Bullet Cache User Guide

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0.7	April 2012		IV

Contents

$Id: UserGuide.txt 490 2012-05-10 12:14:13Z ivoras $

This is the DocBook/PDF version of the User Guide

1 Introduction

Bullet Cache is a fast and flexible memory cache server created to accelerate web application performance in some of the most common cases. It is probably the fastest Open source cache server available.

For more information, please visit the project's web page.

2 General Information

The main component of the *Bullet Cache* project is the cache server program. Its design is somewhat more complex than the other available open source cache servers but the payoff is visible in its higher performance, a consequence of its pervasive multithreaded design, and added features, in particular the usage of the record tags.

Tip

The *Bullet Cache* project started under another name, "Multi domain cache server", hence the name *mdcached* referenced in the documentation and project code. These may be removed in the future. Early research versions of the project were released under a different version scheme, using the "mdcached-r%d" scheme. These versions are highly experimental and not suitable for any kind of use.

This document covers the usage of the cache server and is intended primarily for application developers.

2.1 Starting the Server

The server binary executable is called `mdcached`. The most simple way to start it is without any arguments. The server in its default configuration may not be optimally configured but it is good enough for most workloads. The currently supported command line arguments are:

```
usage: ./mdcached [-h] [-a] [-d] [-f file] [-i] [-k] [-m #] [-n #] [-r] [-s #] [-t #] [-v ↩
    #]
        -a      Disable runtime adaptive reconfiguration
        -d      Debug mode (doesn't fork to background)
        -f file Set dump file name (for use with SIGUSR1)
        -h      Show this help message
        -i #    Automatically create a cache dump every # seconds
        -k      Bind network threads to worker threads
        -m #    Limit memory use to # MiB (default=0, unlimited)
        -n #    Set number of network threads to # (default=2)
        -r      Read cache data from the dump file (prewarm the cache)
        -s #    Set default (median) record size, in bytes (>= 32, default=1500)
        -t #    Set number of worker threads to # (default=0)
        -v #    Set minimum log (verbosity) level to # [0-7]
```

COMMAND LINE ARGUMENTS

-a

Disable runtime monitoring and adaptive reconfiguration of the server

-d
Enable debug mode: the server does not fork to background and displays the log messages on the terminal.

-f _
Set the name of the cache prewarm dump file.

-h
Show the above message with command line arguments.

-i #
Dump the cache server contents into the "prewarm dump" periodically.

-k
Binds the network threads to the worker threads (only if there are equally many network and worker threads).

-m #
Limit server memory usage to approximately # MiB.

-n #
Set the number of network IO threads created in the server.

-r
Prewarm the cache (load data from the file specified by `-f`).

-s #
Set the default record size, in bytes. Correctly specifying the record size may result in significantly lower memory usage.

-t #
Set the number of worker threads created in the server.

-v #
Set logging verbosity level. The exact levels are defined in `syslog(3)` and setting a verbosity level will block all messages with a priority smaller than the given level.

Most of the arguments are very workload-specific and should not be changed without a good reason and careful testing. Generally, workloads with lots of short queries will benefit from more network threads and workloads with a smaller number of complex queries will benefit from more worker threads.

Important
Bullet Cache is a fast, memory-only cache server. It is by design *not* a persistent database; it has no ACID-like properties and all data *is* expected to go away if the server goes down unexpectedly. Currently the only data persistence mechanism is the "prewarm cache dump" which may be used for periodic data snapshots (or check-pointing).

Server memory usage is maintained in an amortized, statistical way, and accounting is only performed for data records, not for internal server bookkeeping structures. The default record expiry decision function expires least used records by looking at record access counts. Somewhat more performance from the cache server can be wrought out by disabling this access statistics collection by not defining the `DEEP_STATS` symbol while compiling *Bullet Cache*, in which case the older records will be expired.

3 Application patterns and use case examples

Bullet Cache is created to address some specific use cases which cannot be solved (or are overly difficult to solve) with pure key-value caches, but at the same time to provide exceptionally high performance on multi-core servers.

The following sections describe examples of how *Bullet Cache* can be applied to common application problems and patterns.

3.1 General notes

Generally, caching can be used in all cases when it is more expensive to create a data object than to put or retrieve it from the cache - which covers a really wide area for its use. As beneficial as caching may be to overall application performance, there are some common problems associated with the idea: efficiency of cache operations (since the point of caching is to replace a slow operation with a fast one) and the problem of data freshness (as the data in the cache can get out of sync with the main data store).

Bullet Cache's performance is achieved by a highly efficient multithreaded program design with careful optimizations for maximum concurrency across multiple simultaneous clients. In addition to this it offers several ways by which data records can be moved in or out of the cache. One of these is by making use of the record tags, which enable applications to issue queries which go beyond simple key-value tags. By making use of record tags, the problem of maintaining cached data freshness can be simplified through expiring (deleting) a set of records matching certain tags.

Though *Bullet Cache* is designed specifically as a fast and feature-full cache server, it can also be used for some tasks which are only tangentially related to this core purpose.

3.2 Application object cache

One of the basic uses for *Bullet Cache* is as a cache for application objects. There are several reasons why such caches are desirable in modern Web application development:

- In stateless environments for Web applications such as PHP and most CGI environments the cache may be used to store "smart" session data, containing initialized objects used in the application across multiple requests.
- In applications where there are CPU-intensive operations with read-mostly final state, such as forums or news portals displaying complex Web pages made of hundreds of items, prepared or even pre-rendered (in HTML) objects can be put into the cache instead of executing the operations on every page visit. Such objects can be put into the cache by a back-end server, or a single front-end and then shared across multiple load-balanced Web servers.
- When generating complex reports which are accessed by multiple clients, the cache can be used to store either the final result (optionally even rendered as HTML) or intermediate results which increase the page display time.

The usual application pattern in this case is to create classes of objects or structures carrying the often accessed data in a way which requires the least processing after the data is fetched from the cache and before it is used (either for further preparation or directly for displaying to the user), make them serializable (marshalable) and create the interface to the cache in the program location where the data is supposed to be used.

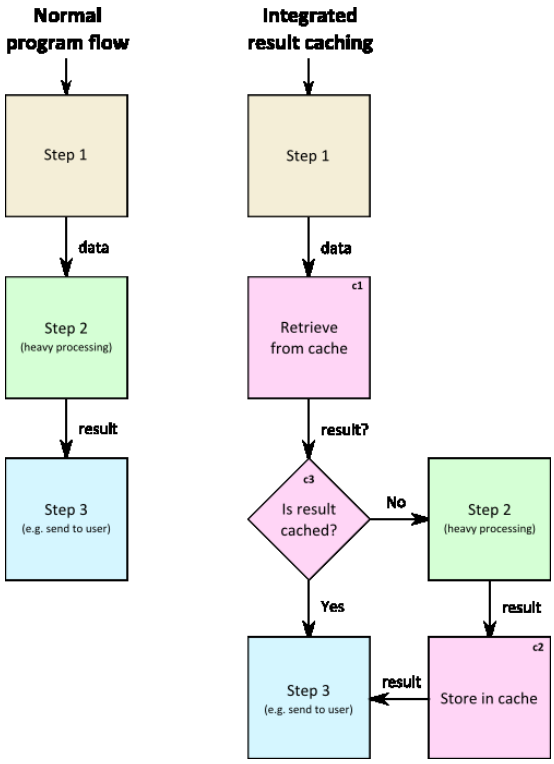

Figure 1: Integration of the cache layer in an application

The cache interface is often inserted in the program flow so that it conditionally replaces certain modules or parts of regular processing.

Tip

You can use application-level identifiers to tag the cached records with metadata. This enables you to issue bulk processing commands based on metadata queries, such as "*delete all cached records belonging to the user X*" or "*retrieve all cached records belonging to object Y*".

3.2.1 Example

Consider a Web page for a busy news site, containing a large number of "news items" in various forms - big news, less important news, stock market results, etc. All these data items need to come from somewhere and be processed / adapted to the Web before being sent to the visitor's browser. Though the majority of Web applications are "read-mostly", meaning that most of the visitors only view most of the content without changing it, the different properties of individual data items prohibit simple pre-rendering and caching of whole pages. The data items may come from different sources and have different "freshness" requirements. Some items may come from a local database, usually containing millions of similar items and accessed with complex queries, which makes it slow, but not as slow as other sources such as live news feeds from other agencies. Some items may change frequently - like stock market data, and others may change daily or less - like travel reports and interviews. All of these items must be combined together to create a single Web page, and a news site could have dozens or even hundreds of pages with similarly complex structure.

Bullet Cache can help! Each data item can be represented by an application object, including its final HTML form, and stored in the cache instead of being retrieved from the source, formatted and then sent to the visitor's browser.

Cached records can have metadata assigned to them in the form of record tags, specifying from which source the data item was originally fetched and on which pages it is presented to visitors. When composing a page, the application can use metadata queries to retrieve all cached records belonging to the page in a single, efficient operation. Similarly, when the data items need to be explicitly expired (e.g. when it is known the data has been changed, possibly by user interaction), all cached records for a

single page, a single site, or from a single source can be deleted in a single operation. Metadata query operations are powerful and can be used in many scenarios, depending only on the structure of tags assigned to cached records.

3.3 Database cache

Databases are the foundation of most non-trivial applications, and especially Web applications where the possibilities for client storage are very limited. Depending on application specifics and the business requirements it is supposed to implement, the database might contain a complex schema with a number of relations and even parts of the business logic - all of which bring great convenience and powerful data manipulation features to the application but at the same time mean that database queries grow complex and slow. Even if the database schema is relatively simple, the sheer number of records in the database might grow to such extent as to saturate available IO or even CPU processing capabilities of the database server.

The use of the application-level database cache allows the application full control over which data is cached and in what way. The principle is similar to that of creating an application object cache but on the level of database records, making it easier to implement in some circumstances.

The most convenient way to implement the database cache in an application is by establishing a central module (or a principle) which the application uses to access the database.

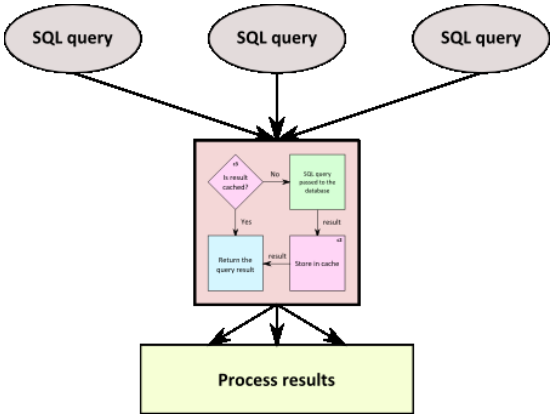

Figure 2: Centralized database module with integrated cache layer

This module can be supplemented with the following functionality:

- Instead of always executing SQL queries, their results may first be looked up in the cache. Techniques such as MD5 sums can be used to construct record keys from plaintext SQL queries in order to increase efficiency. Metadata in the form of tags can be assigned to cache records to enable efficient record expiry when needed. Care must be taken not to "cache" the result of certain database operations such as UPDATE and DELETE.
- If the freshness of data in the cache needs to be tightly controlled instead of relying on simple time-based cached record expiry, the module can detect operations involving certain database operations on respective database objects (such as UPDATE operations on some tables) and use record tags to efficiently expire data which is suspected to be modified. The granularity of such cache control depends on application requirements, but for many read-mostly Web applications, expiry based on table(s) touched by queries is enough.

Integrating a cache on the database layer can quickly make a huge impact on the overall application performance, but care must be taken not to inadvertently cache sensitive queries, such as those used related to application security system (e.g. user permissions).

3.3.1 Example

Some applications need to issue a large number of simple queries to their databases and some issue a small number of complex queries. In both cases, the database may be overloaded and cause the application to appear slow to its users.

Bullet Cache may help by caching the results of database queries! The recommended pattern in implementing a database cache in applications starts by having a central place - a module, or an object class - through which all of the database queries are performed.

There are two database usage patterns common to applications: the first is by directly sending SQL queries and commands (e.g. SELECT, UPDATE, DELETE) to the database, composing the query strings and processing the results in the application, and the second is by creating object classes as wrappers for database entities, which may then either proceed to issue direct SQL commands or use other means such as stored procedures to perform queries and commands.

If SQL strings are used directly, the central module can inspect the strings and if a SELECT query is detected it can proceed to check if the result of the query is cached, and return the cached record data if it is. If object wrappers for relational entities are used, the application can adapt the methods querying the database to check for cached records.

In any case, there is an issue of keeping the cached records "fresh" with respect to the database contents. One way to solve it is to tag the cached records with metadata designating database entities which are involved in their creation. The central module can detect operations which modify these entities (e.g. database tables) and issue metadata queries which explicitly expire (delete) such cached records.

3.4 Primary data store

Although *Bullet Cache* is not intended to be a persistent database, its fast operation and scalability can be desirable features for certain applications, for example:

- *Logging.* Application logs can be very valuable for both developers and system administrators, but the cost of on-disk logging or network logging might be too high with traditional utilities. *Bullet Cache* offers an interface with which simple data-only records can be sent to the server. Such records can contain text or binary data - the interpretation is left to the applications and utilities working with the records.
- *Volatile data.* Certain data needs to be stored only for a short time, or is completely unimportant in the long term. Since *Bullet Cache* is a fast memory-only database, it can be used for efficient storage of such data.
- *Unimportant data.* For certain data, its storage might be delayed and also unimportant in the long term. *Bullet Cache* can be configured to periodically create file dumps of cached data, which can either be used as continuous backups or as the primary data storage. Note that all data sent to the server between file dumps is naturally lost.

In general, *Bullet Cache* can be considered as a special, non-persistent NoSQL database, useful to some applications.

3.5 Application data sharing

Bullet Cache operates as a network server which efficiently supports a large number of connected clients, so it can be used as a "meeting ground" in which applications share data. Some examples for such data sharing are:

- Multiple Web application front-ends sharing state information - session data or information about the currently connected user.
- Stateful applications can make use of *Bullet Cache* to achieve scalability when deployed in a load balancing configuration across multiple servers - the task of sharing data between the application instances can be offloaded to *Bullet Cache*.
- Multiple application or database servers can use *Bullet Cache* to maintain a global serial (autoincrement) counter, which is used safely through the use of "atomic" operations.

When used in this way, *Bullet Cache* behaves as shared storage.

3.6 Distributed lock manager

Among operations supported by *Bullet Cache* are the so-called *ATOMIC operations* on data records which can be used as primitives in creating a distributed lock manager. The atomic operations handle records which contain 64-bit signed integers and perform one of the following (see the ATOMIC API for details):

- `CMPSET`, which sets a record to the given value only if it already contains another specified value
- `ADD`, which adds a value to an existing record (or creates a new record)
- `FETCHADD`, which behaves similarly to `ADD` but returns the value of the record before the addition operation is carried out.
- `READANDCLEAR`, which clears the record (set it to 0) but returns the value the record contained before it was cleared.

The primary use case for this set of operations is to support reliable counters and similar numeric data operations, but they can be used to implement synchronization functions to be used by multiple connected clients.

3.6.1 Example

The CMPSET operation can be used to build a "mutex" synchronization operation for use between multiple clients. The basic mutex implementation is presented here in pseudocode:

```
bool try_lock_mutex(m) {
    if (cmpset(m, 0, 1))
        return true;
    else
        return false;
}

void lock_mutex(m) {
    while (!try_lock_mutex(m))
        sleep();
}

void unlock(m) {
    assert(readandclear(m) == 1);
}
```

Note that mutexes in the above example are not "robust" in the sense that if any of the clients disconnects (e.g. crashes) without unlocking the mutex, all other clients will see the mutex locked, creating a deadlock-like situation.

3.7 Distributed stack

Another type of supported operations are the virtual tag stack (TSTACK) operations. Each record tag can be considered to classify records into virtual stacks (FIFO queues), and there are TSTACK API functions which operate on the records in the cache server as if they belong to such stacks. As there can be arbitrary many different record tags (the whole key-value pair of a tag is considered to designate a virtual stack), there can also be arbitrary many different virtual stacks.

Note that the FIFO property of virtual stacks is only incidental and the "virtual stacks" can also be considered as ordinary unordered lists (i.e. where the ordering is not important).

3.8 Message queue

Bullet Cache can be used as a non-persistent (i.e. memory-only), polling message queue simply by using the virtual stack feature from multiple clients. Applications can use the TSTACK API to pass messages between themselves through the cache server. Each virtual stack can be considered a single message queue.

The structure of these messages is not in any way dictated by the cache server, they are normal opaque records stored in the cache.

Note that *Bullet Cache* is not a persistent database (barring periodical snapshots) and so the message queue is also not persistent. By involving the timed record expiry in the these operations, the messages can be treated as auto-expiring, but this might interfere with the FIFO nature of the queue.

3.9 Interactive application backend

An interesting use case for *Bullet Cache* can be found in interactive applications which may or may not be web-based. Modern web applications tend to rely on AJAX and similar technologies which enable them constant interaction with the web server backend. While the web server could with some care handle this kind of load with a large number of very small requests, it is unlikely that a heavy-weight database would handle it. *Bullet Cache* can help if the application is designed to use it instead of the database for the majority of the requests. On the other hand, non web-based interactive applications have a similar problem where a very large number of clients connect to a small number of servers and are constantly changing application state. Instead of storing this state in a heavy-weight database, it can be stored in *Bullet Cache* and then periodically transferred in bulk to the persistent storage.

3.9.1 Example

An on-line interactive game might need to store data on the players currently playing - such as the player's positions and inventory. This kind of data is likely to change very often (especially the players' positions) and would easily overload a conventional database, but building a game-specific server would be a non-trivial task. Using *Bullet Cache*, players' basic state (position, health) can be stored in one type of record and their positions in another. Since *Bullet Cache* is made specifically to handle a large number of small requests, constant changes in this kind of data will not hinder its performance.

Another benefit to *Bullet Cache* is its ability to query records by tags, which can be used to e.g. find all players in a certain area (if the areas are discretely numbered).

4 Technical details

Though *Bullet Cache* can be used as a simple and fast key-value cache server, more interesting and more efficient use can be achieved by understanding its advanced features and planning the application so it maximally uses these features. The most important efficiency feature of *Bullet Cache* is its data model.

4.1 The Data Model

The core of the *Bullet Cache*'s data model is the key-value dictionary found in many similar servers, where both the key and the value are arbitrary binary strings (i.e. there are no constraints on their contents - they are not limited to ASCII strings and can contain `\0` characters. The key-value pair with its associated metadata is called a *record*. The maximum length for keys is 65534 bytes (64 KiB -2) and the maximum length of values is 2147483646 (2 GiB -2). Each record has the following metadata assigned to it:

- Record expiry time
- Record tags

These are described in the following sections.

4.2 Record expiry time

Record expiry time is an machine-dependent integer (time_t type) representing the classic Unix timestamp: the number of seconds since the Unix epoch. On entry into the cache server, expiry timestamps are interpreted in one of the following ways, depending on their absolute values:

- If the timestamp is 0, it is automatically converted to the largest possible timestamp, representing effective infinity (i.e. the record does not automatically expire).
- If the timestamp is between 0 and REL_TIME_CUTOFF (usually 2592000), it is interpreted as relative to the current server time, and adjusted accordingly. This format allows the quick specification of short expiry times: a value of "30" means the record will expire 30 seconds after it is received at the server.
- Otherwise, the timestamp is interpreted as an absolute Unix timestamp, in the timezone of the server.

On exit from the cache server (i.e. at record retrieval), the original, client-specified timestamp value is returned.

4.3 Record tags

Each key-value pair can have an arbitrary number of *record tags* assigned to it. Record tags are themselves structured as key-value pairs, where both are 32-bit signed integers.

Simplified Bullet Cache record structure

```
+--------------------+-------------------+
| record_key         | record_key_size   |
+--------------------+-------------------+
| record_value       | record_value_size |
+--------------------+-------------------+
| expiry_time        |
+--------------------+---------------------+
| tag_key | tag_value | tag_key | tag_value |
| int32_t |  int32_t  | int32_t |  int32_t  | ...
+--------------------+---------------------+
```

The tags are of deliberately constrained data type to maximize performance while at the same time offering flexibility to application authors. The interpretation of the tags is application-specific - the cache server considers them opaque.

The introduction of record tags makes the biggest difference in efficiency compared to the more common key-value only cache server. It enables applications to issue commands operating on a large number of records at once. Using records, application can retrieve or delete sets of records grouped by some criteria using the tags.

The presence of tags makes implementation of advanced functions possible, such as treating the cache server as a more complex database than a pure key-value database, or accessing the tagged records in the cache server in the form of an array or a stack.

4.4 Data consistency model

Bullet Cache implements a deliberate consistency model in its operation. In general, all operations working on multiple records at once have at least an optional flag that makes the operations be performed atomically on all records. This means that the operations are performed on a snapshot of the cache server date at some time before the operations return their result, and independently of other operations going on in the cache server.

For example, the atomic variants of operations querying and returning a set of records always return all the records found at the start of the operation execution, while operations that would delete those records are postponed / blocked until the records are returned.

Important
Operations on multiple records do not in any way mean that the records are somehow "grouped." Records can be retrieved and deleted individually by other operations, including those of the server's own record expiry mechanisms (either by time or by memory usage).

4.5 Concurrency model

When comparing with the traditional database concurrency models, the *Bullet Cache* can be said to implement page-level locking with shared-exclusive locks.

The usage of shared-exclusive locks means that operations which only access the data for reading can always be concurrently executed with any other read-only operations. All operations which modify records in any way (deleting, inserting) must take an exclusive lock and thus they block concurrent access for all other operations. The granularity of concurrency in *Bullet Cache* is dictated by hash tables of locks which are separate for records and tags, enabling an efficient dispersal of locking operations. In the most common case (light loads) and with random record access, it is reasonable to expect that the locks protect individual records.

Note that this discussion is mostly relevant in the edge cases under high server loads, as the cache operations are very fast themselves and the locks are only acquired for very, very short times.

4.6 Performance considerations

Though maximum record sizes constraints are relatively permissive, performance oriented applications should keep both keys and values as short as possible on general principles. As a recommendation, the keys should be limited to around 100 bytes for best efficiency.

A similar recommendation can be made about the number of distinct record tag *keys* in the cache server database (total across all records) be limited to around 500 for max efficiency and that the number of tags per record be limited to around a dozen.

Note that these are only general recommendations for maximizing the cache server efficiency, not precise rules and not hard limits. The client API documentation includes performance estimates which are very general and may improve in the future.

The *Bullet Cache*'s web site contains some performance measurements.

5 Client API documentation

The primary client API used with *Bullet Cache* is the C API, implemented as a shared library. All other official client APIs are implemented on top of this C library. The *Bullet Cache* uses a binary, machine-specific protocol to communicate with the server (so the client and the server must share the same CPU architecture). This is most easily achieved by using the C library.

The client-server connection is represented by the `mc_connection` structure which is created and returned by the `mc_connect_*` functions. Most of the API functions generally follow this template: they accept a pointer to this structure as their first argument, a `char**` pointer as their last, and they return an integer indicating the function's success. In case this result is not `MC_RESULT_OK`, the error pointer can (optionally) be initialized by the function to point to an error string (which could be presented to the user) and which should be freed by the caller with a simple `free()`.

Unless otherwise documented, functions returning an integer return of the following result indicator constants:

Symbol	Interpretation
MC_RESULT_OK	No error
MC_RESULT_STATUS	Protocol-level error
MC_RESULT_NETWORK	Network-level error (e.g. the connection is closed)
MC_RESULT_NOT_FOUND	The requested records have not been found
MC_RESULT_UNEXPECTED	The server returned an invalid or unexpected response
MC_INVALID_PARAM	The application has passed invalid data to the library
MC_RESULT_STATE	The connection is in invalid state for the operation
MC_ENOMEM	Memory allocation error

5.1 Connecting to the server

C:

```
struct mc_connection *mc_connect_local(char *unix_socket, int do_handshake,
    char **error_msg);
struct mc_connection *mc_connect_tcp(char *host_name, int port, int do_handshake,
    char **error_msg);
void mc_close_connection(struct mc_connection *mc);
```

PHP:

```
resource mc_connect_local(string $local_socket[, bool $do_handshake]);
resource mc_connect_tcp(string $host_name, int $port [, bool $do_handshake]);
void mc_close_connection(resource $mc);
```

The `mc_connect_local()` and `mc_connect_tcp()` functions connect to the *Bullet Cache* server and return a connection structure pointer / resource. The connection is closed and this structure / resource is freed by the `mc_close_connection()` function. The `do_handshake` argument specifies if the client library will issue a "handshake" request to the server before the

functions return, establishing such details as protocol version match. This is only necessary if it is expected that the client and the server can have a version mismatch or CPU architecture mismatch.

5.1.1 Return values

The `mc_connect*` functions return the `mc_connection` structure or resource which is used for all other operations. `NULL` is returned on error.

5.1.2 Performance

Local Unix socket connections can be an order of magnitude faster than TCP connections.

5.1.3 Concurrency

Each client application thread must create its own connection to the server or must otherwise ensure that shared `mc_connection` objects are not used concurrently.

5.2 Getting, putting and deleting a simple record

C:

```
int mc_put_simple(struct mc_connection *cn, char *key, unsigned int key_len,
    char *value, unsigned int data_len, time_t exptime, char **error_msg);
int mc_get_simple(struct mc_connection *cn, char *key, unsigned int key_len,
    char **data, unsigned int *data_len, char **error_msg);
int mc_del_simple(struct mc_connection *cn, char *key, unsigned int key_len,
    char **error_msg);
```

PHP:

```
int mc_put_simple(resource $cn, string $key, string $value[, int $exptime]);
string mc_get_simple(resource $cn, string $key);
bool mc_del_simple(resource $cn, string $key);
```

The simple record functions put, retrieve or delete a single key-value record to or from the cache server in a simple way. The record expiry time (`exptime`) for `mc_put_simple()` can be absolute or relative.

The C API version of `mc_get_simple()` returns the record value in the `data` argument which should be freed by the caller.

5.2.1 Return values

All C functions return `MC_RESULT_OK` on success, or another `MC_RESULT_*` constant. The `mc_get_simple` returns the found record's data in the provided `**data` argument, and the data length in the `*data_len` argument (or `MC_RESULT_NOT_FOUND` if the record is not found).

5.2.2 Performance

The "simple" GET and PUT operations are the most optimized in the *Bullet cache* server. The simple GET operation is the fastest non-trivial operation the server can perform, and it is the basis to which all other read-only operations are compared in performance.

When comparing performance, the simple GET and PUT operations have the base performance index of 1.0. Operations which are estimated to be more costly (i.e. slower) have a performance index larger than 1.0.

5.2.3 Concurrency

The GET operation supports nearly unlimited concurrency: multiple client connections can perform GET operations on all records without waiting, unless the data they have requested is being modified by other connections at the same time.

The PUT operation supports statistical hash bucket concurrency: up to 256 (by default) client connections can perform PUT operations on records with different keys, as long as there are no hash collisions between the keys. A PUT request will block all other concurrent PUT and GET requests on records with hash collisions on the same key.

5.3 Putting and getting a record with tags

C:

```
int mc_put_simple_tags(struct mc_connection *cn, char *key, unsigned int key_len,
    char *value, unsigned int value_len, struct mc_tag *tags, unsigned int n_tags,
    time_t exptime, char **error_msg);

alias: mc_put

int mc_get_simple_tags(struct mc_connection *mc, char *key,
    unsigned int key_len, char **data, unsigned int *data_len,
    struct mc_tag **tags, unsigned int *n_tags, char **error_msg);

alias: mc_get
```

PHP:

```
int mc_put_simple_tags(resource $cn, string $key, string $data, array $tags,
    int $exptime);

alias: mc_put

array mc_get_simple_tags(resource $cn, string $key);
```

Records in the *Bullet Cache* server also contain record tags, and there are API functions which can be used to put tagged records into the cache.

In the C API, the tags are represented by the `struct mc_tag`, or more commonly by a simple array of these structures. The structure is defined as:

```
struct mc_tag {
    tag_key_t key;
    tag_value_t value;
};
```

In the PHP API, the tags are always represented by a key-value array where both the keys and the values are integers.

5.3.1 Return values

The C functions return `MC_RESULT_OK` on success, or another `MC_RESULT_*` constant.

The PHP function `mc_put_simple_tags()` behaves like its C counterpart. The `mc_get_simple_tags` returns an array whose structure is described by the following example:

```
array(
    "record_key" => array(
            "key" => "record_key",
            "value" => "record_value",
            "_tags" => array(
                    1 => 2,
                    1024 => 1025
                    ...
                )
        )
)
```

5.3.2 Performance

Compared to baseline PUT performance, the `mc_put_simple_tags` operation is estimated to have the performance index of at least 1.1, but it depends on the number of record tags and the cardinality of unique tags in the cache.

5.3.3 Concurrency

See simple PUT operation concurrency.

5.4 Querying and deleting the records by their tags

C:

```
int mc_get_by_tag_values(struct mc_connection *cn, tag_key_t key,
    tag_value_t *tag_values, unsigned n_values,
    struct mc_multidata_result **result, char **error_msg);
int mc_del_by_tag_values(struct mc_connection *cn, tag_key_t key,
    tag_value_t *tag_values, unsigned n_values, unsigned *n_del, char **error_msg) ↩
       ;
```

PHP:

```
array mc_get_by_tag_values(resource $cn, int $tag_key, array $tag_values);
int mc_del_by_tag_values(resource $cn, int $tag_key, array $tag_values);
```

The main benefit of record tags is for them to be used in queries such as these. The `mc_get_by_tag_values()` and `mc_del_by_tag_values()` get or retrieve records (respectively) which are tagged with the given tag key and contain any of the given tag values. This enables application developers to efficiently retrieve or delete (expire) cached records or a certain type or belonging to a certain application-specific class (such as owned by a user, present on a web page, etc.). See use case examples.

The C functions pass the list of tag values as a simple array of integers.

The PHP functions are straightforward. The GET function returns a key-value array with the found records. The DEL function returns the number of deleted records.

5.4.1 Return values

All C functions return `MC_RESULT_OK` on success, or another `MC_RESULT_*` constant. The GET function receives the result in the form of `struct mc_multidata_result **result`, which is defined as:

```
struct mc_multidata_result {
     unsigned int n_records;
     struct mc_resp_multidata *pkt;
     struct mc_data_entry **records;
};
```

The application can inspect the `n_records` member of this structure and use the following helper functions to extract information from the `records` member:

```
struct mc_tag* mc_data_entry_tags(struct mc_data_entry *md);
unsigned mc_data_entry_n_tags(struct mc_data_entry *md);

char* mc_data_entry_key(struct mc_data_entry *md);
unsigned mc_data_entry_key_size(struct mc_data_entry *md);

char* mc_data_entry_value(struct mc_data_entry *md);
unsigned mc_data_entry_value_size(struct mc_data_entry *md);
```

These functions are provided for convenience of inspecting and extracting useful data from `struct mc_data_entry` records.

The caller should use `mc_multidata_result_free()` to free the returned `struct mc_multidata_result`. The DEL function returns the number of deleted records in the `n_del` argument (if non-NULL).

5.4.2 Performance

The performance of these functions depends on the number of passed tag values and the cardinality of unique tags in the cache, but the performance index compared to baseline performance of simple operations is expected to be at least 1.4*n_values.

5.4.3 Concurrency

These functions perform their operations atomically. In other words, the returned (or deleted) set of records is a snapshot of the state of the cache server at some time before the functions return. All the relevant found records are locked for the duration of the operations.

The GET and DEL operations have concurrency capabilities similar to the simple operations but with significantly higher constants: the operations themselves are slower and the number of records that can potentially be blocked is higher.

5.5 Resetting record tags

C:

```
int mc_set_tags(struct mc_connection *cn, char *key, unsigned key_len,
    struct mc_tag *tags, unsigned n_tags, char **error_msg);
```

PHP:

```
int mc_set_tags($cn, $key, $tags);
```

As a convenience, the record's tags can be reset to the given list by using the `mc_set_tags()` function. This function does not modify any other data associated with the record.

5.5.1 Return values

The function returns `MC_RESULT_OK` on success, or another `MC_RESULT_*` constant.

5.5.2 Performance

The performance index of `mc_set_tags()` compared to a simple PUT operation is estimated to be around 1.1.

5.5.3 Concurrency

The current implementation of `mc_set_tags()` has very bad concurrency characteristics, effective serializing all other operations which operate on record tags.

5.6 Atomic record value operations

C:

```
int mc_atomic_op(struct mc_connection *cn, int op, char *key,
    unsigned int key_len, int64_t arg1, int64_t arg2, int64_t *result,
    char **error_msg);

int mc_atomic_put(struct mc_connection *cn, char *key, unsigned key_len,
    int64_t value, struct mc_tag *tags, unsigned n_tags, time_t exptime,
    char **error_msg);

int mc_atomic_get(struct mc_connection *cn, char *key, unsigned key_len,
    int64_t *res, char **error_msg);

aliases:
    mc_atomic_cmpset(cn, key, key_len, arg1, arg2, result, err)
    mc_atomic_add(cn, key, key_len, arg1, result, err)
    mc_atomic_fetchadd(cn, key, key_len, arg1, result, err)
    mc_atomic_readandclear(cn, key, key_len, result, err)
```

PHP:

```
bool mc_atomic_cmpset($cn, $key, $arg1, $arg2);
int mc_atomic_add($cn, $key, $arg);
int mc_atomic_fetchadd($cn, $key, $arg);
int mc_atomic_readandclear($cn, $key);
int mc_atomic_put($cn, $key, $arg1 [, $tags, $expiry_time]);
int mc_atomic_get($cn, $key);
```

The atomic helper operations treat records specially: as 64-bit signed integers, and offer a number of "atomic" operations to be performed on them. The "atomicity" of operations simply means that the operations are strongly guaranteed to be performed on the cache server without any interference from other concurrently connected clients (and any other operations). This feature allows the creation of reliable counters and even distributed application locks. See use case examples.

The `CMPSET` operation can be described as: *"if (value == arg1) { value = arg2; return 1; } else { return 0; }"*.

The `ADD` operation can be described as: *"value += arg; return value;"*.

The `FETCHADD` operation can be described as: *"tmp = value; value += arg; return tmp;"*.

The `READANDCLEAR` operation can be described as: *"tmp = value; value = 0; return tmp;"*.

Though the C client really implements only one function from scratch (`mc_atomic_op()`), it is recommended that the applications use the helper aliases (`mc_atomic_cmpset()`, `mc_atomic_add()`, `mc_atomic_fetchadd()` and `mc_atomic_reada` for clarity.

The `ADD` and `CMPSET` operations can be used to create a new record for use with the atomic operations. In case of the `ADD` operation, the newly created record will contain the passed argument. In case of the `CMPSET` operation, the newly created record will contain the second argument.

All operations will fail if the record specified for the operation exists and does not contain a 64-bit quantity.

The `mc_atomic_put()` and `mc_atomic_get()` functions are auxiliary, present for conventience and ease of use. They are a simple way to create or retrieve a record for use with the atomic operations as "cooked" data, already interpreted as a signed 64-bit integer instead of being treated as a binary blob. The `tags` argument is optional for `mc_atomic_put()`.

5.6.1 Return values

All C functions return `MC_RESULT_OK` on success, or another `MC_RESULT_*` constant. The result of the atomic operations is returned in the provided `*result` argument.

The result of PHP functions varies: the `mc_atomic_cmpset()` function returns whether the "set" operation has been carried out, the ADD, FETCHADD and READANDCLEAR functions directly return the result of the operation as previously described, or FALSE on failure. The `mc_atomic_put()` function returns the `MC_RESULT_*` constant, and the `mc_atomic_get()` returns the record value interpreted as a 64-bit signed integer (may overflow on 32-bit systems).

5.6.2 Performance

The performance characteristics of the atomic record value operations are only a bit slower than that of the the simple PUT operation (estimated performance index 1.05).

5.6.3 Concurrency

Concurrency of the atomic record value operations is comparable to that of the simple PUT operation.

5.7 Getting, putting and deleting multiple records

C:

```
int mc_mget(struct mc_connection *cn, uint16_t n_records,
    uint16_t *key_lengths, char **keys, unsigned flags,
    struct mc_multidata_result **result, char **error_msg);
int mc_mput(struct mc_connection *cn, uint16_t n_records,
    struct mc_data_entry **records, unsigned flags, char **error_msg);
int mc_mdel(struct mc_connection *cn, uint16_t n_records, uint16_t *key_lengths,
    char **keys, unsigned flags, unsigned *n_deleted, char **error_msg);
```

PHP:

```
array mc_mget(resource $cn, array $keys[, int $flags]);
int mc_mput(resource $cn, array $records[, int $flags]);
int mc_mdel(resource $cn, array $keys[, int $flags]);
```

These operations retrieve, put or delete multiple records at once, which is useful for avoiding communication overhead when working with a set of records. However, the operations are *slower* than simple operation equivalents if they only operate on a single record.

The `flags` passed to can include `MCMD_FLAG_ATOMICALL` which indicates that the operation is to be performed atomically with regards to other operations on the server.

5.7.1 Return values

All C functions return `MC_RESULT_OK` on success, or another `MC_RESULT_*` constant. The `mc_mget()` function returns one or more results as the `mc_multidata_result` structure, which can be inspected with provided utility functions (or `MC_RESULT_NOT_FOUND` if not found). The function may or may not return all of the records requested (depending on if they are present on the server) and in any order. The `mc_mdel()` function returns the number of deleted records in the `n_deleted` argument.

5.7.2 Performance

The performance index of the multiple-record operations is estimated to be in the ballpark of 1.2+(n_records*0.9), with respect to the simple operations. However, they are usually faster than issuing a large number of simple operations because they avoid the communication overhead present with multiple simple operations.

5.7.3 Concurrency

If the `MCMD_FLAG_ATOMICALL` flag is present on the operation, all relevant records are found and locked before the operations are executed. Otherwise, the concurrency is comparable to that of simple operations.

5.8 Pushing and popping records in a stack

C:

```
int mc_tstack_push(struct mc_connection *cn, struct mc_tag *tags,
    unsigned n_tags, char *data, unsigned data_size, char **new_key,
    unsigned *new_key_size, char **error_msg);
int mc_tstack_pop(struct mc_connection *cn, tag_key_t tkey, tag_value_t tval,
    struct mc_data_entry **mce, char **error_msg);
```

PHP:

```
string mc_tstack_push(resource $cn, string $record_value[, array $tags]);
array mc_tstack_pop(resource $cn, int tag_key, int tag_value);
```

The record tags feature in the cache server can be used to implement virtual stacks (TSTACK). Each such virtual stack is uniquely identified by a tag (the whole key-value pair of a tag), and indeed every tagged record can be said to be inserted to virtual stacks. As such, there are no special operations which insert records to virtual stacks, but the `mc_tstack_push()` operation is provided as a convenience. This method accepts only the record value and its tags, leaving the record key to be auto-generated on the server and returned to the caller.

The `mc_tstack_pop()` operation retrieves and deletes (i.e. "pops") a record from a virtual stack identified by its tag key and tag value.

The C API function `mc_tstack_pop` returns the whole raw record in the `mce` argument, which should be freed by the caller. Helper functions can be used to inspect this record.

5.8.1 Return values

All C functions return `MC_RESULT_OK` on success, or another `MC_RESULT_*` constant. The `mc_tstack_push()` returns the new unique record key in the provided `**new_key` argument. The `mc_tstack_pop()` function returns the found record in the `mc_data_entry` structure which can be inspected with provided utility functions (or `MC_RESULT_NOT_FOUND` if not found).

5.8.2 Performance

The performance index of the `mc_tstack_push()` operation is estimated to be 1.2.

The `mc_tstack_pop()` is a much more complex operation, with the performance index estimated to be around 1.7.

5.8.3 Concurrency

The concurrency of `mc_tstack_push()` is similar to the concurrency of the simple PUT operation.

The concurrency of `mc_tstack_pop()` is somewhat worse than that of the simple operations, with locking required for both the tags and the records. Concurrency is limited to virtual stacks with different tag keys (i.e. there can be no concurrent operation on virtual stacks with the same tag keys or keys which hash to the same value), in addition to "regular" concurrency issues similar to that of the simple record DELETE operation.

Note that concurrent access to the records (specifically, deletion) which are also accessed through the TSTACK API functions can interfere with the FIFO ordering of the stacks. This also applies to automatic timed record expiry.

5.9 API client / driver notes

This section describes client / driver specific behavior important to application developers.

5.9.1 General notes

All official clients are currently based on the C client library. This library (and indeed the network protocol) contains no multi-threading support, meaning that all client application threads which need access to the cache server must either establish their own local connections (`struct mc_connection`) or they must implement serialization (concurrency protection) on the shared connection.

5.9.2 C client notes

With all API functions which return a pointer value (via a double-indirect `**` pointer) to data allocated by the function, this pointer should be freed by the caller, either directly with `free()` or by a provided helper function (depending on the circumstances).

5.9.3 PHP client notes

The PHP driver is implemented as a PHP module written in C, using the standard C API library. The PHP driver exposes errors from the client as regular PHP errors with the severity of `E_ERROR`, and the return value from such functions is usually `FALSE`.

Operations returning records return `NULL` if no records are found.