MySQL Proxy Guide

MySQL Proxy Guide

Abstract

This is the MySQL Proxy extract from the MySQL Reference Manual.

Document generated on: 2009-06-02 (revision: 15161)

MySQL Proxy

The MySQL Proxy is an application that communicates over the network using the MySQL Network Protocol and provides communication between one or more MySQL servers and one or more MySQL clients. In the most basic configuration, MySQL Proxy simply passes on queries from the client to the MySQL Server and returns the responses from the MySQL Server to the client.

Because MySQL Proxy uses the MySQL network protocol, any MySQL compatible client (include the command line client, any clients using the MySQL client libraries, and any connector that supports the MySQL network protocol) can connect to the proxy without modification.

In addition to the basic pass-through configuration, the MySQL Proxy is also capable of monitoring and altering the communication between the client and the server. This interception of the queries enables you to add profiling, and the interception of the exchanges is scriptable using the Lua scripting language.

By intercepting the queries from the client, the proxy can insert additional queries into the list of queries sent to the server, and remove the additional results when they are returned by the server. Using this functionality you can add informational statements to each query, for example to monitor their execution time or progress, and separately log the results, while still returning the results from the original query to the client.

The proxy allows you to perform additional monitoring, filtering or manipulation on queries without you having to make any modifications to the client and without the client even being aware that it is communicating with anything but a genuine MySQL server.

Warning

MySQL Proxy is currently an Alpha release and should not be used within production environments.

Important

MySQL Proxy is compatible with MySQL 5.0.x or later. Testing has not been performed with Version 4.1. Please provide feedback on your experiences via the MySQL Proxy Forum.

Chapter 1. MySQL Proxy Supported Platforms

MySQL Proxy is currently available as a pre-compiled binary for the following platforms:

- Linux (including RedHat, Fedora, Debian, SuSE) and derivatives.
- Mac OS X
- FreeBSD
- IBM AIX
- Sun Solaris
- Microsoft Windows (including Microsoft Windows XP, and Microsoft Windows Server 2003)

Other Unix/Linux platforms not listed should be compatible by using the source package and building MySQL Proxy locally.

System requirements for the MySQL Proxy application are the same as the main MySQL server. Currently MySQL Proxy is compatible only with MySQL 5.0.1 and later. MySQL Proxy is provided as a standalone, statically linked binary. You do not need to have MySQL or Lua installed.

Chapter 2. Installing MySQL Proxy

You have three choices for installing MySQL Proxy:

- Pre-compiled binaries are available for a number of different platforms. See Section 2.1, "Installing MySQL Proxy from a binary distribution".
- You can install from the source code if you want to build on an environment not supported by the binary distributions. See Section 2.2, "Installing MySQL Proxy from a source distribution".
- The latest version of the MySQL proxy source code is available through a development repository is the best way to stay up to date with the latest fixes and revisions. See Section 2.3, "Installing MySQL Proxy from the Subversion repository".

2.1. Installing MySQL Proxy from a binary distribution

If you download the binary packages then you need only to extract the package and then copy the `mysql-proxy` file to your desired location. For example:

```
shell> tar zxf mysql-proxy-0.5.0.tar.gz
shell> cp ./mysql-proxy-0.5.0/sbin/mysql-proxy /usr/local/sbin
```

2.2. Installing MySQL Proxy from a source distribution

If you have downloaded the source package then you will need to compile the MySQL Proxy before using it. To build you will need to have the following installed:

- libevent 1.x or higher (1.3b or later is preferred)
- lua 5.1.x or higher
- glib2 2.6.0 or higher
- pkg-config
- MySQL 5.0.x or higher developer files

Note

On some operating systems you may need to manually build the required components to get the latest version. If you are having trouble compiling MySQL Proxy then consider using one of the binary distributions.

Once these components are installed, you need to configure and then build:

```
shell> tar zxf mysql-proxy-0.5.0.tar.gz
shell> cd mysql-proxy-0.5.0
shell> ./configure
shell> make
```

If you want to test the build, then use the `check` target to `make`:

```
shell> make check
```

The tests try to connect to `localhost` using the `root` user. If you need to provide a password, set the `MYSQL_PASSWORD` environment variable:

```
shell> MYSQL_PASSWORD=root_pwd make check
```

You can install using the `install` target:

```
shell> make install
```

By default `mysql-proxy` is installed into `/usr/local/sbin/mysql-proxy`. The Lua example scripts are copied into `/usr/local/share`.

2.3. Installing MySQL Proxy from the Subversion repository

The MySQL Proxy source is available through a public Subversion repository and is the quickest way to get hold of the latest releases and fixes.

To build from the Subversion repository, you need the following components already installed:

- Subversion 1.3.0 or higher
- `libtool` 1.5 or higher
- `autoconf` 2.56 or higher
- `automake` 1.9 or higher
- `libevent` 1.x or higher (1.3b or later is preferred)
- `lua` 5.1.x or higher
- `glib2` 2.4.0 or higher
- `pkg-config`
- `MySQL` 5.0.x or higher developer files

To checkout a local copy of the Subversion repository, use `svn`:

```
shell> svn co http://svn.MySQL.com/svnpublic/mysql-proxy/ mysql-proxy
```

The above command will download a complete version of the Subversion repository for `mysql-proxy`. The main source files are located within the `trunk` subdirectory. The configuration scripts need to be generated before you can configure and build `mysql-proxy`. The `autogen.sh` script will generate the configuration scripts for you:

```
shell> sh ./autogen.sh
```

The script creates the standard `configure` script, which you can then use to configure and build with `make`:

```
shell> ./configure
shell> make
shell> make install
```

If you want to create a standalone source distribution, identical to the source distribution available for download:

```
shell> make distcheck
```

The above will create the file `mysql-proxy-`*`0.5.0`*`.tar.gz` within the current directory.

Chapter 3. MySQL Proxy Command-Line Options

To start `mysql-proxy` you can just run the command directly. However, for most situations you will want to specify at the very least the address/host name and port number of the backend MySQL server to which the MySQL Proxy should pass on queries.

You can get a list of the supported command-line options using the `--help-all` command-line option. The majority of these options set up the environment, either in terms of the address/port number that `mysql-proxy` should listen on for connections, or the onward connection to a MySQL server. A full description of the options is shown below:

- `--help-all` — show all help options.
- `--help-admin` — show options for the admin-module.
- `--help-proxy` — Show options for the proxy-module.
- `--admin-address=host:port` — specify the host name (or IP address) and port for the administration port. The default is `localhost:4041`.
- `--proxy-address=host:port` — the listening host name (or IP address) and port of the proxy server. The default is `localhost:4040`.
- `--proxy-read-only-backend-address=host:port` — the listening host name (or IP address) and port of the proxy server for read-only connections. The default is for this information not to be set.
- `--proxy-backend-addresses=host:port` — the host name (or IP address) and port of the MySQL server to connect to. You can specify multiple backend servers by supplying multiple options. Clients are connected to each backend server in round-robin fashion. For example, if you specify two servers A and B, the first client connection will go to server A; the second client connection to server B and the third client connection to server A.
- `--proxy-skip-profiling` — disables profiling of queries (tracking time statistics). The default is for tracking to be enabled.
- `--proxy-fix-bug-25371` — gets round an issue when connecting to a MySQL server later than 5.1.12 when using a MySQL client library of any earlier version.
- `--proxy-lua-script=file` — specify the Lua script file to be loaded. Note that the script file is not physically loaded and parsed until a connection is made. Also note that the specified Lua script is reloaded for each connection; if the content of the Lua script changes while `mysql-proxy` is running then the updated content will automatically be used when a new connection is made.
- `--daemon` — starts the proxy in daemon mode.
- `--pid-file=file` — sets the name of the file to be used to store the process ID.
- `--version` — show the version number.

The most common usage is as a simple proxy service (i.e. without addition scripting). For basic proxy operation you must specify at least one `proxy-backend-addresses` option to specify the MySQL server to connect to by default:

```
shell> mysql-proxy
--proxy-backend-addresses=MySQL.example.com:3306
```

The default proxy port is `4040`, so you can connect to your MySQL server through the proxy by specifying the host name and port details:

```
shell> mysql --host=localhost --port=4040
```

If your server requires authentication information then this will be passed through natively without alteration by `mysql-proxy`, so you must also specify the authentication information if required:

```
shell> mysql --host=localhost --port=4040 \
   --user=username --password=password
```

You can also connect to a read-only port (which filters out `UPDATE` and `INSERT` queries) by connecting to the read-only port. By default the host name is the default, and the port is `4042`, but you can alter the host/port information by using the `--proxy-read-only-address` command-line option.

For more detailed information on how to use these command line options, and `mysql-proxy` in general in combination with Lua scripts, see Chapter 5, *Using MySQL Proxy*.

Chapter 4. MySQL Proxy Scripting

You can control how MySQL Proxy manipulates and works with the queries and results that are passed on to the MySQL server through the use of the embedded Lua scripting language. You can find out more about the Lua programming language from the Lua Website.

The primary interaction between MySQL Proxy and the server is provided by defining one or more functions through an Lua script. A number of functions are supported, according to different events and operations in the communication sequence between a client and one or more backend MySQL servers:

- `connect_server()` — this function is called each time a connection is made to MySQL Proxy from a client. You can use this function during load-balancing to intercept the original connection and decide which server the client should ultimately be attached to. If you do not define a special solution, then a simple round-robin style distribution is used by default.
- `read_handshake()` — this function is called when the initial handshake information is returned by the server. You can capture the handshake information returned and provide additional checks before the authorization exchange takes place.
- `read_auth()` — this function is called when the authorization packet (user name, password, default database) are submitted by the client to the server for authentication.
- `read_auth_result()` — this function is called when the server returns an authorization packet to the client indicating whether the authorization succeeded.
- `read_query()` — this function is called each time a query is sent by the client to the server. You can use this to edit and manipulate the original query, including adding new queries before and after the original statement. You can also use this function to return information directly to the client, bypassing the server, which can be useful to filter unwanted queries or queries that exceed known limits.
- `read_query_result()` — this function is called each time a result is returned from the server, providing you have manually injected queries into the query queue. If you have not explicitly inject queries within the `read_query()` function then this function is not triggered. You can use this to edit the result set, or to remove or filter the result sets generated from additional queries you injected into the queue when using `read_query()`.

The table below describes the direction of flow of information at the point when the function is triggered.

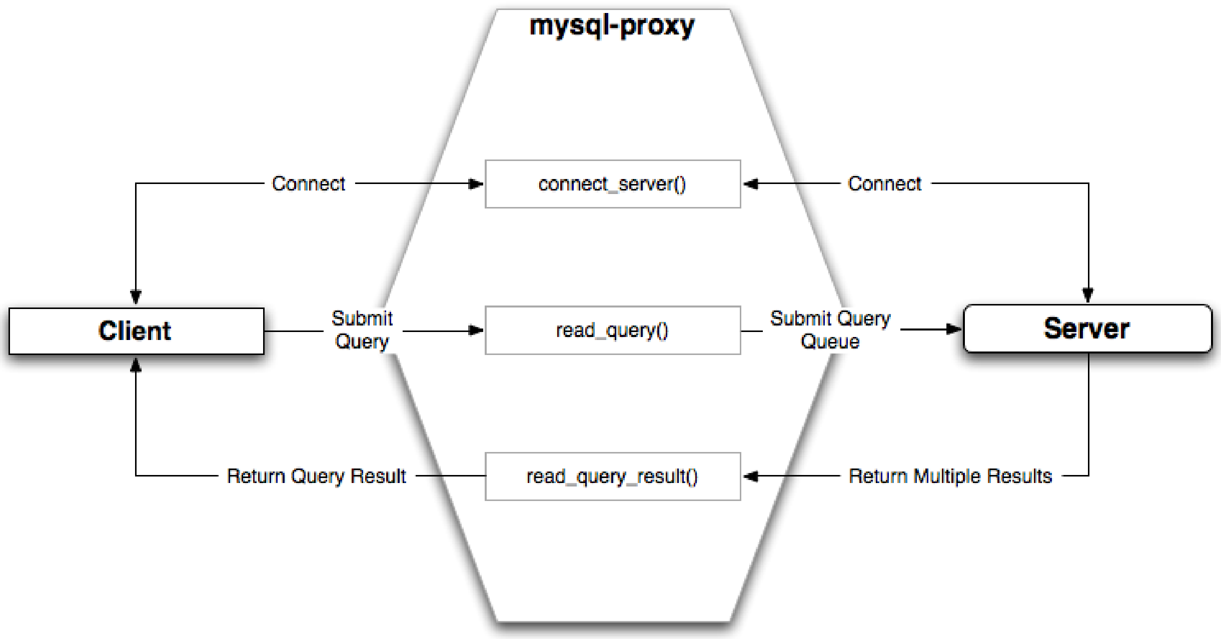
Function	**Supplied Information**	**Direction**
`connect_server()`	None	Client to Server
`read_handshake()`	Handshake packet	Server to Client
`read_auth()`	Authorization packet	Client to Server
`read_auth_result()`	Authorization response	Server to Client
`read_query()`	Query	Client to Server
`read_query_result()`	Query result	Server to Client

By default, all functions return a result that indicates that the data should be passed on to the client or server (depending on the direction of the information being transferred). This return value can be overridden by explicitly returning a constant indicating that a particular response should be sent. For example, it is possible to construct result set information by hand within `read_query()` and to return the resultset directly to the client without ever sending the original query to the server.

In addition to these functions, a number of built-in structures provide control over how MySQL Proxy forwards on queries and returns the results by providing a simplified interface to elements such as the list of queries and the groups of result sets that are returned.

4.1. Proxy Scripting Sequence During Query Injection

The figure below gives an example of how the proxy might be used when injecting queries into the query queue. Because the proxy sits between the client and MySQL server, what the proxy sends to the server, and the information that the proxy ultimately returns to the client do not have to match or correlate. Once the client has connected to the proxy, the following sequence occurs for each individual query sent by the client.



1. The client submits one query to the proxy, the `read_query()` function within the proxy is triggered. The function adds the query to the query queue.
2. Once manipulation by `read_query()` has completed, the queries are submitted, sequentially, to the MySQL server.
3. The MySQL server returns the results from each query, one result set for each query submitted. The `read_query_result()` function is triggered for each result set, and each invocation can decide which result set to return to the client

For example, you can queue additional queries into the global query queue to be processed by the server. This can be used to add statistical information by adding queries before and after the original query, changing the original query:

```
SELECT * FROM City;
```

Into a sequence of queries:

```
SELECT NOW();
SELECT * FROM City;
SELECT NOW();
```

You can also modify the original statement, for example to add `EXPLAIN` to each statement executed to get information on how the statement was processed, again altering our original SQL statement into a number of statements:

```
SELECT * FROM City;
EXPLAIN SELECT * FROM City;
```

In both of these examples, the client would have received more result sets than expected. Regardless of how you manipulate the incoming query and the returned result, the number of queries returned by the proxy must match the number of original queries sent by the client.

You could adjust the client to handle the multiple result sets sent by the proxy, but in most cases you will want the existence of the proxy to remain transparent. To ensure that the number of queries and result sets match, you can use the MySQL Proxy `read_query_result()` to extract the additional result set information and return only the result set the client originally requested back to the client. You can achieve this by giving each query that you add to the query queue a unique ID, and then filter out queries that do not match the original query ID when processing them with `read_query_result()`.

4.2. Internal Structures

There are a number of internal structures within the scripting element of MySQL Proxy. The primary structure is `proxy` and this provides an interface to the many common structures used throughout the script, such as connection lists and configured backend servers. Other structures, such as the incoming packet from the client and result sets are only available within the context of one of the scriptable functions.

Attribute	Description
`connection`	A structure containing the active client connections. For a list of attributes, see `proxy.connection`.
`servers`	A structure containing the list of configured backend servers. For a list of attributes, see `proxy.backends`.
`queries`	A structure containing the queue of queries that will be sent to the server during a single client query. For a list of attributes, see `proxy.queries`.
`PROXY_VERSION`	The version number of MySQL Proxy, encoded in hex. You can use this to check that the version number supports a particular option from within the Lua script. Note that the value is encoded as a hex value, so to check the version is at least 0.5.1 you compare against `0x00501`.

`proxy.connection`

The `proxy.connection` object is read only, and provides information about the current connection.

Attribute	Description
`thread_id`	The thread ID of the connection.
`backend_ndx`	The ID of the server used for this connection. This is an ID valid against the list of configured servers available through the `proxy.backends` object.

`proxy.backends`

The `proxy.backends` table is partially writable and contains an array of all the configured backend servers and the server metadata (IP address, status, etc.). You can determine the array index of the current connection using `proxy.connection["backend_ndx"]` which is the index into this table of the backend server being used by the active connection.

The attributes for each entry within the `proxy.backends` table are shown in this table.

Attribute	Description
`address`	The host name/port combination used for this connection
`connected_clients`	The number of clients currently connected.
`state`	The status of the backend server. See Section 4.2, “Internal Structures” [10]

`proxy.queries`

The `proxy.queries` object is a queue representing the list of queries to be sent to the server. The queue is not populated automatically, but if you do not explicitly populate the queue then queries are passed on to the backend server verbatim. Also, if you do not populate the query queue by hand, then the `read_query_result()` function is not triggered.

The following methods are supported for populating the `proxy.queries` object.

Function	Description
`append(id,packet)`	Appends a query to the end of the query queue. The `id` is an integer identifier that you can use to recognize the query results when they are returned by the server. The packet should be a properly formatted query packet.
`prepend(id,packet)`	Prepends a query to the query queue. The `id` is an identifier that you can use to recognize the query results when they are returned by the server. The packet should be a properly formatted query packet.
`reset()`	Empties the query queue.
`len()`	Returns the number of query packets in the queue.

For example, you could append a query packet to the `proxy.queries` queue by using the `append()`:

```
proxy.queries:append(1,packet)
```

`proxy.response`

The `proxy.response` structure is used when you want to return your own MySQL response, instead of forwarding a packet that you have received a backend server. The structure holds the response type information, an optional error message, and the result set (rows/columns) that you want to return.

Attribute	Description
`type`	The type of the response. The type must be either `MYSQLD_PACKET_OK` or `MYSQLD_PACKET_ERR`. If the `MYSQLD_PACKET_ERR`, then you should set the value of the `mysql.response.errmsg` with a suitable error message.
`errmsg`	A string containing the error message that will be returned to the client.
`resultset`	A structure containing the result set information (columns and rows), identical to what would be returned when returning a results from a `SELECT` query.

When using `proxy.response` you either set `proxy.response.type` to `proxy.MYSQLD_PACKET_OK` and then build `resultset` to contain the results that you want to return, or set `proxy.response.type` to `proxy.MYSQLD_PACKET_ERR` and set the `proxy.response.errmsg` to a string with the error message. To send the completed resultset or error message, you should return the `proxy.PROXY_SEND_RESULT` to trigger the return of the packet information.

An example of this can be seen in the `tutorial-resultset.lua` script within the MySQL Proxy package:

```
if string.lower(command) == "show" and string.lower(option) == "querycounter" then
        ---
        -- proxy.PROXY_SEND_RESULT requires
        --
        -- proxy.response.type to be either
        -- * proxy.MYSQLD_PACKET_OK or
        -- * proxy.MYSQLD_PACKET_ERR
        --
        -- for proxy.MYSQLD_PACKET_OK you need a resultset
        -- * fields
        -- * rows
        --
        -- for proxy.MYSQLD_PACKET_ERR
        -- * errmsg
        proxy.response.type = proxy.MYSQLD_PACKET_OK
        proxy.response.resultset = {
                fields = {
                        { type = proxy.MYSQL_TYPE_LONG, name = "global_query_counter", },
                        { type = proxy.MYSQL_TYPE_LONG, name = "query_counter", },
                },
                rows = {
                        { proxy.global.query_counter, query_counter }
                }
        }
        -- we have our result, send it back
        return proxy.PROXY_SEND_RESULT
elseif string.lower(command) == "show" and string.lower(option) == "myerror" then
        proxy.response.type = proxy.MYSQLD_PACKET_ERR
        proxy.response.errmsg = "my first error"
        return proxy.PROXY_SEND_RESULT
```

`proxy.response.resultset`

The `proxy.response.resultset` structure should be populated with the rows and columns of data that you want to return. The structure contains the information about the entire result set, with the individual elements of the data shown in the table below.

Attribute	Description
`fields`	The definition of the columns being returned. This should be a dictionary structure with the `type` specifying the MySQL data type, and the `name` specifying the column name. Columns should be listed in the order of the column data that will be returned.
`flags`	A number of flags related to the resultset. Valid flags include `auto_commit` (whether an automatic commit was triggered), `no_good_index_used` (the query executed without using an appropriate index), and `no_index_used` (the query executed without using any index).
`rows`	The actual row data. The information should be returned as an array of arrays. Each inner array should contain the column data, with the outer array making up the entire result set.
`warning_count`	The number of warnings for this result set.
`affected_rows`	The number of rows affected by the original statement.
`insert_id`	The last insert ID for an auto-incremented column in a table.
`query_status`	The status of the query operation. You can use the `MYSQLD_PACKET_OK` or `MYSQLD_PACKET_ERR` constants to populate this parameter.

For an example of the population of this table, see Section 4.2, “Internal Structures” [8].

Proxy Return State Constants

The following constants are used internally by the proxy to specify the response to send to the client or server. All constants are exposed as values within the main `proxy` table.

Constant	Description
`PROXY_SEND_QUERY`	Causes the proxy to send the current contents of the queries queue to the server.
`PROXY_SEND_RESULT`	Causes the proxy to send a result set back to the client.
`PROXY_IGNORE_RESULT`	Causes the proxy to drop the result set (nothing is returned to the client).

As constants, these entities are available without qualification in the Lua scripts. For example, at the end of the `read_query_result()` you might return `PROXY_IGNORE_RESULT:`

```
return proxy.PROXY_IGNORE_RESULT
```

Packet State Constants

The following states describe the status of a network packet. These items are entries within the main `proxy` table.

Constant	Description
`MYSQLD_PACKET_OK`	The packet is OK.
`MYSQLD_PACKET_ERR`	The packet contains error information.
`MYSQLD_PACKET_RAW`	The packet contains raw data.

Backend State/Type Constants

The following constants are used either to define the status of the backend server (the MySQL server to which the proxy is connected) or the type of backend server. These items are entries within the main `proxy` table.

Constant	Description
`BACKEND_STATE_UNKNOWN`	The current status is unknown.
`BACKEND_STATE_UP`	The backend is known to be up (available).
`BACKEND_STATE_DOWN`	The backend is known to be down (unavailable).
`BACKEND_TYPE_UNKNOWN`	Backend type is unknown.
`BACKEND_TYPE_RW`	Backend is available for read/write.
`BACKEND_TYPE_RO`	Backend is available only for read-only use.

Server Command Constants

The following values are used in the packets exchanged between the client and server to identify the information in the rest of the packet. These items are entries within the main `proxy` table. The packet type is defined as the first character in the sent packet. For example, when intercepting packets from the client to edit or monitor a query you would check that the first byte of the packet was of type `proxy.COM_QUERY`.

Constant	Description
`COM_SLEEP`	Sleep
`COM_QUIT`	Quit
`COM_INIT_DB`	Initialize database
`COM_QUERY`	Query
`COM_FIELD_LIST`	Field List
`COM_CREATE_DB`	Create database
`COM_DROP_DB`	Drop database
`COM_REFRESH`	Refresh
`COM_SHUTDOWN`	Shutdown

Constant	Description
COM_STATISTICS	Statistics
COM_PROCESS_INFO	Process List
COM_CONNECT	Connect
COM_PROCESS_KILL	Kill
COM_DEBUG	Debug
COM_PING	Ping
COM_TIME	Time
COM_DELAYED_INSERT	Delayed insert
COM_CHANGE_USER	Change user
COM_BINLOG_DUMP	Binlog dump
COM_TABLE_DUMP	Table dump
COM_CONNECT_OUT	Connect out
COM_REGISTER_SLAVE	Register slave
COM_STMT_PREPARE	Prepare server-side statement
COM_STMT_EXECUTE	Execute server-side statement
COM_STMT_SEND_LONG_DATA	Long data
COM_STMT_CLOSE	Close server-side statement
COM_STMT_RESET	Reset statement
COM_SET_OPTION	Set option
COM_STMT_FETCH	Fetch statement
COM_DAEMON	Daemon (MySQL 5.1 only)
COM_ERROR	Error

MySQL Type Constants

These constants are used to identify the field types in the query result data returned to clients from the result of a query. These items are entries within the main `proxy` table.

Constant	Field Type
MYSQL_TYPE_DECIMAL	Decimal
MYSQL_TYPE_NEWDECIMAL	Decimal (MySQL 5.0 or later)
MYSQL_TYPE_TINY	Tiny
MYSQL_TYPE_SHORT	Short
MYSQL_TYPE_LONG	Long
MYSQL_TYPE_FLOAT	Float
MYSQL_TYPE_DOUBLE	Double
MYSQL_TYPE_NULL	Null
MYSQL_TYPE_TIMESTAMP	Timestamp
MYSQL_TYPE_LONGLONG	Long long
MYSQL_TYPE_INT24	Integer
MYSQL_TYPE_DATE	Date
MYSQL_TYPE_TIME	Time
MYSQL_TYPE_DATETIME	Datetime
MYSQL_TYPE_YEAR	Year
MYSQL_TYPE_NEWDATE	Date (MySQL 5.0 or later)
MYSQL_TYPE_ENUM	Enumeration
MYSQL_TYPE_SET	Set
MYSQL_TYPE_TINY_BLOB	Tiny Blob
MYSQL_TYPE_MEDIUM_BLOB	Medium Blob

Constant	Field Type
MYSQL_TYPE_LONG_BLOB	Long Blob
MYSQL_TYPE_BLOB	Blob
MYSQL_TYPE_VAR_STRING	Varstring
MYSQL_TYPE_STRING	String
MYSQL_TYPE_TINY	Tiny (compatible with MYSQL_TYPE_CHAR)
MYSQL_TYPE_ENUM	Enumeration (compatible with MYSQL_TYPE_INTERVAL)
MYSQL_TYPE_GEOMETRY	Geometry
MYSQL_TYPE_BIT	Bit

4.3. Capturing a connection with connect_server()

When the proxy accepts a connection from a MySQL client, the connect_server() function is called.

There are no arguments to the function, but you can use and if necessary manipulate the information in the proxy.connection table, which is unique to each client session.

For example, if you have multiple backend servers then you can set the server to be used by that connection by setting the value of proxy.connection.backend_ndx to a valid server number. The code below will choose between two servers based on whether the current time in minutes is odd or even:

```
function connect_server()
        print("--> a client really wants to talk to a server")
        if (tonumber(os.date("%M")) % 2 == 0) then
                proxy.connection.backend_ndx = 2
                print("Choosing backend 2")
        else
                proxy.connection.backend_ndx = 1
                print("Choosing backend 1")
        end
        print("Using " .. proxy.backends[proxy.connection.backend_ndx].address)
end
```

In this example the IP address/port combination is also displayed by accessing the information from the internal proxy.backends table.

4.4. Examining the handshake with read_handshake()

Handshake information is sent by the server to the client after the initial connection (through connect_server()) has been made. The handshake information contains details about the MySQL version, the ID of the thread that will handle the connection information, and the IP address of the client and server. This information is exposed through a Lua table as the only argument to the function.

- mysqld_version — the version of the MySQL server.
- thread_id — the thread ID.
- scramble — the password scramble buffer.
- server_addr — the IP address of the server.
- client_addr — the IP address of the client.

For example, you can print out the handshake data and refuse clients by IP address with the following function:

```
function read_handshake( auth )
        print("<-- let's send him some information about us")
        print("    mysqld-version: " .. auth.mysqld_version)
        print("    thread-id     : " .. auth.thread_id)
        print("    scramble-buf  : " .. string.format("%q", auth.scramble))
        print("    server-addr   : " .. auth.server_addr)
        print("    client-addr   : " .. auth.client_addr)
        if not auth.client_addr:match("^127.0.0.1:") then
                proxy.response.type = proxy.MYSQLD_PACKET_ERR
                proxy.response.errmsg = "only local connects are allowed"
                print("we don't like this client");
                return proxy.PROXY_SEND_RESULT
        end
end
```

Note that you have to return an error packet to the client by using proxy.PROXY_SEND_RESULT.

4.5. Examining the authentication credentials with read_auth()

The read_auth() function is triggered when an authentication handshake is initiated by the client. In the execution sequence, read_auth() occurs immediately after read_handshake(), so the server selection has already been made, but the connection and authorization information has not yet been provided to the backend server.

The function accepts a single argument, an Lua table containing the authorization information for the handshake process. The entries in the table are:

- username — the user login for connecting to the server.
- password — the password, encrypted, to be used when connecting.
- default_db — the default database to be used once the connection has been made.

For example, you can print the user name and password supplied during authorization using:

```
function read_auth( auth )
        print("    username      : " .. auth.username)
        print("    password      : " .. string.format("%q", auth.password))
end
```

You can interrupt the authentication process within this function and return an error packet back to the client by constructing a new packet and returning proxy.PROXY_SEND_RESULT:

```
proxy.response.type = proxy.MYSQLD_PACKET_ERR
proxy.response.errmsg = "Logins are not allowed"
return proxy.PROXY_SEND_RESULT
```

4.6. Accessing authentication information with read_auth_result()

The return packet from the server during authentication is captured by read_auth_result(). The only argument to this function is the authentication packet returned by the server. As the packet is a raw MySQL network protocol packet, you must access the first byte to identify the packet type and contents. The MYSQLD_PACKET_ERR and MYSQLD_PACKET_OK constants can be used to identify whether the authentication was successful:

```
function read_auth_result( auth )
        local state = auth.packet:byte()
        if state == proxy.MYSQLD_PACKET_OK then
                print("<-- auth ok");
        elseif state == proxy.MYSQLD_PACKET_ERR then
                print("<-- auth failed");
        else
                print("<-- auth ... don't know: " .. string.format("%q", auth.packet));
        end
end
```

4.7. Manipulating Queries with read_query()

The read_query() function is called once for each query submitted by the client and accepts a single argument, the query packet that was provided. To access the content of the packet you must parse the packet contents manually.

For example, you can intercept a query packet and print out the contents using the following function definition:

```
function read_query( packet )
        if packet:byte() == proxy.COM_QUERY then
                print("we got a normal query: " .. packet:sub(2))
        end
end
```

This example checks the first byte of the packet to determine the type. If the type is COM_QUERY (see Section 4.2, “Internal Structures” [10]), then we extract the query from the packet and print it out. The structure of the packet type supplied is important. In the case of a COM_QUERY packet, the remaining contents of the packet are the text of the query string. In this example, no changes have been made to the query or the list of queries that will ultimately be sent to the MySQL server.

To modify a query, or add new queries, you must populate the query queue (proxy.queries) and then execute the queries that you have placed into the queue. If you do not modify the original query or the queue, then the query received from the client is sent to the MySQL server verbatim.

When adding queries to the queue, you should follow these guidelines:

- The packets inserted into the queue must be valid query packets. For each packet, you must set the initial byte to the packet type. If you are appending a query, you can append the query statement to the rest of the packet.
- Once you add a query to the queue, the queue is used as the source for queries sent to the server. If you add a query to the queue to add more information, you must also add the original query to the queue or it will not be executed.
- Once the queue has been populated, you must set the return value from `read_query()` to indicate whether the query queue should be sent to the server.
- When you add queries to the queue, you should add an ID. The ID you specify is returned with the result set so that you identify each query and corresponding result set. The ID has no other purpose than as an identifier for correlating the query and resultset. When operating in a passive mode, during profiling for example, you want to identify the original query and the corresponding resultset so that the results expect by the client can be returned correctly.
- Unless your client is designed to cope with more result sets than queries, you should ensure that the number of queries from the client match the number of results sets returned to the client. Using the unique ID and removing result sets you inserted will help.

Normally, the `read_query()` and `read_query_result()` function are used in conjunction with each other to inject additional queries and remove the additional result sets. However, `read_query_result()` is only called if you populate the query queue within `read_query()`.

4.8. Manipulating Results with `read_query_result()`

The `read_query_result()` is called for each result set returned by the server only if you have manually injected queries into the query queue. If you have not manipulated the query queue then this function is not called. The function supports a single argument, the result packet, which provides a number of properties:

- `id` — the ID of the result set, which corresponds to the ID that was set when the query packet was submitted to the server when using `append(id)` on the query queue.
- `query` — the text of the original query.
- `query_time` — the number of microseconds required to receive the first row of a result set.
- `response_time` — the number of microseconds required to receive the last row of the result set.
- `resultset` — the content of the result set data.

By accessing the result information from the MySQL server you can extract the results that match the queries that you injected, return different result sets (for example, from a modified query), and even create your own result sets.

The Lua script below, for example, will output the query, followed by the query time and response time (i.e. the time to execute the query and the time to return the data for the query) for each query sent to the server:

```
function read_query( packet )
        if packet:byte() == proxy.COM_QUERY then
                print("we got a normal query: " .. packet:sub(2))
                proxy.queries:append(1, packet )
                return proxy.PROXY_SEND_QUERY
        end
end
function read_query_result(inj)
        print("query-time: " .. (inj.query_time / 1000) .. "ms")
        print("response-time: " .. (inj.response_time / 1000) .. "ms")
end
```

You can access the rows of returned results from the resultset by accessing the rows property of the resultset property of the result that is exposed through `read_query_result()`. For example, you can iterate over the results showing the first column from each row using this Lua fragment:

```
for row in inj.resultset.rows do
        print("injected query returned: " .. row[1])
end
```

Just like `read_query()`, `read_query_result()` can return different values for each result according to the result returned. If you have injected additional queries into the query queue, for example, then you will want to remove the results returned from those additional queries and only return the results from the query originally submitted by the client.

The example below injects additional `SELECT NOW()` statements into the query queue, giving them a different ID to the ID of the original query. Within `read_query_result()`, if the ID for the injected queries is identified, we display the result row, and return the `proxy.PROXY_IGNORE_RESULT` from the function so that the result is not returned to the client. If the result is from any other query, we print out the query time information for the query and return the default, which passes on the result set unchanged. We could also have explicitly returned `proxy.PROXY_IGNORE_RESULT` to the MySQL client.

```
function read_query( packet )
        if packet:byte() == proxy.COM_QUERY then
                proxy.queries:append(2, string.char(proxy.COM_QUERY) .. "SELECT NOW()" )
                proxy.queries:append(1, packet )
                proxy.queries:append(2, string.char(proxy.COM_QUERY) .. "SELECT NOW()" )
                return proxy.PROXY_SEND_QUERY
        end
end
function read_query_result(inj)
        if inj.id == 2 then
                for row in inj.resultset.rows do
                        print("injected query returned: " .. row[1])
                end
                return proxy.PROXY_IGNORE_RESULT
        else
                print("query-time: " .. (inj.query_time / 1000) .. "ms")
                print("response-time: " .. (inj.response_time / 1000) .. "ms")
        end
end
```

For further examples, see Chapter 5, *Using MySQL Proxy*.

Chapter 5. Using MySQL Proxy

There are a number of different ways to use MySQL Proxy. At the most basic level, you can allow MySQL Proxy to pass on queries from clients to a single server. To use MySQL proxy in this mode, you just have to specify the backend server that the proxy should connect to on the command line:

```
shell> mysql-proxy --proxy-backend-addresses=sakila:3306
```

If you specify multiple backend MySQL servers then the proxy will connect each client to each server in a round-robin fashion. For example, imagine you have two MySQL servers, A and B. The first client to connect will be connected to server A, the second to server B, the third to server C. For example:

```
shell> mysql-proxy \
     --proxy-backend-addresses=narcissus:3306 \
     --proxy-backend-addresses=nostromo:3306
```

When you have specified multiple servers in this way, the proxy will automatically identify when a MySQL server has become unavailable and mark it accordingly. New connections will automatically be attached to a server that is available, and a warning will be reported to the standard output from `mysql-proxy`:

```
network-mysqld.c.367: connect(nostromo:3306) failed: Connection refused
network-mysqld-proxy.c.2405: connecting to backend (nostromo:3306) failed, marking it as down for ...
```

Lua scripts enable a finer level of control, both over the connections and their distribution and how queries and result sets are processed. When using an Lua script, you must specify the name of the script on the command line using the `--proxy-lua-script` option:

```
shell> mysql-proxy --proxy-lua-script=mc.lua --proxy-backend-addresses=sakila:3306
```

When you specify a script, the script is not executed until a connection is made. This means that faults with the script will not be raised until the script is executed. Script faults will not affect the distribution of queries to backend MySQL servers.

> **Note**
>
> Because the script is not read until the connection is made, you can modify the contents of the Lua script file while the proxy is still running and the script will automatically be used for the next connection. This ensures that MySQL Proxy remains available because it does not have to be restarted for the changes to take effect.

5.1. Using the Administration Interface

The `mysql-proxy` administration interface can be accessed using any MySQL client using the standard protocols. You can use the administration interface to gain information about the proxy server as a whole - standard connections to the proxy are isolated to operate as if you were connected directly to the backend MySQL server. Currently, the interface supports a limited set of functionality designed to provide connection and configuration information.

Because connectivity is provided over the standard MySQL protocol, you must access this information using SQL syntax. By default, the administration port is configured as 4041. You can change this port number using the `--admin-address` command-line option.

To get a list of the currently active connections to the proxy:

```
mysql> select * from proxy_connections;
+------+--------+-------+------+
| id   | type   | state | db   |
+------+--------+-------+------+
|    0 | server | 0     |      |
|    1 | proxy  | 0     |      |
|    2 | server | 10    |      |
+------+--------+-------+------+
3 rows in set (0.00 sec)
```

To get the current configuration:

```
mysql> select * from proxy_config;
+----------------------------+----------------------+
| option                     | value                |
+----------------------------+----------------------+
| admin.address              | :4041                |
| proxy.address              | :4040                |
| proxy.lua_script           | mc.lua               |
| proxy.backend_addresses[0] | mysql:3306           |
| proxy.fix_bug_25371        | 0                    |
| proxy.profiling             | 1                      |
+-----------------------------+----------------------+
6 rows in set (0.01 sec)
```

Chapter 6. MySQL Proxy FAQ

Questions

- 6.1: Is the system context switch expensive, how much overhead does the lua script add?
- 6.2: How do I use a socket with MySQL Proxy? Proxy change logs mention that support for UNIX sockets has been added.
- 6.3: Can I use MySQL Proxy with all versions of MySQL?
- 6.4: If MySQL Proxy has to live on same machine as MySQL, are there any tuning considerations to ensure both perform optimally?
- 6.5: Do proxy applications run on a separate server? If not, what is the overhead incurred by Proxy on the DB server side?
- 6.6: Can MySQL Proxy handle SSL connections?
- 6.7: What is the limit for `max-connections` on the server?
- 6.8: As the script is re-read by proxy, does it cache this or is it looking at the file system with each request?
- 6.9: With load balancing, what happen to transactions ? Are all queries sent to the same server ?
- 6.10: Can I run MySQL Proxy as a daemon?
- 6.11: What about caching the authorization info so clients connecting are given back-end connections that were established with identical authorization information, thus saving a few more round trips?
- 6.12: Could MySQL Proxy be used to capture passwords?
- 6.13: Can MySQL Proxy be used on slaves and intercept binlog messages?
- 6.14: MySQL Proxy can handle about 5000 connections, what is the limit on a MySQL server?
- 6.15: How does MySQL Proxy compare to DBSlayer ?
- 6.16: I currently use SQL Relay for efficient connection pooling with a number of apache processes connecting to a MySQL server. Can MySQL proxy currently accomplish this. My goal is to minimize connection latency while keeping temporary tables available.
- 6.17: The global namespace variable example with quotas does not persist after a reboot, is that correct?
- 6.18: I tried using MySQL Proxy without any Lua script to try a round-robin type load balancing. In this case, if the first database in the list is down, MySQL Proxy would not connect the client to the second database in the list.
- 6.19: Would the Java-only connection pooling solution work for multiple web servers? With this, I'd assume you can pool across many web servers at once?
- 6.20: Is the MySQL Proxy an API ?
- 6.21: If you have multiple databases on the same box, can you use proxy to connect to databases on default port 3306?
- 6.22: Will Proxy be deprecated for use with connection pooling once MySQL 6.x comes out? Or will 6.x integrate proxy more deeply?
- 6.23: In load balancing, how can I separate reads from writes?
- 6.24: We've looked at using MySQL Proxy but we're concerned about the alpha status - when do you think the proxy would be considered production ready?
- 6.25: Will the proxy road map involve moving popular features from lua to C? For example Read/Write splitting
- 6.26: Are these reserved function names (e.g., error_result) that get automatically called?
- 6.27: Can you explain the status of your work with `memcached` and MySQL Proxy?
- 6.28: Is there any big web site using MySQL Proxy ? For what purpose and what transaction rate have they achieved.
- 6.29: So the authentication when connection pooling has to be done at every connection? What's the authentication latency?

Questions and Answers

6.1: Is the system context switch expensive, how much overhead does the lua script add?

Lua is fast and the overhead should be small enough for most applications. The raw packet-overhead is around 400 microseconds.

6.2: How do I use a socket with MySQL Proxy? Proxy change logs mention that support for UNIX sockets has been added.

Just specify the path to the socket:

```
--proxy-backend-addresses=/path/to/socket
```

However it appears that `--proxy-address=/path/to/socket` does not work on the front end. It would be nice if someone added this feature.

6.3: Can I use MySQL Proxy with all versions of MySQL?

MySQL Proxy is designed to work with MySQL 5.0 or higher, and supports the MySQL network protocol for 5.0 and higher.

6.4: If MySQL Proxy has to live on same machine as MySQL, are there any tuning considerations to ensure both perform optimally?

MySQL Proxy can live on any box: application, db or its own box. MySQL Proxy uses comparatively little CPU or RAM, so additional requirements or overhead is negligible.

6.5: Do proxy applications run on a separate server? If not, what is the overhead incurred by Proxy on the DB server side?

You can run the proxy on the application server, on its own box or on the DB-server depending on the use-case

6.6: Can MySQL Proxy handle SSL connections?

No, being the man-in-the-middle, Proxy can't handle encrypted sessions because it cannot share the SSL information.

6.7: What is the limit for `max-connections` on the server?

Around 1024 connections the MySQL Server may run out of threads it can spawn. Leaving it at around 100 is advised.

6.8: As the script is re-read by proxy, does it cache this or is it looking at the file system with each request?

It looks for the script at client-connect and reads it if it has changed, otherwise it uses the cached version.

6.9: With load balancing, what happen to transactions ? Are all queries sent to the same server ?

Without any special customization the whole connection is sent to the same server. That keeps the whole connection state intact.

6.10: Can I run MySQL Proxy as a daemon?

Starting from version 0.6.0, the Proxy is launched as a daemon by default. If you want to avoid this, use the `-D` or `--no-daemon` option. To keep track of the process ID, the daemon can be started with the additional option `--pid-file=file`, to save the

PID to a known file name. On version 0.5.x, the Proxy can't be started natively as a daemon

6.11: What about caching the authorization info so clients connecting are given back-end connections that were established with identical authorization information, thus saving a few more round trips?

There is an option that provides this functionality `--proxy-pool-no-change-user`.

6.12: Could MySQL Proxy be used to capture passwords?

The MySQL network protocol does not allow passwords to be sent in clear-text, all you could capture is the encrypted version.

6.13: Can MySQL Proxy be used on slaves and intercept binlog messages?

We are working on that. See http://jan.kneschke.de/2008/5/30/mysql-proxy-rbr-to-sbr-decoding for an example.

6.14: MySQL Proxy can handle about 5000 connections, what is the limit on a MySQL server?

Se your `max-connections` settings. By default the setting is 150, the proxy can handle a lot more.

6.15: How does MySQL Proxy compare to DBSlayer ?

DBSlayer is a REST->MySQL tool, MySQL Proxy is transparent to your application. No change to the application is needed.

6.16: I currently use SQL Relay for efficient connection pooling with a number of apache processes connecting to a MySQL server. Can MySQL proxy currently accomplish this. My goal is to minimize connection latency while keeping temporary tables available.

Yes.

6.17: The global namespace variable example with quotas does not persist after a reboot, is that correct?

Yes. if you restart the proxy, you lose the results, unless you save them in a file.

6.18: I tried using MySQL Proxy without any Lua script to try a round-robin type load balancing. In this case, if the first database in the list is down, MySQL Proxy would not connect the client to the second database in the list.

This issue is fixed in version 0.7.0.

6.19: Would the Java-only connection pooling solution work for multiple web servers? With this, I'd assume you can pool across many web servers at once?

Yes. But you can also start one proxy on each application server to get a similar behaviour as you have it already.

6.20: Is the MySQL Proxy an API ?

No, MySQL Proxy is an application that forwards packets from a client to a server using the MySQL network protocol. The MySQL proxy provides a API allowing you to change its behaviour.

6.21: If you have multiple databases on the same box, can you use proxy to connect to databases on default port 3306?

Yes, MySQL Proxy can listen on any port. Providing none of the MySQL servers are listening on the same port.

6.22: Will Proxy be deprecated for use with connection pooling once MySQL 6.x comes out? Or will 6.x integrate proxy more deeply?

The logic about the pooling is controlled by the lua scripts, you can enable and disable it if you like. There are no plans to embed the current MySQL Proxy functionality into the MySQL Server.

6.23: In load balancing, how can I separate reads from writes?

There is no automatic separation of queries that perform reads or writes to the different backend servers. However, you can specify to `mysql-proxy` that one or more of the 'backend' MyuSQL servers are read-only.

```
$ mysql-proxy \
--proxy-backend-addresses=10.0.1.2:3306 \
--proxy-read-only-backend-addresses=10.0.1.3:3306 &
```

In the next releases we will add connection pooling and read/write splitting to make this more useful. See also MySQL Load Balancer.

6.24: We've looked at using MySQL Proxy but we're concerned about the alpha status - when do you think the proxy would be considered production ready?

We are on the road to the next feature release: 0.7.0. It will improve the performance quite a bit. After that we may be able to enter the beta phase.

6.25: Will the proxy road map involve moving popular features from lua to C? For example Read/Write splitting

We will keep the high-level parts in the Lua layer to be able to adjust to special situations without a rebuild. Read/Write splitting sometimes needs external knowledge that may only be available by the DBA.

6.26: Are these reserved function names (e.g., error_result) that get automatically called?

Only functions and values starting with `proxy.*` are provided by the proxy. All others are provided by you.

6.27: Can you explain the status of your work with `memcached` and MySQL Proxy?

There are some ideas to integrate proxy and `memcache` a bit, but no code yet.

6.28: Is there any big web site using MySQL Proxy ? For what purpose and what transaction rate have they achieved.

Yes, gaiaonline. They have tested MySQL Proxy and seen it handle 2400 queries per second through the proxy.

6.29: So the authentication when connection pooling has to be done at every connection? What's the authentication latency?

You can skip the round-trip and use the connection as it was added to the pool. As long as the application cleans up the temporary tables it used. The overhead is (as always) around 400 microseconds.

6.30: Is it possible to use the MySQL proxy w/ updating a Lucene index (or Solr) by making TCP calls to that server to update?

Yes, but it isn't advised for now.

6.31: Isn't MySQL Proxy similar to what is provided by Java connection pools?

Yes and no. Java connection pools are specific to Java applications, MySQL Proxy works with any client API that talks the MySQL network protocol. Also, connection pools do not provide any functionality for intelligently examining the network packets and modifying the contents.

6.32: Are there tools for isolating problems? How can someone figure out if a problem is in the client, in the db or in the proxy?

You can set a debug script in the proxy, which is an exceptionally good tool for this purpose. You can see very clearly which component is causing the problem, if you set the right breakpoints.

6.33: Can you dynamically reconfigure the pool of MySQL servers that MySQL Proxy will load balance to?

Not yet, it is on the list. We are working on a administration interface for that purpose.

6.34: Given that there is a `connect_server` function, can a Lua script link up with multiple servers?

The proxy provides some tutorials in the source-package, one is `examples/tutorial-keepalive.lua`.

6.35: Adding a proxy must add latency to the connection, how big is that latency?

In the range of 400microseconds

6.36: In the quick poll, I see "Load Balancer: read-write splitting" as an option, so would it be correct to say that there are no scripts written for Proxy yet to do this?

There is a proof of concept script for that included. But its far from perfect and may not work for you yet.

6.37: Is it "safe" to use `LuaSocket` with proxy scripts?

You can, but it is not advised as it may block.

6.38: How different is MySQL Proxy from DBCP (Database connection pooling) for Apache in terms of connection pooling?

Connection Pooling is just one use-case of the MySQL Proxy. You can use it for a lot more and it works in cases where you can't use DBCP (like if you don't have Java).

6.39: Do you have make one large script and call at proxy startup, can I change scripts without stopping and restarting (interrupting) the proxy?

You can just change the script and the proxy will reload it when a client connects.