MySQL occam-π API

Ulrik Schou, ulriksj@diku.dk
Espen Suenson, expen@diku.dk

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF COPENHAGEN

June 21, 2006

Contents

1 Introduction

This paper describes the development of a MySQL API in occam-π (MoA), including documentation of the API, a benchmark against the MySQL C API, some remarks on developing with the API and a short notice on some demonstration programs distributed with MoA.

The reason for developing MoA is to contribute to the general acknowledgement and spreading of the occam-π programming language. The ease of access to databases is central to many applications, especially web applications which is a field where the parallelism of occam-π is a definite advantage. It is the hope of the authors that MoA will contribute to making occam an attractive language for developers.

The choice of MySQL as the target database for the API was made because MySQL is currently the most widespread free database. An alternative was to make an ODBC API, which would have provided access to more database systems, the cost being a solution less readily applied to MySQL. As the audience is hoped to be upcoming developers and the open-source community, generality was sacrificed for accessibility.

2 The occam-π programming language

occam is a highly parallel safe language based on Hoares Communicating Sequential Processes, a calculus that provides a sound semantic foundation for concurrent execution.

The language was originally developed by the company Inmos in the 80's for their parallel transputer chip. It has since been further developed mainly by a group of researchers at the University of Kent. Notably, the language has been fused with some of the π-calculus for allowing a more dynamic treatment of processes. This is why the language today is called occam-π.

Central to occam is the notion of *processes*. Processes can execute concurrently and possess their own internal data. They can only communicate via. well-defined messages over *channels*. Thus, there is no shared memory in occam, and all communication must be done via. channels. This is the main reason that occam is a very safe and easy language: There are no nasty aliasing effects or surprising concurrent behaviour due to the inappropriate use of the usual concurrency constructs (semaphores etc).

In addition, the compiler is able to check and disallow a variety of unwanted behaviour such as race conditions and out-of-bounds array assignment.

In short, occam is an excellent choice of language for every parallel programming task. The only drawback is that the current compiler onlys supports single-CPU architectures. But even for single-CPU applications occam is worth a thought due to the increased clarity and safety obtained. In addition, efforts are being made continually to further develop occam-π.

3 Design

The main design criteria for the interface to MySQL is that it should be safe and easy to use. The main difficulty in the design lies in the fact that communication with the database is inherently sequential.

Safety is provided by encapsulating access to the database in a process. Communication with the database is then carried out by passing messages to and from the MySQL process. The communication will follow a request-answer model: For each message passed to the process, there will be at least one message received. For safety, there will be separate protocols for incoming and outgoing messages.

The reason for providing access via. a process and not simply as ordinary library calls is that we want to avoid having to expose handles to the database to the application programmer. All state information should be handled internally in the process. For the same reason, each connection to the database will be a separate process. That way, we avoid handles to connections.

The standard way of retrieving data from the database is one row at a time. The basic retrieving method of MoA will work in the same way, but for convenience we will also provide a way of retrieving all rows of data with a single command.

4 Implementation

To lower the development workload, MoA will communicate with MySQL by way of the existing MySQL C API. Thus, MoA is basically built on top of an occam C interface (CIF) process calling the MySQL C API.

As occam is a much safer language than C, as much of the logic as possible should be handled in occam. However, there are some things that are handled in C which could have been handled in occam. It would have been possible to just have occam use blocking C calls to the MySQL C API, and thus avoid having a CIF process running C code altogether. However, that would make it necessary to store the necessary C data structures as byte arrays in occam, which is not very nice.

Instead, we have a C process where the necessary data structures are stored. The communication between the occam and C process is quite low level since channels to C processes are limited in type. Five channels are used to communicate between occam and C. `control`, `argi` and `args` goes from occam to C and `status` and `data` goes from C to occam. `control, argi` and `status` carry type `INT` while `args` and `data` carry type `MOBILE []BYTE`.

`control` is used to send commands to the C process to signal which function in the C API should be called. Subsequently, integer arguments are sent on `argi` and string arguments on `args`, the number and type of arguments being dependent on the command.

If the command can fail, a value is sent back via. the `status` channel indicating success or failure. If any values are returned they are sent back sequentially via. the `status` and `data` channels depending on the type.

All error handling is done in occam, so the only functionality residing in the C process is converting the various C strings and other data types to `MOBILE []BYTE`s and values that occam understands.

It should be okay to have several MySQL processes running in parallel if the MySQL C client library is compiled thread-safe. However, the initialisation call to the API is not thread-safe. For this reason, it is provided in MoA as a process that should terminate before starting any MySQL processes in parallel. This is the only way to provide this functionality, as there is no way for the MySQL processes to atomically

check if they are the first MySQL process to run.

Some of the commands to the MySQL process are very similar, so to avoid code duplication we want the same piece of code to handle these. For this reason, a wrapper process translates the commands for the application programmer before they are passed to the main process of the API for processing. To be really neat, we should translate from the API protocol to an internal protocol, but the translation is so simple that we just re-use the API protocol.

It would have been nice to provide the various field names as part of the `MYSQL.FIELD.INFO` record instead of selecting with a control byte (see section 8.7.9), but the current version of KRoC (1.4.0) doesn't seem to support mobile arrays of mobile records.

5 Applications

This section will describe ways of using MoA in design of CSP applications. The first one is a simple model, shown on figure 1, with only 3 processes. An input, the MoA and an output. An input process reads commands from an interface and then communicates a command to a MoA process. The results from MoA are displayed on an output device. The applications as such cannot interact with the MoA

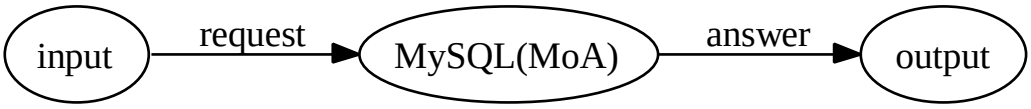

Figure 1: CSP-model for a very simple application.

process i.e. respond intelligent if an error occurs etc. This lead to a another but still simple model. A database process or process network is responsible for intelligent interaction with MoA an then outputs some data. The `mysqlclient` program (see section 7) can be described with this model, as it consists of an input process, an output process and a MoA process. The input and output processes communicate with the MoA process and with each other.

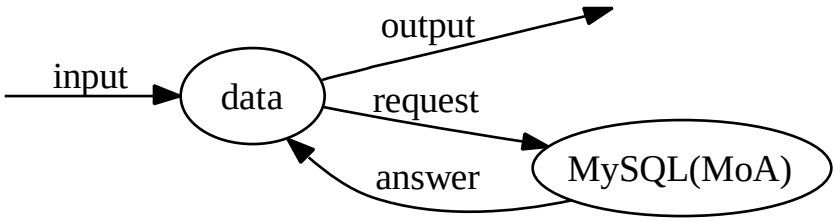

Figure 2: A process called data will now control MoA, receive input and send output.

These two models can be a part of a larger CSP network, and of course many MoA processes can run simultaneously.

6 Benchmark

Operations as `INSERT`, `UPDATE` etc. is not of interest to benchmark, because the workload is on the server side. The most used time consuming operation on the client side is to retrieve larger set of rows with the `SELECT` statement.

The benchmark can be described as follows:

1. Start timer.
2. Query the server: `INSERT INTO t1 (SELECT * FROM s2)` – This will increase the number of rows in `t1` by Δr
3. Query the server: `SELECT * FROM t1`.
4. Retrieve r rows by iterating through the result.
5. Plot r and timer value. Reset timer.
6. Go to 1.

In this case $\Delta r = 10000$. Both the server and client ran on a Intel Pentium 4 CPU running 3 GHz with 3GB of memory. The machine was running Linux 2.6.12.

As can be seen form figure 3, the difference in performance seems to be linear. This is confirmed by figure 4, where the quotient of the performances is drawn. While not perfectly a straight line, the graph suggests that the overhead of using MoA is not above a factor 1.5 for large data sets, less for small sets.

Building an API on top on another API has a price, but in this case the price is a constant.

7 Tests and examples

The distribution of MoA includes two test programs, `test1.occ` and `test2.occ`, that documents the working of most of the API. When run, the programs should result in deadlocks if all is okay, they will give a run-time error if not. See the source for comments on how to make the tests run - currently, they are hard-coded to use a specific server.

The distribution also includes an example program, `simple.occ` (see section 8.3). It is intended as a simple example of how to make a connection and receive data. It will crash in case of any error. As this program is only intended to demonstrate the use of the API, it is hard-coded to use a specific server.

Finally, `mysqlclient.occ` is a simple CLI client for MySQL, providing basic functionality by allowing the user to connect to a database and execute SQL queries.

8 Documentation of the API

The MySQL occam-π API, or MoA for short, is built around the C API for MySQL. The API is developed for version 5.1 of the MySQL server, and is not guaranteed to work with older versions. MoA allows

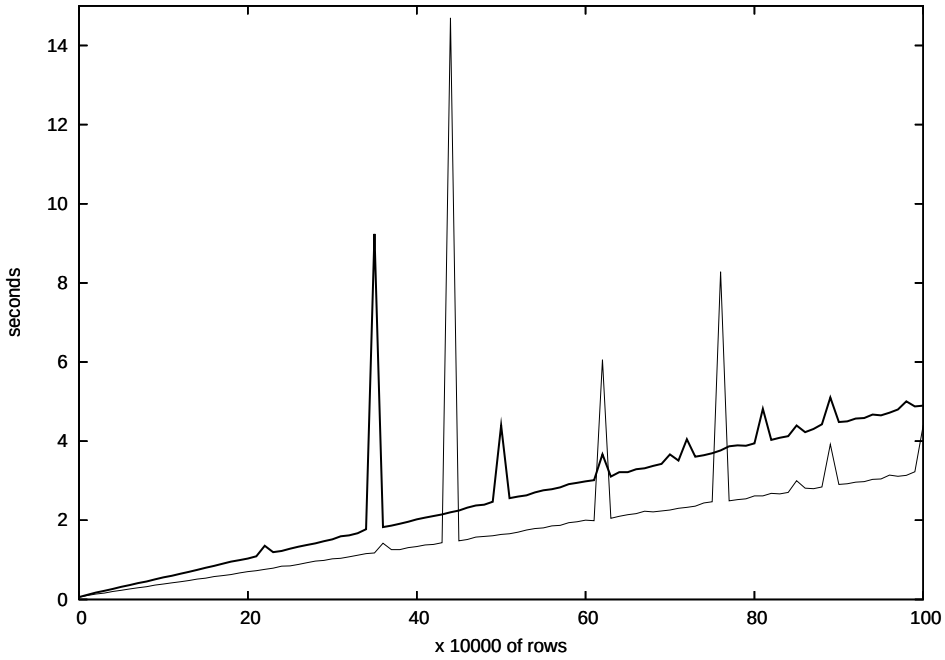

Figure 3: MoA(upper line) compared to the C API. The x-axis is the number of rows the SQL query selected ($*10000$) and the y-axis is the time in seconds. The peaks where queries took up to 14 seconds are caused by disk-caching on the server side either by the operating system or the MySQL-server.

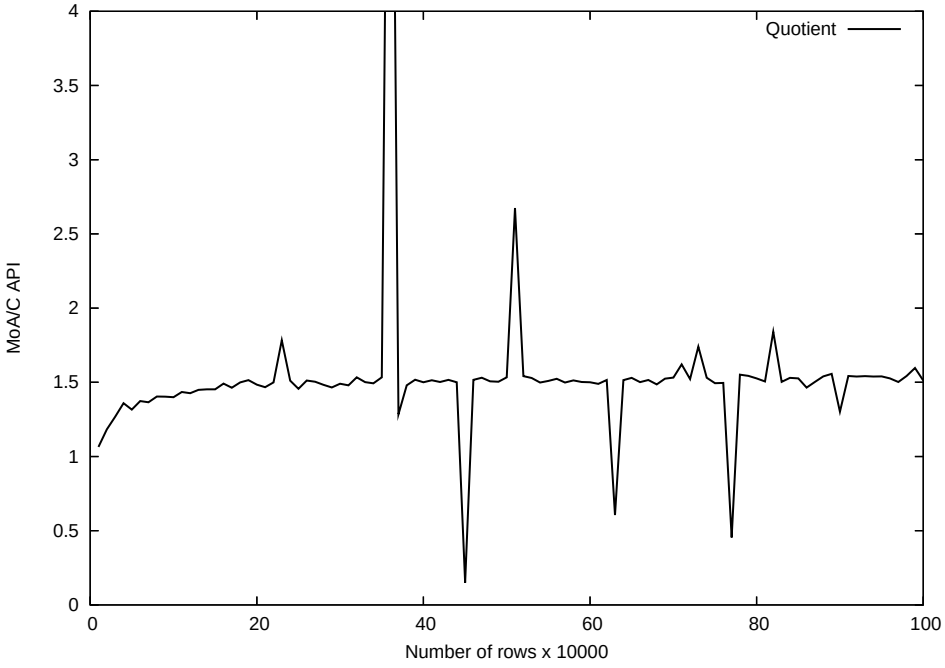

Figure 4: The quotient of the performance of $\frac{MoA}{C\ API}$

.

applications written in occam-π to communicate with a MySQL server. It consists mainly of a pre-defined process definition, `MySQL`, with which you can create processes for communicating with a MySQL server. This is done by sending and receiving messages in the protocols `MYSQL.CONTROL` and `MYSQL.RESULT`. In addition, there is an initialising and a finalizing process, which will be of interest if you are to have more than one connection to MySQL open at a time. Note that, at the present, only the core functionality of the C API is available in MoA.

8.1 Accessing the API

To use MoA put `#INCLUDE "mysql.inc"` and `#USE "mysql.lib"` in your file. The directives must come in this order, or linking will fail. In addition, when compiling, supply KRoC with the flag `-lmoa` for linking with MoA, and a flag for linking with MySQL, this will usually be `-lmysqlclient`.

8.2 Overview of the protocols

MYSQL.CONTROL	MYSQL.RESULT
query; MOBILE []BYTE	mysql.error; INT; MOBILE []BYTE
query.row; MOBILE []BYTE	initialisation.error
query.all; MOBILE []BYTE	no.data.error
next.row	not.connected.error
remaining.rows	connected
affected.rows	data.ready; INT
null.values	data.row; MOBILE []MOBILE []BYTE
field.names	data.null.values; MOBILE []BOOL
field.info; BYTE	data.field.names; MOBILE []MOBILE []BYTE
reconnect	data.field.info; MOBILE []MYSQL.FIELD.INFO
quit	data.affected.rows; INT
	end.of.data
	quit

8.3 An example program: simple.occ

```
#INCLUDE "mysql.inc"
#USE "mysql.lib"

#INCLUDE "consts.inc"
#USE "course.lib"

PROC main(CHAN BYTE kyb, scr, err)
  CHAN MYSQL.CONTROL control:
  CHAN MYSQL.RESULT result:
  BOOL quit:
  PAR
    MySQL(control, result, "bach-1", "root", "latte", "test", 0, "")
    SEQ --control process
      control ! query.all; "SELECT ** FROM t3"
      control ! quit
    SEQ --output process
      quit := FALSE
      WHILE NOT quit
        result ? CASE
          connected
            SKIP
          INT rows:
          data.ready; rows
            SKIP
          MOBILE []MOBILE []BYTE row:
          data.row; row
            SEQ j = 0 FOR SIZE row
              SEQ
                out.string(row[j], 0, scr)
                IF
                  j < ((SIZE row) - 1)
                    scr ! ' '
                  TRUE
                    scr ! '*n'
          end.of.data
            SKIP
          quit
            SEQ
              scr ! FLUSH
              quit := TRUE
:
```

8.4 The MySQL.init process

8.4.1 Signature

```
PROC MySQL.INIT(RESULT BOOL error)
```

8.4.2 Usage

This process initialises the underlying C API for MySQL. If you have only one `MySQL` process in your application you need not call this function, as it will be called automatically. However, if you plan on having several `MySQL` processes running in parallel, this process should be called and allowed to terminate prior to creating any `MySQL` process, as the underlying C function call is not reentrant. In this case, you should also make sure that the C client API is compiled in thread-safe mode (see the MySQL C API documentation for details on this).

Errors Upon termination, `error` will be set to `TRUE` if any error occured, `FALSE` otherwise.

8.5 The MySQL.end process

8.5.1 Signature

```
PROC MySQL.end()
```

8.5.2 Usage

This process finalizes the underlying C API, then terminates. It should not be called until every `MySQL` process has terminated. It might be omitted.

8.6 The MySQL process

8.6.1 Signature

```
PROC MySQL(CHAN MYSQL.CONTROL in, CHAN MYSQL.RESULT out, VAL []BYTE host, user, passwd, db, VAL INT
port, VAL []BYTE socket)
```

8.6.2 Usage

This call will make a process for connecting with a MySQL server.

- The value of host may be either a hostname or an IP address. If host is `""` or the string `"localhost"`, a connection to the local host is assumed. If the OS supports sockets, they are used instead of TCP/IP to connect to the server.
- The `user` parameter contains the user's MySQL login ID. If `user` is the empty string `""`, the current user is assumed. Under Unix, this is the current login name.
- The `passwd` parameter contains the password for user. If `passwd` is `""`, only entries in the user table for the user that have a blank (empty) password field are checked for a match. This allows the database administrator to set up the MySQL privilege system in such a way that users get different privileges depending on whether they have specified a password. Do not attempt to encrypt the password. Password encryption is handled automatically by the C client API.

- `db` is the database name. If `db` is not `""`, the connection sets the default database to this value.
- If `port` is not 0, the value is used as the port number for the TCP/IP connection. Note that the host parameter determines the type of the connection.
- If `socket` is not `""`, the string specifies the socket or named pipe that should be used. Note that the host parameter determines the type of the connection.

Once the process is created it will try to make a connection to the server. It will then send a message to the `out` channel indicating that it is ready to accept commands (see below). Subsequent communication with the process will be done by issuing commands on the `in` channel. Every command will cause one or more answers to be sent on the `out` channel. The process will not terminate until it receives a `quit` message, the only exception to this is noted below under "Errors".

Once the process is created, it is not possible to change the connection, user etc. unless it can be done via. SQL queries.

Answer If the process successfully connects to the server it will send the message `connected` to the `out` channel.

Errors

- If the C client API fails to allocate memory for the connection, the process will send the message `initialisation.error` to the `out` channel. Following this, it will send the message `quit` and then terminate. In all other cases the process must be explicitly killed with the `quit` command. This error should rarely occur.
- If the process fails to connect to the MySQL server, it will send a `mysql.error; INT; MOBILE []BYTE` message containing the error number and description (for information on how to interpret the error numbers, see the MySQL documentation). It will remain alive, but every other commmand than `reconnect` and `quit` will result in an error message.

8.7 Communication with the `MySQL` process

8.7.1 `query; MOBILE []BYTE`

This message carries a string describing an SQL query to be executed. Currently, support for multiple queries in one string (separated by semicolons) is not implemented. Remember that '*' is the escape character in occam, so you should write e.g. "SELECT ** FROM table".

Answer When the query has executed successfully, the process will send a `data.ready; INT` message detailing how many rows were returned from the query. If the query was an UPDATE, DELETE or INSERT statement, or if it was a SELECT statement that didn't match any rows, the integer will be 0.

Errors

- `mysql.error; INT; MOBILE []BYTE` if an SQL error occurred. The error number may be compared with the values `cr.server.gone.error` and `cr.server.lost` to determine if the connection has been lost. If so, the application might try to reconnect.
- `not.connected.error` if the process could not initially connect to the server or if a subsequent attempt to reconnect has failed.

8.7.2 `query.row; MOBILE []BYTE`

This message is similar to `query`, but when the query has executed the process retrieves the first row of the result set without being prompted. As such, `query.row` achieves almost the same effect as `query` followed by `next.row`.

Answer `data.ready` followed by `data.row`, or by `end.of.data` if the query did not return any rows.

Errors The same as for `query`.

8.7.3 `query.all; MOBILE []BYTE`

This message is similar to `query`, but when the query has executed the process will retrieve all the rows of the result set without being prompted. As such, `query.all` achieves almost the same effect as `query` followed by `remaining.rows`.

Answer `data.ready` followed by a number of `data.row` messages and finally an `end.of.data` message. If the query didn't return any data, the answer will just be `data.ready` followed by `end.of.data`.

Errors The same as for `query`.

8.7.4 `next.row`

Retrieves the next row in the current result set.

Answer A `data.row; MOBILE []MOBILE []BYTE` message containing the next row. `end.of.data` if there are no more rows in the result set.

Errors `no.data.error` if there is no current result set, that is, if the last query didn't return any rows, if the last query caused a `mysql.error` or if no query has been executed yet.

8.7.5 `remaining.rows`

Retrieves all remaining rows in the current result set.

Answer A number of `data.row; MOBILE []MOBILE []BYTE` messages followed by `end.of.data`. Just `end.of.data` if there are no more rows in the result set.

Errors `no.data.error` if there is no current result set, that is, if the last query didn't return any rows, if the last query caused a `mysql.error` or if no query has been executed yet.

8.7.6 `affected.rows`

Returns the number of rows affected by the last query.

Answer `data.affected.rows; INT`. For INSERT, DELETE or UPDATE statements, this number indicates how many rows were affected by the update. For SELECT statements this number is the same as the one returned by `data.ready`.

Errors `no.data.error` if the last query caused a `mysql.error` or if no query has been executed yet.

8.7.7 `null.values`

An SQL field can be NULL or an empty string. Since there is no such thing as NULL in occam, both are represented as a `MOBILE []BYTE` of size 0. This message provides a way of distinguishing between the two.

Answer A `data.null.values; MOBILE []BOOL` message. For each field in the last received `data.row` message, the corresponding `BOOL` will be `TRUE` if the field was NULL.

Errors `no.data.error` if the last query didn't return any rows, if the last query caused a `mysql.error`, if no query has been executed yet or if no `data.row` has been received yet.

8.7.8 `field.names`

Provides the names of the fields of the current result set. This message is just a shorthand for `field.info; field.name`.

Answer `data.field.names; MOBILE []MOBILE []BYTE` containing the names of the fields. If the field was given an alias with an AS clause, the alias is given as name.

Errors `no.data.error` if there is no current result set, that is, if the last query didn't return any rows, if the last query caused a `mysql.error` or if no query has been executed yet.

8.7.9 `field.info; BYTE`

Retrieves information about the fields of the current result set. The `BYTE` should have one of the predefined values `field.name, field.org.name, field.table, field.org.table, field.db` or `field.info`. If it has any other value, the effect will be the same is if it was `field.name`.

Answer Depending on whether the control byte was:

- `field.name`: A `data.field.names; MOBILE []MOBILE []BYTE` containing the names of the fields. If the field was given an alias with an AS clause, the alias is given as name.
- `field.org.name`: A `data.field.names` message carrying the names of the fields, but aliases in the names are ignored.
- `field.table`: A `data.field.names` message containing the names of the tables containing the fields. If the field is calculated, the string will be empty. If the table was given an alias with an AS clause, the alias is given as name.
- `field.org.table`: The same as for `field.table`, but aliases in the table names are ignored.
- `field.db`: A `data.field.names` message containing the names of the databases that the fields come from. If the field is calculated, the name is an empty string.
- `field.info`: A `field.info; MOBILE []MYSQL.FIELD.INFO` message containing information about the fields. The `MYSQL.FIELD.INFO` record provides the following data:
  - `INT length`: The width of the field, as specified in the table definition.

– `INT max.length`: The maximum width of the field for the result set (the length of the longest field value for the rows actually in the result set).

– `INT decimals`: The number of decimals for numeric fields.

– `INT charsetnr`: The character set number for the field.

– `MYSQL.TYPE type`: The type of the field:

Type value	Type description
tiny	TINYINT field
short	SMALLINT field
long	INTEGER field
int24	MEDIUMINT field
longlong	BIGINT field
decimal	DECIMAL or NUMERIC field
newdecimal	Precision math DECIMAL or NUMERIC
float	FLOAT field
double	DOUBLE or REAL field
bit	BIT field
timestamp	TIMESTAMP field
date	DATE field
time	TIME field
datetime	DATETIME field
year	YEAR field
string	CHAR or BINARY field
var.string	VARCHAR or VARBINARY field
blob	BLOB or TEXT field (use `max.length` to determine the maximum length)
set	SET field
enum	ENUM field
geometry	Spatial field
null	NULL-type field

To distinguish between binary and non-binary data for string data types, check whether the `charsetnr` value is 63 - or alternatively if `binary.charset` is `TRUE`. If so, the character set is binary, which indicates binary rather than non-binary data. This is how to distinguish between BINARY and CHAR, VARBINARY and VARCHAR, and BLOB and TEXT.

– `BOOL not.null`: Whether the field can't be NULL

– `BOOL pri.key`: Whether the field is part of a primary key

– `BOOL unique.key`: Whether the field is part of a unique key

– `BOOL multiple.key`: Whether the field is part of a non-unique key

– `BOOL unsigned`: Whether the field has the UNSIGNED attribute

– `BOOL zerofill`: Whether the field has the ZEROFILL attribute

– `BOOL binary`: Whether the field has the BINARY attribute

- `BOOL auto.increment`: Whether the field has the AUTO_INCREMENT attribute
- `BOOL binary.charset`: Whether `charsetnr` is 63, indicating binary data for string data types.

8.7.10 `reconnect`

This message will make the process attempt to connect to the SQL server anew. If the connection is already open, it will be shut down beforehand.

Answer `connected` if the attempt was successful.

Errors `mysql.error; INT; MOBILE []BYTE` if an error occurred.

8.7.11 `quit`

This will cause the `MySQL` process to terminate.

Answer `quit`.

Errors None.

References

[1] *P.H. Welch and D.C. Wood* "The Kent Retargetable occam Compiler" - In: Parallel Processing Developments, Proceedings of WoTUG 19. March 1996.

[2] *MySQL AB* "MySQL 5.1 Reference Manual" - 2006. Available from http://dev.mysql.com/doc/

[3] *F.R.M. Barnes* "Interfacing C and occam-pi" - Communicating Process Architectures 2005.

[4] *University of Kent* KRoC with documentation is available from http://www.cs.kent.ac.uk/projects/ofa/kroc/

[5] *SGS-THOMSON Microelectronics Limited* "occam 2.1 reference manual" - 1995. Available from http://www.wotug.org/occam/

[6] *C. A. R. Hoare* "Communicating Sequential Processes" - Prentice Hall International, 1985.