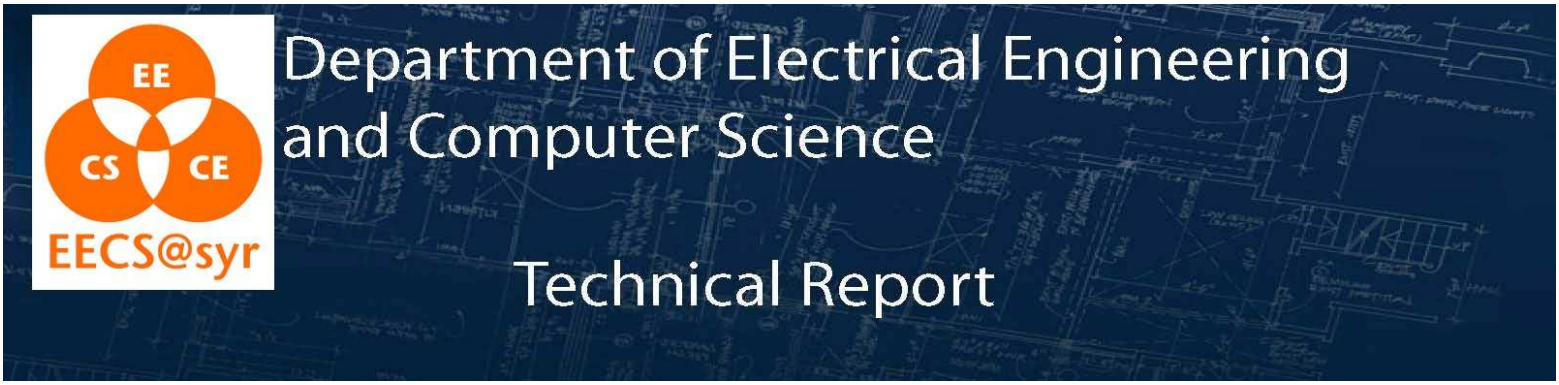
9-15-2011

Exploring non-typical memcache architectures for decreased latency and distributed network usage.

Paul G. Talaga
Syracuse University

Steve J. Chapin
Syracuse University



SYR-EECS-2011-10 September 15, 2011

Exploring Non-typical Memcache Architectures for Decreased Latency and Distributed Network Usage

Paul G. Talaga pgtalaga@syr.edu
Steve J. Chapin chapin@syr.edu

ABSTRACT: Memcache is a distributed in-memory data store designed to reduce database load for web applications by caching frequently used data across multiple machines. While memcache already offers excellent performance, we explore how data-locality can increase performance under certain environments and workloads. We build an analytical model, then compare typical configurations to a handful of proposed schemes which promise to reduce latency and/or reduce core network traffic. An example web application, MediaWiki, is profiled and shown to benefit from alternate Memcache configurations when used in a large web farm. We develop metrics to predict any web application's performance under our schemes.

KEYWORDS: Memcache, Web Technologies, Caching, Data Locality

Syracuse University - Department of EECS,
4-206 CST, Syracuse, NY 13244
(P) 315.443.2652 (F) 315.443.2583
http://eecs.syr.edu

Exploring Non-typical Memcache Architectures for Decreased Latency and Distributed Network Usage

Paul G. Talaga
Syracuse University
pgtalaga@syr.edu

Steve J. Chapin
Syracuse University
chapin@syr.edu

September 15, 2011

Abstract

Memcache is a distributed in-memory data store designed to reduce database load for web applications by caching frequently used data across multiple machines. While memcache already offers excellent performance, we explore how data-locality can increase performance under certain environments and workloads. We build an analytical model, then compare typical configurations to a handful of proposed schemes which promise to reduce latency and/or reduce core network traffic. An example web application, MediaWiki, is profiled and shown to benefit from alternate Memcache configurations when used in a large web farm. We develop metrics to predict any web application's performance under our schemes.

1 Introduction

Originally developed at Danga Interactive for LiveJournal, the Memcache system is designed to reduce database load and speed page construction by providing a scalable key/value caching layer available to all webservers. The system consists of memcache servers (memcached instances) for data storage, and client libraries which provide a storage API to the web application over a network or local file socket connection. No durability guarantees are made, thus memcache is best used to cache regenerable content or in front of a reliable storage system. Data is stored and retrieved via keys, that uniquely determines the storage location of the data via a hash function over the server list. High scalability and speed are achieved with this scheme as a key's location (data location) can easily be computed locally. Complex hashing functions allow addition and removal of memcache servers without severely affecting the location of the already stored data.

This work explores different data placement techniques on top of Memcache to reduce latency and network usage. We present six architecture variants, evaluate their performance in a derived usage model, and offer insights into applications which could benefit from these variants.

Key to this work is the fact that moving data any distance takes time and network resources. While modern LAN networks in a data center environment allow easy and fast transmission, locating the data very close to where it is used can lower latency and overall network usage resulting in better overall system performance [12]. The benefits are amplified for larger or multiple data centers. Just as CPU caches speed computation, we apply the same ideas to data caches in the webfarm.

The rest of the paper consists of: Section 2 reviews the current Memcache server and client architecture, with example code. Section 3 sets up an analytical model for latency, cache usage, network utilization, and configuration variables for a memcache instance in a datacenter which we use later to compare different configurations. A specific usage example using MediaWiki [7], used by Wikipedia, is also given. Standard memcache architectures, obvious extensions, as well as our proposed architectures inspired by multi-cpu caches [4] is outlined in Section 4. In Section 5 we estimate performance for each architecture under our model looking at average latency, network usage, and relative available space. We further discuss the results and future work in Section 6, related work in Section 7, and conclude in Section 8. Appendix A contains all formula used to estimate latency.

2 Memcache Details

As previously mentioned, Memcache is built using a client-server architecture. Clients, in the form of an instance of a web application, set or request data from a memcache server. A Memcache server consist of a daemon listening on a network interface for TCP client connections, UDP messages, or alternatively through a file socket for local clients. Daemons do not communicate with each other, but rather perform the requested memcache commands from a client. An expiration time can be set for data to automatically remove stale data. If the allocated memory is consumed, data is generally evicted in a least-recently-used manner. For speed, data is only stored in memory (RAM) rather than permanent media.

The location of a memcache daemon can vary. For small deployments a single daemon may exist on the webserver itself. Larger multi-webserver deployments generally use dedicated machines to run multiple memcache daemons outfitted with large amounts of memory (RAM). This facilitates management and allows webservers to use as much memory as possible for web script processing.

Clients access data via a library API. As of Memcached version 1.4.7 (8/16/2011), 15 commands are supported, categorized into storage, retrieval, and status commands. Various API's written in many languages communicate via these commands, but not all support every command. Below is an example of a PHP snippet which uses two of the most popular commands, *set* and *get* to quickly return the number of users.

```
function get_num_users(){
    $num = memcached_get("num_users");
    if($num !== FALSE){
     return $num;
    }else{
        $num = get_num_users_from_databaase();
        memcached_set("num_users", $num, 60);
    }
    return $num;
}
```

Rather than query the database on every function call, this caches the result in memcache. A timeout is set for 60 seconds so at most once a second the database would be queried, with the function returning a number at most a minute stale. In this case no relation to the session is made, but session identifying information could be integrated into the key and thus store session data. Using memcache to store session information combined with sticky load balancing (requests from the same session end up in the same server or rack) can allow data to be stored close to where it will be used for greater speed, which we explore later. Alternatively, as in the above example, some data may be used by all clients. Rather than each webserver requesting commonly used data over the network, it may make sense for a local copy to be stored, negating the need for repeated network access.

Some memcache clients, including PHP Memcache, include bulk *set* and *get* commands (multi-get) which sets or gets data to memcache servers in parallel, greatly increasing performance. Rather than loop over a set of keys, this allows an array of keys to be sent to the command. We exploit this feature as much as possible in our implementations as should any memcache enabled web application.

3 Analytic Model

To analyze the strengths and weaknesses of different configurations we devise an analytical model to measure performance. For a realistic evaluation, we profile a popular open-source web application (MediaWiki [7]) for a sample of memcache usage. We then use the model and usage sample to analyze the predicted performance of different configurations.

3.1 Assumed Network Topology

Network topology greatly influences the performance of any large interconnected system. We assume a network topology consisting of multiple web, database, and memcache servers organized into a physical

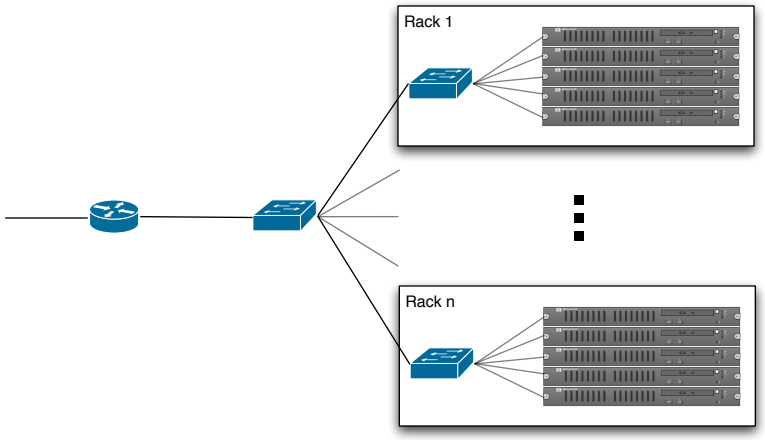


Figure 1: Assumed Web Farm Topology

hierarchical star topology. See Figure 1 for an example.

Web servers, running application code, are grouped conceptually into racks. These can either be physical racks of machines, separate webserver threads in a single machine, or an entire data center. Instances within the same rack can communicate quickly through one network switch, or within the same machine. The choice of what a rack describes depends on the overall size of the webfarm. Whatever the granularity, communication is faster intra-rack over iner-rack.

Racks are connected through a backbone link. Thus, for one webserver in a rack to talk to another in a different rack at least 3 switches must be traversed.

Groups of racks can be aggregated together into floors, rooms, or datacenters. To generalize the different possible configurations further, we assume network latency is linearly related to the switch (or other device) count a signal must travel through. Thus, in our rack topology the estimated latency (round-trip time) from one rack to another is l_3 because three switches are traversed, where $l_3 = 3 * 2 * switch$ for some $switch$ delay value.

Similarly l_2 represents a request traversing 2 switches, such as from a rack to a node on the backbone and back. l_1 represents traversing a single switch, but we'll use the term l_{local} to represent this closest possible network distance. In some cases we'll use $l_{\text{localhost}}$ to represent l_0, where no physical network layer is reached. This metric differs from hop count as we count every device a packet must pass through, rather than only counting routers.

3.2 Network & System Constants

We define a set of variables and constants which are used to compare memcache configurations.

Note we include average memcache command and key length in $size_{object}$ and $size_{message}$ to simplify our calculations. This estimates the on-wire data size. Using these values for storage is not exact, but useful due to extra meta-data in the memcache server per data item.

$l_{\text{localhost}}$ – Average latency to localhost's memcached(l_0)
l_{local} – Average latency to a nearby node through one device(l_1)
l_n – Average latency traversing n devices (round-trip)
where $l_n < l_m | n < m$
r – Number of racks
k – Replication value
ps – Proportion of keys which are only used for a specific session $[0 - 1]$
rw – Percent of commands which are reads
$switch$ – Delay per switch traversed (ms)
$size_{object}$ – Average size of stored object (bytes)
$size_{message}$ – Size of message used in Snoop and Dir (bytes)

3.3 Assumptions

To simplify our model we assume the following traits:

1. Computation is instantaneous - Memcache lookup and code evaluate do not take any time. While false in practice, computers keep getting faster (or more cores), and the speed of signal transmission will forever remain fixed. Network latency is about equivalent to current compute time, with Facebook claiming a 173μs average latency for Memcache handling UDP requests, well within the $30 - 384\mu$s latency for ethernet. [10] [3] [6] Thus, we consider this a constant cost.

2. Linear Network Latency - Network latency is linearly related to network device traversal count. [6]

3. Sticky Sessions - A web request in the same session can be routed to a specific rack or webserver.

We define a usage scenerio as the relative frequency of memcache commands an application uses. Some commands are further broken down into a hit or miss as these may incur different costs totaling 17 different request types. In our case we will use the open source application MediaWiki, which has built-in support for Memcache, for an example usage scenerio. Obtaining the frequencies via memcache instrumentation is discussed next.

3.4 MemcacheTach

Predicting memcache usage is not easy. User demand, web application code changes, network usage and design all can influence the performance of a memcache system. Instrumentation of a running system is therefore needed, but few tools exist. The memcache server itself is capable of returning the keys stored, number of total hits, misses, and their sizes. Unfortunately this is not enough information to answer the following questions: What keys are used the most/least? How many different clients use the same keys? How many memcache requests belong to a single HTTP request? How much time is spent waiting for a memcache request from the client perspective? Which memcache servers are responding slowly? Which webservers are experiencing slow network performance?

To answer these and other questions we developed MemcacheTach, a memcache client wrapper for PHP which intercepts and logs all requests. While currently analyzed after-the-fact, the data could be streamed and analyzed live to give insight into memcache's performance and allow live tuning. Analysis provides values for ps, switch delay, stored data size, the ratio of the 17 memcache request types, as well as other

switch (ms)	3	ps	0.9
Avg. data size (bytes)	92	rw	0.6
Mem. requests per page	6.3		
Set	14%	Add hit	0%
Add miss	0%	Replace hit	0%
Replace miss	0%	CAS hit match	0%
CAS hit missmatch	0%	CAS miss	0%
Delete hit	0%	Delete miss	1%
Inc hit	1%	Inc miss	24%
Dec hit	0%	Dec hit	0%
Flush	0%	Get hit	45%
		Get miss	14%

Figure 2: MediaWiki usage values

useful information about a set of memcache requests. Figure 2 shows the measured values for MediaWiki over a set of three sessions of viewing and editing content totaling 121 pages.

The average page used 6.3 memcache requests and waited 25ms for memcache requests in our experiment.

Notice the only memcache commands observed were set, $increment$, $delete$, and get, a rather limited selection. Surprisingly 60% of keys were only used by one server, showing heavy use of session storage, and thus a good fit for location aware caching.

MemcacheTach is available at `http://fuzzpault.com/memcachetach` with installation and usage instructions.

4 Memcache Architectures

Here we describe the architecture and rules for configurations we analyze later using the previous network model. As mentioned in the Introduction, storage location is determined by each client. Similarly, all alternate configurations below are implemented in client code via wrappers around existing clients.

4.0.1 Standard Deployment Central - SDC

The typical deployment consists of a dedicated set of memcached servers existing on the rack backbone (l_2). Thus, all memcache requests must traverse to the memcache server(s). Data is stored in one location only, not replicated.

4.0.2 Standard Deployment Spread - SDS

This deployment places memcache servers in each webserver rack. Thus, some portion of data ($1/r$) exists close to each webserver (l_1), while the rest is farther away (l_3). Remember the key dictates the storage location, which could be in any rack, not the local.

4.0.3 Standard Deployment Replicated - SDR

To add some durability to data, we store data on more than one memcache daemon (preferably on a different machine or rack). While solutions do exist to handle duplication on the memcache server (repcached [5]), we look at the more general case where the client preforms replication on its own. This allows easy memcached updates and keeps management closer to the application. Implementation can be either through multiple storage pools or, in our case, modifying the key in a known way to choose a different server or rack. Reads have the advantage of choosing which pool/key to use and thus distributing load or reducing latency by choosing a nearby copy.

4.0.4 Snooping Inspired - Snoop

Based on multi-cpu cache snooping ideas, this architecture places memcache server(s) in each rack allowing fast local reads. Writes are local as well, but a data location note is sent to all other racks under the same key. Thus, all racks contain all keys, but data is stored only in the rack where it was written last. An update clears all notes and data with the same key. To avoid race conditions deleting all data, notes are stored first in parallel, followed by the actual data. Thus, in the worst case multiple copies could exist, rather than none. A retrieval request first tries the local server(s), either finding the data, or a note. If a note is found the remote rack is queried and the data returned. Durability and speed can be gained by probabilistically copying the data to the local server if a note was originally found, though this limits available storage space and is not analyzed later.

4.0.5 Directory Inspired - Dir

An alternate mulit-cpu caching system uses a central directory to store location information. In our case, a central memcache cluster is used to store sharing/duplication information. Each rack has its own memcache server(s) allowing local writes, but reads may require retrieval from a distant rack. A retrieval request will try the local server first, and on failure query the directory and subsequent retrieval from the remote rack. A storage request first checks the directory for information, clears the remote data if found, writes locally, and finally sends a note to the directory. The directory note can contain a primary and multiple secondary rack locations allowing duplication for durability and distributed load. We differentiate this scheme with the $Dirk$ option.

4.0.6 Write Through (l_1) - WTR

Here a small memcache server(s) exists in the same rack, with a Standard Deployment Central (SDC) farther away. Data is written locally as well as globally. A retrieval request queries the local cache, and on failure reads the SDC farther away. Due to multiple caches and no removal system it is imperative to have short expire times locally otherwise stale data could be retrieved.

4.0.7 Write Through ($l_{localhost}$) - WTL

Similar to the above WTR, but the local memcache server exists on the same webserver machine and interfaces through the faster socket interface.

5 Comparisons

We evaluate the above architecture options by estimating latency, network usage, and available storage space over each memcache command using our above described model and variables.

Web application usage heavily determines performance of any memcache system. Thus, given a usage scenario, network model, environment variables, and the above architecture policies we can estimate the average memcache latency, network usage, and available storage. This allows a level comparison between architectures and lets us explore how alternate environment variables could improve or hinder performance.

5.1 Latency

Formulas were derived which predict the estimated latency for each of the 17 request types over all architectures given the network model and environment variables.

For example the set command would have a latency of l_2 under SDC, while the SDS would be $\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$ because some portion of the keys would be local. SDR and Snoop would take l_3 because multiple sets can be issued simultaneously. Dir requires a round trip to the directory to learn about other racks which may already contain the same key. Therefore, on a set the other racks must be cleared of that key so the new data is used in the future. Thus ps of the time the key is l_{local}, and the other portion it is

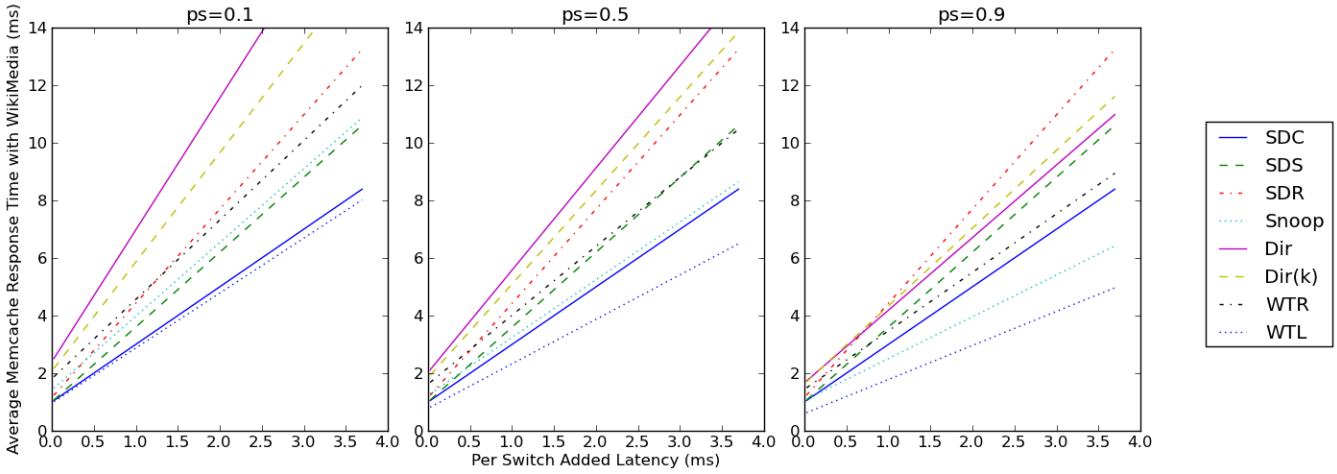


Figure 3: MediaWiki profile under different switch speeds and ps values.

l_3 in the non-duplicated case. Finally, the data can be set and a note sent to the directory. This results in a latency of $2 \times l_2 + ps \times l_{local} + (1 - ps)l_3 + l_{local}$. Write Through (both versions) can set in parallel but must wait for both commands to return, thus taking l_2 time.

Appendix A provides formulas for predicted latency of all 17 request types over all architectures. With all formula, weightings for commands used, and environment variables we can compare how each architecture performs under different parameters.

We first look at how network switch(device) speed can effect performance. Remember we assumed the number of devices linearly relates to network latency, so we vary the single device speed between 12.7μs and $1.85ms$, with an additional $1ms$ OS delay, in the Figure 3 plots. Response time measures round trip time, so a switch speed of $1ms$ would result in a $2ms$ latency. Three plots are shown with ps values of 10%, 50%, and 90% with weightings derived from our MediaWiki profile above.

As seen in Figure 3, ps drives favorable performance for many of these schemes. Next we take a closer look at how ps changes response time in Figure 4 using a fixed switch latency of $0.925ms$ and our MediaWiki usage profile.

Predictibly all 3 location averse schemes (SDC, SDS, and SDR) exhibit no change in performance as ps increases. Additionally, SDC is the best when $ps = 0$ due to all others having added overhead. As ps increases the other 5 schemes improve with Snoop and WTL performing better than SDC.

So far we've analyzed performance using MediaWiki's usage profile. Now we look at the more general case where we split the 17 possible commands into two types: read and write, where read consists of a get request hit or miss, and write is any command which changes data. MediaWiki had 60% reads, or 6 reads per 4 writes. Figure 5 varies the read/write ratio while looking at three ps values.

5.2 Network Usage

While low latency is our end goal, analyzing network usage is also important to predict hotspots or more cost-effective configurations.

Recall our assumed star network topology. Webservers are put in racks possibly including a memcache server, with each rack connected to a backbone. A central memcache server may exist on the backbone. Here we discuss the predict the total bytes transfered for the backbone switch compared to SDC. Fewer bytes transfered equates to fewer packets, meaning less usage. We ignore rack switch analysis for most schemes since lower backbone usage usually equates to lower rack switch usage.

5.2.1 SDC

In the typical situation a single (or cluster of) memcache server sits on the backbone which services all requests. All requests and data must therefore pass through the rack switch as well the backbone switch.

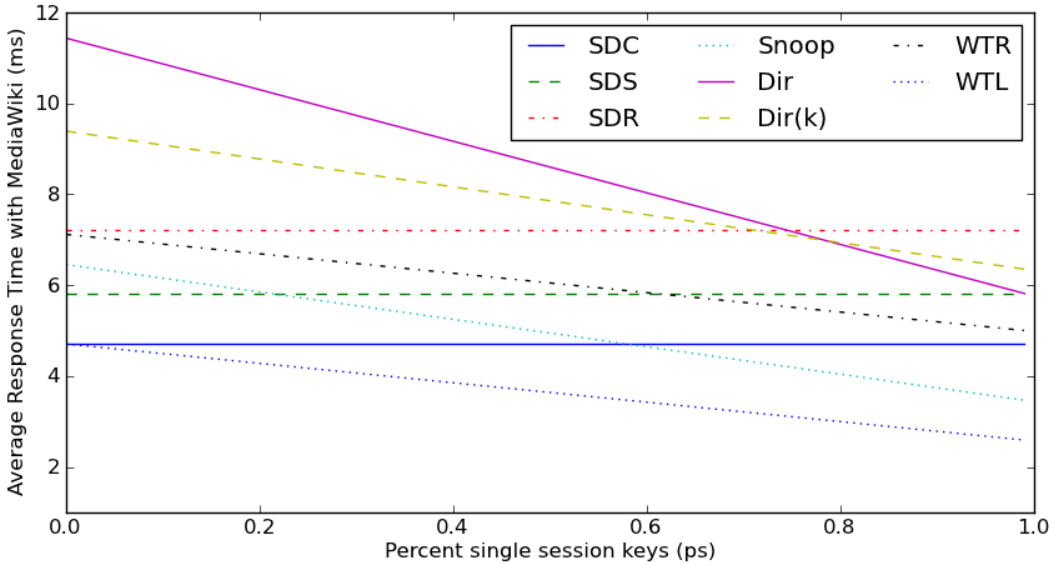


Figure 4: MediaWiki profile under different ps values.

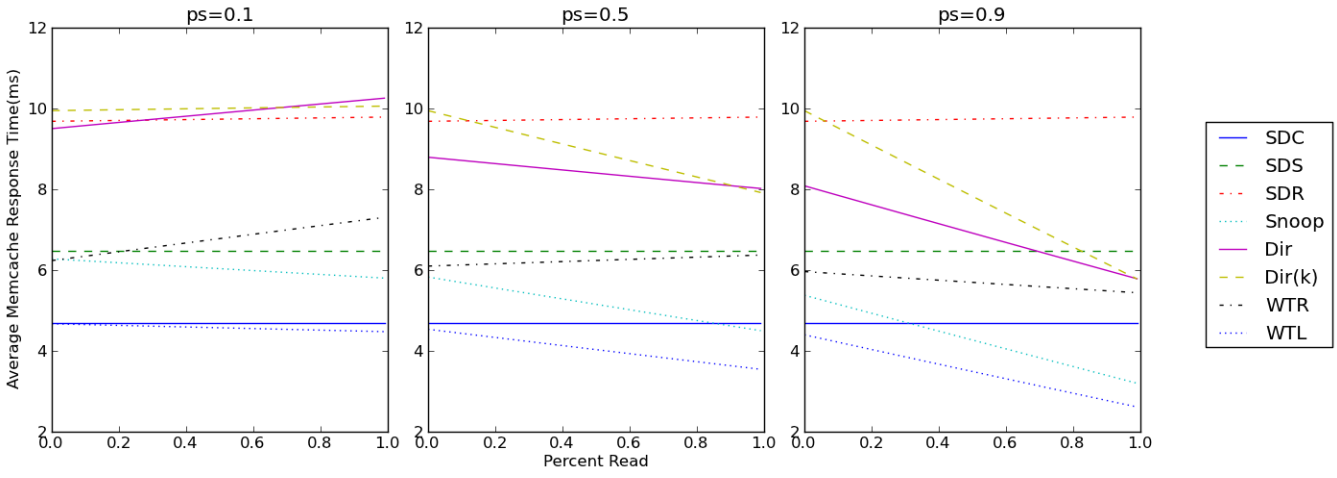


Figure 5: Varying read/write ratio and ps values.

Depending on the number of webservers, the memcache link(s) may become a bottleneck as all traffic is funneled through them.

5.2.2 SDS

Since object placement is not location aware, the majority ($\frac{r-1}{r}$) of data will still pass through the rack and backbone switch. Unfortunately each rack switch will handle twice the memcache load the majority of the time ($\frac{r-1}{r}$) as each request will leave one rack and enter another. The remaining $1/r$ will only pass through one switch.

5.2.3 SDR

Lacking a location aware policy, the majority of data ($\frac{r-1}{r}$) will pass through the rack and backbone switch. Each write will put k times (replication value) as much load on all switches. Reads are the same, though possibly local due to clients having an option where to query the data (choose least loaded or closest).

5.2.4 Snoop

This architecture is location aware and will store data close to where the the store was issued. The downside to this scheme is each set causes a store command note to be sent to all racks. Luckily a set command's data is stored in its own rack's memcache server, thus bypassing the backbone switch. Depending on the ps value of the application, reads can either be identical to SDS (all information passing through the backbone and two rack switches) when $ps = 0$, to the extreme $ps = 1$ of no read using the backbone because it can be serviced locally. To predict backbone switch usage of Snoop we develop the following ratio of total backbone traffic to that of SDC for both read and write:

$$\frac{rw \times (1 - ps) \times size_{object} + (1 - rw) \times size_{message} \times r}{size_{object}}$$

or

$$rw \times (1 - ps) + \frac{(1 - rw) \times size_{message} \times r}{size_{object}}$$

For example, in our MediaWiki scenario we calculate a 94% decrease in backbone traffic using the following parameters with Snoop over SDC:

$size_{message}$	20 (bytes)
$size_{object}$	15 (Kbytes)
ps	0.9
rw	0.6
r	10

Clearly the difference between message, object size, and number of racks is important. For a write in Snoop to have equivalent load on the backbone switch we have the following equality, $size_{message} \times r = size_{object}$. Given our values above we could have 750 racks before a write is equivalent. A small message, larger objects, an application that uses more reads, or a higher ps value will offer lower network usage in this architecture.

5.2.5 Dir

Since objects are stored close to where they were changed, the ps value greatly influences the overall network load. For a ps value of 1 no object data need cross the backbone switch, with only a single message sent to the repository per change, rather than one message per rack for the Snooping inspired. Similar to Snoop, when ps = 0 all objects traverse the backbone. On a local memcache miss the repository will be queried, adding some additional network load, though all object location messages should be less than 20 bytes and thus smaller than the average object.

The k option, where multiple object copies are stored, will have slightly less backbone for reads due to additional copies being local to some queries.

5.2.6 Write Through

Both versions, WTR & WTL, preform identically to the SDC in the worst case, or on startup, but as more local copies are stored we'd expect backbone usage to decrease for reads. Performance is quite difficult to predict due to individual key read/write ratios, local lower expiration values, and global expiration values, plus overall memcache usage. In the best case we'd expect the backbone to carry each key's change, as well as a single query for each rack to fill its local version (ps = 0), or only a single rack if ps = 1.

5.3 Available Storage Space

Each architecture exhibits different performance, but at what additional hardware cost to store the same amount of data? Here we analyze available space for each architecture.

5.3.1 SDC

This serves as our baseline metric. No extra meta-data is stored, so all available space is utilized for object storage. We ignore internal memcache efficiencies as these may change over time.

5.3.2 SDS

The only difference between this and Normal Rack is the location of memcache servers, thus no extra information is stored. 100% efficiencey compared to SDC.

5.3.3 SDR

Duplicating data reduces available space. With a specific k value (replication value), we arrive at a utilization value of $\frac{1}{k} * 100\%$ efficiency compared to SDC. For example with $k = 3$, you would need three times the the storage space to store the same amount of data as a SDC.

5.3.4 Snoop

Snoop's storage of messages reduces the efficiency of this architecture. For values of $size_{object}$, $size_{message}$, and r we arrive at the following storage efficiency formula:

$$\frac{size_{object}}{size_{message} \times (r-1) + size_{object}} = \text{Snoop efficiency compared to SDC}$$

For example in our MediaWiki case where $size_{object} = 15$Kb, $size_{message} = 20$b, and $r = 10$, we arrive at an efficiency of 98%, barely enough to warrant additional hardware over using SDC. Again, a larger average data size compared to message size will increase efficiency.

5.3.5 Directory Inspired

Rather than store many messages through out the system like Snoop, Directory Inspired only stores a single note in a central location, increasing storage efficiency further.

$$\frac{size_{object}}{size_{message} + size_{object}} = \text{Dir efficiency compared to SDC}$$

and if some k duplication is used,

$$\frac{size_{object}}{size_{message} + k \times size_{object}} = \text{Dir k efficiency compared to SDC}$$

With the same MediaWiki 10 rack example a non-duplicated directory configuration has an efficiency of 99%, while simple duplication ($k = 2$), gives a value of 49%.

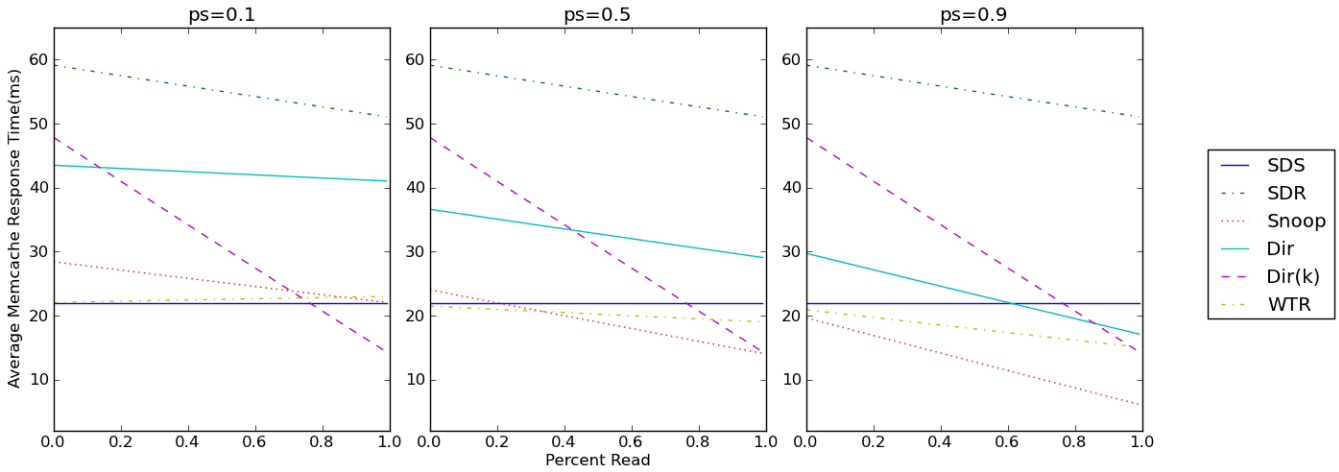


Figure 6: Varying read/write ratio and ps values with an East and West coast DC

5.3.6 Write Through

Many factors influence the performance of write-through (WTR & WTL), such as rw, expiration values, and individual key read/write rations, as well as the fact that local storage will never be used for primary object storage and thus is only useful for reducing network load and latency. Thus, the efficiency is given by the simple formula:

$$\frac{\text{SDC Storage Space}}{\text{SDC Storage Space} + \text{Rack Storage Space}}$$

6 Discussion & Future Work

6.1 Race Conditions

As with any distributed storage system race conditions must be addressed. Luckily for our architectures, and memcache in general, errors are allowed to occur. Data is not guaranteed to be permanent, or correct, and as a result great speed can be gained. Applications using memcache must be aware of its limitations and deal with failures when they occur, usually by requesting the data from a reliable source and re-setting the object in memcache. As mentioned before, implementation changes can be used to minimize conflicts in specific architectures.

6.2 Multi-Datacenter Case

Thus far we have assumed a Memcache installation within the same datacenter, with appropriate estimates on latency. In general running a standard memcache cluster spanning datacenters is not recommended due to high (relative) latencies and expensive bandwidth. The location-averse architectures, SDC, SDS, and partly SDR would not be good choices for this reason. We can apply our same analysis to the multi-datacenter situation by viewing the datacenter as a rack, with a high l_3, though realistic, value for intra-datacenter latency. SDC is no longer possible with its l_2 latency, with SDS taking its place as the typical off-the-shelf architecture. If we assume a l_3 value of 40ms, a best case CA to NY latency, with $l_1 = 4$ms inside the datacenter, we arrive at Figure 6 giving average latencies between the different architectures over different ps and read/write ratios. For Dir's directory we assume it spans both datacenters like SDS.

Here the difference between locality aware and averse is more pronounced. Snoop, Dir(k), and WTR are able to outperform SDC for most cases except high write volume and low ps values. SDR preforms poorly due to consistency checks between the two copies. Interestingly as more datacenters are added SDC becomes worse due to a higher proportion of data being farther away while the location aware architectures can keep it close.

Regardless of the lower latencies over a standard deployment in the multi-datacenter case, caching may not be beneficial over many isolated caches.

6.3 Selective Replication

Replication of a relational database can increase performance by distributing reads. Unfortunately entire tables must be replicated, possibly including seldom used data. In a key/value system such as Memcache replication can offer speed benefits as we saw in SDR. We looked at the static case where all data is replicated, but why not selectively replicate frequently used data so we don't waste space? As mentioned in the Snoop description, data could be probabilistically copied locally. Thus, frequently used but infrequently changed data would be replicated allowing fast local reads. Unused Memcache memory is a waste, so by changing the probability of replication on the fly memory could be used more optimally. We intend to investigate this in further work.

6.4 Object Expiration

In Memcache, if the allocated memory is consumed objects are generally removed in a least-recently-used manner. In a standard deployment this works well, but in our case where meta information is separate for the data a possibility exists where meta expiration may cause a miss followed by a set on an already stressed node. Adjusting expire times, or modifying Memcache to return the set expire time could alleviate this issue by probabilistically returning a miss when the expire time is near zero. This would have the added benefit of placing the subsequent set on a heavily accessed node and place the data closer to where it is used the most.

6.5 User Space Caching

As mentioned in Section 2, the Memcache server is a separate process even when used on the same machine as the client. Inter-process communication is therefore necessary, either through the loopback or file socket interfaces. Either way, transportation costs are incurred. By moving the memcache server, or emulating it, within the same process space as the client the transportation costs could be eliminated. Object serialization overhead may also be reduced. We plan on investigating this option further.

6.6 Overflow

Memcache's standard deployments (SDC,SDS, SDR) all behave well when they reach memory saturation by removing objects evenly in a least-recently-used manner due to the hashing strategy. Our location aware architectures (Snoop & Dir) may over-stress some racks by storing all objects on them while nearby racks remain empty. While possible, this situation should not become a problem as long as web requests are evenly spread across all web servers. Thus, the cache as a whole should not be filled from a single client, but rather from each rack.

7 Related Work

Memcache is part of a larger key/value 'NoSQL' data movement which provides enhanced performance over relational databases by relaxing ACID guarantees. One way key/value systems can scale well is by using a hashing system based on the key to uniquely identify an object's storage location. To achieve locale-aware caching we must use some other key/location mapping system. Here we discuss similar systems to Memcache and concentrate on those which has some locality component.

The Hadoop Distributed File System [2] is designed to manage distributed applications over extremely large data sets while using commodity hardware. They employ a rack-aware placement policy able to place data replicas to improve data reliability, availability, and network bandwidth utilization over random placement. Reads can read from a close copy rather than a remote one. Additionally they identify bandwidth limitations between racks, with intra-rack faster, supporting our architectures. Their mantra of "Moving

Computation is Cheaper than Moving Data" works for large data sets, but in our web case we can write the data close to where it will be used instead. File meta-data is stored and managed in a central location similar to our Dir architectures.

Cassandra can also use a rack and datacenter aware replication strategy [1] for better reliability and to read locally. This is convenient when multiple datacenters are used so a read will never query a remote data center. Voldemort uses a similar system for replicas using zones [9] [12]. While the above three systems use some locality aware policy for replicas they all use a hashing strategy for primary storage, thus possibly writing the data far from where it will be used. We plan on investigating weather locality aware primary storage could be beneficial to these systems, or some mixture.

Microsoft's AppFabric provides very similar services to Memcache with in-memory serializable object storage across many computers using key/value lookup and storage [8]. No mention of key hashing is given to locate data, though they do mention a routing table to locate data similar in practice to our Snoop architecture. No mention of how this table is updated. Their routing table entries reference a specific cache instance, whereas our Snoop note refers to a hash space, or rack, possibly containing thousands of memcache instances, thereby giving more storage space and flexibility. Durability can be added by configuring backup nodes to replicate data, though unlike our architectures all reads go to a single node until it fails, unlike ours where any copy can be read.

EHCACHE Terracotta is a cache system for Java, containing a 'BigMemory' module which permits serializable objects to be stored in memory over many servers. Java's garbage collection (GC) can become a bottleneck for large in-application caching, thus a non-garbage collected self-managed cache system can use large amounts of memory while ignoring typical Java GC issues. Essentially BigMemory implements a key/value store using key hashing for Java objects similar to the Memcache client and server. Unlike memcache, it allows configurations other than the standard memcache case. For example it allows a read and write through mode, backed by a durable key/value storage system, thereby removing all cache decisions from the application code. Replication is done by default (2 copies) for availability, with more available. Additionally, it can delay writing to the durable store for faster apparent write speed. Similar to our work, a small local cache can be maintained, with many policies available. [11]

8 Conclusion

We've seen that when a web application has a high ps value, many reads per write, or slow networking equipment, a location-aware Memcache architectures can lower latency and network usage without significantly reducing available space. Our example application, MediaWiki, showed that a significant amount of keys and data is session specific and thus can exploit such architectures. Our tool, MemcacheTach, can be used to measure a web application's detailed Memcache usage to estimate performance with alternate architectures under a simple model. As web applications grow, and their user base becomes more geographically diverse, the need for systems which can keep latencies low for all users is needed. Our proposed reformulation of multi-processor cache systems for Memcache show that better performance can be currently gained within the web datacenter under certain circumstances, with further gains found for more geographically distributed data centers over current techniques.

References

[1] N. Bailey. Frontpage - cassandra wiki, `http://wiki.apache.org/cassandra/`, 2011.

[2] D. Borthakur. Hdfs architecture guide, `http://hadoop.apache.org/common/docs/current/hdfs_design.html#Data+Blocks`.

[3] S. Cheshire. It's the latency, stupid, `http://www.stuartcheshire.org/rants/Latency.html`.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[5] K. Inc. repcached - add data replication feature to memcached, `http://repcached.lab.klab.org/`.

[6] R. Inc. Latency on a switched ethernet network, http://www.ruggedcom.com/pdfs/application_notes/latency_on_a_switched_ethernet_network.pdf.

[7] mediawiki. Mediawiki, http://www.mediawiki.org/wiki/MediaWiki.

[8] A. N. Nithya Sampathkumar, Muralidhar Krishnaprasad. Introduction to caching with windows server appfabric, http://msdn.microsoft.com/en-us/library/cc645013(en-us).aspx, 2009.

[9] rsumbaly. Voldemort topology awareness capability, https://github.com/voldemort/voldemort/wiki/Topology-awareness-capability, 2011.

[10] P. Saab. Scaling memcached at facebook, http://www.facebook.com/note.php?note_id=39391378919, 2008.

[11] I. Terracotta. Ehcache documentation cache-topologies, http://ehcache.org/documentation/distributed_caching.html.

[12] P. Voldemort. Project voldemort, http://project-voldemort.com/design.php, 2011.

A Latency Costs over each Memcache command for each Architecture

See Section 3.2 for variable definitions.

	Normal	Norm Spread	Replicate	Snoop	Dir	Dir k	WT
Set	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	l_3	l_3	$2 \times l_2 + l_3$	$2 \times l_2 + l_3 + l_{local}$	$max(l_{local}, l_2)$
Add hit	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	$2 \times l_3$	l_{local}	l_2	l_2	$ps \times l_{local} + (1-ps) \times (l_2 + l_{local})$
Add miss	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	$2 \times l_3$	$l_{local} + l_3$	$l_2 + l_{local}$	$l_2 + l_3$	$l_{local} + l_2$
Replace hit	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	$2 \times l_3$	$l_{local} + ps \times l_{local} + (1 - ps) \times l_3$	$l_2 + ps \times l_{local} + (1 - ps) \times l_3$	$l_2 + l_3$	$l_{local} + l_2$
Replace miss	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	$2 \times l_3$	l_{local}	l_2	l_2	$l_{local} + l_2$
CAS hit match	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	$2 \times l_3$	$l_{local} + ps \times l_{local} + (1 - ps) \times l_3$	$2 \times l_2 + l_3$	$2 \times l_2 + l_3$	$l_{local} + l_2$
CAS hit no	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	l_3	$l_{local} + ps \times l_{local} + (1 - ps) \times l_3$	l_2	l_2	l_2
CAS miss	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	l_3	l_{local}	l_2	l_2	l_2
Delete hit	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	l_3	l_3	$2 \times l_2 + ps \times l_{local} + (1 - ps) l_3$	$2 \times l_2 + l_3$	$max(l_{local}, l_2)$
Delete miss	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	l_3	l_3	l_2	l_2	$max(l_{local}, l_2)$
Inc hit	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	$2 \times l_3$	$l_{local} + ps \times l_{local} + (1 - ps) \times l_3$	$ps \times l_{local} + (1 - ps)(l_2 + l_3)$	$l_2 + l_3$	$l_{local} + l_2$
Inc miss	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	l_3	l_{local}	l_2	l_2	l_2
Dec hit	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	$2 \times l_3$	l_3	$ps \times l_{local} + (1 - ps)(l_2 + l_3)$	$l_2 + l_3$	$l_{local} + l_2$
Dec miss	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	l_3	l_{local}	$l_{local} + l_2$	$l_{local} + l_2$	l_2
Flush	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	l_3	l_3	l_3	l_3	l_2
Get hit	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	$\frac{n}{r} \times l_{local} + \frac{r-n}{r} \times l_3$	$l_{local} + (1 - ps) \times l_3$	$l_{local} + (1 - ps)(l_2 + l_3)$	$l_{local} + (1 - ps)(l_2 + l_3)$	$ps \times l_{local} + (1-ps) \times (l_2 + l_{local})$
Get miss	l_2	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3$	$k \times l_3$	l_{local}	$l_{local} + l_2$	$l_{local} + l_2$	$l_{local} + l_2$