

MEMCACHE FOR BEGINNERS

SOME QUESTIONS

- (1) What is cache?
- (2) The cache, we (software engineers) use in our daily life?
- (3) Some caching techniques: APC cache, Query cache (inbuilt cache mechanism in MySQL), WP-cache (file system based caching mechanism), Memcache (In memory based cache mechanism)

SOME FACTS

- (1) Memcache is not a database.
- (2) Memcache is a distributed cache system.
- (3) Memcache is not faster than database, but it's faster than database when connections and requests increase.
- (4) Memcache is not meant for providing any backup support. Its all about simple read and write.
- (5) Memcache is unsecure and so its very fast.

MEMCACHED USERS

1. LiveJournal
2. Wikipedia
3. Flickr
4. Twitter
5. Youtube
6. Dig
7. Wordpress
8. Craigslist
9. Facebook (around 200 dedicated memcache servers)
10. Yahoo! India Movies

WHAT IS MEMCACHE?

[Memcache](#) is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.

> in-memory (volatile) key-value store

```
$memcache->set('unique_key', $value, $flag, $expiration_time);  
$flag = 0 / MEMCACHE_COMPRESSED to store the item compressed.  
$expiration_time = 0 (never expire) / 30 (30 seconds) etc.
```

```
$memcache->get('unique_key');  
NOTE: Missing key makes fetch time doubles.
```

> distributed memory caching system (you can use more than one server to cache your data)

```
$memcache->addServer('host1', 11211);  
$memcache->addServer('host2', 11211);  
$memcache->addServer('host3', 11211);
```

NOTE: You can pass more parameters to [addServer\(\)](#) function

WHAT YOU CAN STORE IN MEMCACHE?

Results of database calls (array, object), API calls (xml as string), page rendering (html as string) etc.

NOTE: Objects are [serialized](#) before being stored to memcache i.e. problem with DOM, XML ??

PHP PROGRAMMER AND SYS ADMIN STORY

(1) php programmer wrote some code (lots of database call, API calls etc.) and launched the application.

Assumptions for this example

- this one page php application has 10 function calls and each function has 2 DB calls i.e. total 20 DB calls
- one DB call take 5 seconds to return result set i.e. ideal application page loading time: $20 * 5 = 100$ seconds = 1 min 40 sec
- 20 DB calls = database load: 1
- database can handle 100 SQL queries at a time and so max load it can take is 100 queries i.e. if one user visit the site, database load: 1, if 5 users visit the site simultaneously, database load is: 5, if 6 users visit the site simultaneously, no result as load is high / database crash

NOTE: All above assumptions are just for understanding the problem and it does not reflect any real data.

What are the problems here?

- > High execution time
- > Heavy load on database

What ideally required for better performance?

- > Low execution time
- > Low database load

(2) php programmer contacts sys admin. Sys admin googles and finds "memcached"

(3) what he (sys admin) does?

1. install memcached server.

```
yum install memcached
```

2. check if memcache server is running

```
memcached -u [your webserver user: apache / nobody / www-data] -d -m [30  
- memory in MB] -l [127.0.0.1] -p [11211 - default port]
```

```
telnet [127.0.0.1] 11211
```

if you can telnet, memcached server is running.

NOTE: [Install memcached & configure manually](#)

3. Now what? he need some sort of APIs to communicate with server.

> application is developed using php so they need memcache client for php, install it:

```
yum install php-pecl-memcache
```

> run phpinfo() and see if you get "memcache" extension activated.

> lets see if apache can communicate to memcache server.
add iptables rules to allow.

```
iptables -A INPUT -m state --state NEW -m tcp -p tcp --dport 11211 -j  
ACCEPT  
iptables -A INPUT -m state --state NEW -m udp -p udp --dport 11211 -j  
ACCEPT
```

4. DONE

(4) sys admin ask programmer to use memcache APIs now.

(5) what php programmer does?

> in one function he made memcache function calls as below for 2 DB calls.

```
$memcache = new Memcache;  
  
$result = $memcache->get('unique_key');  
  
if(empty($result)) {  
    $result = DB_Calls();  
    $memcache->set('unique_key', $results);  
}
```

NOTE: Again, Memcache is faster than DB when connections and requests increase.

> So, let say 5 users are visiting the site simultaneously. What will be the database load now?
5 users > 90 DB calls (instead 100 DB calls) > database load 4.5 > page load time: 1 min 30 sec

> Now programmer writes memcahe calls for each DB calls
5 users > ideally 0 DB call (instead 100 DB calls) > database load 0 > page load time: 0 sec

MYSQL DB IS ON VACATION

(6) php programmer and sys admin both are happy :)

(7) Now what? consider following condition.

- > let say application has lots of DB calls, lots of web pages, lots of API calls
- > all data written to DB and cache is using CRON jobs only.
- > somehow a memcache server got crashed :(
- > what will happen?

memcache read will fail and it won't display anything on web page as we are reading from memcache only.

> whats the solution?

1) addServer() - add more memcache servers.

> will it replicate cache on each new server?

> adding new servers means, you are adding more memory for caching.

i.e. key1 can be on node1, key2 can be on node2, key3 can be on node3 etc. BUT its not possible to have key1 on all 3 servers.

2) you must need database support in such case.

> let say out of 3 servers, only 2 are working, will application get crashed?

References

Memcache wiki: <http://code.google.com/p/memcached/wiki/Start>

External resources: <http://code.google.com/p/memcached/wiki/Resources>

SOME QUESTIONS

(1) What is the maximum key length? 250 characters

(2) What are the limits on setting expire time?

You can set expire times up to 30 days in the future. After that memcached interprets it as a date, and will expire the item after said date.

(3) What is the maximum data size you can store? 1 MB

(4) Memcached is not faster than database. Why?

In a one to one comparison, memcached may not be faster than your SQL queries. However, this is not its goal. Memcached's goal is **scalability**. As connections and requests increase, memcached will perform better than most database only solutions.

(Scalability: the capability of a system to increase total throughput under an increased load when resources (typically hardware) are added.)

(5) What this code will do?

```
$memcache = new Memcache()  
  
$memcache->addServer('node1', 11211);  
$memcache->addServer('node2', 11211);  
$memcache->addServer('node3', 11211);  
  
$memcache->connect('node1', 11211);
```

The last [connect\(\)](#) call clears out the pool and then add and connect node1:11211 making it the only server

NOTE: If you want a pool of memcache servers, do not use the connect() function.

(6) Other [memcache functions](#)?

```
$memcache->delete('unique_key');  
$memcache->close();
```

(7) How to build **highly scalable & high performance web applications?**

3 tier architecture: web server, memcache server, and MySQL database.

(8) When to write?

- decide default cache time for your application (say 10 min)
- write to DB and also to memcache, when information is updated.

MEMCACHED AS PHP SESSION HANDLER

```
session.save_handler = memcache  
session.save_path = "tcp://1.1.1.1:11211,tcp://1.1.1.2:11211,tcp://1.1.1.3:11211"
```

- Its called **session clustering** (distributed sessions) using memcache pool.
- Multiple application server instances shares a common pool of sessions. (Centralized Authentication System?)

NOTE: Each URL may contain parameters which are applied to that server, they are the same as for the Memcache::addServer() method.

For e.g. "tcp://host1:11211?persistent=1&weight=1&timeout=1&retry_interval=15"

- Store sessions to both DB and memcache. Why? If memcache server fails, no user can login to website.
- Write your own session handler that stores to the database and memcache.

MEMCACHED REPLICATION AS A FAIL-OVER SOLUTION (REDUNDANCY)

NOTE: Memcache is not meant for providing any backup support. Its all about simple write and read. Then why we need replication?

If we are storing session only in memcache (no DB sessions) then we need replication.

Replication can be enabled by setting the INI settings (/etc/php.d/memcache.ini)

```
; Enable memcache extension module  
extension=memcache.so  
  
; Options for the memcache module  
  
; Whether to transparently failover to other servers on errors  
;memcache.allow_failover=1  
  
; Defines how many servers to try when setting and getting data.  
;memcache.max_failover_attempts=20  
  
; Data will be transferred in chunks of this size  
;memcache.chunk_size=8192
```

```
; The default TCP port number to use when connecting to the memcached server
;memcache.default_port=11211

; Hash function {crc32, fnv}
;memcache.hash_function=crc32

; Hash strategy {standard, consistent}
;memcache.hash_strategy=standard

; When used from PHP. (default value 1 )
;memcache.redundancy = 2

; When used as a session handler (default value 1 )
;memcache.session_redundancy = 2
```

```
; Options to use the memcache session handler

; Use memcache as a session handler
;session.save_handler=memcache

; Defines a comma separated of server urls to use for session storage
;session.save_path="tcp://localhost:11211?persistent=1&weight=1&timeout=1&retry_interval=15"
```

Query

What is difference between memcache.redundancy and memcache.session_redundancy?
Let say I am setting memcache.session_redundancy = 2 i.e. session will be written to 2 servers.

```
$memcache = new Memcache;
$memcache->addServer('192.168.0.1', 11211);
$memcache->addServer('192.168.0.2', 11211);
$memcache->addServer('192.168.0.3', 11211);
$memcache->addServer('192.168.0.4', 11211);
```

So, how memcache will know which 2 server to choose? or
it will consider following php directives and use 192.168.0.1 and 192.168.0.2?

```
session.save_handler = memcache
session.save_path = "tcp://192.168.0.1:11211,tcp://192.168.0.2:11211"
what will happen if I set memcache.redundancy = 3 for above case with
memcache.session_redundancy = 2?
```

Thanks
Nilesh

Expert's Reply

memcache.redundancy is for stuff you store in memcached yourself with Memcache::add.
memcache.session_redundancy is for PHP session data that's stored in memcached.

For choosing servers, have a look at memcache_pool.c for the mc_pool_find_next function. From a brief glance at the source, it looks like whenever you store a key "foo", if redundancy is enabled it's actually storing "foo-1" and "foo-2" (with redundancy = 2), so each one should (hopefully) hash to a different server.

In your example, memcache.redundancy isn't used because you're dealing with sessions. However if you stored a key with Memcache::add, I think only 2 versions would be stored because the outer loop that calls mc_pool_find_next only counts up to redundancy or # servers:

```
for (i=0; i < redundancy-1 && i < pool->num_servers-1; i++) {
```

Links:

[[memcache_pool.c](#)]

Re:Query

Hello Sean

Thanks for your quick response. It was very helpful. Have quick questions.

- (1) Does that mean, replication is possible without using memcache + repcache?
- (2) If I put memcache.redundancy = 2 along with memcache.session_redundancy = 2, memcache will replicate data other than session data also in 2 memcache server and same time session data will also get replicated over 2 memcache server?
- (3) Do we still need session.save_path directive? Can't I just tell php that use memcache for storing session using session.save_handler = memcache? and servers added using addServer() method will be used by memcached for session storing instead session.save_path.

Re:Expert's Reply

Hi Nilesh,

I haven't used it this way, but based on reading the docs and source:

1. Your data is stored on two servers. I'm not going to get into "is that replication?" ;)
2. Yes
3. save_path is only meaningful if you're storing sessions on disk.

This may be editorializing a bit, but remember that even with the client based replication your data is not persistent. My own opinion and memcache philosophy has been that a cache-miss should not be catastrophic.

This client based replication introduces several consistency problems. For example, if you lose a

server and then bring it back up, you've got one server with no data (that will return a NAK) and one with data.

References

<http://lists.horde.org/archives/horde/Week-of-Mon-20081117/036482.html>

<http://pecl.php.net/bugs/bug.php?id=11637>

CONSISTENT HASHING

The need for consistent hashing arose from limitations experienced while running collections of caching machines - web caches, for example. If you have a collection of n cache machines then a common way of load balancing across them is to put object o in cache machine number $\text{hash}(o) \bmod n$. This works well until you add or remove cache machines (for whatever reason), for then n changes and every object is hashed to a new location. This can be catastrophic since the originating content servers are swamped with requests from the cache machines. It's as if the cache suddenly disappeared. Which it has, in a sense. (This is why you should care - consistent hashing is needed to avoid swamping your servers!)

It would be nice if, when a cache machine was added, it took its fair share of objects from all the other cache machines. Equally, when a cache machine was removed, it would be nice if its objects were shared between the remaining machines. This is exactly what consistent hashing does - consistently maps objects to the same cache machine, as far as is possible, at least.

The basic idea behind the consistent hashing algorithm is to hash both objects and caches using the same hash function. The reason to do this is to map the cache to an interval, which will contain a number of object hashes. If the cache is removed then its interval is taken over by a cache with an adjacent interval. All the other caches remain unchanged.

So how can you use consistent hashing? You are most likely to meet it in a library, rather than having to code it yourself. For example, as mentioned above, memcached, a distributed memory object caching system, now has clients that support consistent hashing.

Reference

<http://www.lexemetech.com/2007/11/consistent-hashing.html>

SCALING OUT (CACHE CONSISTENCY) – FACEBOOK ENGINEERING'S NOTES

A bit of background on our caching model: when a user modifies a data object our infrastructure will write the new value in to a database and delete the old value from memcache (if it was present). The next time a user requests that data object we pull the result from the database and write it to memcache. Subsequent requests will pull the data from memcache until it expires out of the cache or is deleted by another update.

This setup works really well with only one set of databases because we only delete the value from

MEMCACHE FOR ENGINEERS

memcache after the database has confirmed the write of the new value. That way we are guaranteed the next read will get the updated value from the database and put it in to memcache. With a slave database on the east coast, however, the situation got a little tricky.

When we update a west coast master database with some new data there is a replication lag before the new value is properly reflected in the east coast slave database. Normally this replication lag is under a second but in periods of high load it can spike up to 20 seconds.

Now let's say we delete the value from Virginia memcache tier at the time we update the master database in California. A subsequent read from the slave database in Virginia might see the old value instead of the new one because of replication lag. Then Virginia memcache would be updated with the old (incorrect) value and it would be "trapped" there until another delete. As you can see, in the worst case the Virginia memcache tier would always be one "version" behind of the correct data.

Consider the following example:

1. I update my first name from "Jason" to "Monkey"
2. We write "Monkey" in to the master database in California and delete my first name from memcache in California and Virginia
3. Someone goes to my profile in Virginia
4. We don't find my first name in memcache so we read from the Virginia slave database and get "Jason" because of replication lag
5. We update Virginia memcache with my first name as "Jason"
6. Replication catches up and we update the slave database with my first name as "Monkey"
7. Someone else goes to my profile in Virginia
8. We find my first name in memcache and return "Jason"

Until I update my first name again or it falls out of cache and we go back to the database, we will show my first name as "Jason" in Virginia and "Monkey" in California. Confusing? You bet. Welcome to the world of distributed systems, where consistency is a really hard problem.

Fortunately, the solution is a lot easier to explain than the problem. We made a small change to MySQL that allows us to tack on extra information in the replication stream that is updating the slave database. We used this feature to append all the data objects that are changing for a given query and then the slave database "sees" these objects and is responsible for deleting the value from cache after it performs the update to the database.

How'd we do it? MySQL uses a [lex parser and a yacc grammar](#) to define the structure of a query and then parse it. I've simplified the following for ease of explanation, but at the highest level this grammar looks like:

```
query:  
statement END_OF_INPUT {};
```

MEMCACHE FOR ENGINEERS

statement:

alter

| analyze

| backup

| call

... (insert, replace, select, etc.)

Pretty straightforward, right? A **query** is a **statement** which breaks down to one of the MySQL expressions we all know and love. We modified this grammar to allow appending memcache keys to the end of any query, as follows:

query:

```
statement mc_dirty END_OF_INPUT {};
```

mc_dirty:

```
{}
```

```
| MEMCACHE_DIRTY mc_key_list;
```

mc_key_list:

```
mc_key_list ',' text_string { Lex->mc_key_list.push_back($3); }
```

```
| text_string { Lex->mc_key_list.push_back($1); };
```

A query now has an additional component; after the statement comes the **mc_dirty** which is either empty or a keyword **MEMCACHE_DIRTY** followed by a **mc_key_list**. A **mc_key_list** is just a comma-separated list of strings and the rule tells the parser to push all the strings one-by-one on to a vector named `mc_key_list` which is stored inside a per-query parser object.

As an example, an old query might look like:

```
REPLACE INTO profile (`first_name`) VALUES ('Monkey') WHERE `user_id`='jsobel'
```

and under the new grammar it would change to:

```
REPLACE INTO profile (`first_name`) VALUES ('Monkey') WHERE `user_id`='jsobel'
```

```
MEMCACHE_DIRTY 'jsobel:first_name'
```

The new query is telling MySQL that, in addition to changing my first name to Monkey, it also needs to dirty a corresponding memcache key. This is easily implemented. Since the per-query parser object now stores all memcache keys we tack on to a query, we added a small piece of code at the end of `mysql_execute_command` that dirties those keys if the query is successful. Voila, we've hijacked the MySQL replication stream for our own purpose: cache consistency.

The new workflow becomes (changed items in bold):

1. I update my first name from "Jason" to "Monkey"
2. We write "Monkey" in to the master database in California and delete my first name from memcache in California **but not Virginia**

3. Someone goes to my profile in Virginia
4. **We find my first name in memcache and return "Jason"**
5. Replication catches up and we update the slave database with my first name as "Monkey." **We also delete my first name from Virginia memcache because that cache object showed up in the replication stream**
6. Someone else goes to my profile in Virginia
7. **We don't find my first name in memcache so we read from the slave and get "Monkey"**

Reference

http://www.facebook.com/note.php?note_id=23844338919

RATE LIMITING WITH MEMCACHE

On Monday, several high profile “celebrity” Twitter accounts [started spouting nonsense](#), the victims of stolen passwords. Wired [has the full story](#)—someone ran a dictionary attack against a Twitter staff member, discovered their password and used Twitter’s admin tools to reset the passwords on the accounts they wanted to steal.

The Twitter incident got me thinking about rate limiting again. I’ve been wanting a good general solution to this problem for quite a while, for API projects as well as security. Django Snippets has [an answer](#), but it works by storing access information in the database and requires you to run a periodic purge command to clean up the old records.

I’m strongly averse to writing to the database for every hit. For most web applications reads scale easily, but writes don’t. I also want to avoid filling my database with administrative gunk (I dislike database backed sessions for the same reason). But rate limiting relies on storing state, so there has to be some kind of persistence.

Using memcached counters

I think I’ve found a solution, thanks to memcached and in particular the `incr` command. `incr` lets you atomically increment an already existing counter, simply by specifying its key. `add` can be used to create that counter—it will fail silently if the provided key already exists.

Let’s say we want to limit a user to 10 hits every minute. A naive implementation would be to create a memcached counter for hits from that user’s IP address in a specific minute. The counter key might look like this:

```
rateLimit_72.26.203.98_2009-01-07-21:45
```

Increment that counter for every hit, and if it exceeds 10 block the request.

What if the user makes ten requests all in the last second of the minute, then another ten a second later? The rate limiter will let them off. For many cases this is probably acceptable, but we can improve things with a slightly more complex strategy. Let’s say we want to allow up to 30 requests every five minutes. Instead of maintaining one counter, we can maintain five—one for each of the past five minutes (older counters than that are allowed to expire). After a few minutes we might end up with

MEMCACHE FOR ENGINEERS

counters that look like this:

```
ratelimit_72.26.203.98_2009-01-07-21:45 = 13  
ratelimit_72.26.203.98_2009-01-07-21:46 = 7  
ratelimit_72.26.203.98_2009-01-07-21:47 = 11
```

Now, on every request we work out the keys for the past five minutes and use `get_multi` to retrieve them. If the sum of those counters exceeds the maximum allowed for that time period, we block the request.

Are there any obvious flaws to this approach? I'm pretty happy with it—it cleans up after itself (old counters quietly expire from the cache), it shouldn't use much resources (just five active cache keys per unique IP address at any one time) and if the cache is lost the only snag is that a few clients might go slightly over their rate limit. I don't *think* it's possible for an attacker to force the counters to expire early.

Reference

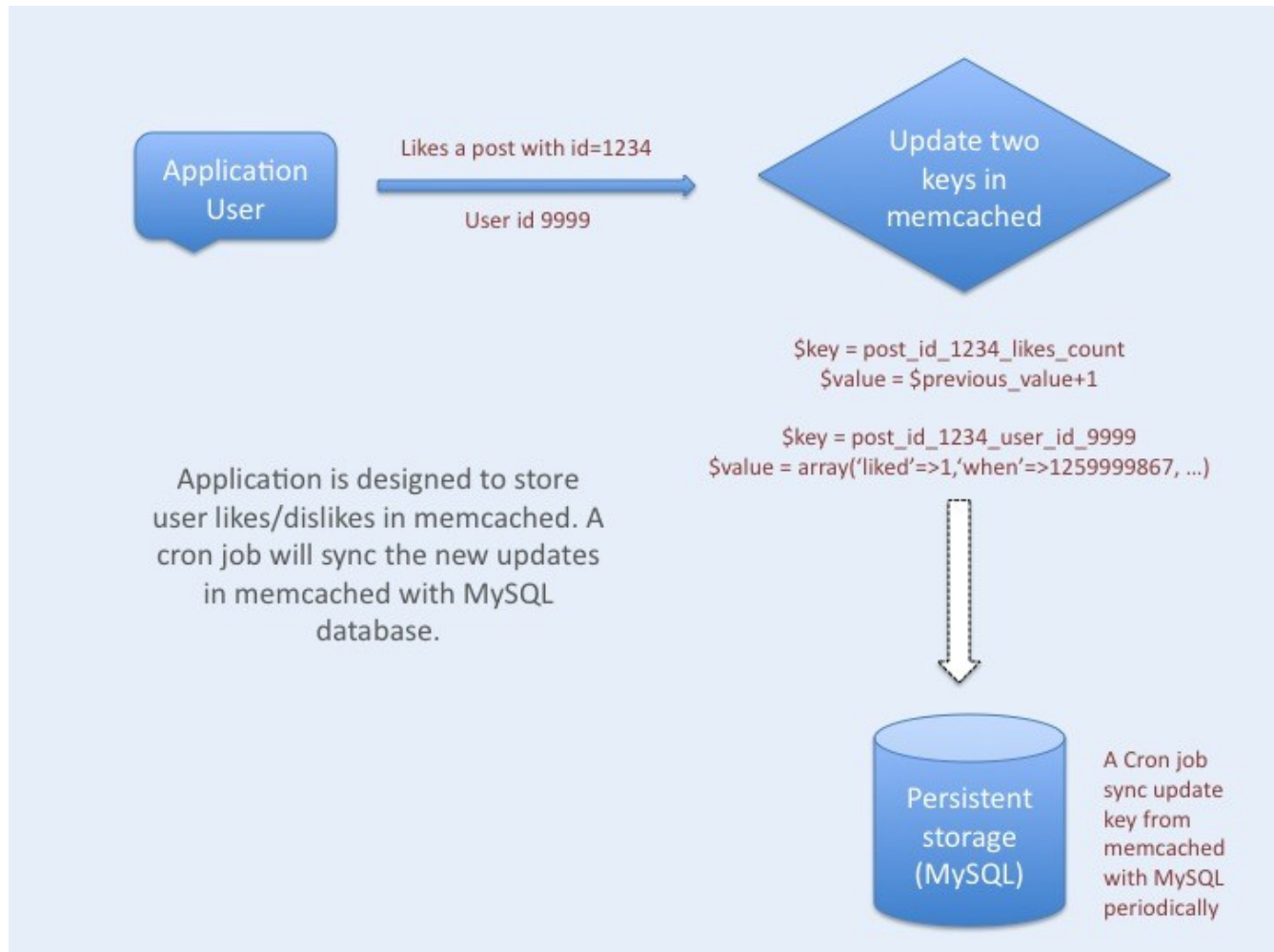
<http://simonwillison.net/2009/Jan/7/ratelimitcache/>

LOCKS FOR ASSURING ATOMIC OPERATION IN MEMCACHED

Memcached provide atomic increment and decrement commands to manipulate integer (key,value) pairs. However special care should be taken to ensure application performance and possible race conditions while using memcached. In this blog post, I will first build a facebook style “like” application using atomic `increment` command of memcached. Also, I will discuss various technical difficulty one would face while ensuring atomicity in this application. Finally, I will demo how to ensure atomicity over a requested process using custom locks in memcached.

Where should I care about it?

Lets consider a sample application as depicted by the flow diagram below:



The above application is similar to facebook "like" feature. In brief, we maintain a key per post e.g. `$key="post_id_1234_likes_count"`, storing count of users who liked this post. Another `$key="post_id_1234_user_id_9999"`, stores `user_id_9999` relationship with `post_id_1234`. Example, "liked" which is set to 1 if liked and "timestamp" which is the time when user liked this post.

Since this application is going to reside on a high traffic website, earlier design decisions are made to have memcached in-front of MySQL database and will act as the primary storage medium with periodic syncs to the database. For me a like/dislike functionality is not so important as compared to other social functionality on my website.

Here is a sample code for the above functionality:

```
$mem = new Memcache;
$mem->addServer("127.0.0.1", 11211);
function incrementLikeCount($post_id) {
    global $mem;
```

```
// prepare post key
$key = "post_id_".$post_id."_likes_count";
// get old count
$old_count = $mem->get($key);
// false means no one liked this post before
if($old_count === FALSE) $old_count = 0;
// increment count
$new_count = $old_count+1;
// set new count value
if($mem->set($key, $new_count, 0, 0)) {
    error_log("Incremented key ".$key." to ".$new_count);
    return TRUE;
}
else {
    error_log("Error occurred in incrementing key ".$key);
    return FALSE;
}
}
// get incoming parameters
$post_id = $_GET['post_id'];
// take action
incrementLikeCount($post_id);
```

Why should I care about it?

Save the above code sample in a file called `memcached_no_lock.php` and hit the url `http://localhost/memcached_no_lock.php?post_id=1234` five times. Verify the key value in memcached:

```
centurydaily-lm:Workspace sabhinav$ telnet localhost 11211
Trying ::1...
Connected to localhost.
Escape character is '^]'
```

MEMCACHE FOR BEGINNERS

```
get post_id_1234_likes_count
VALUE post_id_1234_likes_count 0 2
5
END
```

Alright, application seems to give expected results. Next, let's verify this application for high traffic websites using apache benchmark:

```
centurydaily-lm:Workspace sabhinav$ ab -n 100 -c 10
http://localhost/memcached_no_lock.php?post_id=1234
Concurrency Level: 10
Time taken for tests: 0.090 seconds
Complete requests: 100
Failed requests: 0
Write errors: 0
Total transferred: 22200 bytes
HTML transferred: 0 bytes
Requests per second: 1112.03 [#/sec] (mean)
```

Verify the key value in memcached:

```
centurydaily-lm:Workspace sabhinav$ telnet localhost 11211
Trying ::1...
Connected to localhost.
Escape character is '^]'.
get post_id_1234_likes_count
VALUE post_id_1234_likes_count 0 2
36
END
```

What happened? We expected value for `$key="post_id_1234_likes_count"` to reach 100, but actually it is 36. What went wrong? This behavior can be explained by simply looking at the apache error log file:

```
[Sat Dec 05 14:32:08 2009] [error] [client ::1] Incremented key post_id_1234_likes_count to 1
[Sat Dec 05 14:32:08 2009] [error] [client ::1] Incremented key post_id_1234_likes_count to 1
[Sat Dec 05 14:32:08 2009] [error] [client ::1] Incremented key post_id_1234_likes_count to 1
[Sat Dec 05 14:32:08 2009] [error] [client ::1] Incremented key post_id_1234_likes_count to 2
[Sat Dec 05 14:32:08 2009] [error] [client ::1] Incremented key post_id_1234_likes_count to 2
[Sat Dec 05 14:32:08 2009] [error] [client ::1] Incremented key post_id_1234_likes_count to 3
[Sat Dec 05 14:32:08 2009] [error] [client ::1] Incremented key post_id_1234_likes_count to 3
[Sat Dec 05 14:32:08 2009] [error] [client ::1] Incremented key post_id_1234_likes_count to 3
```

Ohk, from above log we understand concurrency killed our application, since we see \$key being incremented to the same value by more than 1 incoming request.

How should I take care of this?

Below is the modified code sample which will allow us atomic increments:

```

$mem = new Memcache;
$mem->addServer("127.0.0.1", 11211);

function incrementLikeCount($post_id) {
    global $mem;

    // prepare post key
    $key = "post_id_".$post_id."_likes_count";

    $new_count = $mem->increment($key, 1);
    if($new_count === FALSE) {
        $new_count = $mem->add($key, 1, 0, 0);
        if($new_count === FALSE) {
            error_log("Someone raced us for first count on key ".$key);
            $new_count = $mem->increment($key, 1);
            if($new_count === FALSE) {
                error_log("Unable to increment key ".$key);
                return FALSE;
            }
        }
        else {
            error_log("Incremented key ".$key." to ".$new_count);
            return TRUE;
        }
    }
    else {
        error_log("Initialized key ".$key." to ".$new_count);
        return TRUE;
    }
}

// get incoming parameters
$post_id = $_GET['post_id'];

// take action

```

```
incrementLikeCount($post_id);
```

To ensure atomicity, we start with incrementing the `$key="post_id_1234_likes_count"`. Since memcached `increment()` is atomic by itself, we need not put any locking mechanism in here. However, memcached `increment` returns `FALSE`, if the `$key` doesn't already exist.

Hence, if we get a `FALSE` response from the first `increment`, we will try to initialize `$key` using memcached `add()` command. Good thing about memcached `add` is that, it will return a `false FALSE`, if the `$key` is already present. Hence, if more than one thread is trying to initialize `$key`, only one of them will succeed. All the rest of the threads will return `FALSE` for `add` command. Finally, if the response is `FALSE` from the first `add`, we will try to `increment` the `$key` again.

Let's try to test this modified code with `apache benchmark`. Also, this time we will increase concurrency from 10 to 100 threads. Save the above modified code in a file called `memcached_lock.php` and issue the following `ab` command:

```
centurydaily-lm:Workspace sabhinav$ ab -n 10000 -c 100
http://localhost/memcached_lock.php?post_id=1234
Concurrency Level:    100
Time taken for tests:  11.006 seconds
Complete requests:   10000
Failed requests:      0
Write errors:         0
Total transferred:   2224884 bytes
HTML transferred:    0 bytes
Requests per second:  908.61 [#/sec] (mean)
```

Let's verify the key value inside memcached:

```
centurydaily-lm:Workspace sabhinav$ telnet localhost 11211
Trying ::1...
Connected to localhost.
Escape character is '^]'.
get post_id_1234_likes_count
VALUE post_id_1234_likes_count 0 5
10000
END
```

Bingo! As desired we have a value of 10000 for `$key` inside memcached.

Using custom locks for atomicity:

There can be many instances where you SHOULD try to process a request atomically using locks. For e.g. while trying to fetch a query from database or while trying to regenerate a requested page template in your [custom template caching engine](#).

MEMCACHE FOR ENGINEERS

In the example below, I will modify the memcached_lock.php script to ensure atomic increments without using increment command. Instead I will use custom locks using memcached:

```
$mem = new Memcache;
$mem->addServer("127.0.0.1", 11211);

function incrementLikeCount($post_id) {
    global $mem;

    // prepare post key
    $key = "post_id_".$post_id."_likes_count";

    // initialize lock
    $lock = FALSE;

    // initialize configurable parameters
    $tries = 0;
    $max_tries = 1000;
    $lock_ttl = 10;

    $new_count = $mem->get($key); // fetch older value
    while($lock === FALSE && $tries < $max_tries) {
        if($new_count === FALSE) $new_count = 0;
        $new_count = $new_count + 1;

        // add() will return false if someone raced us for this lock
        // ALWAYS USE add() FOR CUSTOM LOCKS
        $lock = $mem->add("lock_".$new_count, 1, 0, $lock_ttl);

        $tries++;
        usleep(100*($tries%($max_tries/10))); // exponential backoff style of
sleep
    }

    if($lock === FALSE && $tries >= $max_tries) {
        error_log("Unable to increment key ".$key);
        return FALSE;
    }
    else {
        $mem->set($key, $new_count, 0, 0);
        error_log("Incremented key ".$key." to ".$new_count);
        return TRUE;
    }
}
```

MEMCACHE FOR ENGINEERS

```
// get incoming parameters
$post_id = $_GET['post_id'];

// take action
incrementLikeCount($post_id);
```

Try testing it using apache benchmark as above and then verify it with memcached.

```
centurydaily-lm:Workspace sabhinav$ telnet localhost 11211
Trying ::1...
Connected to localhost.
Escape character is '^]'.
get post_id_1234_likes_count
VALUE post_id_1234_likes_count 0 3
100
END
```

Benchmarks:

We see a drop in performance from 1112 hits/sec (memcached_no_lock) to 908 hits/sec (memcached_lock using increment). This is majorly because of increased concurrency. At same concurrency level of 10, I received a performance benchmark of 1128 hits/sec with our thread protected code. However, for our custom lock code above, I received a performance benchmark of 275 hits/sec.

Conclusion:

Always use memcached increment/decrement while dealing with locks on integer valued keys. For achieving locks on a process, use custom locks as demoed above using memcached add command. Also custom locks are subjected to configurable options like `$max_tries` and others.

Reference

<http://abhinavsingh.com/blog/2009/12/how-to-use-locks-for-assuring-atomic-operation-in-memcached/>