Moira Server Internals

1. Basic Operation

On startup, `main()` initializes the com_err library, gdb library, and ingres connection to the database. It sets up signal handlers for SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, and SIGHUP, opens the journal file, and finally has GDB open a listening connection. It then loops forever, processing GDB requests and timing out idle connections.

A semophore file, */etc/smsdown*, and the signals SIGUSR1 and SIGUSR2 control the availability of the database. If the semophore exists when the server is first started, or when the last active client connection is closed, then the database will be closed. This means that the Ingres backend exits and no queries will be answered. The contents of the semophore file are returned as the Message of the Day during this time. If the semophore is no longer there when a new client connects to the server, the database will be opened again. Rather than wait until the last client disconnects or first client connects to close and open the database, SIGUSR1 or SIGUSR2 may be sent to the server to cause this to happen as soon as the current query finishes executing, irregardless of how many clients are currently connected. Closing and opening the database also flushes the name to ID cache.

For each connection, the server creates a client structure. The client structure contains connection state, authentication information, and the arguments and return value of the current query. The operations supported are **noop, auth, shutdown, query,** and **access**. In `server/mr_scall.c(do_call)`, these operations are decoded and dispatched.

The authenticate is taken care of in `server/mr_sauth.c`. This requires the client to send a Kerberos ticket for **moira**.*servername@localrealm*. It will extract the client's principal name, and use that for later authentication purposes. To speed up later access checking, it will also resolve the principal name to the users_id of that person in the Moira database. If this operation was invoked as version 2 of the protocol, it also expects to be passed the name of the client program.

The shutdown operation is no longer supported, and will log an error if attempted.

When a query is requested, `server/mr_scall.c(do_retr)` first marshals up the arguments, and determines the protocol version number. **_list_users** is a special query that is intercepted at this point. It then calls `server/qrtn.qc(mr_process_query)` to do the actual work. `server/mr_scall.c(retr_callback)` will be called for each tuple to be returned. This routines takes care of formatting them with GDB and returning them. An access check is done in a similar manner.

1.1. Query processing

`mr_process_query` is the heart of the server. First it catches the special queries **_list_queries** and **_help**. Then it looks up the requested query in the query dispatch table. Note that the protocol version number is needed here, since there is a separate dispatch table for each version. Next if this query will affect the database (i.e. is an append, update, or delete; not a retrieve), then an Ingres transaction is started. Now the query is validated with `mr_verify_query`. This takes care of access checking, argument validation and row validation. If the validation passes, it will perform any specified setup routine. Now it saves the "before" copy of the data for use in incremental updates. Next the actual query is performed, followed by the followup routine if specified and the statistics are updated. Now the "after" copy of the data is saved for possible incremental updates. If we modified the database and everything was successful, we commit the transaction, fire off any necessary incremental updates and log it all in the journal.

An access operation is like an actual query, except that it only does the query validation.

Incremental updates are handled as a queue of pairs of "before" and "after" argument vectors. Each query saves these vectors as it executes, and if the transaction is committed, then incremental_update() is called to fire off any updates. A given database change only starts an incremental update if the "incremental" table in the database indicates that a service has interest in that table and the program BIN_DIR/*service*.**incr** exists.

1.2. Logging

The server itself logs to standard output. However, it is normally started with the `startmoira` program, which will direct the logs into `/u1/sms/moira.log`.

2. Source Organization

Most of the server sources are in the server[1] subdirectory. The **mr_*.c** files are the bulk of the server source:

mr_main.c — Initialization and top-level dispatch routines.

mr_scall.c — Deals with most of the protocol processing, including dealing with GDB format of the arguments and return values.

mr_sauth.c — Performs protocol authenticate operation.

mr_shutdown.c — Catches the no-longer-supported shutdown operation.

mr_sbrk.c, meinitlst.c — These handle a deficiency in the memory allocator used by the Ingres library. **meinitlst.c** is reverse engineered from the Ingres library **~rtingres/lib/compatlib(meinitlst.o** to support better malloc and free routines.

mr_smalloc.c — This is a new version of malloc which includes scribble checking. It is derived from the gnuemacs malloc.

mr_srvdata.c — Contains declarations of statically allocated data.

mr_util.c — Various utility routines needed within the server.

mr_glue.c — Routines to bypass the protocol and glue a client into the back half of the server.

The **q*.qc** and **q*.c** files implement the queries, and should be the only part of the server that depends on RTI Ingres.

qrtn.qc — This has the routines to interpret the query dispatch table and execute queries. It also handles opening and closing the database, and catching Ingres errors.

qsubs.c — This has subroutines needed by the query software which are in pure C-code without embedded equel.

[1]All paths are given as relative to the root of the Moira source tree.

queries.c	This is the dispatch table for version 1 of the protocol.
queries2.c	This is the dispatch table for version 2 of the protocol.
qsupport.qc	These are support routines used in the dispatch table. The access checks, setups, followups, special queries, and validation routines are all in this (too large) file.
qoldsup.qc	These are old support routines only used in version 1 of the protocol.
increment.qc	Routines to handle incremental updates. This touches the database so that it can capture the before and after images of the data.

The server also links against the moira library for some common routines, the gdb library for the transport layer, the zephyr library for error notices, the kerberos and DES libraries for authentication, the com_err library for error reporting and logging, and the math library because it feels like it.

3. Query Dispatch Table

The query table consists of an array of records, one for each query.

```
/* Query Definition Structure */
```
[2]

```
struct query
{
    char *name;                  /* query name */
    char *shortname;             /* abbreviated query name */
    enum query_type type;        /* query type */
    char *rvar;                  /* range variable */
    char *rtable;                /* range table */
    char *tlist;                 /* target list */
    char **fields;               /* input and output field names */
    int  vcnt;                   /* variable count */
    char *qual;                  /* format string for "where" clause */
    int  argc;                   /* number of args for qualifier */
    struct validate *validate;   /* validation support */
};
```

The *name* is the full name of the query, as in "get_user_by_login".

The *shortname* is a 4 character abbreviation of the name. If is important that this be exactly 4 characters long, and must be unique among all of the queries in the table.

The query_type is one of:

```
/* Query Types */
enum query_type {RETRIEVE, UPDATE, APPEND, DELETE, SPECIAL};
```

The *rvar* specifies the range variable to be used in this query, in a range statement in conjunction with rtable. If the query is implemented with a special C/equel routine, this field should be left blank.

The *rtable* specifies the table to be operated on in this query. A range statement will be executed, for example `range of u is users` if *rvar* = "u" and *rtable* = "users". *rtable* is also used to know which entry in tblstats to update. For special queries, this field may be left blank. If this is filled in, the statistics will be updated.

---

[2]All code taken from server/query.h

tlist is the target list to be used in the query about to be executed. This specifies the variables to be retrieved or stored during the query. For retrievals, it should have "%c = ..." for each string to be retrieved. For update and appends, "... = %c" to store a string, or "... = int4(%c)" to parse and store an 4 byte integer. Queries implemented with special routines can have this left blank.

fields input and output field names and displayed by the *_help* query. Help is the only thing they are used for.

The variable count (*vcnt*) is the total number of arguments for an append, update or delete, or the number of return values for a retrieve.

qual is the format string for the qualifier. This will be used in the where clause of the Ingres query, and should have %s and/or %d in it to insert values from the query arguments.

argc is the number of args that will be used in the qualifier, as described above.

Validation support: if *validate* is non-null, it points to a validation structure:

```
/* Query Validation Structure */
struct validate
{
    /* object validation (e.g., user, filesys, type) */
    struct valobj *valobj;      /* object validation descriptors */
    int objcnt;                 /* size of array */
    /* row validation: retrieve (exists=any(rvar.field where qual)) */
    char *field;                /* field to check for */
    char *qual;                 /* format string for "where" clause */
    int  argc;                  /* number of args used in qual */
    /* values field containing current max object id */
    char *object_id;
    /* routine to verify access permission on objects */
    int (*acs_rtn)();
    /* pre-processing routine (var setup only) */
    int (*pre_rtn)();
    /* post-processing routine */
    int (*post_rtn)();
};
```

valobj is a pointer to an array of validation objects. There will be one entry in this array for each argument that needs to be validated. This pointer may be NULL if *objcnt* is zero.

objcnt is the count of entries in the valobj array.

field, *qual*, and *argc* are used together for row validation. *field* names the field to look for, *qual* is a format string for the "where" clause of the query that does the row validation. It is like the *qual* field in the main dispatch table. *argc* is the count of arguments used in the qualifier.

object_id is the name of the ID field in the record being updated. It has two purposes: it is used to allocate new ID's automatically for appends, and it is used to find the proper row to set the modtime information after an append or update.

The *acs_rtn* is the access routine. This is called if the client is not on the query ACL to determine if they will be allowed to perform the query. It will be called as *acs_rtn*(q, argv, cl), where q is the query structure for this query, argv is the arguments for this query, and cl is the client structure for the client requesting the query.

The *pre_rtn* is the query setup routine. If non-NULL, this will be called after the validation and before performing the actual query. It will be called with the same arguments as the access routine.

The *post_rtn* is the query followup routine or the actual query routine for special queries. If a followup routine, this will be called after performing the actual query, as *post_rtn*(). If it is a special query, it will be called as *post_rtn*().

Table of Contents