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
faulure.

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 unqiue> [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).

- <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 blockm 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 either that the
condition for an "add" or a "replace" command wasn't met, or that the
item is in a delete queue (see the "delete" command below).

- "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 or has been deleted.

Retrieval command:

The retrieval commands "get" and "gets" operates 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> [<time>] [noreply]\r\n

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

- <time> is the amount of time in seconds (or Unix time until which)
  the client wishes the server to refuse "add" and "replace" commands
  with this key. For this amount of item, the item is put into a
  delete queue, which means that it won't possible to retrieve it by
  the "get" command, but "add" and "replace" command with this key
  will also fail (the "set" command will succeed, however). After the

time passes, the item is finally deleted from server memory.

The parameter <time> is optional, and, if absent, defaults to 0 (which means that the item will be deleted immediately and further storage commands with this key will succeed).

- "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 commands behave as if the value were 0. 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.

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 vesion
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 unsigner integer. '32u:32u' means two 32-but unsigned
integers separated by a colon.

Name	Type	Meaning
pid	32u	Process id of this server process
uptime	32u	Number of seconds this server has been running
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	32u	Current number of items stored by the server
total_items	32u	Total number of items stored by this server ever since it started
bytes	64u	Current number of bytes used by this server to store items
curr_connections	32u	Number of open connections
total_connections	32u	Total number of connections opened since the server started running
connection_structures	32u	Number of connection structures allocated by the server
cmd_flush	64u	Cumulative number of flush requests
cmd_get	64u	Cumulative number of retrieval requests
cmd_set	64u	Cumulative number of storage requests

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
evictions	64u	Number of valid items removed from cache to free memory for new items
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	32u	Number of bytes this server is allowed to use for storage.
threads	32u	Number of worker threads requested. (see doc/threads.txt)
accepting_conns	32u	Whether or not new connections are being accepted.
listen_disabled_num	64u	Number of times we stopped listening for new connections.

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.

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_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.

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: This command WILL lock up your cache! It iterates over *every item* and examines the size. While the operation is fast, if you have many items you could prevent memcached from serving requests for several seconds.

The data is returned in the following format:

<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.

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 one megabyte in size. Slabs are allocated per 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.
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.
active_slabs	Total number of slab classes allocated.
total_malloced	Total amount of memory allocated to slab pages.

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.).

"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.

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).