

AMGA USER'S AND ADMINISTRATOR'S MANUAL

B. Koblitz, N. Santos

July 30, 2007

Abstract

This is the manual for users and administrators of AMGA. It intends to give an overview on the installation of the client and server packages as well as the client-api packages. Examples of the usage of the command line clients and the client APIs are given.

1 Overview

AMGA is a metadata service for the Grid. In a more general way this is a database access service for Grid applications which allows user jobs running on the Grid to access databases by providing a Grid style authentication as well as an opaque layer which hides the differences of the different underlying database systems from the user. To achieve this, AMGA is a service sitting between the RDBMS and the user's client application.

In addition to this database translation layer, AMGA intends to solve another problem database services face on the Grid which is latencies. AMGA intends to provide a replication layer which makes databases locally available to user jobs and replicate the changes between the different participating databases. A simple implementation based on PostgreSQL asynchronous replication is already working.

All examples given in this manual as well as additional documentation in particular the reference manuals of all APIs are given on the projects home page at <http://project-arda-dev.web.cern.ch/project-arda-dev/metadata/>

2 Installation

The server and the clients (C++, Java and Python) are provided as RPMs packages. RPMs are currently only supported for CERN Scientific Linux or RedHat Enterprise Linux. For users running other operating systems (including Windows and other unix flavours), the Java client is also provided as a platform-independent package (a tar ball), including the Java API, a command line client, some examples and the documentation. You can find the packages in the `download` directory. Download the latest version. As of writing this it is 1.1.0.

You can install the `glite.amga.client` and the `python` and `java api` packages independently of the server package, however the server packages depend on the client package.

2.1 Client installation

To install the command line client and C++ api, you will only need to download the `glite-amga-cli-1.1.0.i386.rpm` itself. By default the package will be installed into `/opt/glite`, so you will need to have write permission for this directory. Install the package via

```
rpm -i glite-amga-cli-1.1.0.i386.rpm
```

Copy the `/opt/glite/etc/mdclient.config` client configuration file into the directory from which you intend to work or into `~/mdclient.config` and customize it according to the instructions in **Configuration of the C++ and Java command line clients**(p.??).

If the `amga` client was built against the `editline` library instead of the standard `unix readline` library, you will need to also install that RPM.

To install the RPM with Java API, download and install the file named `glite-amga-api-java-0.X.Y.rpm`. This RPM is architecture-independent and will install in any Unix platform that supports RPM packages. This RPM contains only the Java API (no command line client or documentation). It installs the file `glite-amga-api-java.jar` in `/opt/glite/share/java/`. To use the Java API, this file must be included on the classpath.

The tar ball with the Java Client API does not need any special installation procedure, apart from unpacking it. It provides both the documentation, the Java API, and two command line utilities similar to the ones installed by the C++ RPM, with the advantage of running in any platform with a Java virtual machine implementation (must be at least Java 1.4 compliant). These command line tools are:

- **mdjavaclient** - an interactive command line shell with the server.
- **mdjavacli** - an utility to submit a single command to the server. For use in shell scripts.

The tar ball includes four scripts to start these command line tools: `mdjavaclient.sh/mdjavaclient.sh` for Linux and `mdjavacli.sh/mdjavacli.sh` for Windows. The interactive command line client can also be started directly by executing the class `arda.md.javaclient.ConsoleClient`.

There is also a Python Client API module available as an RPM package (`glite.amga.api-python-1.1.0-1.noarch.rpm`), which will install the modules under `/opt/glite/python2.2/site-packages/amga`. Alternatively you can use the Python2.3 RPM which works nicely on Debian Sarge. If you prefer a custom installation you can download the source tarball `glite.amga.api-python-1.1.0.tar.gz` and install it via the Python installation mechanism (if you don't know how this works, `./setup.py -help` should get you started).

2.2 Server installation on SLC3

The **AMGA**(p.??) server depends on the C++ client package as well as some external dependencies which are provided by the gLite team for apt-get based installation. See <http://glite.web.cern.ch/glite/packages/APT.asp>

The following packages are necessary to use **AMGA**(p.??) and must be installed first: UnixODBC, libxml2 and Boost-lib. UnixODBC and libxml2 are standard SLC3 packages and can be also found in the usual CERN repositories. You can get the packages via

```
apt-get install unixODBC
apt-get install libxml2
apt-get install boost
```

If **AMGA**(p.??) was compiled against the Globus environment and not the SLC3 environment in order to use the Globus versions of several SLC3 system libraries like openssl, then you will also need to install the Globus RPM. The gLite project currently only provides versions of **AMGA**(p.??) which depend on Globus, the RPMs in the `download` directory work without.

Now install the server RPMs for **AMGA**(p.??) (the server needs the client):

```
su
wget http://amga.web.cern.ch/amga/downloads/glite-amga-server-1.2.2-1.SLC3.i386.rpm http://amga.web.cern.ch/amga/downloads/glite-
rpm -i glite-amga-cli-1.2.2-1.SLC3.i386.rpm glite-amga-server-1.2.2-1.SLC3.i386.rpm
```

You also need a database and the appropriate ODBC driver. **AMGA**(p.??) currently supports 4 different database backends via ODBC drivers. You will need to get at least one of the currently supported 4 database backends installed, including their ODBC driver. You have the choice among PostgreSQL, MySQL, Oracle and SQLite. The default is PostgreSQL and this database should be set-up correctly when installing the **AMGA**(p.??) RPM if you have PostgreSQL and its ODBC driver installed:

```
apt-get install rh-postgresql-server rh-postgresql
```

If you want to use a different database or you want to have a different setup of the ODBC driver, have a look at the more detailed instruction in the **Installation from Source**(p.??) .

Note: For **AMGA**(p.??) to work properly, you need at least version 08.01.0200 of the PostgreSQL ODBC driver. On SLC3 the driver is too old. You can download and install it yourself from `source`. Alternatively we

provide a binary package `amga-odb.tar.gz` which you can download [here](#). Just run the `INSTALL.sh` script after unpacking.

You will need to get at least one of these 4 database backends in order to use the service. ODBC should have been installed correctly by the **AMGA**(p. ??) rpm, if you want to use a standalone installation using PostgreSQL. If you don't know about ODBC, some more hints can be found in the **Installation from Source**(p. ??) or on a general page on ODBC like <http://www.unixodbc.org/doc/User-Manual/> the Unix ODBC User Manual .

If you want to use the PostgreSQL default installation, you can find the `init-arda-psqldb.sh` script in the `download` direcorey which should create a database user and set up the database access and initial tables automatically for PostgreSQL. This will only work on an RHEL3 default installation.

If you want to do the setup manually, you first have to create a DB user, make sure he can connect via a TCP/IP connection (even locally this is required since ODBC does not work via a Unix Domain socket) and then set up an ODBC data source. The steps required are:

```
su root
#Initialize DB configuration:
/etc/init.d/rhdb start
/etc/init.d/rhdb stop

# Uncomment out the line and set the parameter to true
#  tcpip_socket = true
# in /var/lib/pgsql/data/postgresql.conf

# Add the following lines to /var/lib/pgsql/data/pg_hba.conf:
local  all             postgres          sameuser
host   metadata arda   127.0.0.1 255.255.255.255  trust
local  metadata arda          trust

/etc/init.d/rhdb restart
su postgres
createuser arda # Create DB user arda, allow him new DBs
createdb -Oarda metadata # New DB metadata owner is arda
createlang plpgsql # Allow stored procedures
```

The ODBC data source is created by appending the following lines to `/etc/odbc.ini`:

```
[PSQL]
Description      = AMGA metadata catalogue database
Driver           = PostgreSQL
Trace            = No
TraceFil         = /tmp/metadata/odbc.log
Database        = metadata
Servername       = localhost
Port             = 5432
ReadOnly         = No
#Address         = localhost:5432
#User            = arda
```

The Driver needs to name a valid driver description in the `/etc/odbcinst.ini` file:

```
[PostgreSQL]
Description      = PostgreSQL ODBC driver for Linux
Driver           = /usr/local/lib/psqlodbc.so
FileUsage        = 1
```

The ODBC driver in the SLC3 distribution has several bugs to work correctly with **AMGA**(p. ??). Either download the `source code` or our `binary package` and install them.

This setup should be done automatically correctly when installing the `unixODBC` or `postgresql-odbc` RPMs on other distributions. If you want to check your ODBC installation, you can use the `DataManagerII` application which can be found in the `unixODBC-kde` package on SLC3.

Now you can initialise the database using the `createInitialXXX.sql` scripts (where `XXX` has to be replaced with the DB of choice which you will find in `/opt/glite/share/doc/glite-amga-server-1.1.0/` after the installation of the server RPM:

```
For PostgreSQL:
psql -Uarda metadata < /opt/glite/share/doc/glite-amga-server-1.2.2/createInitialPG.sql
```

Finally activate this data source and configure the **AMGA**(p.??) server in the `amgad.config` in `/opt/glite/etc/` file as described in **Configuring the AMGA Server and the Replicatin Daemon**(p.??) .

You should now be able to start the service and verify whether it is running by doing

```
/etc/init.d/mdservice start
tail /var/opt/glite/log/amgad.log
```

If the server is complaining about a missing boost-library (starting `amgad: /opt/glite/bin/amgad: error while loading shared libraries: libboost_thread-gcc-mt-1_32.so.1.32.0: cannot open shared object file: No such file or directory, create one with:`

```
su
cd /usr/lib
ln -s libboost_thread.so.1.32.0 libboost_thread-gcc-mt-1_32.so.1.32.0
```

This is a small inconsistency between SLC3 and RHEL3 (the rpm is in fact compiled on RHEL3).

If you want to change the configuration of the metadata server, you should go on and read **Configuring the AMGA Server and the Replicatin Daemon**(p.??) .

3 Configuration of the C++ and Java command line clients

The **AMGA**(p.??) C++ command line clients as well as all other executables using the MDClient class of the C++ client package need to read an `mdclient.config` file for their configuration. This configuration file is searched first in the current directory, then as `$HOME/.mdclient.config` and finally as `/etc/mdclient.config`. Only the port and server of the configuration file can be overridden on the command line of the clients:

```
mdclient [-p port] [hostname]
```

The following is an example configuration file:

```
# Connection options
Host = localhost
Port = 8822

# User settings
Login = kobnitz
PermissionMask = rw-
GroupMask = r--
Home = /
#Name=

# Security options
UseSSL = require
#AuthenticateWithCertificate = 1
#CertFile=/home/kobnitz/.globus/usercert.pem
#KeyFile=/home/kobnitz/.globus/userkey.pem
UseGridProxy = 1
#Password = secret
#VerifyServerCert = 1
#IgnoreCertificateNameMismatch = 0
# If server certificates are verified, local certificates need to be loaded:
TrustedCertDir = /etc/grid-security/certificates
# RequireDataEncryption = 0
```

The following options are supported:

- **Host:** The name of the host to connect to. This option can be overridden on the command line of `mdclient`. (Default: localhost)

- **Port:** Port of the mdserver to connect to. Can be overridden on the command line of mdclient using the -p option. (Default: 8822)
- **Login:** The login name of the user on the **AMGA(p.??)** server. All entries created in the catalogue will have this owner. This is also the user which you need to authenticate to the **AMGA(p.??)** server if authentication is enabled. (Default: NULL which gives the default role when authenticating with a VO certificate)
- **PermissionMask:** A 3 character string giving the owner permissions of newly created entries in the metadata catalogue.
- **GroupMask:** A 3 character string giving the group permissions of newly created entries in the metadata catalogue.
- **Home:** The home-directory. The default is "/".
- **Name:** Can be used to give a full name to the server, comparable to the comment in an /etc/passwd file. Currently only used for information in the server.
- **UseSSL:** Possible values are no, try, require (synonyme is yes). Default is no. Needed for any authentication using certificates (also proxy certificate). You want this if you intend to use passwords which are not sent in plain text. If you use SSL the entire session will be encrypted. Some servers may require you to use SSL to connect. If you want to be sure that SSL is always used you need to set this to require or yes.
- **AuthenticateWithCertificate:** Set this to 1 to enable certificate based authentication, also grid-proxy certificates. You will need to either enable normal certificates via a CertFile, KeyFile option pair, or use a grid proxy certificate via the UseGridProxy option. If you specify both, then the grid proxy gets precedence. This option does not work if you have not enabled SSL via UseSSL! In fact you will try to authenticate as the user named in Login, so you need to have this field set as well. However, the Password field is ignored unless certificate based authentication fails in which case the password is tried.
- **CertFile:** Path to your x509 certificate in .pem format. For this option to have any effect, you need also options UseSSL and AuthenticateWithCertificate enabled and UseGridProxy disabled!
- **KeyFile:** Path to your x509 private key in .pem format. If the key is encrypted you will be asked for the passphrase on client startup. For this option to have any effect, you need also options UseSSL and AuthenticateWithCertificate enabled and UseGridProxy disabled!
- **UseGridProxy:** Tries to use the a grid proxy certificate in /tmp/x509up_u[user-id].
- **Password:** If a password is given, password based authentication will be tried. You should want to use an SSL connection with this. This is not the password for your private key. For that you will always be prompted on the command line. For a discussion on how to escape special characters (#\) see **Configuring the AMGA Server and the Replicatin Daemon(p.??)** .
- **VerifyServerCert:** Verifies the server certificate against CA certifiates in TrustedCertDir, default is 1: yes.
- **IgnoreCertificateNameMismatch:** Do not try to match the DN in the server certificate to the canonical name of the server. Useful for services that have an alias or are multi-homed. Default is off.
- **TrustedCertDir:** A directory with certificate authority certificates to verify the server certificate.

3.1 Configuration of the Java command line client

The Java client also reads its configuration from a file, called by default `mdjavaclient.config`. This file is searched in the same places as the C++ API looks for the `mdclient.config`, that is, first in the current directory, then in `$HOME/.mdclient.config` and finally on `/etc/mdjavaclient.config`. Like the C++ client, only the port and server of the configuration file can be overridden on the command line of the clients:

```
mdjavaclient.sh [-p port] [hostname]
```

Next are the properties recognized on the `mdjavaclient.config` file. The following have the same syntax as in the C++ configuration.

- Host
- Port
- Login
- PermissionMask
- GroupMask
- Name:

The following exist only on the the Java configuration file or else have a slightly different syntax:

- **AuthMode**: Possible values are GridProxy, Certificate, Password, None. Default is None.
- **Password**: The user password. Only used when the AuthMode is set to Password. If not provided here, it must be provided at runtime by the user.
- **UseSSL**: Possible values are 1 for enable, 0 for disable. Default is 0. Needed for any authentication using certificates and grid proxies. If you use SSL the entire session will be encrypted. Some servers may require you to use SSL to connect.
- **CertFile**: Path to your x509 certificate in .pem format. For this option to have any effect, you need also options UseSSL and the AuthMode set to Certificate.
- **KeyFile**: Path to your x509 private key in .pem format. If the key is encrypted you will be asked for the passphrase on client startup. For this option to have any effect, you need also options UseSSL and the AuthMode set to Certificate.
- **PrivateKeyPassword**: Password for private key. If not defined, the password must be provided at runtime.
- **VerifyServerCert**: Verifies the server certificate against CA certifiates in TrustedCertDir.
- **TrustedCertDir**: A directory with certificate authority certificates to verify the server certificate.

4 Metadata Access from the Shell

You can access the metadata catalogue either with the `mdclient` metadata terminal tool as configured in the last section, or via the `mdcli` (or `mdjavacli` for those using the java package) command line tool which allows you to directly issue metadata commands on the shell, it's output is intended to be easily parseable by scripting languages:

```
> mdcli -p8822 -slocalhost listattr /
t
text
f
float
```

Metadata commands are parsed into pieces which are each separated by white space similarly to shell commands. If you want the white space to be part of one piece of the command itself, for example when you want to set an attribute to a string which contains white space, you must enclose it in single quotes: ' '. Single quotes are part of the command syntax and used when parsing the commands into parts. You need them every time a part shall contain spaces. Double quotes however are used in queries **Metadata Queries**(p.??) (expressions evaluated by the database backend) to distinguish strings from variable references and common values. In order to put a single quote into a command argument or any other character, you can use an octal code, e.g. to get a "'".

Note that quotes may be removed by the shell when parsing a shell command, so if you are using the `mdcli` tool where the **AMGA**(p.??) command is given on the command line, you will need to protect these single quotes from being removed by the shell with double quotes: " ". The various APIs will in contrast usually (that is the Python and Java APIs do so, but not the C-API) automatically quote any arguments you pass to them with single quotes so they are not to be used in those APIs. The following is an example with `mdclient`, `mdcli` and the Python API showing how quotes are being used:

```
> mdclient
Query> find /files '/files:producer="CERN"'
> mdcli find /files "'/files:producer="CERN'"
An in python:
mdclient.find('/files', '/files:producer="CERN"');
```

The metadata server uses a streaming protocol. Some APIs (for example the Java one) allow to interrupt the streaming of a response. The same is true for the `mdclient`. Pressing **CTRL-C once during the transmission of the result** will interrupt the streaming of the result. Only pressing **CTRL-C** a second time will terminate the client.

In the following is given a list of metadata commands. Additionally commands may be available for group or user access management, as described in **Users, Groups and ACLs**(p.??) or **Management of Users using the database backend**(p.??) depending on the server setup. To find out which commands are available on the server you are connected to, use the `help` command.

4.1 Commands for entry manipulation

- `addentry entry (attr value)+`: Creates a new entry and assigns the given values to the provided attributes. Examples:

```
addentry /testdir/a id 10
addentry b id 10 finished '0ct-10-2004'
```

Possible errors are:

- 3 Illegal command: Syntax error
 - 4 Permission denied: You need write permission on the directory to create a file in it.
 - 7 Illegal Key
 - 10 No such key
 - 15 Entry exists
- `insert entry (attr value)+`: Creates a new entry and assigns the given evaluated values to the provided attributes. Same as `addentry`, except that the values are evaluated first, similar to an `INSERT` statement in `SQL`.
 - `addentries (entry)+`: Creates the given entries in the catalogue. This command is done in a transaction, that is either all entries are inserted or none. Entries can be spread over several directories. Example:

```
addentries a
addentries /test1/a /test2/a /test1/b
```

Possible errors are:

- 1 No such file or directory: You tried to insert into a directory which did not exist.
 - 3 Illegal command: Syntax error
 - 4 Permission denied
- `rm [-options] pattern [condition]`: Removes all files matching pattern, where pattern may only contain wild cards in the file part. If a condition is given, than that condition needs to be met by the entry's metadata. In order to remove an entry you need write permissions on the parent directory. The pattern can also be the name of a directory if a condition is given, if the table is a plain table, the pattern `_must_` be a directory name and there must be a condition given. Valid flags are: `-r` (recursively delete).

Possible errors are:

- 1 No such file or directory: You tried to insert into a directory which did not exist.
 - 4 Permission denied
 - 20 Not an entry.
 - 29 Operation not permitted on plain table.
- `listentries [directory/schema]`: Returns the name of all entries in a given directory or schema. This differs from the `dir` command in that it will not show any directories but also that it shows only entries attached to a schema in the case of an **AMGA**(p.??) catalogue collaborating with a file-catalogue. The result is returned in the following way:

```
0
entry 1
...
entry n
EOT
```

- `transaction`: Starts a transaction. Any changes to the backend of **AMGA**(p.??) are done only when committed. To cancel a transaction use `abort`.
- `upload dir (attribute)+`: Starts an upload of entries into the catalogue. Currently a static restriction of the prototype is that there can be only up to 98 attributes assigned like this. After the upload is initialized, the `put`, `abort` and `commit` commands are allowed. Errors are returned by the call immediately, the OK code is delayed till the entire upload is successfully committed.
- `put file (values)+`: Inserts a new entry during upload. Errors are returned by the call immediately, OK is delayed until upload is committed.
- `abort`: Aborts upload or transaction. Errors and OK are returned by the call immediately.
- `commit`: Commits upload or transaction. Errors and OK are returned by the call immediately.

4.2 Commands for Manipulating Attributes

- `addattr dir (key type)+`: Adds new keys to the list of keys of a directory. In a relational database backend these keys become the columns of a table associated to a directory. You should only use one key/type pair currently for compatibility reasons because some older backends like PostgreSQL <=7.4 do not allow to alter a table adding several columns. Possible types are explained in **AMGA data types**(p.??). The type is only used as a hint for the back end to store the data efficiently and allow efficient queries. The type may be ignored by the implementation (e.g. if the back end is a filesystem). In a filesystem the types and defined keys could be stored as attributes of directories. Some storage backend may allow you to define keys on a per-entry basis. Possible errors are:

- 1 No such file or directory
 - 4 Permission denied. You need write permission on the directory.
 - 7 Illegal Key: Keys must be alphanumeric. The following keys are reserved: ACL, CREATED, FILE, GROUP_RIGHTS, GUID, LINK, MD5, OWNER, PERMISSIONS, SIZE.
 - 9 Internal error: All kinds of errors like duplicate keys
 - 23 Not a directory. You can only add attributes to a directory.
- **removeattr dir/entry key+**: Removes the attributes or keys from the list of attributes of the directory dir or the directory of the given entry, or if the implementation allows it from the list of attributes of a given entry. Attributes can only be removed if they are not used by any entry. So you either have to remove all entries for which this key is set or use clearattr to set the value of the attributes to NULL. In order to remove an attribute, write permissions on the directory are necessary. Possible Errors are:
 - 1 No such file or directory
 - 4 Permission denied. You need write permission on the directory.
 - 10 No such key
 - 14 Attribute in use
 - **schema_create dir (attr type)+ [option]**: Creates a new directory with a given schema. This is an atomic replacement for a sequence of createdir and addattr. The meaning of the optional option argument is backend dependent and you should not use it if you want to retain this independence. With a MySQL backend you can give here the name of the table engine, for PostgreSQL the keyword `inherit` will make the table inherit its schema from the parent directory.
 - **setattr file (attribute value)+**: Sets a list of attributes of a file to given values. The attributes must exist.
 - **getattr pattern (attribute)+**: Returns the filename and all attributes in turn for every file matching pattern. The following output is returned on success:

```

0
file 1
attr 1
...
attr n
file 2
attr 1
...
attr n
...
file n
attr 1
...
attr n
EOT

```

- **listattr file**: Returns a list of all attributes and their types in the following format:

```

0
attr 1
type 1
attr 2
.
.
type n
EOT

```

- **clearattr path attribute**: Sets the attributes of all files matching path to NULL. Path may currently contain wildcards only in the file-part. On success returns 0.

4.3 Finding and Updating Entries

- **find path query**: Returns a list of filenames matching path and fulfilling the query with their attributes. The path may currently contain only wild cards in the file name part. Query must be enclosed in single quotes. Strings in the query must be quoted with double quotes. For the supported syntax see the chapter about **Metadata Queries**(p. ??). **WARNING**: Be careful with patterns which also match a subdirectory, the result is undefined. The result is returned in the following way:

```
0
file 1
.
.
.
file n
EOT
```

The following errors may occur: 8 Illegal query: The query could not be parsed or violates security rules. No further information is currently provided on the reasons for this error. 2 Connection to DB failed or syntax error in path or query

- **updateattr pattern (attribute value)+ condition**: Updates attributes of entries matching a pattern in a single collection based on a condition. The values to which the attributes are updated can contain attributes as variables. Complex expressions are allowed as values. The condition may reference attributes of other collections. Updates are atomic. Examples:

```
updateattr /testdir1/* events events+1 'events>100'
updateattr /testdir1/* events events+1 '/testdir2:key > 0'
```

The first example increases the number of events of every file in /testdir1 which has more than 100 events by one. The second example increases the number of events of every file in /testdir1 provided there is an entry in the collection /testdir2 which has the attribute "key" set to anything larger than 1 (useful to do locking by clients: putting such an entry into /testdir2 would lock /testdir1). For the supported syntax for the queries see the chapter about **Metadata Queries**(p. ??).

- **update pattern (attribute value)+ condition**: Same as updateattr, but the values are not evaluated prior to insertion into the table. This command works with bound variables, which will also fix problems with SQL command length limitations in Oracle.
- **selectattr (attribute)+ condition**: Selects attributes from several collections based on a condition doing an inner join on the collections based on a join condition. The FILE attribute is used to select the entry name of an entry. Example:

```
selectattr /jobdir:FILE /configdir:id /jobdir:eventGen /configdir:id
'/jobdir:events>1000 and /configdir:key=/jobdir:key'
```

This selects the entry-name of a job, the id in the configuration, the event generator name of a job and the id in a configuration for all jobs and configurations where the job has more than 1000 events and the keys attributes of the jobs and configurations match. For the supported syntax for the queries see the chapter about **Metadata Queries**(p. ??).

As of **AMGA**(p. ??) 1.2 selectattr also supports constraints to the query similar to a SELECT clause. Queries can now take the form:

```
query [distinct] [limit xx [offset yy]] [order exp]
```

where the **distinct** keyword translates into a SELECT DISTINCT, the **limit** and **offset** clause limits the number of rows returned and with the **order** clause rows can be ordered according to the given expression.

4.4 Manipulating Collections

- `createdir /parentdir/dir [option]`: Creates the directory `dir` if it does not yet exist but `parentdir` already exists. The directory is created with the current owner and the same ACLs as the parent directory. The option field is a comma separated list of options (no spaces allowed). The following options are available as of **AMGA**(p.??) 1.1:
 - `shared`: Subdirectories created under this directory share the same schema and database table of the parent directory
 - `acls`: Creates a directory with acls for every entry, this is currently only supported by PostgreSQL and MySQL5 and only if the necessary supporting stored procedures have been installed first.
 - `type=`: Specifies the data type of the entry column. Explicitely supported are int, float, date.
 - `table=`: Only MySQL. Allows to specify the storage type of the table, e.g. InnoDB. Necessary because not all tables are alike. Specify e.g. MyIsam, this allows for example GIS functionality.
- `dir [directory]`: Returns the name of all subdirectories and files in the directory. The result is returned in the following way:

```
0
entry 1
entry-type-1
...
entry n
entry-type-n
EOT
```

where the entry-type is either 'entry' or 'collection'. If **AMGA**(p.??) collaborates with a file catalogue this command will effectively show the content of the file catalogue. If you want to see which entries have already been attached to a schema in the **AMGA**(p.??) part use the `listentries` command.

- `stat [dir/entry]`: Returns information on a given entry or directory. They can also be a pattern in the case of several entries. You need read permission to get this information. For an entry the following is returned (if the information is not stored in the table, it

```
0
name
type (=entry)
owner-permissions
group-permissions
owner
size
creation time (Format: 1999-12-22 04:05:06)
guid
expires
link
EOT
```

For a directory the following is returned:

```
0
name
type [options]
owner-permissions
owner
acls
EOT
```

- `rmdir path`: Removes all directories matching `path`. Directories are only deleted if they are empty and they have no attributes defined.

- `pwd`: Prints the current directory which you can change with `cd`.
- `cd path`: Changes the directory to the given path. Possible errors are:
 - 1 No such file or directory
 - 4 Permission denied

4.5 Permission Handling

- `whoami` Prints out the name of the current user. Note that this command does not need any connections of the **AMGA**(p.??) server and can thus be also used to do a test on whether an **AMGA**(p.??) server is alive and what response time it has.
- `chown entry/dir new_owner`: Changes the owner of a directory or entry. Only the owner of an entry is allowed to execute this, or the root-user. `chown` does not check whether the user exists, since user management is considered to be handled outside of **AMGA**(p.??) (ideally). Possible errors are:
 - 1 No such file or directory
 - 4 Permission denied
- `chmod entry/dir new_permissions`: Changes the access permissions of an entry or directory. Entries have owner and group-permissions, while directories have owner permissions and group permissions are handled via ACLs. Group permissions for entries allow you to remove privileges granted for all entries in a directory via the directories ACLs. The format of `new_permissions` is `rw-rwx` for entries and `rw-x` for directories where "-"-signs can be substituted for the letters if you do not want to give a certain privilege. The permissions for entries are the concatenation of first the user and then the group rights. The x-Flag allows a user to enter a directory or respectively list an entry. r-and w-flags allow users to read/write metadata while the w-flag for directories allows users to create or delete entries in the given directory. Users cannot list directories for which they don't have read permissions. The command works also for patterns and uses a transaction. Possible errors are:
 - 1 No such file/directory
 - 4 Permission denied

4.6 Capabilities

Capabilities are additional attributes assigned to individual users. They are used for example to allow a user to replicate login information. Currently no mapping of VOMS capabilities is done, but this could be a future use-case.

- `capabilities_add` : Adds the given capability to the user's capabilities. Only root can do this.
- `capabilities_remove` : Removes the given capability from the user's capabilities. Only root can do this.
- `capabilities_list [user]`: Lists all capabilities of a user, default is the current user.

4.7 Index Management

- `createindex name collection '(attribute)+' [algorithm]`: Creates an index name on a collection directory using several attributes and a given algorithm. Algorithms depend on the backend. The index is later referred to `/collection/name` in `index_remove`.
- `index_remove index_remove /path`: Removes an index.

4.8 Table Constraints

- `constraint_add_not_null directory attribute name`: Adds a not NULL constraint for the given attribute of the directory. Name is the name used to refer to the name of the constraint. It must be unique for that directory. Write permissions on the directory are necessary for this operation.
- `constraint_add_unique directory attribute name`: Adds a UNIQUE constraint for the given attribute of the directory. Name is the name used to refer to the name of the constraint. It must be unique for that directory. Write permissions on the directory are necessary for this operation. NOTE: On MySQL you can use `attribute(length)` to set the length of indexed columns.
- `constraint_add_reference directory attribute referenced_attr name`: Adds a foreign key constraint for the given attribute of the directory. The foreign key is given by the referenced attribute which must fully qualify that attribute including the table part, e.g. `/dir:attr`. Name is the name used to refer to the name of the constraint. It must be unique for that directory. Write permissions on the directory are necessary for this operation.
- `constraint_add_check directory check name`: Adds a check constraint to the directory. Check constraints are boolean expression which must be true for all entries inserted into the directory. An example would be `events > 0` requiring the value assigned to the events attribute to be positive. Name is the name used to refer to the name of the constraint. It must be unique for that directory. Write permissions on the directory are necessary for this operation.
- `constraint_add_primary_key directory key(s)`: Adds primary key constraint of the given directory, which by default is on the entry name only. Example:

```
constraint_add_primary_key . t,i
```

In the given example a the primary key becomes the pair of the t, and i attributes.

- `constraint_drop directory name`: Drops the constraint with the given name from the directory. Write permissions on the directory are necessary for this operation. The primary key of a table can be removed by specifying the name "p_key".
- `constraint_list directory`: Prints all constraints of a directory. You need read permissions on the concerned directory.

4.9 Views

Views allow you to create virtual new tables (directories) that combine the information of other tables, similar to what `selectattr` does. **AMGA**(p.??) uses the native support of the database to provide views, so the actual behaviour depends on the database backend. For example some backends (like PostgreSQL) allow you to update an existing view, which actually updates the tables behind it.

An important use-case of views are to support access restrictions to attributes (the columns of the underlying table). This is a typical use-case for views also in normal database usage.

Views can be accessed and deleted like normal directories.

- `view_create name maindir attr_1 ... attr_n condition`: Creates a view with the given name based on the entries in the given directory, attaching the attributes given in the list to these entries, based on the join condition.

In the following examples, the first one shows a use case where a view (view1) is created using all the entries in the current directory, but using only the attr1 or attr2 columns. After assigning the right permissions to the resulting new directory (`./view1`), this can be made readable for users who need to read these attributes, while they will not have access to the resto of the attributes in the `.` directory. In the second example a view (view2) is created combining attributes from the current directory and the `dir` subdirectory.

```
view_create view1 . attr1 attr2 ''
view_create view2 . attr1 ./dir:attr2 'dir:FILE = FILE'
```

4.10 Sequences

Sequences allow the creation of a sequence of integer numbers, which are guaranteed to be unique. They are also monotonically increasing at least during a single **AMGA**(p.??) connection. The exact implementation depends on the database backend, which can optimize handing out parts of the sequence in batches, so that two consecutive connections not necessarily get first a smaller number in the sequence and then the larger. Sequences are not supported by MySQL <5.0 and SQLite. On MySQL and Oracle sequences are implemented through stored procedures. In PostgreSQL the native mechanism is used.

Sequences behave like another directory in a directory. They cannot be deleted with `rmdir`, however, instead `sequence_remove` must be used. The name of the sequence must be lower case due to limitations in some backends.

- `sequence_create name dir [increment] [start value]`: Creates a new sequences with the given name in the given directory. The name of the sequence is then `/dir/name`. It is possible to define the increment as well as the start value. Note that backends may not necessarily follow this behaviour strictly if multiple connections are being used.
- `sequence_next sequence`: Gets the next value from a sequence.
- `sequence_remove sequence`: Deletes a sequence.

4.11 Backing Up Data

- `dump [-sec_all|sec_none] [dir]`: Recursively dumps the contents of a directory and all subdirectories so that they can be recreated by calling the sequence of **AMGA**(p.??) commands printed out. If no directory is specified, it uses `"/`, making a full catalogue dump. The first option controls whether entry permissions and ACLs are included in the dump: `-sec_all` includes them, while `-sec_none` only dumps the metadata.

Only root is allowed to use this command. Possible errors are:

- 4 Permission denied

- `user_dump [dir]`: Dumps the contents of a the user database such that it can be recreated from the sequence of **AMGA**(p.??) commands printed out.

Only root is allowed to use this command. Possible errors are:

- 4 Permission denied

- `grp_dump [dir]`: Dumps out the information on all existing groups so that they can be recreated by calling the sequence of **AMGA**(p.??) commands printed out.

Only root is allowed to use this command. Possible errors are:

- 4 Permission denied

4.12 Site management

For using replication, each **AMGA** server needs to know about the other servers that take part in the system, in order to communicate with them. These will be referred to as sites. The information about other sites is stored in the backend. Sites have the following configuration properties:

```
id
name
hostname
port
login
password
use_ssl
authenticate_with_certificate
```

```
cert_file
key_file
use_grid_proxy
verify_server_cert
trusted_cert_dir
require_data_encryption
```

`id` is a numeric identifier internal to each AMGA instance and generated automatically by AMGA when the site is inserted in the configuration. `name` is an human-readable identifier of the site, which can be freely chosen by the administrator. `hostname` and `port` is the network address of the remote site. The rest of the properties control the security settings of the connection to the master and are similar to the ones defined in the `mdclient.config` file, having a similar meaning. Section **Configuration of the C++ and Java command line clients**(p.??) describes their usage.

The following commands can be used to manage the sites and their configuration:

- `site_list` Lists all sites and their IDs. Shows only a summary for each site, consisting of `id`, `name`, `hostname` and `port`.
- `site_add site_name hostname:port` Registers a site. Only root can perform this operation.
 - 4 Permission denied
- `site_remove site` Removes a site. Only root can perform this operation.
 - 4 Permission denied
- `site_set_properties site property value [property value]*` Updates one or more configuration properties of a site. Only root can perform this operation.
 - 4 Permission denied
- `site_get_properties site property [property]*` Gets one or more configuration properties of a site. Only root can perform this operation.
 - 4 Permission denied
- `site_list_properties site` List all configuration properties of a site, as a list of key/value pairs. Only root can perform this operation.
 - 4 Permission denied
- `site_dump [siteName]*` Outputs the configuration of sites as a list of metadata commands that can be executed by **AMGA**(p.??) to recreate the configuration. If no `siteName` is provided, it dumps all sites in the configuration. Only root can perform this operation.
 - 4 Permission denied

4.13 Various administrative commands

- `import tablename dir` Makes the table given by `tablename` available under the given `dir` as a plain table. Only root can perform this operation.
 - 4 Permission denied
- `execute script [Options]...` Executes a script in the directory given by the `ExecRoot` option in the server configuration file. The script is run with the user-id given in the `ExecUser` configuration variable, if the `amgad` is started as root. The script has access to the `USER` and `DN` environment variables giving the user name of the `amga` user and the distinguished name if a certificate was used to authenticate to **AMGA**(p.??). To enable this feature, `ExecRoot` must be set. The administrator

of **AMGA**(p.??) is responsible to set this up carefully and make sure that scripts are checking the permissions in USER/DN. The stdout of the script is returned through **AMGA**(p.??) to the client. This gives **AMGA**(p.??) a functionality similar to a cgi-bin in a web-server. The following error codes exist:

- 21 Function not implemented: ExecRoot not set.
- 4 Permission denied: Cannot execute anything outside of ExecRoot.
- **backend** Returns the name of the current database backend.

4.14 Replication

The following are the commands used to control replication from the **AMGA**(p.??) command line interface. Some of them are for nodes acting as slaves, others for nodes acting as masters. Nodes acting both as slave and master can use all of them. Section **Replication in AMGA**(p.??) provides the background information required to understand these commands.

4.14.1 Commands for Slave Nodes

Slave nodes are responsible for initiating replication, by contacting the master and requesting the replication of the directories they are interested on. This is done using the following commands:

- **rep_list_mounts**: Lists all local mounts and their current state. It prints out the following information for each mount

```
<nodeID>:<directory> - <currentXID>, <state>
```

where:

- <nodeID> is the master from where the directory is being replicated.
 - <directory> is the root directory of this mount.
 - <currentXID> is the `xid` of the last log that was applied to this mount
 - <current state> is either **Disconnected** if there is no connection to this master, **Reconnecting** (after a broken connection, trying to reconnect to master) or **Receiving** (connected to master and receiving logs).
- **rep_mount [-noperms] nodeID remoteDirectory**: Mounts `remoteDirectory` from the master with id `nodeID`. After mounted, `remoteDirectory` is used to refer to this mount and is therefore also called `mountPoint`. This command fails if `remoteDirectory` already exists locally or if its parent directory does not exist. For instance, to mount `/a/b`, directory `/a` must be created and `/a/b` should not exist. It also synchronizes the local directory by copying the contents of the mount point from the master to the slave, but it does not make the slave start to receive logs. For that, use **rep_start_receive**. After this command executes successfully, the mount is on the **Inactive** state. By default, the permissions and acls associated with the metadata are also replicated. If `-noperms` is specified, the metadata will be replicated without any security information. On the slave, the metadata will be owned by the local user performing replication (which currently must be root).
 - **rep_umount mountPoint**: Unmounts the given mount point. The local contents of `mountPoint` is deleted, as well as the root directory of the mount point. This command only works if the connection to this master is inactive, that is, it should not be receiving logs. See **rep_stop_receive**. The `nodeID` has to be assigned via **site_add** with the port of the site pointing to the **replication daemon**, and where at least the login site-property has to be set.

- `rep_mount_users nodeID`: Replicates user and group information from the given master. The root user and any groups owned by root are not replicated. The slave must not have any user or group defined, except root or groups owned by root. Each slave can only replicate users and groups from a single master. Apart from these restrictions, the replication of users and groups is handled by any other mount.
- `rep_umount_users`: Stops replicating users and deletes all users and groups imported from the master. This command only works if the connection to the master is inactive, that is, it should not be receiving logs. See `rep_stop_receive`.
- `rep_users_allow user`: Allows the given user to replicate user (login) information. This needs to be allowed done on the master as well as on the slave.
- `rep_users_disallow user`: Disallows the given user to replicate user information
- `rep_start_receive master`: Connects to `master` starts receiving the replication logs for all mount points originating from this node. This opens a TCP connection that is kept open while the slave is running. If the slave is shutdown, the next time it restarts it will try to reestablish the connection to the master to continue receiving logs. The state of all mounts from this master is changed to `Receiving`.
- `rep_stop_receive master`: Stops receiving logs from `master`. The TCP connection is closed and the state of all mount points originating from this node is changed to `Disconnected`. In this state, connections will not be reestablished when the slave is restarted.

4.14.2 Commands for Master Nodes

The main responsibility of master nodes is to configure the access control rights of slaves. Slaves connect to the master using the standard `AMGA(p.??)` users and authenticate in the same way. Access control is done using the replication right, which is granted to users to control the directories they are allowed to replicate. The following commands allow granting and removing this right:

- `rep_allow directory user`: Grants to `user` the right to replicate `directory` and all its subdirectories. Having this permission is a necessary and sufficient condition for a user to be allowed to replicate a directory. In particular, the normal user permissions play no role in deciding to allow a user to replicate a directory.
- `rep_disallow directory user`: Remove from `user` the right to replicate `directory` and all its subdirectories.
- `rep_users_allow user`: Allows the given user to replicate user (login) information. This needs to be allowed done on the master as well as on the slave.
- `rep_users_disallow user`: Disallows the given user to replicate user information.
- `rep_show_permissions`: Show the replication rights for all directories.
- `rep_list_subscribers`: Show the list of the current subscribers, with their subscriptions. This includes information about whether users/groups and permissions are being replicated and the last xid acknowledged by the subscribers.

`AMGA(p.??)` also has special commands for user and group management. They are optional and may not be available on your installation for example if it collaborates with a file catalogue and uses the permission system of that catalogue. For more information see `Users, Groups and ACLs(p.??)` and `User Management(p.??)`.

5 Metadata Queries

AMGA(p.??) provides its own query language which is similar to the SQL query language. It tries to offer a large subset of the common functionality of database systems in a transparent way to the user. The biggest difference to SQL is that in AMGA's query language tables are referred to as references to directories. **AMGA(p.??)** will ensure access restrictions on the data so that users cannot infer data from queries if they have no read-access to that data.

Queries are performed in the following **AMGA(p.??)** commands:

```
find entry_pattern query_condition
selectattr column_1_query ... column_n_query query_condition
updateattr attr_1 update_query_1 .... attr_n update_query_n query_condition
```

A query condition is a query which returns a boolean in order to select or not select an entry for retrieval or update. Examples are

```
/jobdir:events>1000 and /configdir:key=/jobdir:key
like(/jobdir:FILE, "t%")
```

Query conditions are used in the WHERE statements of the SQL queries which are passed to the backends.

The other queries used in the **AMGA(p.??)** commands return general values which are returned to the user in the `selectattr` command or which are used to update attributes in the `updateattr` command.

Queries can contain either literal values like numbers or strings which are marked by double quotes. Make sure to use single quotes around double quotes if the string contains spaces. Queries can also contain attributes which are evaluated in the query by filling in the values of the attributes for the current value. In **AMGA(p.??)** all queries are in fact inner joins over all tables mentioned in any of the queries of command, that is the all possible combinations of all entries of all tables are made and those selected matching the query condition. Inner joins are the most common type of join. Some database systems provide no other kind.

References to attributes take the form:

```
<directory>:<attribute>
```

where relative paths to the directory (which is synonyme for table or schema, here) are allowed. Examples are:

```
selectattr /test:t 'like(t, "Test%")'
selectattr count(/test:t) 'like(t, "Test%")'
```

From the above example you can see that also functions are allowed. Function names are case-sensitive and lowercase. The following functions are available:

- `lower(string)`: Converts string to lower case.
- `upper(string)`: Converts string to upper case.
- `count(x)`: Aggregate function. Counts how often the attribute is set (not = NULL)
- `abs(x)`: Absolute value of x.
- `sin(x)`: The sine of x.
- `cos(x)`: The cosine of x.
- `tan(x)`: The tangens of x.
- `atan(x)`: The arc-tangens of x.
- `sqrt(x)`: The square root of x.
- `ln(x)`: The natural logarithm of x.

- `log(x)`: The base 10 logarithm of x.
- `rnd()`: A random number between 0 and 1.
- `sum(x)`: The aggregate sum of x.
- `max(x)`: The aggregate maximum of x.
- `min(x)`: The aggregate minimum of x.
- `avg(x)`: The aggregate average of x.
- `length(string)`: The length of the string.
- `pow(x, y)`: x to the power of y.
- `mod(x, y)`: x modulo y.
- `concat(str1, str2)`: The concatenation of str1 and str2.
- `like(str, pattern)`: Whether str is like pattern. The pattern is an SQL90 pattern.
- `substr(str, n, m)`: The substring of length m of str starting at n.
- `isnull(arg)`: Checks that the argument is NULL;
- `nonnull(arg)`: Checks that the argument is not NULL;
- `is(condition, t, e)`: Evaluates to t if condition is fulfilled, otherwise to e.

Queries can contain the following operators: +, -, *, /, =, and, or, not, >=, <=, <> or !=
 Special attribute names refer to the properties of an entry:

- `FILE`: The name of the entry.
- `LINK`: The link pointed to.
- `OWNER`: The owner of the entry.
- `PERMISSIONS`: The owner's permission.
- `GROUP_RIGHTS`: The group-rights. These names could e.g. be used to restrict access to entries using a `VIEW`.

As of version 1.2.11, **AMGA**(p.??) supports other table joins apart from the implicit cross join. The supported joins are left and right outer joins and the inner join. The following shows some examples

```
selectattr /t1:num /t1:name /t2:num /t2:value 'join_left_on(/t1:, /t2:,
/t1:num = /t2:num) limit 1'
selectattr /t1:name /t2:value 'join_right_on(/t1:, /t2:, /t1:num = /t2:num)'
```

The following is a list of the supported joins and their translation into SQL:

- `join_left_on(<left>, <right>, <condition>)` left LEFT OUTER JOIN right ON condition
- `join_right_on(<left>, <right>, <condition>)` left RIGHT OUTER JOIN right ON condition
- `join_inner_on(<left>, <right>, <condition>)` left INNER JOIN right ON condition
- `join_cross(<left>, <right>)` left C JOIN right ON condition

The exact syntax of **AMGA**(p.??) queries is described in the annotated `parser.y++` and `lexer.l++` sourcecodes.

6 AMGA data types

The **AMGA(p.??)** metadata server supports a set of generic datatypes for all the possible backends. It is guaranteed that if you add an attribute of with one of these types that it is translated into a type supported by the database. **AMGA(p.??)** also guarantees that a `listattr` command will return the same data type.

Other datatypes supported by a given back end can be used, however they will be not portable and it is not guaranteed that `listattr` will return the type you specified with `addattr`.

6.1 AMGA data types

The following table lists the generic metadata data types supported by **AMGA(p.??)** and their internal representation in the back ends:

| | PostgreSQL | MySQL | Oracle | SQLite | Python |
|---------------------|----------------------|----------------------|--------------|--------------|----------------|
| int | integer | int | number(38) | int | int |
| float | double precision | double precision | float | float | float |
| varchar(n) | character varying(n) | character varying(n) | varchar2(n) | varchar(n) | string |
| timestamp | timestamp w/o TZ | datetime | timestamp(6) | unsupported | time (unsupp.) |
| text | text | text | long | text | string |
| numeric(p,s) | numeric(p,s) | numeric(p,s) | numeric(p,s) | numeric(p,s) | float |

The datatypes have the following properties:

- **int**: at least 32 bit integer value.
- **float**: 64 bits IEEE double precision floating point number.
- **varchar(n)**: A string of up to at least $n = 254$ characters. The low limit of 254 is imposed by MySQL 4.0 and smaller version. MySQL may also truncate any trailing white space, see MySQL documentation.
- **timestamp**: Timestamp with format 'YYYY-MM-DD HH:MI:SS'.
- **text**: Long text string (2GB size limit).
- **numeric(p,s)**:SQL numeric type with precision p and scale s .

Note that SQLite not really has a strong typing system. SQLite is in fact typeless, so you can try to store anything into a given column, although internally it distinguishes between text strings and 64 bit double precision values. A column is of type text if the column type contains any of the following substrings: BLOB, CHAR, CLOB, TEXT.

The pure Python back end distinguishes strings, floats (64 bit), integers (64 bit) and timestamps.

Both pure python and SQLite will accept any type and guarantee you that the same type is returned by `listattr` with the default datatypes being numeric and string respectively.

7 Configuring the AMGA Server and the Replicatin Daemon

The **AMGA(p.??)** metadata server `amgad` the **AMGA(p.??)** replication daemon `mdservermt.config` are configured using the `amgad.config` file. By default, they share the same configuration file, but by using the `-c` command line option it is possible to specify a different configuration file for each. Nevertheless, the format of the configuration is similar for both programs.

7.1 Format of the Configuration File

Every line of the file contains a key value pair separated by an equal (=) sign. Blank space is ignored around the equal sign and the end of the line. Keys may not contain any white space but white space in the value is allowed. To have white space at the beginning of a value (after the =) or at the end (at the end of the line), escape it with a backslash (\). Lines can be continued with a backslash (\) at the end of a line (white space after the \ is ignored). Comments can be put at the beginning or end of a line after a hash-sign (#). The escape character is the backslash (\). The following escape sequences exist:

- \ : Gives \
- # : Gives #
- \ (): Insert a space at beginning/end of option

The configuration file is divided into two sections. The first, consisting of settings defined in the global section, contains generic server options. The second section is called `Replication` and contains all the settings controlling replication. Sections are opened like this:

```
[SectionA]
...

[SectionB]
...
```

7.2 Example of a server configuration file

```
# Server options
Port=8822 # Default 8822
MinProcesses = 2
MaxProcesses = 50 # Default: 50
# MaxConnectsPerProcess = 1000000

# Database options
# The DataSource option must match a data source name in the odbc.ini file
#DataSource=sqlitedb
#DataSource=Oracle10g
DataSource=PSQL
DBUser=arda

# Session options
Sessions = allow          # Values are: no allow force
IdleTimeout = 1200       # Timeout for a connection/query in sec [20m default]
SessionTimeout = 86400   # Timeout for a session in the cache in sec

# Connection limits
#MaxConnectsPerUser = -1 # Maximum concurrent connections per user
#ReservedConnections = -1 # Number of connections reserved for root

# Secure Connections
UseSSL = 1

# Authentication options
RequireAuthentication = 0 # If this is off, no authentication is done!
AllowCertificateAuthentication = 1
AllowPasswordAuthentication = 1
# If you use SSL, you need to load server certificates:
CertFile = cert.pem
KeyFile = key.pem
# If Certificate based authentication allowed, you need to load server certs
# TrustedCertDir = /etc/grid-security/certificates
# AllowGridProxyLogin = 1          # Requires also AllowCertificateAuthentication

# Authorization options, choose 0 or more
#MapFile = /etc/grid-map-file      # Authorization based on certs
# Authorization based on certificates, put a list of VOMS URL and assigned users, here:
#VOMSGroups = https://lcg-voms.cern.ch:8443/voms/lhcb/services/VOMSAdmin?method=listMembers, lhcb, \
#             https://kuiken.nikhef.nl:8443/voms/picard/services/VOMSAdmin?method=listMembers, picard
UserDB = 1 # Authorization based on certs & passwords
#VOGroupMap =
#VirtualOrganizations = gildav(gilda) # vo1(defaultUser1), vo2, vo3(defaultUser2)...
#VOGroupMap = gildav:/gildav(gildav:users)
#VOUserMap = gildav:/gildav/Role=TrailersManager(gildav)
#MyProxyHack = 1 # Allow roles in MyProxy certificates for login
```

```
#####
# Replication settings
#####
[Replication]
# Settings for the master
EnableMaster = 0
ReplicationDaemonPort=8833

# The remaining slaves are for slave nodes
EnableSlave = 1
NodeName=gridpc1
```

7.3 Description of the server options

The following options are supported:

- **Port:** The number of the port the server will listen on. This can be overridden on the command line with the `-p` switch. The default port is 8822.
- **MinProcesses:** This is the minimum number of processes waiting for client connections the server must offer. When the server starts up or there are no client connections for some time, `MinProcesses` is the number of processes spawned waiting for connections.
- **MaxProcesses:** This is the maximum number of processes the server will spawn in total. The server always tries to have 1/3 of the processes in the awaiting connection state. To achieve this, the server will spawn new processes until the number of `MaxProcess` is reached. Please make sure that your database backend can support as many client connections.
- **MaxConnectsPerProcess:** To prevent any very rare memory leaks or other resource leaks to reduce the stability of the service, server processes can be asked to terminate themselves after serving a certain number of connections. The default is not to do this.
- **DataSource:** An ODBC database source which you need to have configured in an `odbc.ini` file. Note: you can use programs like `gODBCConfig` or `iobcdm-gtk` to configure ODBC. The underlying database needs to be prepared for **AMGA**(p.??) by running the `createInitial.sql` script in the `scripts` directory of the **AMGA**(p.??) distribution.
- **DBUser:** The user with which the server will contact the database backend. The user name given in the `odbc.ini` file is the default.
- **DBPass:** The password the server will give when contacting to the database backend. The password in `odbc.ini` is the default.
- **DBSchema:** The schema within the database **AMGA**(p.??) shall use for its own tables. Via the import feature it is possible to then access tables in different schemas. The default depends on the database backend and is the user-name on Oracle, the database name on MySQL and for SQLite and PostgreSQL it is the default schema of the database.
- **Sessions:** This defines whether you want to allow sessions. Sessions create an overhead on the protocol if they are enforced, so the performance of individual clients may reduce while you will be able to support more clients which share the available connections (there is a maximum of `MaxProcesses` connections, if they are all hogged by a client, then no new clients will be able to connect). Such a denial-of-service situation can be prevented by forcing sessions. Values are: `no`, `allow`, `force`. Default is `allow`.
- **IdleTimeout:** Timeout for an idle connection (that is a connection that waits for a client command) in seconds. There are no timeouts currently for database queries apart from how the database is configured. The default is 20 minutes. This is what will make your `mdclient` command line tool time out.

- **SessionTimeout:** Timeouts for session. The lifetime of a session in seconds. Default is 1 day.
- **MaxConnectsPerUser:** Maximum number of concurrent connections for a user. Default is -1: No limits.
- **ReservedConnections:** Number of connections reserved for the root user. Default is -1: No reserved connections.
- **UseSSL:** Whether the server will offer SSL as a connection protocol. This is also required to allow certificate based authentication and if you want to use passwords this is recommended if you want to be sure no one listens in. Values are 0 and 1, default is 1. Note that you cannot force the client to use an SSL connection.
- **RequireAuthentication:** Whether users need to be authenticated. Default is 0: no.
- **AllowCertificateAuthentication:** Whether you allow users to authenticate with their certificate. You will need to have CA certificates loaded for the server for this to work in order to be able to verify the client's certificate. See **TrustedCertDir**. You should also look at **User Management**(p.??) because you will need to have a user manager module running for this to work. Default is 0: no.
- **AllowPasswordAuthentication:** Allow authentication with a password. You need a user manager module running for this to work. See **User Management**(p.??) . Default is 0: no.
- **CertFile:** The path to the server certificate in PEM format. Preferably without encryption. Necessary to use SSL.
- **KeyFile:** The path to the private key of the server in PEM file format. Necessary to use SSL.
- **TrustedCertDir:** Path to a directory with trusted CA certificates. Needed to verify a client certificate.
- **AllowGridProxyLogin:** Whether you allow users to authenticate with a proxy certificate. Default is 0: no.

7.3.1 Replication Settings

The following options control the replication system. They should be defined on the replication section of the configuration file, otherwise **AMGA**(p.??) will either not recognize them or will understand them as referring to one of the generic settings, since some names are equal.

These two settings are used by master nodes:

- **EnableMaster:** Allow this node to act as a master for replication. In particular, this setting activates saving of replication logs to the hard disk. Therefore, if the node is not going to be used as a master, this setting should be disabled for performance reasons. Default: 0. (Disabled)
- **ReplicationDaemonPort:** The port used by the replication daemon to listen for connections from slaves. Default: 8823.

The following settings are used by slave nodes:

- **EnableSlave:** Allow this node to act as a slave for replication. Leaving this setting active has no impact on performance, the only different is that it allows the use of the client side replication commands. Default: 0. (Disabled)
- **NodeName:** The identifier of the local node. Should be unique among all the nodes on the system.

For the rest of the options see **User Management**(p.??) .

8 Replication in AMGA

This section presents the Replication mechanisms that are part of AMGA since version 1.2.

8.1 Overview

In this section we provide the technical background necessary to understand replication in AMGA. Reading this section is highly recommended for anyone interested in using this feature. Further details can be found in the following article, available on the AMGA web page:

N. Santos and B. Koblitz, *Distributed Metadata with the AMGA(p.??) Metadata Catalog*, In *Workshop on Next-Generation Distributed Data Management - HPDC-15*, Paris, France, June 2006
Some more background is also explained [here](#).

8.1.1 Features Overview

Replication in AMGA follows an **asynchronous, master-slave** model, and supports **partial replication** of the directory hierarchy.

In **master-slave** replication only one of the replicas, the master, is writable, with all other replicas being always read-only. This model is sufficient for all applications with read-only metadata or where the metadata is written only at a single geographical location, which covers most of the Grid applications.

Slaves can replicate any sub-tree of the metadata hierarchy. This allows slaves to copy only the data they are interested on, reducing the load both on the slave and on the master, as well as the bandwidth requirements. If desired, a full replica of the master can be obtained by replicating the root directory.

8.1.2 Concepts

Next is the description of the main concepts used in AMGA Replication.

- **Master:** Any node that exports part of its metadata hierarchy for replication.
- **Slave:** Any node that replicates metadata from a master. A node can be both a slave and a master at the same time, or neither if working standalone.
- **Replication log:** Shipped from a master to a slave, containing a metadata command plus some context information that allows the slave to replay the original command executed at the master.
- **Subscriptions:** To replicate a directory, the slave creates a subscription for that directory with the master that will allow it to receive the replication logs for that particular directory.
- **Mount:** On the slave side, the root of the sub-tree subscribed from the master is called the mount. There is a mount for each subscription and it corresponds to the path of the directory that is replicated. Each mount contains also all the sub-directories of the mounted directory.
- **Initial Synchronization:** When a slave subscribes for the first time to a directory, it must copy the existing data for which there are no replication logs.

8.1.3 Architecture Overview

In AMGA replication, the master ships to the slaves logs containing the metadata commands executed at the master, much in the same way as it is done in Oracle Streams and MySQL replication. When updated by a client, the master saves a replication log containing the command and some context information to the `logs` table on its database backend. A separate program, the replication daemon, queries the database periodically looking for new logs and shipping them to the slaves with subscriptions to the directories updated by the command of the log. This replication daemon is also responsible for managing the list of slaves that are replicating from the master. At the slave, the logs are replayed to update the slave's metadata. The logs contain only metadata commands, and they are totally independent of the underlying database backend.

Logs are assigned a unique sequence number at creation time, the log `xid`. Each AMGA instance generates its own identifiers in an independent way, so the `xid` is unique only inside an AMGA instance. The `xid` is necessary for synchronization between the slave and the master. For each directory replicated, the slave keeps the `xid` of the last log it received and applied, so that it knows from what point to resume after connection failures or shutdowns. The master also keeps track of the subscriptions by storing them persistently on its

database backend, on the `subscribers` table. For each subscription, it stores the most recent xid that the slave has acknowledge. This is necessary to know when there are no more slaves waiting for a particular log, so it can be removed from the `logs` table.

Replication Daemon The bulk of the operations necessary for replication on master nodes are implemented outside the `amgad` process, by the `mdrepdaemon` program, also called the replication daemon. The `amgad` is only responsible for writing the replication logs to the database, while the replication daemon does everything else related to replication, including:

- Managing subscriptions
- Accepting replication requests from slaves
- Sending the initial snapshot of replicated directories to slaves
- Polling the logs table and shipping logs to the interested subscribers
- Cleaning up the logs table, removing any unnecessary logs.

The replication daemon is independent of the AMGA server it is working for, and needs only to connect to the database backend used by the AMGA server. Apart from that, there is no communication between them and they can run separately, even on different machines for better load balancing.

8.1.4 Operation

Nodes interested in replicating from another node must subscribe to the directories they wish to receive by contacting the replication daemon of that node. After connecting, the slave informs the master of the directories it is interested on, and begins copying the contents of the database. This is done using the `dump` feature of AMGA, which generates the commands that must be executed on another AMGA instance to recreate a directory hierarchy. The replication daemon internally executes a `dump` and forwards the commands to the slave, that replays them. Each directory is shipped using a database transaction to isolate it from updates that may be happening concurrently. It is also tagged with the xid of the last log generated for that directory before the synchronization started, so the slave will know from what point to start receiving logs after finishing the initial synchronization. Updates generated during the synchronization will be saved as logs and shipped to the slave after the synchronization is done.

The initial synchronization might be a lengthy process and currently there are no provisions for resuming from failed synchronizations. Nevertheless, if the synchronization is interrupted, the slave will reestablish a consistent state by discarding all the information received. In the future, we plan to implement mechanisms to allow resuming partial synchronizations.

After having the initial snapshot, the slave can start receiving and applying logs. In the current implementation the slave connects to the master using a TCP connection, sends the xid of the earliest log that it wants to receive and waits for incoming logs. When the slave receives a log, it executes the log locally, and after making sure that the log is safely committed to the database, it sends an `acknowledge` to the master. After receiving the `acknowledge`, the master is free to delete the log. To ensure good performance over high-latency connections, this communication between the master and the slave is asynchronous, that is, the master sends the logs without waiting for `acknowledges` of previous logs, and polls the socket periodically with a non-blocking operation looking for incoming `acknowledgments`.

The replication daemon stores the information about the subscriptions in the local database in order for it to survive eventual crashes of the master node. This information includes the slave's id, address, directories subscribed, and the id of the last log acknowledged.

Logs must be deleted when they are no longer needed. This is done also in the replication daemon, by the same thread that monitors the logs. After polling for new logs, shipping them to subscribers, and polling for new `acknowledges` from clients, this thread goes over the logs table and deletes the logs that are no longer needed by any subscriber. Under normal conditions, that is, when all subscribers are connected, logs are deleted shortly after being generated. A log is kept for a longer time only when a subscriber is disconnected.

To tolerate failures resulting in disconnections of the subscribers, subscriptions are persistent, in the sense that if a subscriber disconnects without having first requested to be unsubscribed, the master will preserve the subscription and continue saving logs for the directories subscribed by the slave. When the subscriber reconnects, the subscription is resumed from the point it was interrupted. Currently, there is no provision for dealing with slaves that are disconnected for too long. In this case, the logs will accumulate on the logs table, eventually causing problems on the database backend. In the future, we plan to implement mechanisms for controlling the growth of the log table by removing old subscriptions when certain conditions are met, like the log table exceeding a maximum size or a subscriber being disconnected for too long.

8.2 Setup

8.2.1 Setting up a Master node

A master node consists of an amga daemon configured to save replication logs on the database plus an associated replication daemon. To simplify the setup, both processes use exactly the same format for the configuration file, allowing them to share the same configuration file.

The amgad is configured to save logs by setting the following property on the configuration file:

```
EnableMaster = 1
```

As an optimization, logs are saved only if there is at least one slave interested on the directory updated by the log, so if there are no subscriptions, the cost of having replication enabled is only an extra database query (to look for active subscriptions).

The replication daemon must be configured to point to the same database as the amga daemon. The simplest way of doing this is to reuse the same configuration file for both programs, although it is possible to have separate files. Additionally, the replication daemon needs to be assigned to a port where it will listen for incoming connections. This is configured by the `Port` property on the `Replication` section of the configuration file:

```
[Replication]
...
ReplicationDaemonPort=8823
```

This is all that needs to be configured for the replication daemon. After this, it can be started with:

```
$ mdrepdaemon [-c <amgad.config>]
```

The accepted options are:

```
-p <port>      : listen port
-d            : Activate debug output. Very verbose.
-c <configFile> : configuration file
```

If no options are given, it will look for `amgad.config` in the same way as `amgad` does.

After starting up, the replication daemon waits for connections from slaves and polls the database periodically looking for new logs. It will also go over the `logs` table periodically to delete logs that are no longer needed.

8.2.2 Setting up a Slave node

The slave functionality is implemented fully on the `mdserver`, and therefore, there is no need to run an external program like the replication daemon. To activate replication as a slave, the following setting must be enabled on the configuration of the `mdserver`:

```
EnableSlave = 1
```

Activating the slave mode enables the `rep_*` commands for the slave, and has no impact on the performance.

A slave node needs also to be configured with the settings of master nodes that it might be required to contact. This configuration is defined on a per-node basis, which is done through the `site_*` commands, which can be used to define sites and set their parameters like the login name the slave will use or the port of the replication daemon to contact on this site. The minimum configuration necessary is (pointing to a replication daemon listening on port 8832 and logging into it as root):

```
site_add SiteA localhost:8832
site_set_properties SiteA login root
```

Please consult Section **Replication**(p.??) for further information on how to define the master nodes and **Site management**(p. ??) for how to configure sites. Once configured as a slave node, the server is started normally. There are no special command line options related to replication.

On startup, the server will consult its database backend to check if there are any subscriptions on the active state. If so, it will try to restart receiving logs from these masters, by reopening a TCP connection to each master and asking for the logs after the last one it received. If the connection can't be established immediately, it will continue to try, waiting 60 seconds between attempts, until either it succeeds or the subscriptions is stopped using the `rep_stop_receive` command.

8.3 Security

8.3.1 Replicating Security Information

There are two types of security information managed by AMGA:

- **permissions and acls** - This is associated with the metadata.
- **groups and users** - Global to the catalogue instance, and not associated directly with the metadata.

Both types of information can be replicated in AMGA. The first is replicated as part of the metadata replication. The `rep_mount` command by default imports this information from the master. In this case, the permissions and owner of the metadata on the slave will be exactly the same as in the master and will be kept synchronized with the master. This behavior can be disabled by specifying the `-noperms` option, so that the slave replicates the metadata without any security information. The resulting replica will be created using the security settings in effect at the slave at the time of replication. After a directory being mounted in a slave, it is not possible to change its configuration concerning replication of permissions and acls.

Groups and users are replicated on their own, using the commands `rep_mount_users` and `rep_umount_users`. They are handled like a virtual mount, in the sense that they have to be synchronized initially by executing `rep_mount_users` and will be updated after establishing a connection to the master with `rep_start_receive`. The hash of the password of the user is also replicated. This allows the slave to authenticate the replicated users locally. A slave can only replicate users and groups from a single master and it should have no local users or groups of its own. The exception is the root user and any group owned by root, which are always considered local users and are not replicated.

8.3.2 Security at Slave Nodes

Only the root user is allowed to initiate or to control replication, that is, to execute the `rep_*` commands. For any other users, the replicated directories will be readable depending on their permissions and access control lists, which may or may not have been replicated from the master depending on the options given to `rep_mount`. A replicated directory is always read-only, regardless of the write permission or access control right.

8.3.3 Connections from the Slave to the Master

Connections from slaves to the replication daemon use a similar protocol as the ones from the clients to the `mdserver`, including the support for authentication and encryption. Therefore, the connection settings

(e.g., use SSL, authenticate with password, certificates or grid proxies,...) used by the `mdrepdaemon` can be configured in the same way as the `mdserver`. In fact, it's even possible to use the same configuration file, in which case the connection settings will be the same. To have different settings, it's enough to specify different configuration files using the command line options of each program.

When connecting to the replication daemon, slaves may need to authenticate. Once again, this is done just like if the slave were a client connecting directly to the master. The replication daemon will accept the same credentials and users as the `mdserver` of that node. The slave node is configured in the configuration file of the slave's `mdserver`, on the [Replication] section. The slave can specify several different master nodes, with different connection settings for each. It is possible to check the list of known nodes at a slave during an interactive client session using the `rep_list_nodes` command.

8.3.4 Granting the replication right on the master

The master controls what groups can replicate which directories by granting the replication right. This right is granted to groups, allowing them to replicate the specified directory, including the full sub-tree rooted on it and all their entries. The right is granted using an interactive session to the master with the commands `rep_allow` and `rep_disallow`.

An important point about this right is that for replication it is the only access control performed by the master. This is easier to illustrate with an example. Suppose that user `joe` does not have permission to read or write directory `/jobs` when connected as a client, but has the replicate permission over that directory. Then, if a slave connects to this master authenticating as `joe` it will be able to replicate the full contents of `/jobs`. Once this information is on the slave, it is completely exposed to the slave's administrator. If the administrator is trustworthy, he will allow the system to enforce the original access permissions. But malicious administrators can easily expose the information to any user.

8.4 Limitations

8.4.1 Concurrent Updates on Master

When two or more clients are updating the master concurrently, there is a small probability that the slaves become inconsistent with the master. For this to happen, the clients must be writing (updating, deleting or inserting) to the same collection and to intersecting sets of rows. This is a concurrency problem, so it will happen randomly depending on the interleaving of the execution of two or more concurrent commands. This problem can be avoided in two ways:

- Prevent concurrent updates to the same directory.
- Set the transaction isolation mode of the database to serializable.

The second solution is generic but has a high overhead on the database backend. Another disadvantage is that the database backend will abort the commands that cannot be serialized due to conflicts with other concurrent clients, and this will create a new failure mode that clients might not be expecting.

Technical Details

The replication mechanisms of AMGA are based on the assumption that it is possible to order the write commands received by the master into a sequence that if replayed in the slave will produce the exact same results. But this is not necessarily true, depending on the transaction isolation level used by the database backend. Most database systems use an isolation mode (Read Committed) that allows transactions to interleave in a way that makes them non-serializable. This is, there is no ordering of transactions that if replayed sequentially on a different database would produce the same results. This basically means that with the default isolation levels used by databases, there is no way of guaranteeing consistency between the master and the slave in all possible situations. Databases support stricter isolation levels, including a Serializable level. This mode will provide the guarantees needed by AMGA, but has a high overhead and therefore should be used only when the server needs to support concurrent updates.

8.5 Tutorial - Setting up a Master to Single Slave Setup

This section provides a step by step tutorial of how to prepare a simple replication setup. It will show how to setup replication from a master to a single slave. These are the names of the hosts that will be used:

- master - gridpc1.cern.ch, port 8822
- slave - gridpc2.cern.ch, port 8822

The tutorial assumes that the master and the slave nodes have already valid AMGA instances configured and operational, but with replication disabled. On the master, we assume the following directory hierarchy:

```
\users
\files
\files\2005
\files\2006
```

We assume that the slave is interested in replicating the `\files\2006` directory.

8.5.1 Configuring the master

- Configure the AMGA instance at `gridpc1` to act as a master and configure the port where the replication daemon will listen. This is done on the `mdserver.config` file, with the following properties:

```
[Replication]
EnableMaster = 1
ReplicationDaemonPort=8823
```

- Setup authentication and security for connections from slaves. For this tutorial we will require no security and accept plain text connections. Edit `amgad.config` at the server adding the following lines to the main section:

```
UseSSL = 0
RequireAuthentication = 0
```

- Configure access permissions to replication. This is done by connecting to the `mdserver` instance of the master over a normal interactive session using `mdclient`. First we create a group for users allowed to replicate. We will call it `replicators`, but any group name will do:

```
$ mdclient gridpc1
Connecting to localhost:8822...
ARDA Metadata Server 1.2.0
Query> grp_create replicators
```

Now we must add to this group the users allowed to replicate. For this tutorial, we will create a new user called `joe` but any existing user can be used:

```
Query> user_create joe
Query> grp_adduser replicators joe
```

Add the `replicators` group to the list of groups allowed to replicate the `/files` directory.

```
Query> rep_allow /files replicators
```

This will allow the `replicators` group to replicate all the contents and subdirectories of the `/files` directory.

- Back at the shell, start the replication daemon:

```
$ mdrepdaemon -c amgad.config
```

8.5.2 Configuring the slave

Now we are going to configure the AMGA instance at `gridpc2` to act as a slave. We start by editing the `mdserver.config` file.

- Activate the slave functionality:

```
[Replication]
...
EnableSlave = 1
...
```

- Configure the list of master nodes. In this case, it's only a single master, you need to be root to set up the sites:

```
Query> site_add GridPC1 gridpc1.cern.ch:8823
Query> site_set_properties GridPC1 login joe
Query> site_set_properties GridPC1 use_ssl 0
Query> site_set_properties GridPC1 authenticate_with_certificate 0
```

Note that only the login property is mandatory, the other settings are the default, anyway.

We can now start the AMGA server on the slave and connect to it using the `mdclient`. The rest of the tutorial is done from inside the `mdclient` shell.

- Create the parent directory of the directory we are going to mount.

```
Query> createdir /files
```

The slave expects this directory to exist, otherwise it will fail. We should also make sure that there is no directory with the same name as the one we are going to mount, in this case `/files/2006`.

- Mount the directory.

```
Query> rep_mount GridPC1 /files/2006
```

The slave connects to the master, subscribes to this directory and copies its contents. When the command finishes executing, we should have a local copy of the remote directory. We can check the status of the mount:

```
Query> rep_list_mounts
>> GridPC1:/files/2006 - 23628, Disconnected
```

The `Disconnected` state means that we are currently not receiving logs. The number before the state is the xid corresponding to the position of our local snapshot on the log sequence.

- Start receiving logs. The slave now must connect to the master and wait for logs. This is done with the command:

```
Query> rep_start_receive GridPC1
Query> rep_list_mounts
>> gridpc11:/files/2006 - 23645, Receiving
```

The slave will keep the connection to the Master open until it is shutdown or until the we stop the subscription. If the slave is restarted, it will resume automatically the subscriptions that were active when it was shutdown. It will also try to reconnect if the connection fails for some external reason.

- Removing the subscriptions. If we want to cancel the subscription we have first to stop receiving the logs and then unmount the remote directory:

```
Query> rep_stop_receive GridPC1
Query> rep_umount /files/2006
Query> rep_list_mounts
Query>
```

9 User Management

The standalone **AMGA**(p.??) server comes with a powerful system to manage users as well as to control access to entries and metadata. If **AMGA**(p.??) is run as an add on to a file catalogue, however, these features are not available and the access controls of the file catalogue is used instead.

To understand the user management of the **AMGA**(p.??) server it is necessary to know that the server does not really manage users but only their authentication and authorization. When changing the owner of an entry for example, the server does not check that this owner exists. Users are only relevant for logging in. This allows to manage users outside of the server, e.g. in a VOMS.

IMPORTANT NOTE: When you use sessions or connections, then changes to the way a user is authenticated or how the authorization is done through the mapping to an **AMGA**(p.??) user, will not affect active sessions or connections. You must restart the server to propagate changes to the user management to active sessions or connections.

9.1 Configuration

To use the metadata service, a user must be authenticated and authorized. Authentication can be done via a certificate or a password, see **Configuring the AMGA Server and the Replicatin Daemon**(p.??) . After the authenticity of a user is established in the handshaking of the client with the server, the client needs to be authorized to use the role of a certain user. Authorization is optional, if authorization is not enabled for the server, any authenticated user can assume any role he wishes. Authorization is controlled via the `mdserver.config` configuration file:

```
# Authorization options, choose 0 or more
# MapFile = /etc/grid-map-file           # Authorization based on certs
# Authorization based on certificates, put a list of VOMS URL and assigned users, here:
VOMSGroups = https://lcg-voms.cern.ch:8443/voms/lhcb/services/VOMSAdmin?method~=listMembers, lhcb, \
             https://kuiken.nikhef.nl:8443/voms/picard/services/VOMSAdmin?method~=listMembers, picard
UserDB = 1 # Authorization based on certs & passwords
```

Authorization can be done via certificates or passwords (password authentication actually includes authorization), both must be explicitly enabled. **For authentication via certificates to work, both the server and the client must have SSL enabled (UseSSL).** Four ways are foreseen to accessing the necessary information to match user names with their credentials, one or more must be enabled for the `RequireUserAuthorization` to work:

- A grid-map file mapping certificate subjects, that is distinguished names (DN of users) to users. This is a static setup and no new users can be added at runtime. No password authorization is possible via a grid map file. Option `MapFile`.
- A user database using the database backend. This allows creation of users and the management of their credential at runtime. This is the only option which allows password based login. Option `UserDB`.
- Authorization using a VOMS. All users registered with a VO will be assigned to the user specified here. You can give several VOMS-URL user pairs here. Option `VOMSGroups`.
- Authorization via VOMS certificates. All users connecting with a VO information-enriched certificate obtained via `voms-proxy-init` will be assigned to specific **AMGA**(p.??) users depending on the role within the VO. Option `VirtualOrganizations`. Note that only the DB based user management module is able to make changes to the user setup. If you have several user management modules activated at the same time, then listing users and checking their credentials for authorization will go through the users in all of the modules. A user is authorized as soon as he has been found in **any** of the modules.

9.2 Management via a Grid Map file

You can give a location of a grid map file using the `MapFile` option for user authorization. This file contains pairs of distinguished names and user names. The DN must be enclosed in double quotes and must be

in the form where its fields are separated by commas on one line (output of `openssl x509 -subject -in usercert.pem -nameopt oneline -noout`):

```
> cat mapfile
"/C=CH/O=CERN/OU=GRID/CN=Birger Koblitz 9904" koblitz
```

There are no wild cards currently allowed. The map file will be read only once at server startup. It is not possible to add or change users using the command line tool.

9.3 Management of Users using the database backend

To enable user management using your database backend, you need to enable this feature by setting `UserDB = 1`. If you have run the `createInitial.sql` script, to set up your database, the necessary tables have already been created. You can now manage users via the `mdclient` command line tool:

- `user_list`: Lists all users known to the authentication subsystem.
- `user_listcred user`: Lists the credentials with which the user can be authenticated. Returns first the user name, then whether there is a password and then the certificates which are mapped in a Grid-Mapfile and via the user database. Finally the different VO and VO roles which allow you to become that user are listed. Only root is allowed to see the credentials of other users.
- `user_create user [password]`: Creates a new user and assigns a password if given. This command is for the root user only. Only hashes of passwords are stored in the database backend.
- `user_remove user`: Deletes a user. This command is for root only. It does not check whether there are still files or directories owned by that user.
- `user_password_change user password`: Changes the password of a user. Only root can change the password of any user. Non-privileged users may only change their own passwords.
- `user_subject_add user subject`: Adds a certificate identified by its subject line to be used to authenticate a user. While every user can only have one password. Several certificates can point to the same user. Remember that in order to have spaces in the subject, you need to enclose it by single quotes (""). See **Definition of the Client Server Protocol**(p.??). The form of the subject needs to be the one where parts are separated by commas as in the output of e.g. `openssl x509 -subject -in usercert.pem -nameopt oneline -noout`.
- `user_subject_remove user subject`: Removes a certificate from the list of certificates which allow to login as a certain user.

9.4 Management via a VOMS

Giving pairs of VOMS member list URLs and user names in the `VOMSGroups` option, you can assign all members of a VO to a user (role would be the better word here).

9.5 Management via VO-Certificates

You can allow users to log in with VO-enabled certificates by using the `VirtualOrganizations` option and assigning it a list of `VO(default_user)` definitions. By enabling `MyProxyHack` this works also with certificates issued by a `MyProxy` server. The `VOGroupMap` and `VOUserMap` options allow to map VO groups to `AMGA`(p.??) groups and special VO roles to `AMGA`(p.??) users with the syntax used by `VirtualOrganizations`.

10 Users, Groups and ACLs

The standalone **AMGA**(p.??) server comes with a powerful system to manage users as well as to control access to entries and metadata. If **AMGA**(p.??) is run as an add on to a file catalogue, however, these features are not available and the access controls of the file catalogue is used instead.

The permission schema tries to copy the semantics of POSIX APIs. Some of the semantics are different from the POSIX semantics for a file system as **AMGA**(p.??) is a metadata catalogue. As an example, it is necessary to have the 'x' permission for a directory to read the attribute list, while 'r' permissions for any file are necessary to read the values of the attributes for a file. The exact behaviour is described together with the respective commands.

10.1 Users

The size of a username is limited to 64 lower-case latin alphabet characters.

10.2 Groups

Any user can create groups. Group names are scoped with the name of the user creating them. A fully qualified group name has the form user:groupname. If the user scope of the group is the current user, it does not need to be specified in a command. The size of groupname is limited to 64 lower-case latin alphabet characters.

A special group exists and is maintained by **AMGA**(p.??) internally, the system:anyuser group which contains automatically any user which is authenticated to the system. Using this group it is possible to emulate the permissions for 'other'-users in a Unix filesystem which are missing in **AMGA**(p.??).

The following commands can be used to manage groups:

- **grp_create groupname**: Creates a new group with name groupname. It is not possible to create groups belonging to others.
- **grp_delete groupname**: Deletes a group with name groupname. Only root can delete groups of other users.
- **grp_show groupname**: Shows all the members belonging to group groupname. You can only look into groups of which you are a member or your own groups. Root can list all groups.
- **grp_adduser groupname user**: Adds a user to a group. Only owners of a group or root can change group membership.
- **grp_removeuser groupname user**: Removes a user from a group. Only owners of a group or root can change group membership.
- **grp_member [user]**: Shows to which groups a user belongs. Only root can ask this question for other users.
- **grp_list [-a] [user]**: Shows the groups owned by user, by default the current user. If the -a option is given, all groups are shown.

10.3 Access Control Lists

ACLs (Access Control Lists) can be assigned to any directory.

The following commands exist to manipulate ACLs of a directory.

- **acl_add directory group rights**:
- **acl_remove directory group**: You can use the * to remove all ACLs of a directory.
- **acl_show directory**:

On MySQL5 or PostgreSQL you can create directories with the "acls" option, which will allow you to put ACLs also on individual files.

11 Installation from Source

To install the ARDA metadata server from source you will need to first download the source distribution from the `download` directory.

For compilation you need to install a development package for ODBC (e.g. `unixodbc`) This should be part of any standard distribution. On CERN SLC3 you simply should be able to do:

```
apt-get install unixODBC unixODBC-devel
```

You will then need the `libxml2` development library, which should also be part of any distribution. On SLC3 you can simply install it using

```
apt-get install libxml2-devel
```

For the server you will finally need to install the boost libraries:

```
apt-get install boost-devel
```

The SOAP based service will in addition need `gSOAP`. You can directly download the binary package for Linux. It suffices to unpack to `/opt` to immediately get started.

Now you should be ready to compile and install the **AMGA**(p.??) server:

```
tar xvzf glite-amga-server-1.1.0.tar.gz
cd glite-amga-server-1.1.0
./configure
make
su
make install
```

If you want to make rpm packages for your architecture and install them so that you will be able to deinstall them easily later or install them also on other machines, do

```
tar xvzf glite-amga-server-1.2.2.tar.gz
cd glite-amga-server-1.2.2
./configure --with-readline --enable-rpm-rules --prefix=/opt/glite --without-globus
make rpm
```

You will need to get at least one of the currently supported 4 database backends installed, including their ODBC driver. You have the choice among PostgreSQL, MySQL, Oracle and SQLite:

- PostgreSQL. This is the easiest solution, since ODBC drivers are included in any distribution package. On SLC3 you can install the Postgres ODBC driver using

```
apt-get install postgresql-odbc
```

PostgreSQL needs a little setup: You need to create a database and a user. Access should be made possible via TCP/IP from localhost. The `scripts/init-arda-psqldb.sh` script should be able to do this for you.

- MySQL. Again, ODBC drivers are included in any distribution. On SLC3 you can install the MySQL ODBC driver using

```
apt-get install MyODBC
```

Again, make sure you have a user created which gets access to the database on MySQL.

- SQLite is a file-based database. You need to get the ODBC driver and compile if it is not part of your distribution.

- Oracle can be used from CERN. You need to get the instant client and ODBC driver from Oracle. Which you can freely download after a registration [here](#). . You will need to set up Oracle for the service names at you place. At CERN you might try to learn something from the [Oracle Linux pages](#).

Examples of ODBC configuration files can be found in the `scripts` directory. Copy the `odbc.ini` and `odbcinst.ini` configurations into `/etc` or into your home directory (but then called `.odbc.ini` and `.odbcinst.ini`). In `odbc.ini` you need to configure the database used by the server and which server is being connected to. Examples are given for all 4 databases. The users(and passwords if required) must then be setup in the `amgad.config` file. The ODBC configuration can be checked with e.g. `g0DBCConfig` or `DataManagerII` which are probably installed along the ODBC package or other ODBC clients like OpenOffice. If you don't know about ODBC, some more hints can be found in <http://www.unixodbc.org/doc/User\Manual/> the Unix ODBC User Manual .

Some initial tables need to be setup in the database. You need to run one of

```
sqlplus user/passwd@endpoint <scripts/createInitial.sql
psql -User database <scripts/createInitial.sql
sqlite3 dbfile.db <scripts/createInitial.sql
mysql database <scripts/createInitialMySQL.sql
```

to setup the database.

You should now proceed to configure the server (**Configuring the AMGA Server and the Replicatin Daemon(p.??)**) and start it up.

The following database and ODBC versions are known to work:

- SQLite 3.2.1, but not 3.2.7 with the 0.64 and 0.65 ODBC drivers. In 3.2.7 the ODBC driver is incompatible to the library.
- MySQL 4.0.x and 4.1.x work, 3.x does not.
- PostgreSQL works starting from version 7.2, however only one attribute can be added at a time in versions before 8.0. Explicitely tested were versions 7.2, 7.3, 7.4 and 8.0 allways in their latest sub-versions.

12 Using the C++ Client API

There are two different C++ client APIs available for the **AMGA(p.??)** metadata service. One is through the `md_api` which provides several api functions, the other is directly through the `MDClient` class which also serves as a backend to the `md_api`.

The `MDClient` class offers an interface which allows to issue **AMGA(p.??)** commands directly but does not understand the semantics of the commands and thus does not parse the responses of the server into suitable structures, while this is done by the `md_api`. However, the control on the connection to the server is much better in the case of the `MDClient` class, for example it allows you to abort a query easily. It may also happen that some commands are not available in the `md_api` yet.

In any case, both ways to access the metadata service from C++ depend on an existing and accessible `mdclient.config` file being either in the current working directory or in the home directory as `~/.mdclient.config`. See **Configuration of the C++ and Java command line clients(p.??)** for explanations how to set up the client configuration.

The following is an example of a program using the `md_api` to

```
#include "client/md_api.h"
#include <iostream>

int main (int argc, char *argv[])
{
    std::cout << "Listing attributes of /test\";
    std::list< std::string > attrList;
    std::list< std::string > types;
    if( (res=listAttr("/test", attrList, types)) == 0){
        std::cout << " Result:" << std::endl;
```


All capabilities of the `MDCClient` like cancellation of requests or the catching of `CTRL_C` are explained in the reference at <http://project-arda-dev.web.cern.ch/project-arda-dev/metadata/class\MDCClient.html> a short(!) example of how to make use of them is the `mdclient.cc` program itself.

13 Using the Java Client API

The `AMGA(p.??)` Java API is distributed in two forms. As an RPM and as a tar ball. The tar ball is provided so that the Java API can be used in other platforms other than Linux, including Windows and MacOS.

To use the Java API it is necessary to include the `glite-amga-api-java.jar` file in the classpath. If the Java API was installed from the RPM, then this file is typically located at `<GLITE_HOME>/share/java`, where `<GLITE_HOME>` is the base directory where the gLite software is installed (typically, `/opt/glite`). If the Java API was installed directly from the tar ball available on the `AMGA(p.??) Web Site(p.??)`, the `glite-amga-api-java.jar` is located on the top level directory to where the tar ball was unpacked.

A jar can be included in the classpath in two ways: by setting the `CLASSPATH` environment variable or by using the `-classpath` option in the command line arguments when running java.

To set the classpath variable:

- Unix (bash): `export CLASSPATH=./glite-amga-api-java.jar`
- Windows: `set CLASSPATH=./glite-amga-api-java.jar`

After setting the `CLASSPATH`, to run a Java program it is only necessary to do the following to run a class called, for instance, `QueryMetadata`:

```
java QueryMetadata
```

To specify the jar file directly on the command, one must do:
Unix:

```
java -classpath ./glite-amga-api-java.jar QueryMetadata
```

On Windows the command line is similar, except the path separator is `;` instead of `./`.

The Javadocs for the Java API can be found here <http://project-arda-dev.web.cern.ch/project-arda-dev/metadata/java>.

Like the C++ API, the Java API can be used in two ways. Either through the higher-level interface exposed by the class `arda.md.javaclient.MDCClient` or by sending the commands directly to the server using the low-level API in `arda.md.javaclient.MDServerConnection`. Next are the two examples given for the C++ client API rewritten using the Java API. The first uses the higher-level Java API.

```
import java.io.IOException;
import arda.md.javaclient.*;

public class MDJavaAPI {

    public static void main(String[] args) throws IOException {
        MDServerConnection serverConn = new MDServerConnection(
            MDServerConnectionContext.loadDefaultConfiguration());
        MDCClient mdClient = new MDCClient(serverConn);

        System.out.println("Listing attributes of /test");
        try {
            AttributeDef[] attrs = mdClient.listAttr("/test");
            System.out.println("Result: ");
            for (int i = 0; i < attrs.length; i++) {
                System.out.println(">" + attrs[i].name + ":" + attrs[i].type);
            }
        } catch (CommandException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

```

    }

    System.out.println("Getting gen and events attributes of /test");
    try {
        String[] keys = {"gen", "events"};
        NamedAttributesIterator attrs = mdClient.getAttr("/test", keys);
        while (attrs.hasNext()) {
            NamedAttributes entry = attrs.next();
            System.out.println("File: " + entry.getEntryName());
            String[] keys1 = entry.getKeys();
            for (int i = 0; i < keys1.length; i++) {
                System.out.println(" >" + keys1[i] + "=" + entry.getValue(keys1[i]));
            }
        }
    } catch (CommandException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
}
}

```

The following example uses the low-level API directly:

```

import java.io.IOException;
import arda.md.javaclient.*;

public class DirectServerConnection {

    public static void main(String[] args) throws IOException
    {
        // Loads default configuration and connects to server
        MDServerConnection serverConn = new MDServerConnection(
            MDServerConnectionContext.loadDefaultConfiguration());
        try {
            serverConn.execute("pwd");
            while (!serverConn.eot()) {
                String row = serverConn.fetchRow();
                System.out.println(">" + row);
            }
        } catch (CommandException e) {
            System.out.println("Error executing command: " + e.getMessage());
        }
    }
}
}

```

14 Using the Python Client API

The Python client for **AMGA**(p.??) is distributed in the in the `glite.amga.api-python` RPM. After installation the `amga` package is available with the `mdclient` and `mdinterface` modules. The `mdclient` class offers an interface similar to the `MDCClient` interface in C++ plus methods for most **AMGA**(p.??) command. All arguments are automatically quoted before being sent to the server.

The following is an example script which creates a directory, cd's into it and then gets the "sin" and "events" attributes of all entries in the directory.

```

#!/usr/bin/env python

#import amga classes
from amga import mdclient, mdinterface

#instantiate an AMGA client connecting to localhost:8822 as 'guest'
client = mdclient.MDCClient('localhost', 8822, 'guest')

try:
    print "Creating directory /pytest ..."
    client.createDir("/pytest")

```

```

except mdinterface.CommandException, ex:
    print "Error:", ex

try:
    print "cd /pytest"
    client.cd("/pytest")
except mdinterface.CommandException, ex:
    print "Error:", ex

try:
    print "Getting all attributes of the files in /pytest..."
    client.getattr('/pytest', ['sin', 'events'])
    while not client.eot():
        file, values=client.getEntry()
        print "->",file, values
except mdinterface.CommandException, ex:
    print "Error:", ex

```

15 Definition of the Client Server Protocol

The protocol is a streamed ASCII protocol which is line oriented. Three bytes are special control characters: \n (012) is the line-ending byte which needs to be attached to any line, EOT (004) is the end of transmission sent by the server after any response because server responses can have many (also empty) lines and CAN (030) is the cancel byte which can be sent out-of-band by the client to abort the request or inline in the servers response if during response processing an error occurs.

This defines the full client server protocol including the handshaking with four example commands(first one OK, second cancelled by user, third has an execution error, fourth is cancelled by the server):

| SERVER | CLIENT |
|---|---------------------------|
| Greeting\n | |
| Protocol <protocol number>\n | |
| <space sep. list of serv opts>\n | |
| | <requested serv option>\n |
| | <more opts>\n |
| | <...>\n |
| | \n |
| OK\n | |
| ----- SSL handshaking if required ----- | |
| SSL_accept() | SSL_connect() |
| ----- | ----- |
| | <command> |
| 0\n | |
| <line 1 of response>\n | |
| ... | |
| <line n of response>\n | |
| EOT[<session-id>]EOT | |
| | <command> |
| <error-code> <literal err.>\n | |
| ... | |
| CAN | CAN (Out-of-Band!) |
| <abort error code>\n | |
| EOT[<session-id>]EOT | |
| | <command> |
| <err-code> <comment>\n | |
| EOT[<session-id>]EOT | |
| | <command> |
| 0\n | |
| <line 1 of response>\n | |
| ... | |
| CAN (timeouts, back end-error...) | |
| <err-code> <comment>\n | |
| EOT[<session-id>]EOT | |

Remarks: The greeting is a free-form greeting string sent by the server. The client cannot make any assumptions about the content, apart that it ends in a version number, which may contain dots. The intention is to allow the client to display this greeting. The client should only depend on the protocol version number, which is integer for any assumptions on the protocol or the available calls.

As of protocol version 2 the server options can include "plain", "ssl" and "statistics" giving the possible ways for the connection security/encryption. The statistics option, used for monitoring is explained in the **Monitoring support in AMGA**(p.??) section.

The requested server option sent by the client is either "plain" or "ssl", requesting an ssl connection. Alternatively the client may resume a session by responding "resumeSSL<sessionID>" or "resume<session-ID>". The other options sent one by one on a line consist of an integer number and the value for that option:

- 0
- 1
- 2
- 3
- 4
- 5 Where the login name is the requested user id string, the full name is entirely optional and should contain the full user name of the user as in the comment field of /etc/passwd. The group and user permission masks are 3 character fields like umask. All of these fields are optional, unless e.g. the server requires that particular user to use a password for authentication.

In case a session is resumed, the server ignores any additional options requested by the client, and will expect a command next if no SSL session is requested, that is the "OK\\n" will only be sent if SSL is being used. This reduces round-trips in case of No-SSL, but the OK is necessary to synchronize SSL startup in case of an SSL session.

Any authentication failure is sent by the server as the response to the first command, to save round-trips in the case of sessions (see below) and because the result of the authentication may only be known after the initialization of the SSL session if the certificate is being used for authentication.

Commands are strings terminated by a newline as explained in the **Metadata Access from the Shell**(p.??) section . Sending a wrong command does not violate the protocol, but results in an error "3 Illegal command". Three special commands are part of the protocol:

- quit Asks the server to close the connection.
- exit The same as quit.
- close Ask the server to send a session ID and then close the connection.

Every command is answered by an integer error code, which is 0 in case of everything OK. Error codes are not part of the protocol, but part of the responses to commands.

If sessions are forced on the server, then the server will always send a sessionID at the end of any response to the command and then simply close the connection. This also saves round-trips.

The CAN bytes to interrupt the streaming of a result sent by the server are sent in a TCP out-of-band message. However if the server decides to interrupt the streaming of a response the CAN byte is sent in-band.

16 Monitoring support in AMGA

AMGA(p.??) supports monitoring via the <http://sourceforge.net/projects/monami/> MonAMI monitoring service which can also forward information into MonALISA. The following is an example script in Python which requests information from an **AMGA**(p.??) server. The server is contacted on the standard port:

```

#!/usr/bin/env python
import socket

# Open TCP socket to AMGA server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("localhost", 8822))

# Request statistics
s.send('statistics\n\n')

# Read response
result = ''
while 1:
    r = s.recv(1000)
    result = result + r
    if not r:
        break

start = result.find('<Service>') # Skip response header
result = result[start:]
print result

```

The server is contacted and monitoring is requested via the "statistics\n\n" command. In the response of the server the server greeting needs to be skipped. The rest of the result is then an XML encoded status report according to the GLUE schemas as described [here](#).

The following is an example result:

```

<Service>
<Name>AMGA</Name>
<Version>1.1.0</Version>
<Data>
  <Key>MaxConnections</Key>
  <Value>50</Value>
  <Key>PreparedConnections</Key>
  <Value>1</Value>
  <Key>UsedConnections</Key>
  <Value>1</Value>
  <Key>MaxSessions</Key>
  <Value>1024</Value>
  <Key>UsedSessions</Key>
  <Value>0</Value>
  <Key>SessionStorage</Key>
  <Value>shm</Value>
</Data>
</Service>

```