
 2. (cluster    ) Work on Clusters.
 3. (service    ) Service Clusters.
 4. (map        ) Machine to Cluster Mapping.
 5. (toggle     ) Toggle Delete Confirmation.
 6. (list       ) List All Valid Queries.
 7. (help       ) Help.
 r. (return     ) Return to previous menu.
 q. (quit       ) Quit.
Command:

Command:1
                    Machine Maintenence
 1. (get        ) Get Machine by Name.
 2. (add        ) Add Machine.
 3. (update     ) Update Machine.
 4. (delete     ) Delete Machine.
 5. (put        ) Add Machine to Cluster.
 6. (remove     ) Delete Machine from Cluster.
 7. (map        ) Machine to Cluster Mapping.
 r. (return     ) Return to previous menu.
 q. (quit       ) Quit.
Command:

Command:2
                    Cluster Information
 1. (get        ) Get Cluster Information.
 2. (delete     ) Delete Cluster.
 3. (add        ) Add a Cluster.
 4. (update     ) Update Cluster Info.
 5. (map        ) Machine to Cluster Mapping.
 r. (return     ) Return to previous menu.
 q. (quit       ) Quit.
Command:

Command:3
                    Service Cluster Maintenence
 1. (get        ) Get All Service Clusters.
 2. (add        ) Add a Service Cluster.
 3. (delete     ) Delete a Service from a Cluster.
 4. (update     ) Update Service Cluster.
 r. (return     ) Return to previous menu.
 q. (quit       ) Quit.
Command:

Command:4
Machine to Cluster Mapping
Machine Name:
Cluster Name:

Command:5
Toggle Delete Confirmation
Delete functions will first confirm

Command:q
%
```

PROGRAM NAME: DB_MAINT - Data base integrity checker/intersective constrainer.

DESCRIPTION - This program systematically checks the database for user date expirations, INACTIVE status fields, and does a complete integrity check of the lists when a user or list is deleted. The program will also alert the system administrator if a user's expiration time has been reached and if there is an inconsistency with the database. This is the program which provides interactive constraint capability. The program is invoked automatically every 24 hours.

PRE-DEFINED QUERIES USED:

- To be determined

SUPPORTED SERVICE(S):

- SMS

END USER: Administrator. Cron (automatically invoked).

PROGRAM NAME: DCM_MAINT - Add/update DCM table entries.

DESCRIPTION: This program allows the administrator to "check in" server description files, and associated information. The dcm reads the server table and uses the information to update the system. It is through this program, therefore, that update interval, target path, and files used, to name a few, are entered. This program is a menu-driven program and is invoked with the command dcm_maint.

PRE-DEFINED QUERIES USED:

- get_server_info
- add_server
- update_server

manipulates the following fields: (update_interval, target_dir) SERVERS relation

SUPPORTED SERVICE(S):

- SMS Hesiod - sloc.db

END USERS: Administrator.

A SESSION WITH DCM_MAINT:

Data Control Manager Maintenance
1. (change_host_info) Modify host-specific info for a server.
2. (add_host_info) Create new entry for the table.
3. (delete_host_info) Remove an entry from the table.
4. (list_host_info) List entries by host or service.
5. (change_service) Modify info for an existing service.
6. (add_service ) Create new service.
7. (delete_service) Remove an entry from the table.
8. (list_service) List services.
r. (return ) Return to previous menu.
q. (quit ) Quit.
Command:

```
Command:1
    Change table entry [host: ATHENA-PO-1.MIT.EDU, service: ]
1. (show      ) Show values of entry.
2. (last_time  ) Change the last_time field.
3. (success    ) Change the success field.
4. (override   ) Change the override field.
5. (enable     ) Change the enable field.
6. (value1     ) Change the value1 field.
7. (value2     ) Change the value2 field.
r. (return     ) Return to previous menu.
q. (quit      ) Quit.
Command:

Command: 2
Create new entry for the table
Which host[]:
Which service[]:

Command: 3
Remove an entry from the table
Which host[]:
Which service[]:

Command:4
Modify info for an existing service
Which service[]:

Command: 5
Modify info for an existing service
Which service[]:

    Modify existing service
1. (show      ) Show fields of service.
2. (interval   ) Change the interval field.
3. (target_path ) Change the target_path field.
4. (instructions) Change the instructions field.
5. (dfgen      ) Change the dfgen field.
r. (return     ) Return to previous menu.
q. (quit      ) Quit.
Command:

Command: 6
Create new service
Which service[]:
service_name  interval     target_path   instructions  dfgen

Add service to database? [y]:
Service created

Command: 7
Remove an entry from the table
Which service[]:
No entry found for service .

Command: 8
List services
Which service[]:
service_name  interval     target_path   instructions  dfgen
hesiod                /dev/null     /dev/null
pop                   /dev/null     /dev/null
```

PROGRAM NAME: LIST_MAINT - List Administration

DESCRIPTION: This program handles mailing lists, and group lists. The general approach to this program is to identify the list type which will need updating. The program is, among other things, a combination of the current madm and gadm programs.

The notion of a list in the SMS world is an entity which contains members. These members are not limited to users, and, in fact, can be machines, users, mail addresses, and even other lists. Additionally, a list has an owning access control list, acl. For a user who wishes to have himself own a list the process is simple. Upon registering, a user will be placed on a list, where the list's name is the user's name and the list's contents are the user himself. This will allow that user to "own", or more precisely, associate an acl, to a new list. In listmaint, where a user creates lists, the entry "administrator ACL" must be another list. Based on the above, however, this entry may be the user's name.

A list also may be associated with many functionalities. For example, a list may be a mailing list and a group list. This allows the same list to be used differently. Listmaint provides the mechanism to associated a list with a given type of function. Because of this, however, the user must create a list first. This seems obvious in discussion, although a deficiency the the program does not intuitively presume. The sample session below highlights this case. The command "listmaint" invokes this program.

PRE-DEFINED QUERIES USED:

- get_all_mail_lists
- add_mail_list
- delete_mail_list
- get_list_info
- add_list
- update_list
- delete_list
- add_member_to_list
- delete_member_from_list
- get_members_of_list
- get_all_groups
- add_group
- delete_group

manipulates the following fields: (name, type, list_id, flags, description, expdate, modtime) LIST relation

(list_id, member_type, member_id) MEMBERS relation

(member_id, string) STRINGS relation

(capability, list_id) CAPACLS relation

(list_id) GROUPS relation

END USERS: All.

SUPPORTED SERVICE(S):

- /usr/lib/aliases

A SESSION WITH LISTMAINT:

```
%listmaint
                    List Maintenance Functions
1. (list_menu   ) Manage List Parameters.
2. (member_menu ) Manage Membership of Lists.
3. (group_menu  ) Manage Groups.
4. (mail_menu   ) Manage Mailing Lists.
r. (return      ) Return to previous menu.
q. (quit        ) Quit.
Command:1

                    List Creation and Deletion
1. (get_list_info) Get information about a list.
2. (add_list    ) Create a new list.
3. (delete_list ) Delete a list.
4. (update_list ) Update characteristics of a list.
r. (return      ) Return to previous menu.
q. (quit        ) Quit.
Command:


Command:2
                    Change membership
1. (list_members) List all members of a list.
2. (list_lists  ) List all lists to which a given (user,list,string)belongs.
3. (add_member  ) Add a new member to a list.
4. (delete_member) Delete a member from a list.
r. (return      ) Quit.n to previous menu.
Command:

Command:3
                    Groups
1. (list_groups ) List all groups.
2. (add_group   ) Add a group.
3. (delete_group) Delete a group.
r. (return      ) Return to previous menu.
q. (quit        ) Quit.
Command:

Command:4
                    Mailing Lists
1. (list_mail   ) List all visible mailing lists.
2. (add_mail    ) Add a mailing list.
3. (delete_mail ) Delete a mailing list.
r. (return      ) Return to previous menu.
q. (quit        ) Quit.
Command:

Command:q
%
```

PROGRAM NAME: PRINTER_MAINT - Printer maintenance.

DESCRIPTION: This program will manage printers and queues for the Multiple Device Queueing System (MDQS). It handles associations between printers and machines, printers and abilities (such as windowdumps, postscript, etc.), machines, printers, and queues, and printers and printcaps.

PRE-DEFINED QUERIES USED:

- get_all_printers
- get_printer_info
- add_printer
- update_printer
- delete_printer
- get_printer_ability
- add_printer_ability
- delete_printer_ability
- get_all_queues
- get_queue_info
- add_queue
- update_queue
- delete_queue
- add_printer_to_queue
- get_qdev
- add_qdev
- update_qdev
- delete_qdev
- add_queue_device_map
- update_queue_device_map
- delete_queue_device_map
- get_all_printcap
- get_printcap
- add_printcap
- update_printcap
- delete_printcap

updates the following fields:

(name, printer_id, type, machine_id) PRINTER relation

(printer_id, ability) PRABILITY relation

(name, queue_id, machine_id, ability, status) QUEUE relation

(printer_id, queue_id) PQM relation

(machine_id, qdev_id, name, device, status) QDEV relation

(machine_id, queue_id, device_id, server) QDM relation

SUPPORTED SERVICE(S):

- Hesiod - printers.db

END USERS: Administrator. Staff.

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 SMS database.

PRE-DEFINED QUERIES USED:

- update_user
- update_user_status

manipulates the following fields: (status, expdate) USERS relation.

SUPPORTED SERVICE(S):

- SMS

END USERS: Administrator.

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.

- PROGRAM NAME: RVD_MAINT - Create/update an RVD server.

  DESCRIPTION: This administrative program will allow for the master copy of rvddb's to be updated and created. The DCM will distribute the RVD information automatically to the servers requiring RVD data. Presently, the system administrator keeps an up-to-date file of RVD data and then copies the data to the RVD server.

  This program will handle three "tables" in the SMS database: rvdsrv (machine id, operations pwd, admin pwd, shutdown pwd)[6], rvdphys (machine id, device, size create-time, modify-time) , and rvdvirt (machine id, physical device, name, pack id, owner, rocap, excap, shcap, modes, offset, blocks, ownhost, create-time, modify-time)

  It will become the responsibility of SMS to maintain the present file /site/rvd/rvddb. The DCM will automatically load the RVD server with information.

  PRE_DEFINED QUERIES USED:
  - get_rvd_server
  - add_rvd_server
  - delete_rvd_server

  manipulates the following fields: (machine id, operations pwd, admin pwd, shutdown pwd) RVDSRV relation
  - get_rvd_physical
  - add_rvd_physical
  - delete_rvd_physical

  manipulates the following fields: (machine id, device, size create-time, modify-time) RVDPHYS relation
  - get_rvd_virtual
  - add_rvd_virtual
  - delete_rvd_virtual
  - update_rvd_virtual

  manipulates the following fields: (machine id, physical device, name, pack id, owner, rocap, excap, shcap, modes, offset, blocks, ownhost, create-time, modify-time) RVDVIRT relation

  SUPPORTED SERVICE(S):
  - RVD - rvddb.

  END USERS: Administrators.

  A SESSION WITH RVD_MAINT:

[6]See the section on database structure for explanations of these fields.

```
%rvd_maint

                    SMS RVD Maintenance
1. (modvd     ) Modify a virtual disk.
2. (addvd     ) Add a new virtual disk.
3. (addpd     ) Add a new physical disk.
4. (rmvd      ) Delete a virtual disk.
5. (rmpd      ) Delete a virtual disk.
6. (exchvd    ) Exchange virtual disk names.
7. (lookvd    ) Look up virtual disk names.
8. (lookpd    ) Look up physical disk names.
9. (rvdhelp   ) Get help with RVD commands.
r. (return    ) Return to previous menu.
q. (quit      ) Quit.
Command:
```

•

PROGRAM NAME: SERVICE_MAINT - Services management.

DESCRIPTION: This program manages what today is /etc/services: it informs Hesiod of the association between services and reserved ports.

PRE-DEFINED QUERIES USED:

- get_all_services
- add_service
- delete_service
- update_service
- get_all_service_aliases
- add_service_alias
- delete_service_alias

manipulates the following fields:

(service, protocol, port) SERVICES relation

(name, type, trans) ALIAS relation

SUPPORTED SERVICE(S):

- Hesiod - service.db

END USERS: Administrator. Staff

- A SESSION WITH SERVICE_MAINT:

```
%servermaint

                    Server Maintenence
1. (get      ) Services Info.
2. (add      ) Add Service.
3. (update   ) Update Service.
4. (delete   ) Delete Service.
5. (toggle   ) Toggle Verbosity of Delete.
6. (list     ) List All Valid Queries.
r. (return   ) Return to previous menu.
q. (quit     ) Quit.
Command:

Command: 1
Services Info
Service Name:
```

[Continued on next page]

- Command: 2
  Add Service
  Name:
  Protocol:
  Port:

  Command: 3
  Update Service
  Name:
  Protocol:
  Port:

  Command: 4
  Delete Service
  Service Name:
  Protocol:

  Command: 5
  Toggle Verbosity of Deletey handle
  Delete function will first confirm

  Command:q
  %

•

PROGRAM NAME: USER_MAINT - Adding/changing user information, including NFS and post office information.

DESCRIPTION: Presently, there are two programs which the system administrator uses to register a new user: register and admin. These programs register the user and enter the private key information to Kerberos, respectively. The new application will provide these, and offer a third feature which allows the administrator to check the fields of the SMS database and verify that all of the database fields are correct. (Currently this is done by exiting the program and using INGRES query commands to verify data.)

PRE-DEFINED QUERIES USED

- get_user_by_login
- get_user_by_firstname
- get_user_by_lastname
- get_user_by_first_and_last
- add_user
- update_user_shell
- update_user_status
- update_user_home
- update_user

manipulates the following fields: (login, mit_id, first, last, mid_init, exp_date, shell, status, users_id, modtime, home) USER relation

This program will also allocate and change home directory storage space. It will allow the administrator to check storage allocation on a server and allocate or change a storage space for a user. The information will be held in the SMS database and will be passed to the name service. The allocation of a user's quota can be done automatically at register time, using the amount of quota already allocated on each server, and querying each server as to the amount of disk space actually available. There are other circumstances, such as the user's living group, that should be taken into consideration when assigning a server. (Trying to assign people to servers reasonably "near" them is an attempt to decrease the load on the network.)

For home directory allocation and change, the following predefined queries are used:

- get_nfs_quota
- add_nfs_quota
- update_nfs_quota

manipulates the following fields: (machine, login, quota) NFSQUOTA relation

- get_server_info
- add_server
- update_server

manipulates the following fields: (value) SERVERS relation

- get_value - for default PO allocation

manipulates the following field: (value) VALUE relation.

In the SERVERS relation, the *value* field represents the total currently allocated space (but not necessarily used). In the VALUE relation, the *value* field represents the default nfs quota for a user (used in new user home allocation). For example, if 20 users have been allocated to a machine and each has a filesys quota of 2 Meg, then the value field (SERVER relation) will be 40 Meg. If the server reports back that its free space is 80 Meg, then another 20 users can be given allocated space on this disk. As long as the free space minus the allocated space is greater or equal to the quota of the current allocation, the disk is OK to use. This mechanism will prevent over allocation of home directory storage space.
END USER: Administrator.

SUPPORTED SERVICE(S):

- Hesiod - passwd.db Kerberos

A SESSION USING USERMAINT:

SMS User Maintenance

1. (add ) Add new user.
2. (modify ) Modify user.
3. (chsh ) Change a user's login shell.
4. (chdir ) Change a user's home directory.
5. (chstat ) Change a user's status.
6. (chpw ) Change a user's password.
7. (show_login ) Show user entry by login name.
8. (show_last ) Show user entry by last name.
9. (show_first ) Show user entry by first name.
10. (show_full ) Show user by first and last names.

r. (return ) Return to previous menu.
q. (quit ) Quit.
Command:

- PROGRAM NAME: SMS_MAINT - Master SMS program.

  DESCRIPTION: This program can do anything that any of the above-described programs can do, but attendant with that ability is an increase in complexity unsuited for random users or faculty administrators. Given this program (and a listing in the appropriate database acls), there is nothing in the SMS database that you cannot view and update. It is intended for the one or two people whose main responsibility is the care and feeding of SMS.

  SUPPORTED SERVICES:
  - All.

  PRE-DEFINED QUERIES USED:

  ALL

  manipulates the following fields:

  ALL

  END USER: Administrator. God.

## 11. Addendum

The following information supplements the formal text. In general the addendum will provide critical information regarding programming application and use.

### 11.1. Application Programmers Library

The following is an application library used by consumers of the SMS database. The library reflects a direct mapping between its functions and the above listed pre-defined query handles.

To be furnished.

### 11.2. Catastrophic Failure Recovery Procedure

This section reviews, from an operational standpoint, the procedures necessary to bring SMS up after a catastrophic failure. Catastrophic failure is defined as any system crash which cannot recover on its own accord, or any automatic system recovery procedure which results in database inconsistency or operational failure of the SMS system.

To be furnished.

## References

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

[2] BCN, SPM, JIS, JHS, and friends.
*Whatever the Kerberos document is called*
Athena, DEC, Telecom, and Athena, 1987.