MySQL Proxy

Abstract

The MySQL Proxy is an application that communicates over the network using the MySQL client/server protocol and provides communication between one or more MySQL servers and one or more MySQL clients. Because MySQL Proxy uses the MySQL client/server protocol, it can be used without modification with any MySQL-compatible client that uses the protocol. This includes the `mysql` command-line client, any clients that uses the MySQL client libraries, and any connector that supports the MySQL network protocol.

For notes detailing the changes in each release of MySQL Proxy, see MySQL Proxy Release Notes.

For legal information, see the Legal Notices.

For help with using MySQL, please visit either the MySQL Forums or MySQL Mailing Lists, where you can discuss your issues with other MySQL users.

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML and PDF formats, see the MySQL Documentation Library.

Document generated on: 2016-04-07 (revision: 47334)

Table of Contents

Preface and Legal Notices

This is the MySQL Proxy manual. This document covers MySQL Proxy 0.8.5.

Legal Notices

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Chapter 1 Introduction to MySQL Proxy

The MySQL Proxy is an application that communicates over the network using the MySQL client/server protocol and provides communication between one or more MySQL servers and one or more MySQL clients. Because MySQL Proxy uses the MySQL network protocol, it can be used without modification with any MySQL-compatible client that uses the protocol. This includes the `mysql` command-line client, any clients that uses the MySQL client libraries, and any connector that supports the MySQL network protocol.

In the most basic configuration, MySQL Proxy simply interposes itself between the server and clients, passing queries from the clients to the MySQL Server and returning the responses from the MySQL Server to the appropriate client. In more advanced configurations, the MySQL Proxy can also monitor and alter the communication between the client and the server. Query interception enables you to add profiling, and 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 return the results from the original query to the client while adding informational statements to each query, for example, to monitor their execution time or progress, and separately log the results.

The proxy enables you to perform additional monitoring, filtering, or manipulation of queries without requiring you 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.

This documentation covers MySQL Proxy 0.8.5.

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 or later. Testing has not been performed with Version 4.1. Please provide feedback on your experiences using the MySQL Proxy Forum.

For notes detailing the changes in each release of MySQL Proxy, see MySQL Proxy Release Notes.

Chapter 2 MySQL Proxy Supported Platforms

MySQL Proxy is currently available as a precompiled binary for the following platforms:

- Linux (including Red Hat, Fedora, Debian, SuSE) and derivatives
- OS X
- FreeBSD
- IBM AIX
- Sun Solaris
- Microsoft Windows (including Microsoft Windows XP, Microsoft Windows Vista, Microsoft Windows Server 2003, Microsoft Windows Server 2008)

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 binary. It is not necessary to have MySQL or Lua installed.

Chapter 3 Installing MySQL Proxy

Table of Contents

You have three choices for installing MySQL Proxy:

- Precompiled binaries are available for a number of different platforms. See Section 3.1, “Installing MySQL Proxy from a Binary Distribution”.
- You can install from the source code to build on an environment not supported by the binary distributions. See Section 3.2, “Installing MySQL Proxy from a Source Distribution”.
- The latest version of the MySQL Proxy source code is available through a development repository. This is the best way to stay up to date with the latest fixes and revisions. See Section 3.3, “Installing MySQL Proxy from the Bazaar Repository”.

3.1 Installing MySQL Proxy from a Binary Distribution

If you download a binary package, you must extract and copy the package contents to your desired installation directory. The package contains files required by MySQL Proxy, including additional Lua scripts and other components required for execution.

To install, unpack the archive into the desired directory, then modify your `PATH` environment variable so that you can use the `mysql-proxy` command directly:

```
shell> cd /usr/local
shell> tar zxf mysql-proxy-0.8.5-platform.tar.gz
shell> PATH=$PATH:/usr/local/mysql-proxy-0.8.5-platform/bin
```

To update the path globally on a system, you might need administrator privileges to modify the appropriate `/etc/profile`, `/etc/bashrc`, or other system configuration file.

On Windows, you can update the `PATH` environment variable using this procedure:

1. On the Windows desktop, right-click the **My Computer** icon, and select **Properties**.
2. Next select the **Advanced** tab from the **System Properties** menu that appears, and click the **Environment Variables** button.
3. Under **System Variables**, select **Path**, then click the **Edit** button. The **Edit System Variable** dialogue should appear.

The Microsoft Visual C++ runtime libraries are a requirement for running MySQL Proxy as of version 0.8.2. Users that do not have these libraries must download and install the Microsoft Visual C++ 2008 Service Pack 1 Redistributable Package MFC Security Update. Use the following link to obtain the package:

Note

This is not required on Windows 8 or higher as the Visual C++ runtime libraries are bundled with that version of Windows.

```
http://www.microsoft.com/download/en/details.aspx?id=26368
```

3.2 Installing MySQL Proxy from a Source Distribution

You can download a source package and compile the MySQL Proxy yourself. To build from source, you must have the following prerequisite components installed:

- `libevent` 1.4 or higher.
- `lua` 5.1.x only. lua 5.2.x is not currently supported.
- `glib2` 2.16.0 or higher.
- `pkg-config`.
- `libtool` 1.5.x or higher.
- `autoconf` 2.56 or higher.
- `automake` 1.10 or higher, up to 1.12.x. It is not possible to use version 1.13 or higher.
- `flex` 2.5.37.
- `gtk-doc` 1.18.
- MySQL 5.0.x or higher developer files.

Note

On some operating systems, you might need to manually build the required components to get the latest version. If you have trouble compiling MySQL Proxy, consider using a binary distributions instead.

After verifying that the prerequisite components are installed, configure and build MySQL Proxy:

```
shell> tar zxf mysql-proxy-0.8.5.tar.gz
shell> cd mysql-proxy-0.8.5
shell> ./configure
shell> make
```

To test the build, use the `check` target to `make`:

```
shell> 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 installed into `/usr/local/share`.

3.3 Installing MySQL Proxy from the Bazaar Repository

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

A build from the Bazaar repository requires that the following prerequisite components be installed:

- Bazaar 1.10.0 or later.
- `libevent` 1.4 or higher.

- `lua` 5.1.x only. lua 5.2.x is not currently supported.
- `glib2` 2.16.0 or higher.
- `pkg-config`.
- `libtool` 1.5.x only. Version 1.6 or higher is not currently supported.
- `autoconf` 2.56 or higher.
- `automake` 1.9 or higher.
- `flex` 2.5.37.
- `gtk-doc` 1.18.
- MySQL 5.0.x or higher developer files.

The `mysql-proxy` source is hosted on Launchpad. To check out a local copy of the Bazaar repository, use `bzr`:

```
shell> bzr branch lp:mysql-proxy
```

The preceding command downloads a complete version of the Bazaar repository for `mysql-proxy`. The main source files are located within the `trunk` subdirectory. The configuration scripts must be generated before you can configure and build `mysql-proxy`. The `autogen.sh` script generates the required configuration scripts for you:

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

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

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

To create a standalone source distribution, identical to the source distribution available for download, use this command:

```
shell> make distcheck
```

The preceding command creates the file `mysql-proxy-0.8.5.tar.gz` (with the corresponding current version) within the current directory.

3.4 Setting Up MySQL Proxy as a Windows Service

The MySQL distribution on Windows includes the `mysql-proxy-svc.exe` command that enables a MySQL Proxy instance to be managed by the Windows service control manager. You can control the service, including automatically starting and stopping it during boot, reboot and shutdown, without separately running the MySQL Proxy application.

To set up a MySQL Proxy service, use the `sc` command to create a new service using the MySQL Proxy service command. Specify the MySQL Proxy options on the `sc` command line, and identify the service with a unique name. For example, to configure a new MySQL Proxy instance that will automatically start when your system boots, redirecting queries to the local MySQL server:

```
C:\> sc create "Proxy" DisplayName= "MySQL Proxy" start= "auto" »
  binPath= "C:\Program Files\MySQL\mysql-proxy-0.8.5\bin\mysql-proxy-svc.exe »
```

```
  --proxy-backend-addresses=127.0.0.1:3306"
```

Note

The space following the equal sign after each property is required; failure to include it results in an error.

The preceding command creates a new service called `Proxy`. You can start and stop the service using the `net start|stop` command with the service name. The service is not automatically started after it is created. To start the service:

```
C:\> net start proxy
The MySQL Proxy service is starting.
The MySQL Proxy service was started successfully.
```

You can specify additional command-line options to the `sc` command. You can also set up multiple MySQL Proxy services on the same machine (providing they are configured to listen on different ports and/or IP addresses.

You can delete a service that you have created:

```
C:\> sc delete proxy
```

For more information on creating services using `sc`, see How to create a Windows service by using Sc.exe.

Chapter 4 MySQL Proxy Command Options

To start MySQL Proxy, you can run it directly from the command line:

```
shell> mysql-proxy
```

For most situations, you specify at least the host name or address and the port number of the backend MySQL server to which the MySQL Proxy should pass queries.

You can specify options to `mysql-proxy` either on the command line, or by using a configuration file and the `--defaults-file` command-line option to specify the file location.

If you use a configuration file, format it as follows:

- Specify the options within a `[mysql-proxy]` configuration group. For example:

  ```
  [mysql-proxy]
  admin-address = host:port
  ```

- Specify all configuration options in the form of a configuration name and the value to set.
- For options that are a simple toggle on the command line (for example, `--proxy-skip-profiling`), use `true` or `false`. For example, the following is invalid:

  ```
  [mysql-proxy]
  proxy-skip-profiling
  ```

  But this is valid:

  ```
  [mysql-proxy]
  proxy-skip-profiling = true
  ```

- Give the configuration file Unix permissions of `0660` (readable and writable by user and group, no access for others).

Failure to adhere to any of these requirements causes `mysql-proxy` to generate an error during startup.

The following tables list the supported configuration file and command-line options.

Table 4.1 `mysql-proxy` Help Options

Format	Description
--help	Show help options
--help-admin	Show admin module options
--help-all	Show all help options
--help-proxy	Show proxy module options

Table 4.2 `mysql-proxy` Admin Options

Format	Description
--admin-address	The admin module listening host and port
--admin-lua-script	Script to execute by the admin module
--admin-password	Authentication password for admin module
--admin-username	Authentication user name for admin module
--proxy-address	The listening proxy server host and port

Table 4.3 `mysql-proxy` Proxy Options

Format	Description	Removed
--no-proxy	Do not start the proxy module	
--proxy-backend-addresses	The MySQL server host and port	
--proxy-fix-bug-25371	Enable the fix for Bug #25371 for older libmysql versions	0.8.1
--proxy-lua-script	Filename for Lua script for proxy operations	
--proxy-pool-no-change-user	Do not use the protocol CHANGE_USER command to reset the connection when coming from the connection pool	
--proxy-read-only-backend-addresses	The MySQL server host and port (read only)	
--proxy-skip-profiling	Disable query profiling	

Table 4.4 `mysql-proxy` Applications Options

Format	Description
--basedir	The base directory prefix for paths in the configuration
--daemon	Start in daemon mode
--defaults-file	Read only named option file
--event-threads	Number of event-handling threads
--keepalive	Try to restart the proxy if a crash occurs
--log-backtrace-on-crash	Try to invoke the debugger and generate a backtrace on crash
--log-file	The file where error messages are logged
--log-level	The logging level
--log-use-syslog	Log errors to syslog
--lua-cpath	Set the LUA_CPATH
--lua-path	Set the LUA_PATH
--max-open-files	The maximum number of open files to support
--pid-file	File in which to store process ID
--plugin-dir	Directory containing plugin files
--plugins	List of plugins to load
--user	The user to use when running mysql-proxy
--version	Show version information

Except as noted in the following details, all of the options can be used within the configuration file by supplying the option and the corresponding value. For example:

```
[mysql-proxy]
log-file = /var/log/mysql-proxy.log
log-level = message
```

- `--help`, `-h`

Command-Line Format	`--help`

  Show available help options.

- `--help-admin`

Command-Line Format	--help-admin

Show options for the admin module.

- --help-all

Command-Line Format	--help-all

Show all help options.

- --help-proxy

Command-Line Format	--help-proxy

Show options for the proxy module.

- --admin-address=*host*:*port*

Command-Line Format	--admin-address=host:port	
Permitted Values	Type	string
	Default	:4041

The host name (or IP address) and port for the administration port. The default is localhost:4041.

- --admin-lua-script=*file_name*

Command-Line Format	--admin-lua-script=file_name	
Permitted Values	Type	file name
	Default	

The script to use for the proxy administration module.

- --admin-password=*password*

Command-Line Format	--admin-password=password	
Permitted Values	Type	string
	Default	

The password to use to authenticate users wanting to connect to the MySQL Proxy administration module. This module uses the MySQL protocol to request a user name and password for connections.

- --admin-username=*user_name*

Command-Line Format	--admin-username=user_name	
Permitted Values	Type	string
	Default	root

The user name to use to authenticate users wanting to connect to the MySQL Proxy administration module. This module uses the MySQL protocol to request a user name and password for connections. The default user name is root.

- --basedir=*dir_name*

Command-Line Format	--basedir=dir_name

Permitted Values	Type	directory name

The base directory to use as a prefix for all other file name configuration options. The base name should be an absolute (not relative) directory. If you specify a relative directory, `mysql-proxy` generates an error during startup.

- `--daemon`

Command-Line Format	--daemon

Starts the proxy in daemon mode.

- `--defaults-file=file_name`

Command-Line Format	--defaults-file=file_name	
Permitted Values	Type	file name

The file to read for configuration options. If not specified, MySQL Proxy takes options only from the command line.

- `--event-threads=count`

Command-Line Format	--event-threads=count	
Permitted Values	Type	numeric
	Default	1

The number of event threads to reserve to handle incoming requests.

- `--keepalive`

Command-Line Format	--keepalive

Create a process surrounding the main `mysql-proxy` process that attempts to restart the main `mysql-proxy` process in the event of a crash or other failure.

Note

The `--keepalive` option is not available on Microsoft Windows. When running as a service, `mysql-proxy` automatically restarts.

- `--log-backtrace-on-crash`

Command-Line Format	--log-backtrace-on-crash

Log a backtrace to the error log and try to initialize the debugger in the event of a failure.

- `--log-file=file_name`

Command-Line Format	--log-file=file_name	
Permitted Values	Type	file name

The file to use to record log information. If this option is not given, `mysql-proxy` logs to the standard error output.

- `--log-level=level`

Command-Line Format	--log-level=level	
Permitted Values	Type	enumeration

	Default	`critical`
	Valid Values	`critical`
		`error`
		`warning`
		`info`
		`message`
		`debug`

The log level to use when outputting error messages. Messages with that level (or lower) are output. For example, `message` level also outputs message with `info`, `warning`, and `error` levels.

- `--log-use-syslog`

Command-Line Format	`--log-use-syslog`

Log errors to the syslog (Unix/Linux only).

- `--lua-cpath=`*`dir_name`*

Command-Line Format	`--lua-cpath=dir_name`	
Permitted Values	**Type**	`directory name`

The `LUA_CPATH` to use when loading compiled modules or libraries for Lua scripts.

- `--lua-path=`*`dir_name`*

Command-Line Format	`--lua-path=dir_name`	
Permitted Values	**Type**	`directory name`

The `LUA_CPATH` to use when loading modules for Lua.

- `--max-open-files=`*`count`*

Command-Line Format	`--max-open-files=count`	
Permitted Values	**Type**	`numeric`

The maximum number of open files and sockets supported by the `mysql-proxy` process. Certain scripts might require a higher value.

- `--no-proxy`

Command-Line Format	`--no-proxy`

Disable the proxy module.

- `--plugin-dir=`*`dir_name`*

Command-Line Format	`--plugin-dir=dir_name`	
Permitted Values	**Type**	`directory name`

The directory to use when loading plugins for `mysql-proxy`.

- `--plugins=`*`plugin`*

Command-Line Format	`--plugins=plugin,...`

Permitted Values	Type	string

Loads a plugin.

When using this option on the command line, you can specify the option multiple times to specify multiple plugins. For example:

```
shell> mysql-proxy --plugins=proxy --plugins=admin
```

When using the option within the configuration file, you should separate multiple plugins by commas. The equivalent of the preceding example would be:

```
...
plugins=proxy,admin
```

- `--proxy-address=host:port`, `-P host:port`

Command-Line Format	--proxy-address=host:port	
Permitted Values	Type	string
	Default	:4040

The listening host name (or IP address) and port of the proxy server. The default is `:4040` (all IPs on port 4040).

- `--proxy-read-only-backend-addresses=host:port`, `-r host:port`

Command-Line Format	--proxy-read-only-backend-addresses=host:port	
Permitted Values	Type	string

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.

Note

Setting this value only configures the servers within the corresponding internal structure (see `proxy.global.backends`). You can determine the backend type by checking the `type` field for each connection.

You should therefore only use this option in combination with a script designed to make use of the different backend types.

When using this option on the command line, you can specify the option and the server multiple times to specify multiple backends. For example:

```
shell> mysql-proxy --proxy-read-only-backend-addresses=192.168.0.1:3306 --proxy-read-only-backend-addres
```

When using the option within the configuration file, you should separate multiple servers by commas. The equivalent of the preceding example would be:

```
...
proxy-read-only-backend-addresses = 192.168.0.1:3306,192.168.0.2:3306
```

- `--proxy-backend-addresses=host:port`, `-b host:port`

Command-Line Format	--proxy-backend-addresses=host:port	
Permitted Values	Type	string

	Default	`127.0.0.1:3306`

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.

When using this option on the command line, you can specify the option and the server multiple times to specify multiple backends. For example:

```
shell> mysql-proxy --proxy-backend-addresses 192.168.0.1:3306 --proxy-backend-addresses 192.168.0.2:3
```

When using the option within the configuration file, you should separate multiple servers by commas. The equivalent of the preceding example would be:

```
...
proxy-backend-addresses = 192.168.0.1:3306,192.168.0.2:3306
```

- `--proxy-pool-no-change-user`

Command-Line Format	`--proxy-pool-no-change-user`

Disable use of the MySQL protocol `CHANGE_USER` command when reusing a connection from the pool of connections specified by the `proxy-backend-addresses` list.

- `--proxy-skip-profiling`

Command-Line Format	`--proxy-skip-profiling`

Disable query profiling (statistics time tracking). The default is for tracking to be enabled.

- `--proxy-fix-bug-25371`

Removed	0.8.1
Command-Line Format	`--proxy-fix-bug-25371`

Enable a workaround for an issue when connecting to a MySQL server later than 5.1.12 when using a MySQL client library of any earlier version.

This option was removed in `mysql-proxy` 0.8.1. Now, `mysql-proxy` returns an error message at the protocol level if it sees a `COM_CHANGE_USER` being sent to a server that has a version from 5.1.14 to 5.1.17.

- `--proxy-lua-script=`*`file_name`*, `-s` *`file_name`*

Command-Line Format	`--proxy-lua-script=file_name`	
Permitted Values	**Type**	`file name`

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, the updated content is automatically used when a new connection is made.

- `--pid-file=`*`file_name`*

Command-Line Format	`--pid-file=file_name`	
Permitted Values	**Type**	`file name`

The name of the file in which to store the process ID.

- `--user=user_name`

Command-Line Format	--user=user_name	
Permitted Values	Type	string

Run `mysql-proxy` as the specified `user`.

- `--version`, `-V`

Command-Line Format	--version

Show the version number.

The most common usage is as a simple proxy service (that is, without additional 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, this will be passed through natively without alteration by `mysql-proxy`, so you must also specify the required authentication information:

```
shell> mysql --host=localhost --port=4040 \
   --user=user_name --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-backend-addresses` command-line option.

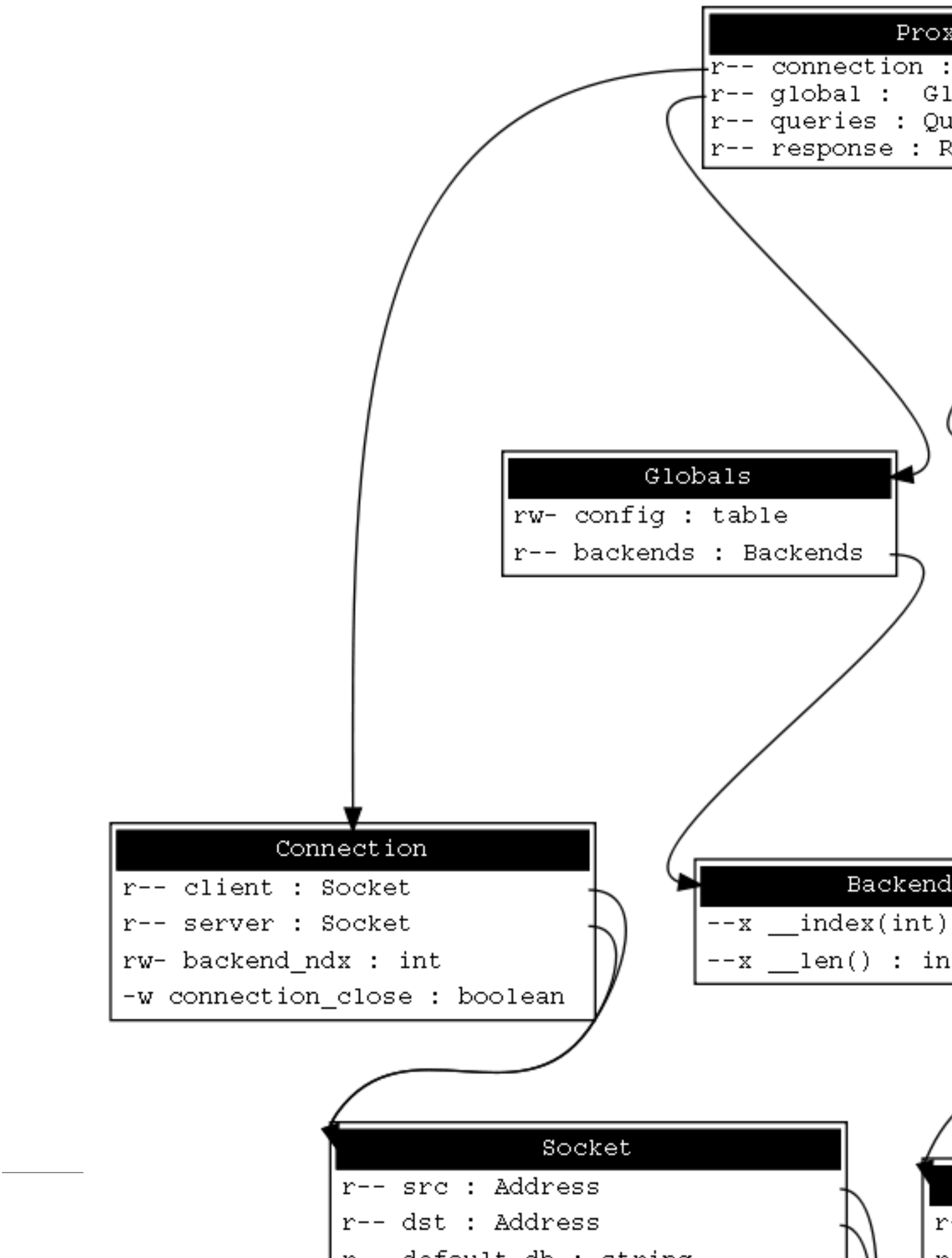
For more detailed information on how to use these command-line options, and `mysql-proxy` in general in combination with Lua scripts, see Chapter 6, *Using MySQL Proxy*.

Chapter 5 MySQL Proxy Scripting

Table of Contents

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 Web site.

The following diagram shows an overview of the classes exposed by MySQL Proxy.


Prox
r-- connection :
r-- global : Gl
r-- queries : Qu
r-- response : R
Globals
rw- config : table
r-- backends : Backends
Connection
r-- client : Socket
r-- server : Socket
rw- backend_ndx : int
-w connection_close : boolean
Backend
--x __index(int)
--x __len() : in
Socket
r-- src : Address
r-- dst : Address

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, 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 injected queries within the `read_query()` function, 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 following table lists MySQL proxy and server communication functions, the supplied information, and the direction of information flow when the function is triggered.

Function	Supplied Information	Direction
`connect_server()`	None	Client to Server
`read_handshake()`	None	Server to Client
`read_auth()`	None	Client to Server
`read_auth_result()`	None	Server to Client
`read_query()`	Query	Client to Server
`read_query_result()`	Query result	Server to Client

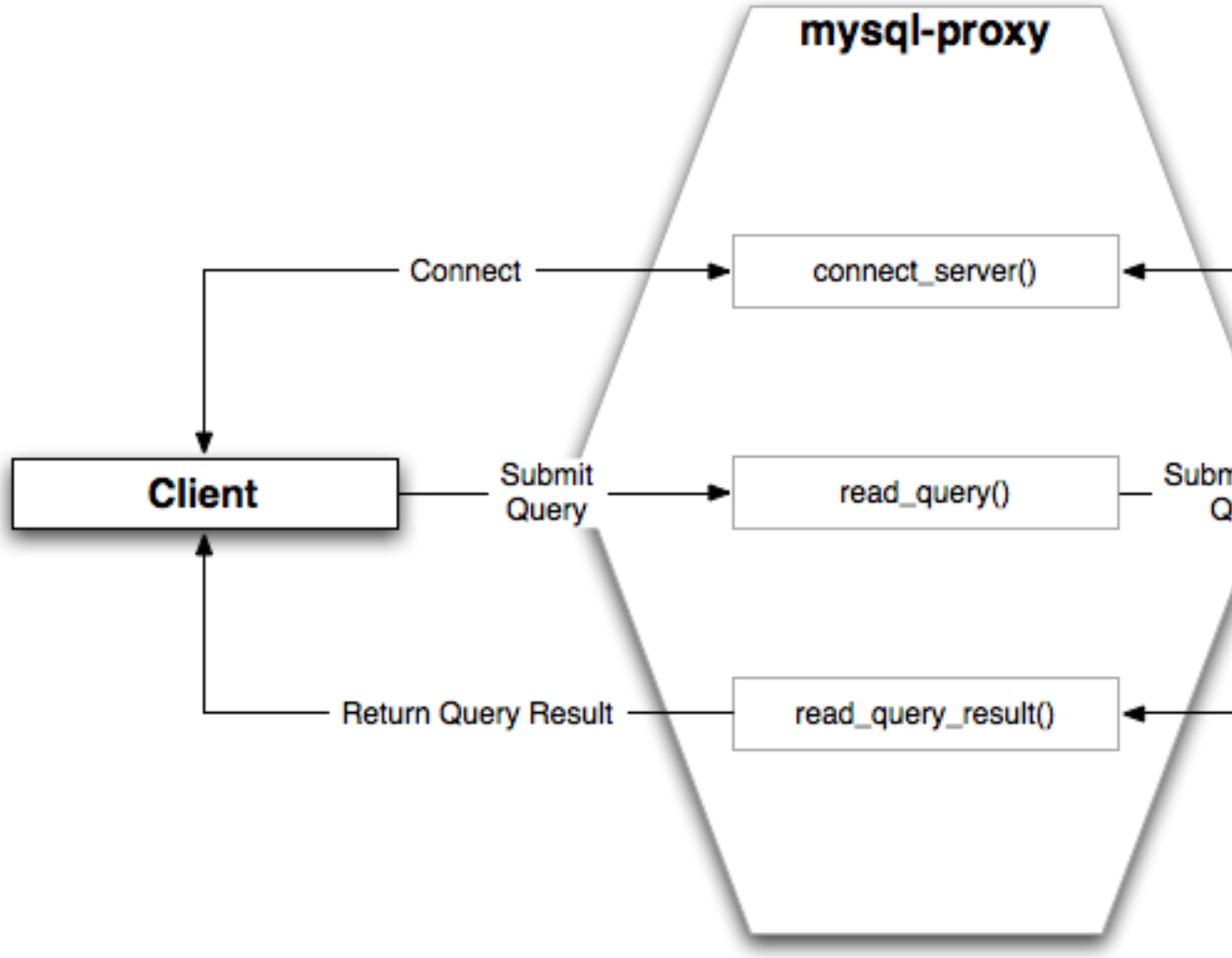
By default, all functions return a result that indicates whether the data should be passed on to the client or server (depending on the direction of the information being transferred). This return value can be overridden by explicitly returning a constant indicating that a particular response should be sent. For example, it is possible to construct result set information by hand within `read_query()` and to return the result set 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 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.

5.1 Proxy Scripting Sequence During Query Injection

The following figure 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, need not match or

correlate. Once the client has connected to the proxy, the sequence shown in the following diagram occurs for each individual query sent by the client.



1. When 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, then filter out queries that do not match the original query ID when processing them with `read_query_result()`.

5.2 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.

The following table describes common attributes of the MySQL `proxy` scripting element.

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.global.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. 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, and is split into a `client` and `server` tables. This enables you to examine information about both the incoming client connections to the proxy (`client`), and to the backend servers (`server`).

The following table describes the client and server attributes of the `proxy.connection` object.

Attribute	Description
`client.default_db`	Default database requested by the client
`client.username`	User name used to authenticate
`client.scrambled_password`	The scrambled version of the password used to authenticate
`client.dst.name`	The combined `address:port` of the Proxy port used by this client (should match the `--proxy-address` configuration parameter)

Attribute	Description
client.dst.address	The IP address of the of the Proxy port used by this client
client.dst.port	The port number of the of the Proxy port used by this client
client.src.name	The combined address:port of the client (originating) TCP/IP endpoint
client.src.address	The IP address of the client (originating) TCP/IP port
client.src.port	The port of the client (originating) TCP/IP endpoint
server.scramble_buffer	The scramble buffer used to scramble the password
server.mysqld_version	The MySQL version number of the server
server.thread_id	The ID of the thread handling the connection to the current server
server.dst.name	The combined address:port for the backend server for the current connection (i.e. the connection to the MySQL server)
server.dst.address	The address for the backend server
server.dst.port	The port for the backend server
server.src.name	The combined address:port for the TCP/IP endpoint used by the Proxy to connect to the backend server
server.src.address	The address of the endpoint for the proxy-side connection to the MySQL server
server.src.port	The port of the endpoint for the proxy-side connection to the MySQL server

proxy.global.backends

The proxy.global.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.global.backends table are shown in the following table.

Attribute	Description
dst.name	The combined address:port of the backend server.
dst.address	The IP address of the backend server.
dst.port	The port of the backend server.
connected_clients	The number of clients currently connected.
state	The status of the backend server. See Backend State/Type Constants [25].
type	The type of the backend server. You can use this to identify whether the backed was configured as a standard read/write backend, or a read-only backend. You can compare this value to the proxy.BACKEND_TYPE_RW and proxy.BACKEND_TYPE_RO.

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, queries are passed on to the backend server verbatim. Also, if you do not populate the query queue by hand, the read_query_result() function is not triggered.

The following functions are supported for populating the proxy.queries object.

Function	Description
`append(id,packet, [options])`	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. The optional `options` should be a table containing the options specific to this 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)
```

The optional third argument to `append()` should contain the options for the packet. To have access to the result set through the `read_query_result()` function, set the `resultset_is_needed` flag to `true`:

```
proxy.queries:append( 1, packet, { resultset_is_needed = true } )
```

If that flag is `false` (the default), proxy will:

- Send the result set to the client as soon as it is received
- Reduce memory usage (because the result set is not stored internally for processing)
- Reduce latency of returning results to the client
- Pass data from server to client unaltered

The default mode is therefore quicker and useful if you only want to monitor the queries sent, and the basic statistics.

To perform any kind of manipulation on the returned data, you must set the flag to `true`, which will:

- Store the result set so that it can be processed.
- Enable modification of the result set before it is returned to the client.
- Enable you to discard the result set instead of returning it to the client.

`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) to return.

The following table describes the attributes of the `proxy.response` structure.

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`, you should set the value of the `mysql.response.errmsg` with a suitable error message.

Attribute	Description
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 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 result set 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 to return. The structure contains the information about the entire result set, with the individual elements of the data shown in the following table.

The following table describes the attributes of the proxy.response.resultset structure.

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 result set. Valid flags include auto_commit (whether an automatic commit was triggered),

Attribute	Description
	`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 showing how to use this structure, see `proxy.response`.

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 or type of the backend MySQL server to which the proxy is connected. 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

Constant	Description
BACKEND_TYPE_RW	Backend is available for read/write
BACKEND_TYPE_RO	Backend is available only for read-only use

Server Command Constants

The values described in the table below 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
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
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

5.3 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, you can specify which server that connection should use by setting the value of proxy.connection.backend_ndx to a valid server number. The following code chooses 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.global.backends[proxy.connection.backend_ndx].dst.name)
end
```

This example also displays the IP address/port combination by accessing the information from the internal `proxy.global.backends` table.

5.4 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 the `proxy.connection` structure.

- `proxy.connection.server.mysqld_version`: The version of the MySQL server.
- `proxy.connection.server.thread_id`: The thread ID.
- `proxy.connection.server.scramble_buffer`: The password scramble buffer.
- `proxy.connection.server.dst.name`: The IP address of the server.
- `proxy.connection.client.src.name`: 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()
        print("<-- let's send him some information about us")
        print("    mysqld-version: " .. proxy.connection.server.mysqld_version)
        print("    thread-id     : " .. proxy.connection.server.thread_id)
        print("    scramble-buf  : " .. string.format("%q",proxy.connection.server.scramble_buffer))
        print("    server-addr   : " .. proxy.connection.server.dst.name)
        print("    client-addr   : " .. proxy.connection.client.dst.name)

        if not proxy.connection.client.src.name: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
```

You must return an error packet to the client by using `proxy.PROXY_SEND_RESULT`.

5.5 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.

You can obtain the authentication information by examining the `proxy.connection.client` structure. For more information, see `proxy.connection`.

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

```
function read_auth()
        print("    username      : " .. proxy.connection.client.username)
        print("    password      : " .. string.format("%q", proxy.connection.client.scrambled_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
```

5.6 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
```

If a long-password capable client tries to authenticate to a server that supports long passwords, but the user password provided is actually short, read_auth_result() will be called twice. The first time, auth.packet:byte() will equal 254, indicating that the client should try again using the old password protocol. The second time read_auth_result()/ is called, auth.packet:byte() will indicate whether the authentication actually succeeded.

5.7 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 Server Command Constants [26]), we extract the query from the packet and print it. 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`), then execute the queries that you have placed into the queue. If you do not modify the original query or the queue, 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 result set. When operating in a passive mode, during profiling for example, you identify the original query and the corresponding result set so that the results expected 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()`.

5.8 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, 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. You must have set the `resultset_is_needed` flag to `append` to intercept the result set before it is returned to the client. See proxy.queries [22].
- `query`: The text of the original query.
- `query_time`: The number of microseconds required to receive the first row of a result set since the query was sent to the server.
- `response_time`: The number of microseconds required to receive the last row of the result set since the query was sent to the server.
- `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 following Lua script, for example, will output the query, followed by the query time and response time (that is, 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 result set 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, remove the results returned from those additional queries and return only the results from the query originally submitted by the client.

The following example 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()", {resultset_is_n
                proxy.queries:append(1, packet, {resultset_is_needed = true})
                proxy.queries:append(2, string.char(proxy.COM_QUERY) .. "SELECT NOW()", {resultset_is_n

                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 6, *Using MySQL Proxy*.

Chapter 6 Using MySQL Proxy

Table of Contents

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

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

If you specify multiple backend MySQL servers, the proxy connects each client to each server in a round-robin fashion. Suppose that you have two MySQL servers, A and B. The first client to connect is connected to server A, the second to server B, the third to server A. For example:

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

When you specify multiple servers in this way, the proxy automatically identifies when a MySQL server has become unavailable and marks it accordingly. New connections are automatically attached to a server that is available, and a warning is 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 are not raised until the script is executed. Script faults will not affect the distribution of queries to backend MySQL servers.

Note

Because a 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 modified script is automatically used for the next connection. This ensures that MySQL Proxy remains available because it need not be restarted for the changes to take effect.

6.1 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.

In `mysql-proxy` 0.8.0 and earlier, a rudimentary interface was built into the proxy. In later versions this was replaced so that you must specify an administration script to be used when users connect to the administration interface.

To use the administration interface, specify the user name and password required to connect to the admin service, using the `--admin-username` and `--admin-password` options. You must also

specify the Lua script to be used as the interface to the administration service by using the `admin-lua-script` script option to point to a Lua script.

For example, you can create a basic interface to the internal components of the `mysql-proxy` system using the following script, written by Diego Medina:

```
--[[

   Copyright 2008, 2010, Oracle and/or its affiliates. All rights reserved.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; version 2 of the License.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

--]]

-- admin.lua

--[[

    See http://www.chriscalender.com/?p=41
    (Thanks to Chris Calender)
    See http://datacharmer.blogspot.com/2009/01/mysql-proxy-is-back.html
    (Thanks Giuseppe Maxia)

--]]

function set_error(errmsg)
    proxy.response = {
        type = proxy.MYSQLD_PACKET_ERR,
        errmsg = errmsg or "error"
    }
end

function read_query(packet)
    if packet:byte() ~= proxy.COM_QUERY then
        set_error("[admin] we only handle text-based queries (COM_QUERY)")
        return proxy.PROXY_SEND_RESULT
    end

    local query = packet:sub(2)
    local rows = { }
    local fields = { }

    -- try to match the string up to the first non-alphanum
    local f_s, f_e, command = string.find(packet, "^%s*(%w+)", 2)
    local option

    if f_e then
            -- if that match, take the next sub-string as option
            f_s, f_e, option = string.find(packet, "^%s+(%w+)", f_e + 1)
    end

    -- we got our commands, execute it
    if command == "show" and 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 = "query_counter", },
                    },
                    rows = {
                            { proxy.global.query_counter }
                    }
            }

            -- we have our result, send it back
            return proxy.PROXY_SEND_RESULT
    elseif command == "show" and option == "myerror" then
            proxy.response.type = proxy.MYSQLD_PACKET_ERR
            proxy.response.errmsg = "my first error"

            return proxy.PROXY_SEND_RESULT

    elseif string.sub(packet, 2):lower() == 'select help' then
            return show_process_help()

    elseif string.sub(packet, 2):lower() == 'show proxy processlist' then
            return show_process_table()

    elseif query == "SELECT * FROM backends" then
        fields = {
            { name = "backend_ndx",
              type = proxy.MYSQL_TYPE_LONG },

            { name = "address",
              type = proxy.MYSQL_TYPE_STRING },
            { name = "state",
              type = proxy.MYSQL_TYPE_STRING },
            { name = "type",
              type = proxy.MYSQL_TYPE_STRING },
        }

        for i = 1, #proxy.global.backends do
            local b = proxy.global.backends[i]

            rows[#rows + 1] = {
                i, b.dst.name, b.state, b.type
            }
        end
    else
        set_error()
        return proxy.PROXY_SEND_RESULT
    end

    proxy.response = {
        type = proxy.MYSQLD_PACKET_OK,
        resultset = {
            fields = fields,
            rows = rows
        }
    }
    return proxy.PROXY_SEND_RESULT
end


function make_dataset (header, dataset)
    proxy.response.type = proxy.MYSQLD_PACKET_OK

    proxy.response.resultset = {
        fields = {},
        rows = {}
```

```
    }
    for i,v in pairs (header) do
        table.insert(proxy.response.resultset.fields, {type = proxy.MYSQL_TYPE_STRING, name = v})
    end
    for i,v in pairs (dataset) do
        table.insert(proxy.response.resultset.rows, v )
    end
    return proxy.PROXY_SEND_RESULT
end

function show_process_table()
    local dataset = {}
    local header = { 'Id', 'IP Address', 'Time' }
    local rows = {}
    for t_i, t_v in pairs (proxy.global.process) do
        for s_i, s_v in pairs ( t_v ) do
            table.insert(rows, { t_i, s_v.ip, os.date('%c',s_v.ts) })
        end
    end
    return make_dataset(header,rows)
end

function show_process_help()
    local dataset = {}
    local header = { 'command',  'description' }
    local rows = {
        {'SELECT HELP',                 'This command.'},
        {'SHOW PROXY PROCESSLIST',      'Show all connections and their true IP Address.'},
    }
    return make_dataset(header,rows)
end

function dump_process_table()
    proxy.global.initialize_process_table()
    print('current contents of process table')
    for t_i, t_v in pairs (proxy.global.process) do
        print ('session id: ', t_i)
        for s_i, s_v in pairs ( t_v ) do
            print ( '\t', s_i, s_v.ip, s_v.ts )
        end
    end
    print ('---END PROCESS TABLE---')
end

--[[    Help

we use a simple string-match to split commands are word-boundaries

mysql> show querycounter

is split into
command = "show"
option  = "querycounter"

spaces are ignored, the case has to be as is.

mysql> show myerror

returns a error-packet

--]]
```

The script works in combination with a main proxy script, `reporter.lua`:

```
--[[

   Copyright 2008, 2010, Oracle and/or its affiliates. All rights reserved.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; version 2 of the License.
```

```
   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

--]]

-- reporter.lua

--[[

    See http://www.chriscalender.com/?p=41
    (Thanks to Chris Calender)
    See http://datacharmer.blogspot.com/2009/01/mysql-proxy-is-back.html
    (Thanks Giuseppe Maxia)

--]]

proxy.global.query_counter = proxy.global.query_counter or 0

function proxy.global.initialize_process_table()
    if proxy.global.process == nil then
        proxy.global.process = {}
    end
    if proxy.global.process[proxy.connection.server.thread_id] == nil then
        proxy.global.process[proxy.connection.server.thread_id] = {}
    end
end

function read_auth_result( auth )
    local state = auth.packet:byte()
    if state == proxy.MYSQLD_PACKET_OK then
        proxy.global.initialize_process_table()
        table.insert( proxy.global.process[proxy.connection.server.thread_id],
            { ip = proxy.connection.client.src.name, ts = os.time() } )
    end
end

function disconnect_client()
    local connection_id = proxy.connection.server.thread_id
    if connection_id then
        -- client has disconnected, set this to nil
        proxy.global.process[connection_id] = nil
    end
end


---
-- read_query() can return a resultset
--
-- You can use read_query() to return a result-set.
--
-- @param packet the mysql-packet sent by the client
--
-- @return
--   * nothing to pass on the packet as is,
--   * proxy.PROXY_SEND_QUERY to send the queries from the proxy.queries queue
--   * proxy.PROXY_SEND_RESULT to send your own result-set
--
function read_query( packet )
        -- a new query came in this connection
        -- using proxy.global.* to make it available to the admin plugin
        proxy.global.query_counter = proxy.global.query_counter + 1

end
```

To use the script, save the first script to a file (`admin.lua` in the following example) and the other to `reporter.lua`, then run `mysql-proxy` specifying the admin script and a backend MySQL server:

```
shell> mysql-proxy --admin-lua-script=admin.lua --admin-password=password \ »
     --admin-username=root --proxy-backend-addresses=127.0.0.1:3306 -proxy-lua-script=reporter.lua
```

In a different window, connect to the MySQL server through the proxy:

```
shell> mysql --user=root --password=password --port=4040
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1798669
Server version: 5.0.70-log Gentoo Linux mysql-5.0.70-r1

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

In another different window, connect to the `mysql-proxy` admin service using the specified user name and password:

```
shell> mysql --user=root --password=password --port=4041 --host=localhost
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.99-agent-admin

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

To monitor the status of the proxy, ask for a list of the current active processes:

```
mysql> show proxy processlist;
+---------+---------------------+--------------------------+
| Id      | IP Address          | Time                     |
+---------+---------------------+--------------------------+
| 1798669 | 192.168.0.112:52592 | Wed Jan 20 16:58:00 2010 |
+---------+---------------------+--------------------------+
1 row in set (0.00 sec)

mysql>
```

For more information on the example, see MySQL Proxy Admin Example.

Appendix A MySQL Proxy FAQ

Questions

- A.24: What about caching the authorization information so clients connecting are given back-end connections that were established with identical authorization information, thus saving a few more round trips?
- A.25: Is there any big web site using MySQL Proxy? For what purpose and what transaction rate have they achieved?
- A.26: How does MySQL Proxy compare to DBSlayer?
- A.27: 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.
- A.28: Is it “safe” to use `LuaSocket` with proxy scripts?
- A.29: How different is MySQL Proxy from DBCP (Database connection pooling) for Apache in terms of connection pooling?
- A.30: MySQL Proxy can handle about 5000 connections, what is the limit on a MySQL server?
- A.31: Would the Java-only connection pooling solution work for multiple web servers? With this, I would assume that you can pool across many web servers at once?

Questions and Answers

A.1: 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” MySQL servers are read only.

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

A.2: How do I use a socket with MySQL Proxy? Proxy change logs mention that support for UNIX sockets has been added.

Specify the path to the socket:

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

A.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.

A.4: Can I run MySQL Proxy as a daemon?

Use the `--daemon` option. To keep track of the process ID, the daemon can be started with the `--pid-file=file` option to save the PID to a known file name. On version 0.5.x, the Proxy cannot be started natively as a daemon.

A.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.

A.6: With load balancing, what happens 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.

A.7: Is it possible to use MySQL Proxy with updating a Lucene index (or Solr) by making TCP calls to that server to update?

Yes, but it is not advised for now.

A.8: 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.

A.9: How much latency does a proxy add to a connection?

In the range of 400 microseconds per request.

A.10: Do you have to make one large script and call it 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.

A.11: 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, database, or its own box. MySQL Proxy uses comparatively little CPU or RAM, with negligible additional requirements or overhead.

A.12: 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.

A.13: Are these reserved function names (for example, `error_result()`) that get automatically called?

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

A.14: As the script is re-read by MySQL 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.

A.15: Given that there is a `connect_server()` function, can a Lua script link up with multiple servers?

MySQL Proxy provides some tutorials in the source package; one is `examples/tutorial-keepalive.lua`.

A.16: 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 behavior.

A.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.

A.18: Can MySQL Proxy handle SSL connections?

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

A.19: Could MySQL Proxy be used to capture passwords?

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

A.20: Are there tools for isolating problems? How can someone figure out whether a problem is in the client, the database, or 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.

A.21: Is 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.

A.22: So authentication with connection pooling has to be done at every connection? What is 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.

A.23: 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, provided that none of the MySQL servers are listening on the same port.

A.24: What about caching the authorization information 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 `--proxy-pool-no-change-user` option that provides this functionality.

A.25: 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.

A.26: 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.

A.27: 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.

A.28: Is it “safe” to use `LuaSocket` with proxy scripts?

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

A.29: 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 cannot use DBCP (for example, if you do not have Java).

A.30: MySQL Proxy can handle about 5000 connections, what is the limit on a MySQL server?

The server limit is given by the value of the `max_connections` system variable. The default value is version dependent.

A.31: Would the Java-only connection pooling solution work for multiple web servers? With this, I would assume that 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 behavior as you have it already.