Protocol

--------

Clients of memcached communicate with server through TCP connections.

(A UDP interface is also available; details are below under "UDP

protocol.") A given running memcached server listens on some

(configurable) port; clients connect to that port, send commands to

the server, read responses, and eventually close the connection.

There is no need to send any command to end the session. A client may

just close the connection at any moment it no longer needs it. Note,

however, that clients are encouraged to cache their connections rather

than reopen them every time they need to store or retrieve data. This

is because memcached is especially designed to work very efficiently

with a very large number (many hundreds, more than a thousand if

necessary) of open connections. Caching connections will eliminate the

overhead associated with establishing a TCP connection (the overhead

of preparing for a new connection on the server side is insignificant

compared to this).

There are two kinds of data sent in the memcache protocol: text lines

and unstructured data. Text lines are used for commands from clients

and responses from servers. Unstructured data is sent when a client

wants to store or retrieve data. The server will transmit back

unstructured data in exactly the same way it received it, as a byte

stream. The server doesn't care about byte order issues in

unstructured data and isn't aware of them. There are no limitations on

characters that may appear in unstructured data; however, the reader

of such data (either a client or a server) will always know, from a

preceding text line, the exact length of the data block being

transmitted.

Text lines are always terminated by \r\n. Unstructured data is \_also\_

terminated by \r\n, even though \r, \n or any other 8-bit characters

may also appear inside the data. Therefore, when a client retrieves

data from a server, it must use the length of the data block (which it

will be provided with) to determine where the data block ends, and not

the fact that \r\n follows the end of the data block, even though it

does.

Keys

----

Data stored by memcached is identified with the help of a key. A key

is a text string which should uniquely identify the data for clients

that are interested in storing and retrieving it. Currently the

length limit of a key is set at 250 characters (of course, normally

clients wouldn't need to use such long keys); the key must not include

control characters or whitespace.

Commands

--------

There are three types of commands.

Storage commands (there are six: "set", "add", "replace", "append"

"prepend" and "cas") ask the server to store some data identified by a

key. The client sends a command line, and then a data block; after

that the client expects one line of response, which will indicate

success or failure.

Retrieval commands (there are two: "get" and "gets") ask the server to

retrieve data corresponding to a set of keys (one or more keys in one

request). The client sends a command line, which includes all the

requested keys; after that for each item the server finds it sends to

the client one response line with information about the item, and one

data block with the item's data; this continues until the server

finished with the "END" response line.

All other commands don't involve unstructured data. In all of them,

the client sends one command line, and expects (depending on the

command) either one line of response, or several lines of response

ending with "END" on the last line.

A command line always starts with the name of the command, followed by

parameters (if any) delimited by whitespace. Command names are

lower-case and are case-sensitive.

Expiration times

----------------

Some commands involve a client sending some kind of expiration time

(relative to an item or to an operation requested by the client) to

the server. In all such cases, the actual value sent may either be

Unix time (number of seconds since January 1, 1970, as a 32-bit

value), or a number of seconds starting from current time. In the

latter case, this number of seconds may not exceed 60\*60\*24\*30 (number

of seconds in 30 days); if the number sent by a client is larger than

that, the server will consider it to be real Unix time value rather

than an offset from current time.

Error strings

-------------

Each command sent by a client may be answered with an error string

from the server. These error strings come in three types:

- "ERROR\r\n"

means the client sent a nonexistent command name.

- "CLIENT\_ERROR <error>\r\n"

means some sort of client error in the input line, i.e. the input

doesn't conform to the protocol in some way. <error> is a

human-readable error string.

- "SERVER\_ERROR <error>\r\n"

means some sort of server error prevents the server from carrying

out the command. <error> is a human-readable error string. In cases

of severe server errors, which make it impossible to continue

serving the client (this shouldn't normally happen), the server will

close the connection after sending the error line. This is the only

case in which the server closes a connection to a client.

In the descriptions of individual commands below, these error lines

are not again specifically mentioned, but clients must allow for their

possibility.

Storage commands

----------------

First, the client sends a command line which looks like this:

<command name> <key> <flags> <exptime> <bytes> [noreply]\r\n

cas <key> <flags> <exptime> <bytes> <cas unique> [noreply]\r\n

- <command name> is "set", "add", "replace", "append" or "prepend"

"set" means "store this data".

"add" means "store this data, but only if the server \*doesn't\* already

hold data for this key".

"replace" means "store this data, but only if the server \*does\*

already hold data for this key".

"append" means "add this data to an existing key after existing data".

"prepend" means "add this data to an existing key before existing data".

The append and prepend commands do not accept flags or exptime.

They update existing data portions, and ignore new flag and exptime

settings.

"cas" is a check and set operation which means "store this data but

only if no one else has updated since I last fetched it."

- <key> is the key under which the client asks to store the data

- <flags> is an arbitrary 16-bit unsigned integer (written out in

decimal) that the server stores along with the data and sends back

when the item is retrieved. Clients may use this as a bit field to

store data-specific information; this field is opaque to the server.

Note that in memcached 1.2.1 and higher, flags may be 32-bits, instead

of 16, but you might want to restrict yourself to 16 bits for

compatibility with older versions.

- <exptime> is expiration time. If it's 0, the item never expires

(although it may be deleted from the cache to make place for other

items). If it's non-zero (either Unix time or offset in seconds from

current time), it is guaranteed that clients will not be able to

retrieve this item after the expiration time arrives (measured by

server time). If a negative value is given the item is immediately

expired.

- <bytes> is the number of bytes in the data block to follow, \*not\*

including the delimiting \r\n. <bytes> may be zero (in which case

it's followed by an empty data block).

- <cas unique> is a unique 64-bit value of an existing entry.

Clients should use the value returned from the "gets" command

when issuing "cas" updates.

- "noreply" optional parameter instructs the server to not send the

reply. NOTE: if the request line is malformed, the server can't

parse "noreply" option reliably. In this case it may send the error

to the client, and not reading it on the client side will break

things. Client should construct only valid requests.

After this line, the client sends the data block:

<data block>\r\n

- <data block> is a chunk of arbitrary 8-bit data of length <bytes>

from the previous line.

After sending the command line and the data block the client awaits

the reply, which may be:

- "STORED\r\n", to indicate success.

- "NOT\_STORED\r\n" to indicate the data was not stored, but not

because of an error. This normally means that the

condition for an "add" or a "replace" command wasn't met.

- "EXISTS\r\n" to indicate that the item you are trying to store with

a "cas" command has been modified since you last fetched it.

- "NOT\_FOUND\r\n" to indicate that the item you are trying to store

with a "cas" command did not exist.

Retrieval command:

------------------

The retrieval commands "get" and "gets" operate like this:

get <key>\*\r\n

gets <key>\*\r\n

- <key>\* means one or more key strings separated by whitespace.

After this command, the client expects zero or more items, each of

which is received as a text line followed by a data block. After all

the items have been transmitted, the server sends the string

"END\r\n"

to indicate the end of response.

Each item sent by the server looks like this:

VALUE <key> <flags> <bytes> [<cas unique>]\r\n

<data block>\r\n

- <key> is the key for the item being sent

- <flags> is the flags value set by the storage command

- <bytes> is the length of the data block to follow, \*not\* including

its delimiting \r\n

- <cas unique> is a unique 64-bit integer that uniquely identifies

this specific item.

- <data block> is the data for this item.

If some of the keys appearing in a retrieval request are not sent back

by the server in the item list this means that the server does not

hold items with such keys (because they were never stored, or stored

but deleted to make space for more items, or expired, or explicitly

deleted by a client).

Deletion

--------

The command "delete" allows for explicit deletion of items:

delete <key> [noreply]\r\n

- <key> is the key of the item the client wishes the server to delete

- "noreply" optional parameter instructs the server to not send the

reply. See the note in Storage commands regarding malformed

requests.

The response line to this command can be one of:

- "DELETED\r\n" to indicate success

- "NOT\_FOUND\r\n" to indicate that the item with this key was not

found.

See the "flush\_all" command below for immediate invalidation

of all existing items.

Increment/Decrement

-------------------

Commands "incr" and "decr" are used to change data for some item

in-place, incrementing or decrementing it. The data for the item is

treated as decimal representation of a 64-bit unsigned integer. If

the current data value does not conform to such a representation, the

incr/decr commands return an error (memcached <= 1.2.6 treated the

bogus value as if it were 0, leading to confusion). Also, the item

must already exist for incr/decr to work; these commands won't pretend

that a non-existent key exists with value 0; instead, they will fail.

The client sends the command line:

incr <key> <value> [noreply]\r\n

or

decr <key> <value> [noreply]\r\n

- <key> is the key of the item the client wishes to change

- <value> is the amount by which the client wants to increase/decrease

the item. It is a decimal representation of a 64-bit unsigned integer.

- "noreply" optional parameter instructs the server to not send the

reply. See the note in Storage commands regarding malformed

requests.

The response will be one of:

- "NOT\_FOUND\r\n" to indicate the item with this value was not found

- <value>\r\n , where <value> is the new value of the item's data,

after the increment/decrement operation was carried out.

Note that underflow in the "decr" command is caught: if a client tries

to decrease the value below 0, the new value will be 0. Overflow in

the "incr" command will wrap around the 64 bit mark.

Note also that decrementing a number such that it loses length isn't

guaranteed to decrement its returned length. The number MAY be

space-padded at the end, but this is purely an implementation

optimization, so you also shouldn't rely on that.

Touch

-----

The "touch" command is used to update the expiration time of an existing item

without fetching it.

touch <key> <exptime> [noreply]\r\n

- <key> is the key of the item the client wishes the server to touch

- <exptime> is expiration time. Works the same as with the update commands

(set/add/etc). This replaces the existing expiration time. If an existing

item were to expire in 10 seconds, but then was touched with an

expiration time of "20", the item would then expire in 20 seconds.

- "noreply" optional parameter instructs the server to not send the

reply. See the note in Storage commands regarding malformed

requests.

The response line to this command can be one of:

- "TOUCHED\r\n" to indicate success

- "NOT\_FOUND\r\n" to indicate that the item with this key was not

found.

Slabs Reassign

--------------

NOTE: This command is subject to change as of this writing.

The slabs reassign command is used to redistribute memory once a running

instance has hit its limit. It might be desirable to have memory laid out

differently than was automatically assigned after the server started.

slabs reassign <source class> <dest class>\r\n

- <source class> is an id number for the slab class to steal a page from

A source class id of -1 means "pick from any valid class"

- <dest class> is an id number for the slab class to move a page to

The response line could be one of:

- "OK" to indicate the page has been scheduled to move

- "BUSY [message]" to indicate a page is already being processed, try again

later.

- "BADCLASS [message]" a bad class id was specified

- "NOSPARE [message]" source class has no spare pages

- "NOTFULL [message]" dest class must be full to move new pages to it

- "UNSAFE [message]" source class cannot move a page right now

- "SAME [message]" must specify different source/dest ids.

Slabs Automove

--------------

NOTE: This command is subject to change as of this writing.

The slabs automove command enables a background thread which decides on its

own when to move memory between slab classes. Its implementation and options

will likely be in flux for several versions. See the wiki/mailing list for

more details.

The automover can be enabled or disabled at runtime with this command.

slabs automove <0|1>

- 0|1|2 is the indicator on whether to enable the slabs automover or not.

The response should always be "OK\r\n"

- <0> means to set the thread on standby

- <1> means to return pages to a global pool when there are more than 2 pages

worth of free chunks in a slab class. Pages are then re-assigned back into

other classes as-needed.

- <2> is a highly aggressive mode which causes pages to be moved every time

there is an eviction. It is not recommended to run for very long in this

mode unless your access patterns are very well understood.

LRU Tuning

----------

Memcached supports multiple LRU algorithms, with a few tunables. Effort is

made to have sane defaults however you are able to tune while the daemon is

running.

The traditional model is "flat" mode, which is a single LRU chain per slab

class. The newer (with `-o modern` or `-o lru\_maintainer`) is segmented into

HOT, WARM, COLD. There is also a TEMP LRU. See doc/new\_lru.txt for details.

lru <tune|mode|temp\_ttl> <option list>

- "tune" takes numeric arguments "percent hot", "percent warm",

"max hot factor", "max warm age factor". IE: "lru tune 10 25 0.1 2.0".

This would cap HOT\_LRU at 10% of the cache, or tail is idle longer than

10% of COLD\_LRU. WARM\_LRU is up to 25% of cache, or tail is idle longer

than 2x COLD\_LRU.

- "mode" <flat|segmented>: "flat" is traditional mode. "segmented" uses

HOT|WARM|COLD split. "segmented" mode requires `-o lru\_maintainer` at start

time. If switching from segmented to flat mode, the background thread will

pull items from HOT|WARM into COLD queue.

- "temp\_ttl" <ttl>: If TTL is less than zero, disable usage of TEMP\_LRU. If

zero or above, items set with a TTL lower than this will go into TEMP\_LRU

and be unevictable until they naturally expire or are otherwise deleted or

replaced.

The response line could be one of:

- "OK" to indicate a successful update of the settings.

- "ERROR [message]" to indicate a failure or improper arguments.

LRU\_Crawler

-----------

NOTE: This command (and related commands) are subject to change as of this

writing.

The LRU Crawler is an optional background thread which will walk from the tail

toward the head of requested slab classes, actively freeing memory for expired

items. This is useful if you have a mix of items with both long and short

TTL's, but aren't accessed very often. This system is not required for normal

usage, and can add small amounts of latency and increase CPU usage.

lru\_crawler <enable|disable>

- Enable or disable the LRU Crawler background thread.

The response line could be one of:

- "OK" to indicate the crawler has been started or stopped.

- "ERROR [message]" something went wrong while enabling or disabling.

lru\_crawler sleep <microseconds>

- The number of microseconds to sleep in between each item checked for

expiration. Smaller numbers will obviously impact the system more.

A value of "0" disables the sleep, "1000000" (one second) is the max.

The response line could be one of:

- "OK"

- "CLIENT\_ERROR [message]" indicating a format or bounds issue.

lru\_crawler tocrawl <32u>

- The maximum number of items to inspect in a slab class per run request. This

allows you to avoid scanning all of very large slabs when it is unlikely to

find items to expire.

The response line could be one of:

- "OK"

- "CLIENT\_ERROR [message]" indicating a format or bound issue.

lru\_crawler crawl <classid,classid,classid|all>

- Takes a single, or a list of, numeric classids (ie: 1,3,10). This instructs

the crawler to start at the tail of each of these classids and run to the

head. The crawler cannot be stopped or restarted until it completes the

previous request.

The special keyword "all" instructs it to crawl all slabs with items in

them.

The response line could be one of:

- "OK" to indicate successful launch.

- "BUSY [message]" to indicate the crawler is already processing a request.

- "BADCLASS [message]" to indicate an invalid class was specified.

lru\_crawler metadump <classid,classid,classid|all>

- Similar in function to the above "lru\_crawler crawl" command, this function

outputs one line for every valid item found in the matching slab classes.

Similar to "cachedump", but does not lock the cache and can return all

items, not just 1MB worth.

Lines are in "key=value key2=value2" format, with value being URI encoded

(ie: %20 for a space).

The exact keys available are subject to change, but will include at least:

"key", "exp" (expiration time), "la", (last access time), "cas",

"fetch" (if item has been fetched before).

The response line could be one of:

- "OK" to indicate successful launch.

- "BUSY [message]" to indicate the crawler is already processing a request.

- "BADCLASS [message]" to indicate an invalid class was specified.

Watchers

--------

Watchers are a way to connect to memcached and inspect what's going on

internally. This is an evolving feature so new endpoints should show up over

time.

watch <fetchers|mutations|evictions>

- Turn connection into a watcher. Options can be stacked and are

space-separated. Logs will be sent to the watcher until it disconnects.

The response line could be one of:

- "OK" to indicate the watcher is ready to send logs.

- "ERROR [message]" something went wrong while enabling.

The response format is in "key=value key2=value2" format, for easy parsing.

Lines are prepending with "ts=" for a timestamp and "gid=" for a global ID

number of the log line. Given how logs are collected internally they may be

printed out of order. If this is important the GID may be used to put log

lines back in order.

The value of keys (and potentially other things) are "URI encoded". Since most

keys used conform to standard ASCII, this should have no effect. For keys with

less standard or binary characters, "%NN"'s are inserted to represent the

byte, ie: "n%2Cfoo" for "n,foo".

The arguments are:

- "fetchers": Currently emits logs every time an item is fetched internally.

This means a "set" command would also emit an item\_get log, as it checks for

an item before replacing it. Multigets should also emit multiple lines.

- "mutations": Currently emits logs when an item is stored in most cases.

Shows errors for most cases when items cannot be stored.

- "evictions": Shows some information about items as they are evicted from the

cache. Useful in seeing if items being evicted were actually used, and which

keys are getting removed.

Statistics

----------

The command "stats" is used to query the server about statistics it

maintains and other internal data. It has two forms. Without

arguments:

stats\r\n

it causes the server to output general-purpose statistics and

settings, documented below. In the other form it has some arguments:

stats <args>\r\n

Depending on <args>, various internal data is sent by the server. The

kinds of arguments and the data sent are not documented in this version

of the protocol, and are subject to change for the convenience of

memcache developers.

General-purpose statistics

--------------------------

Upon receiving the "stats" command without arguments, the server sents

a number of lines which look like this:

STAT <name> <value>\r\n

The server terminates this list with the line

END\r\n

In each line of statistics, <name> is the name of this statistic, and

<value> is the data. The following is the list of all names sent in

response to the "stats" command, together with the type of the value

sent for this name, and the meaning of the value.

In the type column below, "32u" means a 32-bit unsigned integer, "64u"

means a 64-bit unsigned integer. '32u.32u' means two 32-bit unsigned

integers separated by a colon (treat this as a floating point number).

|-----------------------+---------+-------------------------------------------|

| Name | Type | Meaning |

|-----------------------+---------+-------------------------------------------|

| pid | 32u | Process id of this server process |

| uptime | 32u | Number of secs since the server started |

| time | 32u | current UNIX time according to the server |

| version | string | Version string of this server |

| pointer\_size | 32 | Default size of pointers on the host OS |

| | | (generally 32 or 64) |

| rusage\_user | 32u.32u | Accumulated user time for this process |

| | | (seconds:microseconds) |

| rusage\_system | 32u.32u | Accumulated system time for this process |

| | | (seconds:microseconds) |

| curr\_items | 64u | Current number of items stored |

| total\_items | 64u | Total number of items stored since |

| | | the server started |

| bytes | 64u | Current number of bytes used |

| | | to store items |

| max\_connections | 32u | Max number of simultaneous connections |

| curr\_connections | 32u | Number of open connections |

| total\_connections | 32u | Total number of connections opened since |

| | | the server started running |

| rejected\_connections | 64u | Conns rejected in maxconns\_fast mode |

| connection\_structures | 32u | Number of connection structures allocated |

| | | by the server |

| reserved\_fds | 32u | Number of misc fds used internally |

| cmd\_get | 64u | Cumulative number of retrieval reqs |

| cmd\_set | 64u | Cumulative number of storage reqs |

| cmd\_flush | 64u | Cumulative number of flush reqs |

| cmd\_touch | 64u | Cumulative number of touch reqs |

| get\_hits | 64u | Number of keys that have been requested |

| | | and found present |

| get\_misses | 64u | Number of items that have been requested |

| | | and not found |

| get\_expired | 64u | Number of items that have been requested |

| | | but had already expired. |

| get\_flushed | 64u | Number of items that have been requested |

| | | but have been flushed via flush\_all |

| delete\_misses | 64u | Number of deletions reqs for missing keys |

| delete\_hits | 64u | Number of deletion reqs resulting in |

| | | an item being removed. |

| incr\_misses | 64u | Number of incr reqs against missing keys. |

| incr\_hits | 64u | Number of successful incr reqs. |

| decr\_misses | 64u | Number of decr reqs against missing keys. |

| decr\_hits | 64u | Number of successful decr reqs. |

| cas\_misses | 64u | Number of CAS reqs against missing keys. |

| cas\_hits | 64u | Number of successful CAS reqs. |

| cas\_badval | 64u | Number of CAS reqs for which a key was |

| | | found, but the CAS value did not match. |

| touch\_hits | 64u | Number of keys that have been touched |

| | | with a new expiration time |

| touch\_misses | 64u | Number of items that have been touched |

| | | and not found |

| auth\_cmds | 64u | Number of authentication commands |

| | | handled, success or failure. |

| auth\_errors | 64u | Number of failed authentications. |

| idle\_kicks | 64u | Number of connections closed due to |

| | | reaching their idle timeout. |

| evictions | 64u | Number of valid items removed from cache |

| | | to free memory for new items |

| reclaimed | 64u | Number of times an entry was stored using |

| | | memory from an expired entry |

| bytes\_read | 64u | Total number of bytes read by this server |

| | | from network |

| bytes\_written | 64u | Total number of bytes sent by this server |

| | | to network |

| limit\_maxbytes | size\_t | Number of bytes this server is allowed to |

| | | use for storage. |

| accepting\_conns | bool | Whether or not server is accepting conns |

| listen\_disabled\_num | 64u | Number of times server has stopped |

| | | accepting new connections (maxconns). |

| time\_in\_listen\_disabled\_us |

| | 64u | Number of microseconds in maxconns. |

| threads | 32u | Number of worker threads requested. |

| | | (see doc/threads.txt) |

| conn\_yields | 64u | Number of times any connection yielded to |

| | | another due to hitting the -R limit. |

| hash\_power\_level | 32u | Current size multiplier for hash table |

| hash\_bytes | 64u | Bytes currently used by hash tables |

| hash\_is\_expanding | bool | Indicates if the hash table is being |

| | | grown to a new size |

| expired\_unfetched | 64u | Items pulled from LRU that were never |

| | | touched by get/incr/append/etc before |

| | | expiring |

| evicted\_unfetched | 64u | Items evicted from LRU that were never |

| | | touched by get/incr/append/etc. |

| evicted\_active | 64u | Items evicted from LRU that had been hit |

| | | recently but did not jump to top of LRU |

| slab\_reassign\_running | bool | If a slab page is being moved |

| slabs\_moved | 64u | Total slab pages moved |

| crawler\_reclaimed | 64u | Total items freed by LRU Crawler |

| crawler\_items\_checked | 64u | Total items examined by LRU Crawler |

| lrutail\_reflocked | 64u | Times LRU tail was found with active ref. |

| | | Items can be evicted to avoid OOM errors. |

| moves\_to\_cold | 64u | Items moved from HOT/WARM to COLD LRU's |

| moves\_to\_warm | 64u | Items moved from COLD to WARM LRU |

| moves\_within\_lru | 64u | Items reshuffled within HOT or WARM LRU's |

| direct\_reclaims | 64u | Times worker threads had to directly |

| | | reclaim or evict items. |

| lru\_crawler\_starts | 64u | Times an LRU crawler was started |

| lru\_maintainer\_juggles |

| | 64u | Number of times the LRU bg thread woke up |

| slab\_global\_page\_pool | 32u | Slab pages returned to global pool for |

| | | reassignment to other slab classes. |

| slab\_reassign\_rescues | 64u | Items rescued from eviction in page move |

| slab\_reassign\_evictions\_nomem |

| | 64u | Valid items evicted during a page move |

| | | (due to no free memory in slab) |

| slab\_reassign\_chunk\_rescues |

| | 64u | Individual sections of an item rescued |

| | | during a page move. |

| slab\_reassign\_inline\_reclaim |

| | 64u | Internal stat counter for when the page |

| | | mover clears memory from the chunk |

| | | freelist when it wasn't expecting to. |

| slab\_reassign\_busy\_items |

| | 64u | Items busy during page move, requiring a |

| | | retry before page can be moved. |

| slab\_reassign\_busy\_deletes |

| | 64u | Items busy during page move, requiring |

| | | deletion before page can be moved. |

| log\_worker\_dropped | 64u | Logs a worker never wrote due to full buf |

| log\_worker\_written | 64u | Logs written by a worker, to be picked up |

| log\_watcher\_skipped | 64u | Logs not sent to slow watchers. |

| log\_watcher\_sent | 64u | Logs written to watchers. |

|-----------------------+---------+-------------------------------------------|

Settings statistics

-------------------

CAVEAT: This section describes statistics which are subject to change in the

future.

The "stats" command with the argument of "settings" returns details of

the settings of the running memcached. This is primarily made up of

the results of processing commandline options.

Note that these are not guaranteed to return in any specific order and

this list may not be exhaustive. Otherwise, this returns like any

other stats command.

|-------------------+----------+----------------------------------------------|

| Name | Type | Meaning |

|-------------------+----------+----------------------------------------------|

| maxbytes | size\_t | Maximum number of bytes allowed in the cache |

| maxconns | 32 | Maximum number of clients allowed. |

| tcpport | 32 | TCP listen port. |

| udpport | 32 | UDP listen port. |

| inter | string | Listen interface. |

| verbosity | 32 | 0 = none, 1 = some, 2 = lots |

| oldest | 32u | Age of the oldest honored object. |

| evictions | on/off | When off, LRU evictions are disabled. |

| domain\_socket | string | Path to the domain socket (if any). |

| umask | 32 (oct) | umask for the creation of the domain socket. |

| growth\_factor | float | Chunk size growth factor. |

| chunk\_size | 32 | Minimum space allocated for key+value+flags. |

| num\_threads | 32 | Number of threads (including dispatch). |

| stat\_key\_prefix | char | Stats prefix separator character. |

| detail\_enabled | bool | If yes, stats detail is enabled. |

| reqs\_per\_event | 32 | Max num IO ops processed within an event. |

| cas\_enabled | bool | When no, CAS is not enabled for this server. |

| tcp\_backlog | 32 | TCP listen backlog. |

| auth\_enabled\_sasl | yes/no | SASL auth requested and enabled. |

| item\_size\_max | size\_t | maximum item size |

| maxconns\_fast | bool | If fast disconnects are enabled |

| hashpower\_init | 32 | Starting size multiplier for hash table |

| slab\_reassign | bool | Whether slab page reassignment is allowed |

| slab\_automove | bool | Whether slab page automover is enabled |

| slab\_automove\_ratio |

| | float | Ratio limit between young/old slab classes |

| slab\_automove\_window |

| | 32u | Internal algo tunable for automove |

| slab\_chunk\_max | 32 | Max slab class size (avoid unless necessary) |

| hash\_algorithm | char | Hash table algorithm in use |

| lru\_crawler | bool | Whether the LRU crawler is enabled |

| lru\_crawler\_sleep | 32 | Microseconds to sleep between LRU crawls |

| lru\_crawler\_tocrawl |

| | 32u | Max items to crawl per slab per run |

| lru\_maintainer\_thread |

| | bool | Split LRU mode and background threads |

| hot\_lru\_pct | 32 | Pct of slab memory reserved for HOT LRU |

| warm\_lru\_pct | 32 | Pct of slab memory reserved for WARM LRU |

| hot\_max\_factor | float | Set idle age of HOT LRU to COLD age \* this |

| warm\_max\_factor | float | Set idle age of WARM LRU to COLD age \* this |

| temp\_lru | bool | If yes, items < temporary\_ttl use TEMP\_LRU |

| temporary\_ttl | 32u | Items with TTL < this are marked temporary |

| idle\_time | 0 | Drop connections that are idle this many |

| | | seconds (0 disables) |

| watcher\_logbuf\_size |

| | 32u | Size of internal (not socket) write buffer |

| | | per active watcher connected. |

| worker\_logbuf\_size| 32u | Size of internal per-worker-thread buffer |

| | | which the background thread reads from. |

| track\_sizes | bool | If yes, a "stats sizes" histogram is being |

| | | dynamically tracked. |

| inline\_ascii\_response |

| | bool | If yes, stores numbers from VALUE response |

| | | inside an item, using up to 24 bytes. |

| | | Small slowdown for ASCII get, faster sets. |

|-------------------+----------+----------------------------------------------|

Item statistics

---------------

CAVEAT: This section describes statistics which are subject to change in the

future.

The "stats" command with the argument of "items" returns information about

item storage per slab class. The data is returned in the format:

STAT items:<slabclass>:<stat> <value>\r\n

The server terminates this list with the line

END\r\n

The slabclass aligns with class ids used by the "stats slabs" command. Where

"stats slabs" describes size and memory usage, "stats items" shows higher

level information.

The following item values are defined as of writing.

Name Meaning

------------------------------

number Number of items presently stored in this class. Expired

items are not automatically excluded.

number\_hot Number of items presently stored in the HOT LRU.

number\_warm Number of items presently stored in the WARM LRU.

number\_cold Number of items presently stored in the COLD LRU.

number\_temp Number of items presently stored in the TEMPORARY LRU.

age\_hot Age of the oldest item in HOT LRU.

age\_warm Age of the oldest item in WARM LRU.

age Age of the oldest item in the LRU.

evicted Number of times an item had to be evicted from the LRU

before it expired.

evicted\_nonzero Number of times an item which had an explicit expire

time set had to be evicted from the LRU before it

expired.

evicted\_time Seconds since the last access for the most recent item

evicted from this class. Use this to judge how

recently active your evicted data is.

outofmemory Number of times the underlying slab class was unable to

store a new item. This means you are running with -M or

an eviction failed.

tailrepairs Number of times we self-healed a slab with a refcount

leak. If this counter is increasing a lot, please

report your situation to the developers.

reclaimed Number of times an entry was stored using memory from

an expired entry.

expired\_unfetched Number of expired items reclaimed from the LRU which

were never touched after being set.

evicted\_unfetched Number of valid items evicted from the LRU which were

never touched after being set.

evicted\_active Number of valid items evicted from the LRU which were

recently touched but were evicted before being moved to

the top of the LRU again.

crawler\_reclaimed Number of items freed by the LRU Crawler.

lrutail\_reflocked Number of items found to be refcount locked in the

LRU tail.

moves\_to\_cold Number of items moved from HOT or WARM into COLD.

moves\_to\_warm Number of items moved from COLD to WARM.

moves\_within\_lru Number of times active items were bumped within

HOT or WARM.

direct\_reclaims Number of times worker threads had to directly pull LRU

tails to find memory for a new item.

hits\_to\_hot

hits\_to\_warm

hits\_to\_cold

hits\_to\_temp Number of get\_hits to each sub-LRU.

Note this will only display information about slabs which exist, so an empty

cache will return an empty set.

Item size statistics

--------------------

CAVEAT: This section describes statistics which are subject to change in the

future.

The "stats" command with the argument of "sizes" returns information about the

general size and count of all items stored in the cache.

WARNING: In versions prior to 1.4.27 this command causes the cache server to

lock while it iterates the items. 1.4.27 and greater are safe.

The data is returned in the following format:

STAT <size> <count>\r\n

The server terminates this list with the line

END\r\n

'size' is an approximate size of the item, within 32 bytes.

'count' is the amount of items that exist within that 32-byte range.

This is essentially a display of all of your items if there was a slab class

for every 32 bytes. You can use this to determine if adjusting the slab growth

factor would save memory overhead. For example: generating more classes in the

lower range could allow items to fit more snugly into their slab classes, if

most of your items are less than 200 bytes in size.

In 1.4.27 and after, this feature must be manually enabled.

A "stats" command with the argument of "sizes\_enable" will enable the

histogram at runtime. This has a small overhead to every store or delete

operation. If you don't want to incur this, leave it off.

A "stats" command with the argument of "sizes\_disable" will disable the

histogram.

It can also be enabled at starttime with "-o track\_sizes"

If disabled, "stats sizes" will return:

STAT sizes\_status disabled\r\n

"stats sizes\_enable" will return:

STAT sizes\_status enabled\r\n

"stats sizes\_disable" will return:

STAT sizes\_status disabled\r\n

If an error happens, it will return:

STAT sizes\_status error\r\n

STAT sizes\_error [error\_message]\r\n

CAVEAT: If CAS support is disabled, you cannot enable/disable this feature at

runtime.

Slab statistics

---------------

CAVEAT: This section describes statistics which are subject to change in the

future.

The "stats" command with the argument of "slabs" returns information about

each of the slabs created by memcached during runtime. This includes per-slab

information along with some totals. The data is returned in the format:

STAT <slabclass>:<stat> <value>\r\n

STAT <stat> <value>\r\n

The server terminates this list with the line

END\r\n

|-----------------+----------------------------------------------------------|

| Name | Meaning |

|-----------------+----------------------------------------------------------|

| chunk\_size | The amount of space each chunk uses. One item will use |

| | one chunk of the appropriate size. |

| chunks\_per\_page | How many chunks exist within one page. A page by |

| | default is less than or equal to one megabyte in size. |

| | Slabs are allocated by page, then broken into chunks. |

| total\_pages | Total number of pages allocated to the slab class. |

| total\_chunks | Total number of chunks allocated to the slab class. |

| get\_hits | Total number of get requests serviced by this class. |

| cmd\_set | Total number of set requests storing data in this class. |

| delete\_hits | Total number of successful deletes from this class. |

| incr\_hits | Total number of incrs modifying this class. |

| decr\_hits | Total number of decrs modifying this class. |

| cas\_hits | Total number of CAS commands modifying this class. |

| cas\_badval | Total number of CAS commands that failed to modify a |

| | value due to a bad CAS id. |

| touch\_hits | Total number of touches serviced by this class. |

| used\_chunks | How many chunks have been allocated to items. |

| free\_chunks | Chunks not yet allocated to items, or freed via delete. |

| free\_chunks\_end | Number of free chunks at the end of the last allocated |

| | page. |

| mem\_requested | Number of bytes requested to be stored in this slab[\*]. |

| active\_slabs | Total number of slab classes allocated. |

| total\_malloced | Total amount of memory allocated to slab pages. |

|-----------------+----------------------------------------------------------|

\* Items are stored in a slab that is the same size or larger than the

item. mem\_requested shows the size of all items within a

slab. (total\_chunks \* chunk\_size) - mem\_requested shows memory

wasted in a slab class. If you see a lot of waste, consider tuning

the slab factor.

Connection statistics

---------------------

The "stats" command with the argument of "conns" returns information

about currently active connections and about sockets that are listening

for new connections. The data is returned in the format:

STAT <file descriptor>:<stat> <value>\r\n

The server terminates this list with the line

END\r\n

The following "stat" keywords may be present:

|---------------------+------------------------------------------------------|

| Name | Meaning |

|---------------------+------------------------------------------------------|

| addr | The address of the remote side. For listening |

| | sockets this is the listen address. Note that some |

| | socket types (such as UNIX-domain) don't have |

| | meaningful remote addresses. |

| state | The current state of the connection. See below. |

| secs\_since\_last\_cmd | The number of seconds since the most recently |

| | issued command on the connection. This measures |

| | the time since the start of the command, so if |

| | "state" indicates a command is currently executing, |

| | this will be the number of seconds the current |

| | command has been running. |

|---------------------+------------------------------------------------------|

The value of the "state" stat may be one of the following:

|----------------+-----------------------------------------------------------|

| Name | Meaning |

|----------------+-----------------------------------------------------------|

| conn\_closing | Shutting down the connection. |

| conn\_listening | Listening for new connections or a new UDP request. |

| conn\_mwrite | Writing a complex response, e.g., to a "get" command. |

| conn\_new\_cmd | Connection is being prepared to accept a new command. |

| conn\_nread | Reading extended data, typically for a command such as |

| | "set" or "put". |

| conn\_parse\_cmd | The server has received a command and is in the middle |

| | of parsing it or executing it. |

| conn\_read | Reading newly-arrived command data. |

| conn\_swallow | Discarding excess input, e.g., after an error has |

| | occurred. |

| conn\_waiting | A partial command has been received and the server is |

| | waiting for the rest of it to arrive (note the difference |

| | between this and conn\_nread). |

| conn\_write | Writing a simple response (anything that doesn't involve |

| | sending back multiple lines of response data). |

|----------------+-----------------------------------------------------------|

Other commands

--------------

"flush\_all" is a command with an optional numeric argument. It always

succeeds, and the server sends "OK\r\n" in response (unless "noreply"

is given as the last parameter). Its effect is to invalidate all

existing items immediately (by default) or after the expiration

specified. After invalidation none of the items will be returned in

response to a retrieval command (unless it's stored again under the

same key \*after\* flush\_all has invalidated the items). flush\_all

doesn't actually free all the memory taken up by existing items; that

will happen gradually as new items are stored. The most precise

definition of what flush\_all does is the following: it causes all

items whose update time is earlier than the time at which flush\_all

was set to be executed to be ignored for retrieval purposes.

The intent of flush\_all with a delay, was that in a setting where you

have a pool of memcached servers, and you need to flush all content,

you have the option of not resetting all memcached servers at the

same time (which could e.g. cause a spike in database load with all

clients suddenly needing to recreate content that would otherwise

have been found in the memcached daemon).

The delay option allows you to have them reset in e.g. 10 second

intervals (by passing 0 to the first, 10 to the second, 20 to the

third, etc. etc.).

"cache\_memlimit" is a command with a numeric argument. This allows runtime

adjustments of the cache memory limit. It returns "OK\r\n" or an error (unless

"noreply" is given as the last parameter). If the new memory limit is higher

than the old one, the server may start requesting more memory from the OS. If

the limit is lower, and slabs\_reassign+automove are enabled, free memory may

be released back to the OS asynchronously.

The argument is in megabytes, not bytes. Input gets multiplied out into

megabytes internally.

"version" is a command with no arguments:

version\r\n

In response, the server sends

"VERSION <version>\r\n", where <version> is the version string for the

server.

"verbosity" is a command with a numeric argument. It always succeeds,

and the server sends "OK\r\n" in response (unless "noreply" is given

as the last parameter). Its effect is to set the verbosity level of

the logging output.

"quit" is a command with no arguments:

quit\r\n

Upon receiving this command, the server closes the

connection. However, the client may also simply close the connection

when it no longer needs it, without issuing this command.

Security restrictions

---------------------

In the debug build the following commands are available for testing the

security restrictions:

"misbehave" is a command with no arguments:

misbehave\r\n

This command causes the worker thread to attempt a) opening a new socket, and

b) executing a shell command. If either one is successful, an error is

returned. Otherwise memcached returns OK.

The check is available only in Linux builds with seccomp enabled.

UDP protocol

------------

For very large installations where the number of clients is high enough

that the number of TCP connections causes scaling difficulties, there is

also a UDP-based interface. The UDP interface does not provide guaranteed

delivery, so should only be used for operations that aren't required to

succeed; typically it is used for "get" requests where a missing or

incomplete response can simply be treated as a cache miss.

Each UDP datagram contains a simple frame header, followed by data in the

same format as the TCP protocol described above. In the current

implementation, requests must be contained in a single UDP datagram, but

responses may span several datagrams. (The only common requests that would

span multiple datagrams are huge multi-key "get" requests and "set"

requests, both of which are more suitable to TCP transport for reliability

reasons anyway.)

The frame header is 8 bytes long, as follows (all values are 16-bit integers

in network byte order, high byte first):

0-1 Request ID

2-3 Sequence number

4-5 Total number of datagrams in this message

6-7 Reserved for future use; must be 0

The request ID is supplied by the client. Typically it will be a

monotonically increasing value starting from a random seed, but the client

is free to use whatever request IDs it likes. The server's response will

contain the same ID as the incoming request. The client uses the request ID

to differentiate between responses to outstanding requests if there are

several pending from the same server; any datagrams with an unknown request

ID are probably delayed responses to an earlier request and should be

discarded.

The sequence number ranges from 0 to n-1, where n is the total number of

datagrams in the message. The client should concatenate the payloads of the

datagrams for a given response in sequence number order; the resulting byte

stream will contain a complete response in the same format as the TCP

protocol (including terminating \r\n sequences).