

Contents

- [Telnet Interface](#)
 - [How To Connect](#)
 - [Supported Commands](#)
 - [Traffic Statistics](#)
 - [Memory Statistics](#)
 - [Which Keys Are Used?](#)
- [Troubleshooting](#)
 - [1MB Data Limit](#)
 - [Never Set a Timeout > 30 Days!](#)
 - [Disappearing Keys on Overflow](#)
- [Replication](#)
- [Monitoring](#)
 - [memcached-top](#)
 - [statsproxy](#)
 - [memcache.php](#)
- [Memcached Bindings](#)
 - [Dumping Memcache Keys](#)
 - [Dumping Tools](#)
 - [Using Consistent Hashing](#)
 - [nginx](#)
 - [PHP](#)
 - [Perl](#)
- [Memcache Alternatives](#)
- [Suggested Reading](#)

Memcached Cheat Sheet

Telnet Interface

This is a short summary of everything important that helps to inspect a running [memcached](#) instance. You need to know that memcached requires you to connect to it via telnet. The following post describes the usage of this interface.

How To Connect

Use "ps -ef" to find out which IP and port was passed when memcached was started and use the same with telnet to connect to memcache. Example:

```
telnet 10.0.0.2 11211
```

Supported Commands

The supported commands (the official ones and some unofficial) are documented in the [doc/protocol.txt](#) document.

Sadly the syntax description isn't really clear and a simple help command listing the existing commands would be much better. Here is an overview of the commands you can find in the [source](#) (as of 16.12.2008):

Command	Description	Example
get	Reads a value	get mykey
set	Set a key unconditionally	set mykey 0 60 5
add	Add a new key	add newkey 0

		60 5
replace	Overwrite existing key	replace key 0 60 5
append	Append data to existing key	append key 0 60 15
prepend	Prepend data to existing key	prepend key 0 60 15
incr	Increments numerical key value by given number	incr mykey 2
decr	Decrements numerical key value by given number	decr mykey 5
delete	Deletes an existing key	delete mykey
flush_all	Invalidate specific items immediately	flush_all
	Invalidate all items in n seconds	flush_all 900
stats	Prints general statistics	stats
	Prints memory statistics	stats slabs
	Prints memory statistics	stats malloc
	Print higher level allocation statistics	stats items
		stats detail
		stats sizes
	Resets statistics	stats reset
version	Prints server version.	version
verbosity	Increases log level	verbosity
quit	Terminate telnet session	quit

Traffic Statistics

You can query the current traffic statistics using the command

```
stats
```

You will get a listing which serves the number of connections, bytes in/out and much more.

Example Output:

```
STAT pid 14868
STAT uptime 175931
STAT time 1220540125
STAT version 1.2.2
STAT pointer_size 32
STAT rusage_user 620.299700
STAT rusage_system 1545.703017
STAT curr_items 228
STAT total_items 779
STAT bytes 15525
STAT curr_connections 92
STAT total_connections 1740
STAT connection_structures 165
STAT cmd_get 7411
STAT cmd_set 28445156
STAT get_hits 5183
STAT get_misses 2228
STAT evictions 0
STAT bytes_read 2112768087
STAT bytes_written 1000038245
STAT limit_maxbytes 52428800
STAT threads 1
END
```

Memory Statistics

You can query the current memory statistics using

```
stats slabs
```

Example Output:

```
STAT 1:chunk_size 80
STAT 1:chunks_per_page 13107
STAT 1:total_pages 1
STAT 1:total_chunks 13107
STAT 1:used_chunks 13106
STAT 1:free_chunks 1
STAT 1:free_chunks_end 12886
STAT 2:chunk_size 100
STAT 2:chunks_per_page 10485
STAT 2:total_pages 1
STAT 2:total_chunks 10485
STAT 2:used_chunks 10484
STAT 2:free_chunks 1
STAT 2:free_chunks_end 10477
[...]
STAT active_slabs 3
STAT total_malloced 3145436
END
```

If you are unsure if you have enough memory for your memcached instance always look out for the "evictions" counters given by the "stats" command. If you have enough memory for the instance the "evictions" counter should be 0 or at least not increasing.

Which Keys Are Used?

There is no builtin function to directly determine the current set of keys. However you can use the

```
stats items
```

command to determine how many keys do exist.

```
stats items
STAT items:1:number 220
STAT items:1:age 83095
STAT items:2:number 7
STAT items:2:age 1405
[...]
END
```

This at least helps to see if any keys are used. To dump the key names from a PHP script that already does the memcache access you can use the PHP code from 100days.de [↗](#).

Troubleshooting

1MB Data Limit

Note that prio to memcached 1.4 you cannot store objects larger than 1MB due to the default maximum slab size.

Never Set a Timeout > 30 Days!

If you try to "set" or "add" a key with a timeout bigger than the allowed maximum you might not get what you expect because memcached then treats the value as a Unix timestamp. Also if the timestamp is in the past it will do nothing at all. Your command will silently fail.

So if you want to use the maximum lifetime specify 2592000.

Example:

```
set my_key 0 2592000 1
1
```





Disappearing Keys on Overflow

Despite the documentation saying something about wrapping around


64bit overflowing a value using "incr" causes the value to disappear. It needs to be created using "add"/"set" again.

Replication

memcached itself does not support replication. If you really need it you need to use 3rd party solutions:

- [repcached](#) : Multi-master async replication (memcached 1.2 patch set)
- [Couchbase memcached interface](#) : Use CouchBase as memcached drop-in
- [yrmcuds](#) : memcached compatible Master-Slave key value store
- [twemproxy](#)  (aka nutcracker): proxy with memcached support


Monitoring

When using [memcached](#)  or memcachedb everything is fine as long as it is running. But from an operating perspective memcached is a black box. There is no real logging you can only use the -v/-vv/-vvv switches when not running in daemon mode to see what your instance does. And it becomes even more complex if you run multiple or distributed memcache instances available on different hosts and ports.


So the question is: **How to monitor your distributed memcache setup?**

There are not many tools out there, but some useful are. We'll have a look at the following tools. Note that some can monitor multiple memcached instances, while others can only monitor a single instance at a time.

Name	Multi-Instances	Complexity/Features
telnet	no	Simple CLI via telnet
memcached-top	no	CLI

stats-proxy	yes	Simple Web GUI
memcache.php	yes	Simple Web GUI
PhpMemcacheAdmin 	yes	Complex Web GUI
Memcache Manager	yes	Complex Web GUI

memcached-top

You can use [memcache-top](#)  for live-monitoring a single memcached instance. It will give you the I/O throughput, the number of evictions, the current hit ratio and if run with "--commands" it will also provide the number of GET/SET operations per interval.

```

memcache-top v0.6          (default port: 11211, color: on, re

INSTANCE          USAGE    HIT %   CONN    TIME    EVI
10.50.11.5:11211   88.9%   69.7%   1661    0.9ms   0.3
10.50.11.5:11212   88.8%   69.9%   2121    0.7ms   1.3
10.50.11.5:11213   88.9%   69.4%   1527    0.7ms   1.7
[...]



AVERAGE:          84.7%    72.9%   1704    1.0ms   1.3

TOTAL:             19.9GB/ 23.4GB          20.0K   11.7ms  15.
(ctrl-c to quit.)

```

(Example output)

statsproxy

Using the [statsproxy](#)  tool you get a browser-based statistics tool for multiple memcached instances. The basic idea of statsproxy is to provide the unmodified memcached statistics via HTTP. It also provide a synthetic health check for service monitoring tools like [Nagios](#) . To

compile statsproxy on Debian:

```
# Ensure you have bison
sudo apt-get install bison

# Download tarball
tar zxvf statsproxy-1.0.tgz
cd statsproxy-1.0
make
```

Now you can run the "statsproxy" binary, but it will inform you that it needs a configuration file. I suggest to redirect the output to a new file e.g. "statsproxy.conf" and remove the information text on top and bottom and then to modify the configuration section as needed.

```
./statsproxy > statsproxy.conf 2>&1
```

Ensure to add as many "proxy-mapping" sections as you have memcached instances. In each "proxy-mapping" section ensure that "backend" points to your memcached instance and "frontend" to a port on your webserver where you want to access the information for this backend. Once finished run:

```
./statsproxy -F statsproxy.conf
```


Below you find a screenshot of what stats-proxy looks like:

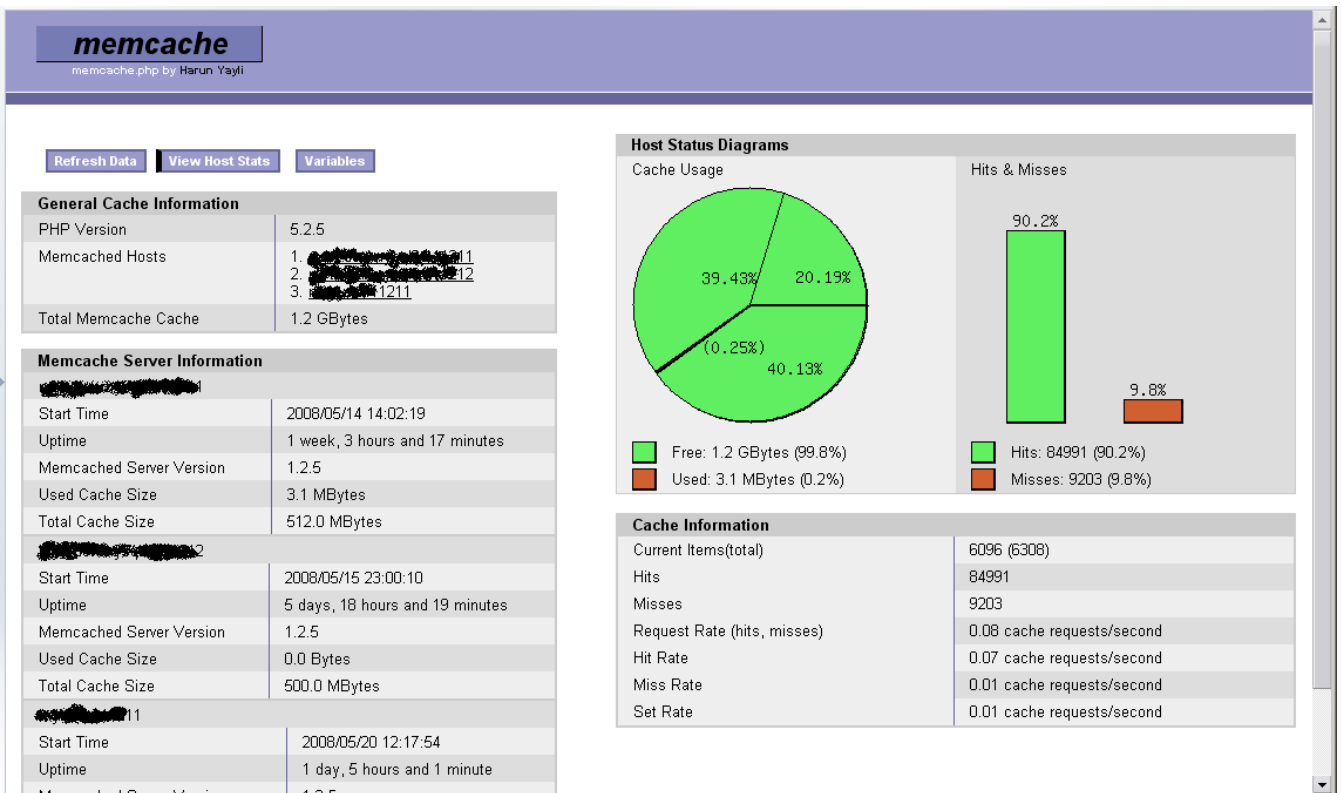
Raw stats: [basic](#) [health](#) [memory](#) [items](#) [storage](#) [slabs](#) [sizes](#)
[replication](#)

*[polling interval: 10000ms, webpage refresh interval: 10000ms,
connect/read/write timeout: 5000/5000/5000ms]*

STAT *pid* **3296**
STAT *uptime* **1341**
STAT *time* **1344324365**
STAT *version* **1.4.13**
STAT *libevent* **2.0.16-stable**
STAT *pointer_size* **64**
STAT *rusage_user* **0.044002**
STAT *rusage_system* **0.088005**
STAT *curr_connections* **5**
STAT *total_connections* **15**
STAT *connection_structures* **7**
STAT *reserved_fds* **20**
STAT *cmd_get* **0**
STAT *cmd_set* **3**
STAT *cmd_flush* **0**

memcache.php

Using [this PHP script](#)  you can quickly add memcached statistics to a webserver of your choice. Most useful is the global memory usage graph which helps to identify problematic instances in a distributed environment. Here is how it should look (screenshot from the project homepage):



When using this script ensure access is protected and not to trigger the "flush_all" menu option by default. Also on large memcached instances refrain from dumping the keys as it might cause some load on your server.


Memcached Bindings

- Tomcat: [memcached-session-manager](#)
- libketama: <https://github.com/RJ/ketama> (supports C, PHP, Java, Python)
- libmemcached: <http://libmemcached.org> (used by PHP and others)
- python-memcached <https://pypi.python.org/pypi/python-memcached> (implemented in Python)
- pylibmc <https://pypi.python.org/pypi/pylibmc/1.4.1> (implemented in C)

Dumping Memcache Keys

You spent already 50GB on the memcache cluster, but you still see many evictions and the cache hit ratio doesn't look good since a few

days. The developers swear that they didn't change the caching recently, they checked the code twice and have found no problem. What now? How to get some insight into the black box of memcached? One way would be to add logging to the application to see and count what is being read and written and then to guess from this about the cache efficiency. For to debug what's happening we need to set how the cache keys are used by the application. **An Easier Way** Memcache itself provides a means to peek into its content. The memcache protocol provides [commands](#) to peek into the data that is organized by slabs (categories of data of a given size range). There are some significant limitations though:

1. You can only dump keys per slab class (keys with roughly the same content size)
2. You can only dump one page per slab class (1MB of data)
3. This is an unofficial feature that [might be removed anytime](#). 

The second limitation is propably the hardest because 1MB of several gigabytes is almost nothing. Still it can be useful to watch how you use a subset of your keys. But this might depend on your use case. If you don't care about the technical details just skip to the [tools section](#) to learn about what tools allow you to easily dump everything.

Alternatively follow the following guide and try the commands [using telnet](#) against your memcached setup. **How it Works** First you need to know how memcache organizes its memory. If you start memcache with option "-vv" you see the slab classes it creates. For example

```
$ memcached -vv
slab class 1: chunk size 96 perslab 10922
slab class 2: chunk size 120 perslab 8738
slab class 3: chunk size 152 perslab 6898
slab class 4: chunk size 192 perslab 5461
[...]
```

In the configuration printed above memcache will keep fit 6898 pieces

of data between 121 and 152 byte in a single slab of 1MB size (6898*152). All slabs are sized as 1MB per default. Use the following command to print all currently existing slabs:

```
stats slabs
```

If you've added a single key to an empty memcached 1.4.13 with

```
set mykey 0 60 1
1
STORED
```

you'll now see the following result for the "stats slabs" command:

```
stats slabs
STAT 1:chunk_size 96
STAT 1:chunks_per_page 10922
STAT 1:total_pages 1
STAT 1:total_chunks 10922
STAT 1:used_chunks 1
STAT 1:free_chunks 0
STAT 1:free_chunks_end 10921
STAT 1:mem_requested 71
STAT 1:get_hits 0
STAT 1:cmd_set 2
STAT 1:delete_hits 0
STAT 1:incr_hits 0
STAT 1:decr_hits 0
STAT 1:cas_hits 0
STAT 1:cas_badval 0
STAT 1:touch_hits 0
STAT active_slabs 1
STAT total_malloced 1048512
END
```

The example shows that we have only one active slab type #1. Our key being just one byte large fits into this as the smallest possible

chunk size. The slab statistics show that currently on one page of the slab class exists and that only one chunk is used. **Most importantly it shows a counter for each write operation (set, incr, decr, cas, touch) and one for gets. Using those you can determine a hit ratio!** You can also fetch another set of infos using "stats items" with interesting counters concerning evictions and out of memory counters.

```
stats items
STAT items:1:number 1
STAT items:1:age 4
STAT items:1:evicted 0
STAT items:1:evicted_nonzero 0
STAT items:1:evicted_time 0
STAT items:1:outofmemory 0
STAT items:1:tailrepairs 0
STAT items:1:reclaimed 0
STAT items:1:expired_unfetched 0
STAT items:1:evicted_unfetched 0
END
```

What We Can Guess Already... Given the statistics infos per slabs class we can already guess a lot of thing about the application behaviour:

1. How is the cache ratio for different content sizes?
 - How good is the caching of large HTML chunks?
2. How much memory do we spend on different content sizes?
 - How much do we spend on simple numeric counters?
 - How much do we spend on our session data?
 - How much do we spend on large HTML chunks?
3. How many large objects can we cache at all?

Of course to answer the questions you need to know about the cache objects of your application. **Now: How to Dump Keys?** Keys can be dumped per slabs class using the "stats cachedump" command.

```
stats cachedump <slab class> <number of items to dump>
```

To dump our single key in class #1 run

```
stats cachedump 1 1000
ITEM mykey [1 b; 1350677968 s]
END
```







The "cachedump" returns one item per line. The first number in the braces gives the size in bytes, the second the timestamp of the creation. Given the key name you can now also dump its value using


```
get mykey
VALUE mykey 0 1
1
END
```

This is it: iterate over all slabs classes you want, extract the key names and if need dump there contents.

Dumping Tools



There are different dumping tools sometimes just scripts out there that help you with printing memcache keys:

PHP	simple script 	Prints key names.
Perl	simple script 	Prints keys and values
Ruby	simple script 	Prints key names.
Perl	memdump 	Tool in CPAN module Memcached-libmemcached 
PHP	memcache.php 	Memcache Monitoring GUI that also allows dumping keys
		Does freeze your memcached

libmemcached	peep 	process!!! Be careful when using this in production. Still using it you can workaround the 1MB limitation and really dump all keys.
--------------	--	--

Using Consistent Hashing

Papers:

- [Web Caching with Consistent Hashing](#) 
- [Consistent Hashing and Random Trees](#)  (PDF)

nginx

```
upstream somestream {  
    consistent_hash $request_uri;  
    server 10.0.0.1:11211;  
    server 10.0.0.2:11211;  
    ...  
}
```

PHP

Note: the order of `setOption()` and `addServers()` is important. When using `OPT_LIBKETAMA_COMPATIBLE` the hashing is compatible with all other runtimes using libmemcached.

```
$memcached = new Memcached();  
$memcached->setOption(Memcached::OPT_DISTRIBUTION, Memcached::OPT_LIBKETAMA_COMPATIBLE,  
$memcached->setOption(Memcached::OPT_LIBKETAMA_COMPATIBLE,  
$memcached->addServers($servers);
```

Perl



```





$m = new Memcached('mymemcache');
$m->setOptions(array(
    ...
    Memcached::OPT_LIBKETAMA_COMPATIBLE => true,
    Memcached::OPT_DISTRIBUTION => Memcached::DISTRIBUTION_C
    ...
));
$m->addServers(...);


```

Memcache Alternatives

Below is a list of tools competing with memcached in some manner and a probably subjective rating of each.

Name	Difference	Why [Not] Use It?
memcached	%	Because it simple and fast
memcachedb	Persistence with BDB	Because it is a simple and fast as memcached and allows easy persistence and backup. But not maintained anymore since 2008!
BDB	Simple and old	Use when you want an embedded database. Rarely used for web platforms. Has replication.
CouchBase 	HTTP(S) transport, authentication, buckets, memcached compatible default bucket. Includes moxi proxy that can run on	Sharding, replication and online rebalancing. Often found in small Hadoop setup. Easy drop-in for memcached

	client side or with CouchBase instances.	caching with no consistent hashing.
DynamoDB 	HTTP transport, Amazon cloud	If you are in AWS anyway and want sharding and persistency
Redis 	Key difference is the rich data types: Hashes, Lists, Scanning for Keys, Replication	Great bindings. Good documentation. Flexible yet simple data types. Slower than memcached (read more ).
Riak 	Sharded partitioning in a commercial cloud.	Key-value store as a service. Transparent scaling. Automatic sharding. Map reduce support.
Sphinx	Search Engine with SQL query caching	Supports sharding and full text search. Useful for static medium data sets (e.g. web site product search)
MySQL 5.6	Full RDBMS with memcached API	Because you can run queries against the DB via memcached protocol.

There are many more key-value stores. If you wonder what else is out there look at the [db-engines.com](#)  rankings.

Suggested Reading

- [Scaling Memcache at Facebook](#)  (PDF)

