Memcached

Jeremy Leishman

Ben Robison

Josh Taylor

Memcached

Memcached is a fairly new technology aimed at decreasing heavy database loads by adding a scalable object-caching layer to an application. There are several people using Memcached and some you've definitely heard of. LiveJournal, Wikipedia, Slashdot, and Digg are among the most notable.

As with any valuable new technology, Memcached evolved to meet an identified problem in the market. We'll talk about this problem, and how Memcached provides an excellent solution. Then we'll go into a more technical explanation of how Memcached actually works. Following that, we'll compare Memcached to a few alternatives and then discuss some of the general positives and negatives to implementing and using Memcached. Then we'll wrap it all up by talking about out own experience with its implementation and usage.

Problem & Solution

Let's imagine a scenario. Pretend you are in charge of a large, interactive, database-driven web site. It started out small, but has grown dramatically and now has literally thousands of users who hit the site every day. Some users are just reading, some users are posting messages, some users are uploading photos, and many users are doing a combination of all those activities. Each user has different levels of access, which needs to be managed as well. Since you are in charge of this website you are concerned with the user's experience.

You are worried that with a large and growing user base perhaps the speed of the site will become a problem. You're already using several servers to balance the load, but your database clusters in particular are having large trouble keeping up with the demands being placed on it.

LiveJournal

Brad Fitzpatrick found himself in this position working with LiveJournal.com for the past eight years. Brad needed to optimize LiveJournal's speed to meet the demands of a growing user base.

Brad needed to find a cheap and scalable solution that would evolve with his user-base. One option to speed up site access is to pre-generate static pages. However, Brad knew from previous assignments that static pages don't work well with dynamic, context-aware content. In addition, it is impossible to predict a page's popularity in advance and Brad had no idea which pages he should pre-generate and which should continue to be served directly.

Caching

Brad knew that caching should have its place in the ultimate solution. He also knew that caching works best when used at the right granularity and in the right layer. Previous project solutions had cached the whole page with unsatisfying results, and Brad found that caching individual objects on a page was more efficient, reduced wasted processing time, and eliminated duplicate caching. Memcached became the software that met LiveJournal's needs. It resides in between the database and the application providing a single global, scalable object cache.

Brad also knew that caching could really be described in terms of a series of trade-offs. Both processors and disks continue to evolve, but they evolve in different ways. Processors decrease in size and increase in speed. The physical size of disks increasingly shrinks while capacity grows, but for the most part, disk access speeds have not gotten any faster for many years. Brad needed a solution that would provide speed.

Another trade-off relates to the freshness of your data. You can cache data to reduce load on your systems, or you can have piping fresh data from your database. These two goals conflict at a very basic level, so the amount of caching you do is very specific to an application. The important thing to remember is that the database can't make that trade-off. The application is responsible for doing that. A good caching mechanism is much simpler to implement and much easier to scale horizontally than anything you can do at the database layer.

Memcached Emerges

LiveJournal started on a single server, grew to two, then four. The scaling up process was extremely painful but proved to be an excellent learning experience. Brad learned many lessons the hard way and Memcached emerged as the proud cure for LiveJournal's growing pains. Memcached handles 90-93% of the hits that would otherwise end up in the database.

LiveJournal’s web nodes are all diskless, netbooting off a common, redundant NFS boot image. Brad likes the solution for two reasons: it's faster and it's cheaper.

Today LiveJournal uses ten different database clusters, each with two or more machines. Nine clusters are used solely for maintaining user data, while the tenth is dedicated to non-user data and a table that maps each user to the appropriate cluster. LiveJournal handles 40-50 million dynamic hits per day with 700-800 hits per second during peak hours.

How Memcached Works

Now that we've established the background of Memcached and talked about the business problem involved, let's get a little more technical and talk about how Memcached actually works.

Memcached was developed with the specific underlying assumptions that networks are fast, memory is cheap, and memory storage should be spread out across multiple machines to compensate for unreliability in a single server. A global hash table could then manage a cache that multiple web processes can simultaneously access each seeing the changes made by the other and reacting appropriately.

So how does Memcached work? There are two important pieces: server and client.

Server Instances

In general terms a bunch of Memcached server instances are running throughout a network wherever free memory is available. Each instance of a Memcached server listens on a specified port and IP address. Specific levels of memory on each machine are allocated and the application utilizing Memcached can then use all the spare memory dedicated to it on the network.

It is also beneficial to use multiple instances of Memcached running on the same machine in cases where the total memory of a machine is greater than the amount that the kernel makes available to a single process. Multiple instances are easily handled by listening on different ports.

Client Read Process

On the client side, you use Memcached just like you would use any other cache. When the application determines what object is needed, it runs the key (object id or similar identifier) through a hashing algorithm and then checks the hash against Memcached to see if the object is available. If the object is available, it is returned to the web process. If the object is not available, it is fetched from the database, and then put in the cache.

Client Write Process

First the object is fetched from the database or the cache, updated, saved to the database, and then saved to the cache. Again we run into tradeoffs here. Memcached does not support transactions, so it makes sense to pull from the database, update the object, save to the database, and save to the cache. This maintains the integrity of the data, but also involves an extra hit on the database.

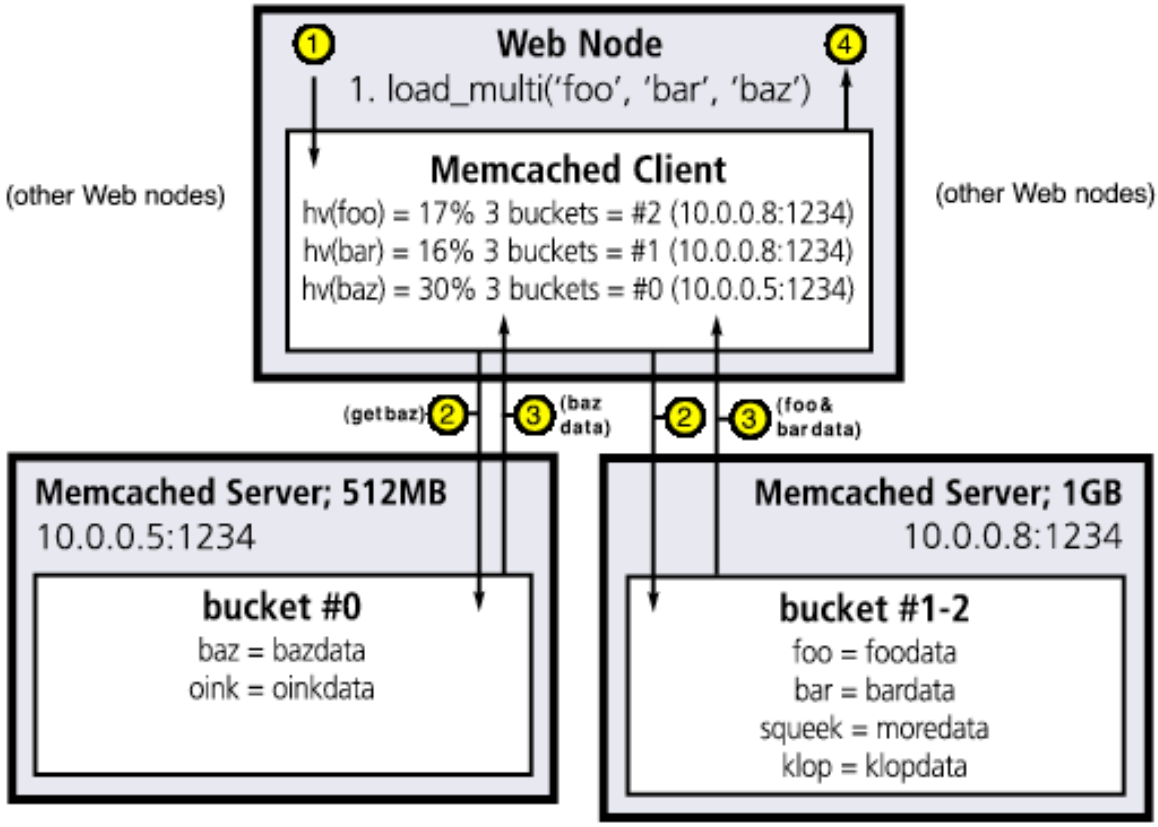
Hashing and Keys

These server instances store the data that various web servers and processes will later request in a client-server type relationship. The web application maintains a hash table, which stores information about what information is stored in which Memcached instance. Object requests are then sent to the appropriate server instance in the distributed cache before the web application accesses a database in order to fulfill requests.

Memcached uses a set of keys to look up cached results in the hash table. Storage keys are evenly spread out across the multiple nodes running Memcached. An application uses the hash table to determine which server to send the request to.

As the web server processes data and discovers a need for an object that it doesn't have, it first checks to see whether a Memcached value with a corresponding unique key exists. If it exists the corresponding data will be sent back. If it did not exist in memory the application would then be directed to read from a database as usual.

Internally, this hashing and key system is implemented at two layers. The first layer tells the application which server the necessary key is stored on, and the second layer is the actual serialized object that needs to be retrieved. This two-layer system guarantees that only the server with the appropriate data will be asked to retrieve it. One request will only ever be sent to one server to get a response. This is illustrated graphically below.



Step 1: The application requests keys foo, bar and baz. Key hash values are calculated determining which Memcached server should receive the requests.
Step 2: The Memcached client sends parallel requests to all relevant Memcached servers.
Step 3: The Memcached servers send responses to the client.
Step 4: The Memcached client aggregates responses for the application.

Source: http://www.linuxjournal.com/article/7451

So what happens when there is a key collision? Well the simple answer is that data gets lost. Either you'll end up with an object that isn't what you thought it was, or you overwrite something in the cache with something completely different. Good design of an application can completely prevent this from happening however. Each individual application is responsible to define and use it's own key-creating structure, so if you start running into key collisions, you need to take a closer look at your code.

Independence

Each Memcached server instance operates completely independent of the others. If a server fails inside the Memcached farm, the remaining active servers function as normal. Requests to the failed server will of course not be resolved; however clients can be configured to route around failed machines. This is an important point to bring out though. If a Memcached server instance fails, all data contained within that instance will be lost.

Expiration

Memcached implements the least recently used (LRU) caching principle discarding the least recently used items first based on available memory space. Thus by design, the most often used objects will remain in the cache and the ones that aren't used as often will be phased out as new objects enter the cache.

Performance

Empirical data on the performance of Memcached is incredibly difficult to come by. APIs implemented in different languages have completely different performance specs. There is a similar difficulty on the server end of things depending on the skill of the programmer that ported the source code into a usable binary on all the different flavors of *nix.

To illustrate the difficulty, consider two different Ruby implementations of the Memcached client API (yes, Ruby is a great language, but it's the only one with two different clients). The following tests were performed on the TownX Blog using Memcached to cache sessions for a web application.

Library	Total time for 2000 requests (seconds)	Mean time per request (ms)	Requests per second
Ruby-MemCache	36.310	18.155	55.08
memcache-client	31.787	15.894	62.92

You can see the variability even with the same server and simply using two different client APIs.

So that being said, we've uncovered one set of benchmark statistics on the MySQL Performance Blog. Just understand that the results should be taken with a grain of salt and an understanding of the variations and unpredictability involved.

In the next set of benchmarks, the PHP Associative Array is provided as a baseline along with the Table Selects. The Table Selects was a very simple lookup on a primary key, so it's typical of the absolute peak of performance you can expect from MySQL. The cost of a normal database query would be much larger and thus, slower.

In our Ruby example we were caching user sessions, so in order to test, we made a series of HTTP GETs from the front end of the web application. In the next example, we're just pounding the database with SELECTs and avoiding all the overhead of HTTP in general. We explain this to show that comparison between the two tests is unwise.

The following tests were performed on a 2 GHz Sempron CPU using MySQL 4.1 and PHP 5. The APC Cache is a PHP only cache that works well on a single machine, but does not scale across multiple machines.

Cache Type	Gets/Second
PHP Associative Array	365,000
APC Cache	98,000
File Cache	27,000
MySQL Query Cache (Unix Socket)	13,500
Memcached (TCP/IP)	12,200
MySQL Query Cache (TCP/IP)	9,900
Table Select (Unix Socket)	7,400
Table Select (TCP/IP)	5,100

So we can see that under ideal conditions the MySQL query cache using a Unix socket performs better than Memcached in terms of straight gets/second, but you may or may not be to describe your environment as ideal.

In addition, items returned from Memcached are usable objects while those returned from the MySQL cache must then be constructed into objects. As queries get more complex, we can see how Memcached's advantage over other caching solutions begins to grow.

While it won't ever catch up to the speed of a PHP Associative Array or an APC Cache, Memcached has other benefits that may make it a better fit for the job at hand.

Implementation

So now that we've seen how Memcached works, we need to take a look at how to use it. A successful implementation of Memcached consists of at least one running server instance and a client that has been integrated into the desired application.

Server

The Memcached server instance is by far the simplest piece of implementation. Because Memcached is open source, it is available for free download in any number of places. If compiling from source seems a little advanced, then you can use the very available versions that are packaged for nearly every flavor of Unix and Linux. It has even been ported over to the Windows platform. This means that in most circumstances, you can install the Memcached server with one simple command or a download and double-click.

Once your server is installed, firing up an instance is a matter of simplicity. There are no configuration files, just a command at the command line with a few parameters to specify how you want it to run. Common runtime configuration options are port number, bind address, amount of memory, and thread type (Memcached can run in verbose mode in the foreground or daemon mode in the background).

Client

Installation of the client is a little more difficult than the server. Integrating Memcached

into an application is probably the hardest part of all. The reason that client installation is difficult is because it will vary greatly depending on which programming language you are using. The client and server are independent of each other, so a client on any architecture can access and use a server on any other architecture. This provides good abstraction, but also introduces an element of complexity.

In most cases, client installation consists of downloading libraries and storing them in a particular place in your file system. Then your application will have to load the appropriate libraries, create the needed objects, and call the necessary functions. APIs are available in compiled languages such as Java, C, and C# but have also been created for the popular scripting languages Ruby, PHP, Python, and Perl.

Integration

Finally we'll look at integrating Memcached into an application (new or existing). This will vary greatly based on language and architecture as well, but there are a few basic components that you'll have to address regardless.

First, you'll face the decision of what you want to cache. As mentioned earlier Memcached stores objects, but what kinds of objects would you like to store. Maybe you'd like to execute queries on your database, construct objects from those queries and store the objects in your cache. Maybe you'd rather store the entire user session in the cache. Maybe you even want to build a whole page and cache that.

Second, you'll have to tell your application how you want it to use Memcached. You can specify options for compression, access-level, debugging, and namespace. A namespace is a way of defining which application an object belongs to. That means that you could potentially have any number of different applications using the same cache and Memcached would keep track of everything for you and only give you access to objects in the appropriate namespace. An analogy would be that several different applications get their own room inside of one big Memcached house.

Finally, you'll have to tell your application where all your Memcached servers are. This instruction is simply an array of IP Address and port numbers to the locations of all usable Memcached servers. Because Memcached uses the IP protocol for transportation of information, you can even use Memcached servers that are not on the same network, though there are security issues with going remote and obvious speed advantages to keeping things local.

Alternatives and Complements

Memcached is not the cure-all for all memory caching. There are a number of traditional alternatives that may prove to be sufficient enough in order to optimize your application and database resource usage. Memcached is not meant for small implementations, though you could use it for that if desired. In some cases, Memcached does not function as a replacement at all, but rather a complement to the existing architecture.

Another thing to keep in mind as we head into this section, Memcached's target is the high-end performance applications on multiple servers. That is where you'll begin to realize major benefits.

Traditional Shared Memory

Many applications will use what is known as shared memory where applications will cache items within web processes. Some web servers will spawn multiple threads to handle all the processing requests. Depending on the server software being used, these separate threads may or may not have access to shared memory. Still, shared memory may be fine on a single machine, but what happens when you begin to scale up and out?

Once you introduce multiple web servers behind a load balancer, Memcached really starts to shine. If you're using a load balancer, each request is evaluated by the load balancer and then sent to the appropriate server based on the balancing algorithm. It is conceivable that a single user will make multiple requests that would include some of the same data. If the load balancer distributes these requests to different servers, then each web process will likely cache the data. The individual servers will not have the data in their cache, so they'll have to query the database again. The result is duplicate caching, extra database hits, and wasted memory.

Let's go back to two of the basic principles of Memcached to find the advantage. Networks are fast and memory is cheap. Memcached creates a single global cache available to all web processes across their standard blazing fast network connections. Individual web processes and threads do not maintain their own caches in this case, but rather use the single global cache. When one web process puts something in the cache, it is still available for all others to access.

MySQL Query Caching

MySQL query caching is a viable solution in particular in situations where the data that is cached is not too large and is rarely updated. If the majority of your database queries are simply SELECT statements and not INSERTs and UPDATEs then Memcached will yield few, if any, improvement. In other situations however, Memcached can provide dramatic improvements.

If your database is highly dynamic, query caching is not a viable solution. A cached query must be re-cached each time a table changes so that new data will not be excluded when the cached result is returned. On very dynamic databases this becomes very cumbersome and may actually slow the process down as the MySQL process is constantly trying to reconstruct the cache.

Another advantage is that the Memcached cache is potentially limitless as new server instances can be added at any time. MySQL caches are limited in size and for potentially large databases this may be unacceptable.

Memcached is an object cache rather than a query cache. A typical response from a MySQL SELECT statement is given in the form of a result set. The application must receive this result set and transform it into objects for the application to use. Anyone who's written a data access layer in Java knows how complicated this can get. However, this means that even though the result set is cached and can be instantly returned, the result set must still be translated into usable objects. Memcached can provide significant benefits in environments where this object construction is complex caching the already created objects, thus eliminating the need to construct them again.

Finally, there are speed advantages to handling the bulk of your data needs from memory rather than disk. Memcached speeds up your application because requests are often returned from a large memory base without having to read a database. Through reference counts internal objects can be in multiple states to multiple clients at the same

time. By using a hash table and key combination to allocate memory to various server instances, virtual memory does not get externally fragmented.

Database Replication

Database replication and/or distribution are often used to spread out the number of hits to a database. Database management systems have been designed to make this process as painless as possible, but designing and implementing a replication system can still be a painful and potentially dangerous process.

In this case, Memcached is not really an alternative but rather a complement that allows you to scale much larger before needing to add more database servers and clusters.

Typically distribution and replication schemes only spread out the reads to a database, while all the writes still have to go to a single server. Memcached cannot help to alleviate the number of writes to a database. What Memcached will do is reduce the read load to a mere fraction freeing up the database servers to perform writes and maintain ACID properties (we'll talk about ACID properties in the next section). In Brad Fitzpatrick's words, it allows you to "get more bang for your buck."

General Advantages

While the alternatives section focuses on specific advantages that Memcached can offer over some alternatives. This next section focuses on general benefits that cannot necessarily be tied to a specific alternative.

Blocking vs. Non-blocking

Every DBMS is responsible for maintaining the integrity of the data and the related transactions. How well the DBMS accomplishes this goal is measured in terms of four properties. These properties are atomicity, consistency, isolation, and durability, collectively referred to as the ACID properties.

By way of explanation, atomicity refers to the ability to guarantee a transaction occurs; that either all pieces commit or none do. Consistency refers to the database being in a legal state before and after the transaction, that a transaction did not violate any integrity rules on the database. Isolation means that each transaction is completely separate from another. Finally, durability means that once a user has been notified of success the transaction will persist and not be undone.

There is a lot of overhead in a DBMS that guarantees that the database successfully performs these functions. By their very nature, these functions conflict with the goal of optimization. The traditional way of being ACID compliant is through the use of locking. Tables are locked when in use which creates a queue of transactions waiting to access the database. Implementing ACID properties is a complex task especially for large databases and requires a lot of resources.

Memcached can compensate for insufficient ACID properties and it never blocks. It has a non-blocking network I/O meaning that a thread invoking an I/O function, like a request for a Memcached value, does not have to wait on any previous operation before executing. The request will simply succeed or fail immediately just as if it were the only request being made.

Objects in the cache have multiple versions and are all reference counted. As a result, there is never a queue to process transactions because clients cannot block other clients from requesting or receiving data while the multiple versions preserve transaction integrity.

Cross Platform

Memcached is cross-platform in the same way that we typically speak of web services. An application being served from Mac OS X can use Memcached server instances running Windows, Unix, and Linux servers with complete transparency.

Cross-DBMS

Memcached was not developed specific to any particular database platform so whether you use MySQL, PostgreSQL, Oracle, MSSQL or any other database on the backend of your application Memcached can integrate into your application and help optimize your architecture.

Cheap

Memcached is free. It is open source software so it is free to download. There are no licensing fees or restrictions so a network administrator can run as many daemons as desired. Having a rather impressive client list and a collection of geeks behind it means that development is regular and there are good contributions from the user community.

In addition to being open source software and freely available, let's take another look at one of the foundations of Memcached. Memory is cheap. Where other alternatives require expensive investments to scale out, Memcached only requires more RAM. There is no limit to the number of server instances that can be created and one hash table manages all the separate machines as one global cache.

General Disadvantages

Although, many have been able to optimize database performance through the use of Memcached it may not be the best solution for every situation.

Size Requirement

Memcached becomes more valuable the larger the DBMS and the more reads and writes required in a particular application. Sites like LiveJournal, Wikipedia, Slashdot, and Digg have realized significant improvements using Memcached. That being said, not everyone will see dramatic increases from using Memcached. There is a certain amount of size and dynamics required before the benefits really begin to manifest.

For smaller applications, simple query caching may be sufficient. Of course you could use Memcached, but if your application lives on a single server and your processors and disks are handling the load without difficulty, there is little sense in using it. Once you start scaling up, it can provide significant benefits, but the age-old adage still applies to IT: If it's not broke, don't fix it.

Documentation

Partially due to its open-source status there is not a whole lot of documentation or support for Memcached. While it is a conceptually simple technology the support is all on a voluntary basis. While the user community is friendly and helpful, if you run into real trouble you don't have a lot of options.

Memcached is not mature enough to have an established user support group nor are there any forums dedicated for Memcached related posts. The technology has not been highly publicized and is far from being widely used resulting in only a few real experts on implementation and execution.

Volatility

Memcached is volatile. If a Memcached server instance crashes, any data stored within the session is gone. Of course your database is still up to date, but calls to that particular instance that should be returning data will instead return nothing. This could have potentially damaging effects on your application. Once the instance comes back up, it will begin to function normally again, but all objects stored there are irretrievably lost.

If your cache was storing sessions for example, the application will continue to function, but the sessions stored in that instance will be completely broken. Even if it comes immediately back up, the end user will have to start their session over again. For some sites, particularly retail sites with session store shopping carts etc, this may be an unacceptable risk.

Security

There is no authentication built into Memcached. The only built-in security measures relate to the naming conventions of the keys used. If someone knows the proper keys to request, then the data in the cache is readily available to anyone who wants it. Each key has a namespace that is unique to an application, and each key is produced by an application in whatever hashing algorithm the designer chooses to use, which could be difficult to duplicate, but not impossible. Networks can be designed in secure ways that would help protect data as well, but this may also be an unacceptable limitation in some scenarios.

Our Experience Using Memcached

Server

Implementing Memcached is actually a very straightforward operation. We were quite nervous when we saw that it was distributed as source code which would require compiling, but were then pleased to learn that it has been packaged up by most Linux distributions and is a very simple to install. MacPorts has even made it available for the Mac users of the world through their packaging system. For us on an Ubuntu server, the command to install was simply

```
sudo apt-get install memcached
```

On the server side, Memcached is simply a single process that runs in the background (windows users can even set it up to run as a service). There are no configuration files, just a few command line parameters that you pass in when you fire up the daemon. The

options that we used allowed us to daemonize the process (run it in the background), specify the amount of memory, define the IP address to bind, and choose the port. For us, the command was:

```
memcached -d -m 64 -l 127.0.0.1 -p 11211
```

We only dedicated 64 Megs of ram because we're not trying to support a huge dynamic database. We're just trying to learn about the issues faced in implementation. As mentioned above, Memcached is very scalable. You can simply add another Memcached instance and further expand the size of the global cache. Your web server(s) only needs to know IP addresses and port numbers for where to reach the cache machines. The client API provides for dynamically adding server:port combinations to your array of servers, so you can scale on the fly without bringing your application down.

So now that we've got a running Memcached server, let's look at the implementation on the client side. Client-side setup was more difficult. We decided to cache user sessions into our cache.

Client

We used Ruby on Rails for our client, specifically used the Memcache-client, which is simpler and faster than Ruby-Memcache.

```
sudo gem install memcache-client
```

Integration

In a matter of minutes, we had Memcached caching our sessions. We simply installed the Memcache-client 1.3.0 with RubyGems, and tell our application how to use the cache, and where the available caches were. It looks like this:

```
memcache_options = {
   :compression => true,
   :debug => false,
   :namespace => "mem-#{RAILS_ENV}",
   :readonly => false,
   :urlencode => false
}

memcache_servers = [ 'benrobb.hopto.org:11211' ]

CACHE = MemCache.new(memcache_options)
CACHE.servers = memcache_servers
ActionController::Base.session_options[:cache] = CACHE
```

The compression option tells Memcached to compress the data if it can. The debug option is obvious. The namespace gives our different Ruby environments unique keys to the cache so that our development and test data don't tread all over the production data. The readonly option is turned off so that we can write to our cache in addition to reading from it. The memcache_servers array is expandable at will as we add more Memcache servers.

So Memcache is now storing our user sessions for us. Our application knows where the cache is, so if we want to store the results of complex queries in the cache as well, that is an available option for us. To wrap up, implementation was fairly simple and painless.

Conclusion

The adoption of Memcached is growing, development is advancing, and support is improving. The re is an impressive list of users realizing extreme value from integrating Memcached into their architectures. That being said, there should be a serious conversation about advantages and disadvantages before anyone goes into this large-scale.

The Wikipedias and Slashdots of the world experience thousands of changes every minute. Each individual needs to research and weigh the alternatives to evaluate whether Memcached is a good fit for their business needs.

Specifically, what kinds of objects would you cache? How will this help? How fast is your network? How much spare memory is floating around your network? How big is your application? How much expertise have you got on your web team? How much spare memory can be allocated?

Questions such as these will help companies make a viable business case for or against implementing Memcached. Lastly, Memcached rocks and this report is very typical of an 'A++' student.

References

Memcached: A distributed memory object caching system: http://www.danga.com/memcached/

Distributed Caching with Memcached: http://www.linuxjournal.com/article/7451

Wikipedia: Memcached: http://en.wikipedia.org/wiki/Memcached

Memcached: http://www.lifewiki.net/sixapart/memcached

What I've Learned About Ruby and Memcache: http://townx.org/blog/elliot/ruby_tuesday_what_ive_learned_about_rails_and_memcache

Memcached Basics for Rails: http://nubyonrails.com/articles/2006/08/17/memcached-basics-for-rails http://nubyonrails.com/articles/2007/01/22/memcached-basics-for-rails-part-ii

Cache Performance Comparison: http://www.mysqlperformanceblog.com/2006/08/09/cache-performance-comparison/

MySQL Cluster (NDB) or Vertical Replication: http://www.feedblog.org/2006/10/mysql_cluster_n.html

MySQL Cluster vs Memcached: http://lists.danga.com/pipermail/memcached/2006-October/002889.html

Structure of the Memcached Key: http://lists.danga.com/pipermail/memcached/2006-November/003048.html

Optimising and Using Multiple Memcache Servers in a Pool: http://lists.danga.com/pipermail/memcached/2007-January/003479.html

Hierarchy of Caches for High Performance & High Capacity Memcached: http://www.planetmysql.org/entries/4538