

# A simple distributed lock with memcached

*Posted on Wed 28 October 2009*

When you have a cluster of web application servers, you often need to coordinate the activity of your servers to avoid the same expensive work being done at the same time when a condition triggers it.

Most people use [memcached](http://code.google.com/p/memcached/) (<http://code.google.com/p/memcached/>) as a simple key/value store but it can also be used as a simple distributed lock manager: along with the `put(key, value)` operation, it also has an `add(key, value)` operation that succeeds only if the cache wasn't already holding a value for the key.

Locking then becomes easy:

```
if (cache.add("lock:xyz", "1", System.currentTimeMillis() + 60000)) {
    try {
        doSomeExpensiveStuff();
    } finally {
        cache.delete("lock:xyz");
    }
} else {
    // someone else is doing the expensive stuff
}
```

The code above tries to get the lock by adding a dumb value for our lock's identifier, with an expiration of one minute. This is the lock lease time, and should be more than the estimated maximum time for the lengthy operation. This avoids the lock being held forever if ever things go really bad such as your server crashing.

Once the operation is completed, we delete the lock, et voilà.

If you want the system to be rock-solid, you should check that you still own the lock before deleting it (in case the lease time expired), but in most cases this simple approach works nicely.

And if the expensive operation resets in the database the condition that triggered it, the lock should be released once the transaction has been committed to prevent a race condition in the time interval between the end of the expensive operation and the actual commit that would allow other servers to restart the same work. [Spring's transaction synchronization](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/transaction/support/TransactionSynchronization) (<http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/transaction/support/TransactionSynchronization>) helps doing that.

**Update:** as [Leo points out](http://twitter.com/lsimons/statuses/5225112451) (<http://twitter.com/lsimons/statuses/5225112451>), the above works as long as memcache doesn't decide to flush your lock to have more room. Practically, the small size and short life time of a lock should ensure this almost always works. If locking is critical though, either use a dedicated memcache server for locks or use another solution like [ZooKeeper](http://hadoop.apache.org/zookeeper/) (<http://hadoop.apache.org/zookeeper/>).

---

## 8 Comments

Type Comment Here (at least 3 chars)

Name (optional)

E-mail (optional)

Website (optional)

Submit



Christopher Chan • 10 years ago

Hi, Sylvain Wallez: Is it possible that two servers executing the code `cache.add("lock:xyz", "1", System.currentTimeMillis() + 60000)` at the same time will both return true? how did the memcached ensure atomic for test and set actions?

1 ^ | v **Reply**



**Sylvain Wallez** • 10 years ago

It's not possible: memcached operations are atomic, and the contract of `add()` is to fail if there's already a value for that key.

So if two processes send `add("lock:xyz", "1")` to a given memcached server at the same time, requests will be serialized internally by memcached: one will succeed and the other will fail.

See also [http://code.google.com/p/memcached/wiki/FAQ#Is\\_memcached\\_atomic](http://code.google.com/p/memcached/wiki/FAQ#Is_memcached_atomic) ([http://code.google.com/p/memcached/wiki/FAQ#Is\\_memcached\\_atomic](http://code.google.com/p/memcached/wiki/FAQ#Is_memcached_atomic))?

^ | v **Reply**



**Anonymous** • 2 years ago

Hi, How about a cluster of memcached server ? any idea of to keep atomicity accross the memcached cluster ?

^ | v **Reply**



**Sylvain Wallez (<https://bluxte.net>)** • 2 years ago

The term "memcached cluster" is misleading as memcached has no clustering feature. It's generally used to designate a set of (independent) memcached server on which the load is spread by application-level hashing of keys. So to answer your question, it would work the same with a memcached "cluster" since the entry for the lock would exist only on one server.

^ | v **Reply**



**Rajul** • last year

This approach does not work if the memcache server owning the lock dies. At this point that key is owned by another server in the ring and another client can acquire the lock while our original client still thinks it has exclusive ownership of the lock.

^ | v **Reply**



**Sylvain Wallez (<https://bluxte.net>)** • last year

Absolutely. This is why it's a "simple" distributed lock. This post is almost 9 years old, and nowadays we have products that provide this distributed resiliency that memcached doesn't provide. The main option available 9 years ago was Zookeeper.

^ | v **Reply**



**Anonymous** • last year

Which products? This site still comes up as a top hit for those searching for distributed locking solutions using memcached, so pointing us to newer, more robust technologies would be appreciated.






^ | v **Reply**



**Michael** • 12 months ago

Some newer, more robust technologies are etcd and consul. The thing is, they are complex. They are marginally less complicated than zookeeper, (they use RAFT instead of PAXOS locking) but using a very simple approach like memcached, redis or dynamodb still has value. Because of how memcached works, it will always lose the data it has when it dies. If people are searching for memcached distributed locking, they need to take that into account.

-1 ^ | v **Reply**

-  (<https://twitter.com/bluxte>)
-  (<https://www.linkedin.com/in/swallez>)
-  (<https://github.com/swallez>)
-  (<http://www.slideshare.net/swallez>)
-  (<https://www.facebook.com/sylvain.wallez>)