EXPLORING NON-TYPICAL MEMCACHE ARCHITECTURES FOR DECREASED LATENCY AND DISTRIBUTED NETWORK USAGE

Paul G. Talaga and Steve J. Chapin
Syracuse University, Syracuse, NY, U.S.A.

Keywords: Memcache, Latency, Network Utilization, Caching, Web-farm.

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. In a distributed web serving environment applications rely on many network services to complete each request. While faster processors have lowered computation time and available network bandwidth has increased, signal propagation delay is a constant and will become a larger proportion of latency in the future. We explore how data-locality with Memcache can be exploited to reduce latency and minimize core network traffic. A model is developed to predict how alternate Memcache configurations would perform for specific applications followed by an evaluation using the MediaWiki open-source web application in a miniature web farm setting. Our results verified our model and we observed a 66% reduction in core network traffic and a 23% reduction in Memcache response time under certain network conditions.

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 web servers. 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. 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.

As web farms and cloud services grow and use faster processors, the relative delay from network access will increase as signal propagation is physically limited. Developing methods for measuring and addressing network latencies is necessary to continue to provide rich web experiences with fast, low latency interactions.

As an example, consider a webserver and a Memcache server on opposite ends of a datacenter the size of a football field. The speed of light limits the fastest round-trip time in fiber to about 1μs. Current network hardware claim a round-trip time (RTT) for this situation between 22μs and 3.7ms(RuggedCom, 2011), more than an order of magnitude slower than the physical minimum. Assuming Memcache is using TCP with optimal ACKs or UDP, 100 Memcache requests would take between 2.2ms and 370ms, ignoring all processing time. Compare this to the 200ms recommended page response time for web content and Memcache latency becomes significant in some situations. This is backed by multiple measurements of latencies in both Google App Engine and Amazon EC2 showing between 300μs and 2ms RTT between two instances(Newman, 2011; Cloudkick, 2011).

Related to latency is network load. More network utilization will translate into more latency, leading to costly network hardware to keep utilization low.(RuggedCom, 2011). Reducing network load, especially the highly utilized links, will keep latency low.

This work explores how data locality can be exploited to benefit Memcache to reduce latency and core network utilization. While modern LAN networks in a data center environment allow easy and fast transmission, locating the data close to where it is used can lower latency and distribute network usage resulting in better system performance(Voldemort, 2011). When looking at inter-datacenter communi-

36 Talaga P. and Chapin S..
EXPLORING NON-TYPICAL MEMCACHE ARCHITECTURES FOR DECREASED LATENCY AND DISTRIBUTED NETWORK USAGE.
DOI: 10.5220/0003933200360046
In *Proceedings of the 8th International Conference on Web Information Systems and Technologies* (WEBIST-2012), pages 36-46
ISBN: 978-989-8565-08-2
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

cation latency becomes even more pronounced.

We present five architecture variants, two novel to this area based on prior multi-processor caching schemes, two natural extensions, and the last the typical Memcache architecture. In addition to showing performance for a single application and hardware configuration, we develop a network and usage model able to predict the performance of all variants under different configurations. This allows mathematical derivation of best and worst case situations, as well as the ability to predict performance.

Our contributions are outlined below:

1. Develop a model for predicting web caching performance
2. Present a tool for gathering detailed Memcache usage statistics
3. Reformulate two multi-CPU caching methods into the web-caching area and compare them to three other typical architectures
4. Evaluate MediaWiki performance under various Memcache architectures

The rest of the paper consists of: Section 2 reviews background material, such as Memcache operation, assumed web farm network topology, and a definition of constants we use to characterize a web application. Next, Section 3 describes our developed tool for logging and analyzing Memcache's performance which is used to build an application's usage profile. Section 4 describes the five Memcache architectures along with estimation formula for network usage and storage efficiency. Section 5 analyzes the five architectures with respect to latency, with analysis of best and worst case scenarios. Our experimental results are shown in Section 6 in which we show how well our model and estimation formula match actual performance.A discussion of relevant issues is given in Section 7, followed by related work in Section 8, and our conclusion in Section 9.

2 BACKGROUND

2.1 Memcache

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 consists of a daemon listening on a network interface for TCP client connections, UDP messages, or alternatively through a file socket. Daemons do not communicate with each other, but rather perform the requested Memcache commands from a client. An expiration time can be set to 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 configured for optimal performance (large RAM, slow processor, small disk). 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 1.4.13 (2/2/2012), 16 commands are supported, categorized into storage, retrieval, and status commands. Various APIs 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 and retrieve the last login time.

```
function get_num_users(){
    $num = memcached_get("num_users");
    if($num === FALSE){
        $num = get_num_users_from_database();
        memcached_set("num_users", $num, 60);
    }
    return $num;
}
function last_login($user_id){
    $date = memcached_get('last_login' . $user_id);
    if($date === FALSE){
        $date = get_last_login($user_id);
        memcached_set('last_login' . $user_id, $date, 0);
    }
    return $date;
}
```

Rather than query the database on every function call, these functions cache the result in Memcache. In `get_num_users`, a cache timeout is set for 60 seconds which causes the cached value to be invalidated 60 seconds after the *set*, causing a subsequent database query. Thus, at most once a minute the database would be queried, with the function returning a number at most a minute stale. To cache session information, the `last_login` function stores the time of last login by including the `$user_id` in the key. On logout (or timeout) another function clears the cached data. During an active session the `last_login` function will only access data the current session stored and will be of no use to other users. Thus, if sticky load balancing (requests from the same session are routed to the same server or rack) is used the data could be stored locally to speed access and reduce central network load. Alternatively, as in `get_num_users`, 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, avoiding the need for repeated network access. Caching data locally, when possible, is the basis for the proposed architectures.

2.2 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 hierarchical star topology.

Web servers, running application code, are grouped 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 over at most two network segments, or within the same machine. The choice of what a rack describes depends on the overall size of the web farm. Whatever the granularity, communication is faster intra-rack than inter-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 4 network segments connected with 3 switches must be traversed.

To generalize the different possible configurations, we assume network latency is linearly related to the switch count a signal must travel through, or any other devices connecting segments. Thus, in our rack topology the estimated RTT from one rack to another is l_3 because three switches are traversed, where $l_3 = 3*2*switch + base$ for some *switch* and *base* delay values.

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.

2.3 Network & System Constants

We define a set of variables and constants which define a particular instance of an application. This includes datacenter size, switch performance, Memcache usage statistics, and application use parameters.

Set Hit	Set Miss	Add Hit	Add Miss
Replace Hit	Replace Miss	Delete Hit	Delete Miss
Increment Hit	Increment Miss	Decrement Hit	Decrement Miss
Get Hit	Get Miss	App/Prepend Hit	App/Prepend Miss
CAS1[1]	CAS2[1]	CAS3[1]	Flush

Figure 1: 20 monitored Memcache commands.

To simplify our model we assume the following traits:

1. Linear Network Latency - Network latency is linearly related to network device traversal count.(RuggedCom, 2011)
2. Sticky Sessions - A web request in the same session can be routed to a specific rack or webserver.

$l_{\text{localhost}}$	– Average RTT to localhost's memcached (ms) (l_0)
l_{local}	– Average RTT to a nearby node through one device (ms) (l_1)
l_n	– Average RTT traversing n devices (round-trip) (ms)
	where $l_n < l_m \mid n < m$
r	– Number of racks
k	– Replication value
ps	– Proportion of session specific keys on a page $[0-1]$
rw_{cmd}	– Percent of commands which are reads [0-1]
rw_{net}	– Percent of network traffic which are reads [0-1]
$switch$	– Delay per switch traversed (ms)
$base$	– Constant OS and Memcache processing overhead (ms)
$size_{object}$	– Average size of typical on-wire object (bytes)
$size_{message}$	– Size of message used in Snoop and Dir (bytes)

The *ps* value is central to our approaches. It gives a measure of how many Memcache requests are for data pertinent only to that session per returned page. If only session information is stored in Memcache a *ps* value of 1 would be the result for the application. If half the keys are for session and half not, then $ps = 0.5$. For example if an application used the above `get_num_users()` and `last_login($id)` once per page then for 100 user sessions Memcache would store 101 data items, of which 100 are session specific. In our page oriented view of *ps* only half the Memcache requests per page could take advantage of locality caching, thus producing a *ps* value of 0.5.

For a more detailed application model we define a usage scenario as the above values plus 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 latency costs. Figure 1 lists the 20 different request types.

3 MEMCACHETACH

Predicting Memcache usage is not easy. User demand, network usage and design all can influence the

switch (ms)	0.11	*ps*	0.56
base (ms)	4.4	rw_{cmd}	0.51
Avg. data size (Kbytes)	3.3	Mem. requests per page	16.7
Avg. net size (bytes)	869		
Set hit	24%	Set miss	0%
Add miss	0%	Replace hit	0%
Replace miss	0%	CAS1	0%
CAS2	0%	CAS3	0%
Delete hit	2%	Delete miss	0%
Inc hit	0%	Inc miss	21%
Dec hit	0%	Dec miss	0%
App/Prepend hit	0%	App/Prepend miss	0%
Flush	0%	Get hit	44%
Add hit	0%	Get miss	7%

Figure 2: MediaWiki usage values (full caching).

State	Avg page generation time (ms)	std.dev	samples (pages)
Off	1067	926	13,800
Logging	1103	876	13,800

Figure 3: MemcacheTach overhead.

performance of a Memcache system. Instrumentation of a running system is therefore needed. 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 important questions: What keys are used the most/least? How many 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?

To answer these and other questions we developed MemcacheTach, a Memcache client wrapper which intercepts and logs all requests. While currently analyzed after-the-fact, the log data could be streamed and analyzed live to give insight into Memcache's performance and allow live tuning. Analysis provides values for l_n, ps, rw_{cmd}, rw_{net}, $switch$, $base$, $size_{object}$, plus the ratio of the 20 Memcache request types, as well as other useful information about a set of Memcache requests. Figure 2 shows the measured values for MediaWiki from a single run in our mock datacenter with full caching enabled of 100 users requesting 96 pages each. See Section 6 for further run details.

The average page used 16.7 Memcache requests, waited 46ms for Memcache requests, and took 1067ms to render a complete page.

56% of keys used per page were used by a single webserver, showing good use of session storage, and thus a good candidate for location aware caching.

As implemented, MemcacheTech is written in PHP, not compiled as a module, and writes uncompressed log data. Thus, performance could improve with further development. Additionally, every Memcache request issues a logfile write for reliability. Two performance values are given in Figure 3. Off did not use MemcacheTech, while *Logging* saved data on each Memcache call. See Section 6 for implementation details. On average MemcacheTech had a statistical significant overhead of 36ms.

MemcacheTach is available at http://fuzzpault.com/memcache.

4 MEMCACHE ARCHITECTURES

Here we describe and compare Memcache architectures currently in use, two natural extensions, and our two proposed versions. All configurations are implemented on the client via wrappers around existing Memcache clients, thus requiring no change to the Memcache server.

Estimation formula for network usage of the central switch and space efficiency are given using the variables defined in Section 2.3. An in-depth discussion of latency is given in Section 5.

4.1 Standard Deployment Central - SDC

The typical deployment consists of a dedicated set of memcached servers existing on the backbone (l_2). Thus, all Memcache requests must traverse to the Memcache server(s) typically over multiple network devices. Data is stored in one location, not replicated.

This forms the standard for network usage as all information passes through the central switch:

Network Usage: 100%

All available Memcache space is used for object storage:

Space Efficiency: 100%

4.2 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 that the key dictates the storage location, which could be in any rack, not the local. This architecture requires no code changes compared to SDC, but rather a change in Memcache server placement.

With some portion of the data local, the central

[1]CAS - Compare-and-Swap
CAS1 = Key exists, correct CAS value.
CAS2 = Key exists, wrong CAS value.
CAS3 = Key does not exist.

switch will experience less traffic:

Network Usage: $\frac{r-1}{r}\%$

All available space is used for object storage:

Space Efficiency: 100%

4.3 Standard Deployment Replicated - SDR

To add durability to data, we store k copies of the data on different Memcache daemons, preferably on a different machine or rack. While solutions do exist to duplicate the server (repcached(KLab, 2011)), we duplicate on the client and use locality to read from the closest resource possible. This can be implemented either through multiple storage pools or, in our case, modifying the key in a known way to choose a different server or rack. A write must be applied to all replicas, but a read contacts the closest replica first, reducing latency and core network load.

Reading locally can lower central switch usage over pure duplication:

Network Usage: $rw_{net} \times (1 - \frac{k}{r}) + (1 - rw_{net}) \times (k - \frac{k}{r})\%$

The replication value lowers space efficiency:

Space Efficiency: $1/k\%$

4.4 Snooping Inspired - Snoop

Based on multi-CPU cache snooping ideas, this architecture places Memcache server(s) in each rack allowing fast local reads(Hennessy and Patterson, 2006; Li and Hudak, 1989). 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. This scheme is analogous to a local-write protocol using forwarding pointers(Tanenbaum and Steen, 2001). An update overwrites all notes and data with the same key. To avoid race conditions deleting 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, either finding the data, a note, or nothing. If a note is found the remote rack is queried and the data returned. If nothing is found then the data does not exist.

The broadcast nature of a set could be more efficiently sent if UDP was used with network broadcast or multicast. Shared memory systems have investigated using a broadcast medium, though none in the web arena(Tanenbaum et al., 1994).

Based on the metric ps, the proportion of keys used during one HTTP request which are session specific, and the message size $size_{message}$, we have the following estimation for central switch traffic:

Network Usage: $rw_{net} \times (1 - ps) + \frac{(1-rw_{net}) \times size_{message} \times r}{size_{object}}\%$

Storage efficiency depends on the size of the messages compared to the average object size:

Space Efficiency: $\frac{size_{object}}{size_{message} \times (r-1) + size_{object}}\%$

4.5 Directory Inspired - Dir

An alternate multi-CPU caching system uses a central directory to store location information(Hennessy and Patterson, 2006; Li and Hudak, 1989). In our case, a central Memcache cluster is used to store sharing 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.

Rather than many notes being sent per write as with Snoop, Dir is able to operate with two requests, one to retrieve the current note, and the second to *set* the new one, no matter how many racks are used. This allows Dir to stress the central switch the least.

Network Usage: $rw_{net} \times (1 - ps) \times \frac{size_{message} + size_{object}}{size_{object}} + \frac{(1-rw_{net}) \times size_{message} \times 2}{size_{object}}\%$

Likewise with Snoop, message size dictates storage efficiency:

Space Efficiency: $rw_{net} \times (1 - ps) \times \frac{size_{message} + size_{object}}{size_{object}} + \frac{(1-rw_{net}) \times size_{message} \times 3}{size_{object}}\%$

5 LATENCY ESTIMATION

We evaluate the above architecture options by estimating latency over each Memcache command using variables defined in Section 2.3. We use individual Memcache commands divided into a hit or miss for a more accurate latency estimation as covered in Section 2.3.

Formulas were derived which predict the estimated latency for each of the 20 request types over all architectures assuming the star network model and environment variables. All 85 formula are viewable with detailed descriptions at http://fuzzpault.com/memcache.

As an example, the *set* command would have a latency of l_2 under SDC, while using SDS a *set* 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 would take l_3 because multiple sets can be issued simultaneously. Snoop would need $l_{local} + l_3$ with the data being sent to the local and messages sent to all other racks.

Dir requires a round trip to the directory to learn about other racks which may already contain the same key. Thus the formula for a *set* using Dir is more complex: $2 \times l_2 + ps \times l_{local} + (1 - ps)l_3 + l_{local}$

Using a set of variables, here from a run of MediaWiki(mediawiki, 2011), we can vary parameters to gain an understanding of the performance space under different environments.

We first look at how network switch 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 1.85*ms*, with an additional 4.4*ms* OS delay, in the Figure 4 plots (at end of document). Latency measures round trip time, so our X axis varies from 0.025ms to 3.7ms. Three plots are shown with *ps* values of 10%, 50%, and 90% with weightings derived from our MediaWiki profile.

As seen in Figure 4, *ps* compresses the plots vertically, showing improved performance for location-aware schemes using higher *ps* values. When ps=0.9 and a switch latency of 0.3ms SDS and Snoop are equivalent, with Snoop preforming better as switch latency increases further.

Next we take a closer look at how *ps* changes response time in Figure 5 using a fixed switch latency of 1.0*ms* and our MediaWiki usage profile.

Predictably all 3 location-averse schemes (SDC, SDS, and SDR) exhibit no change in performance as *ps* increases. As *ps* increases Snoop and Dir improve with Snoop eventually overtaking SDC when ps=0.86.

So far we've analyzed performance using MediaWiki's usage profile. Now we look at the more general case where we split the 20 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 51% reads when fully caching, or about one read per write. Figure 6 varies the read/write ratio while looking at three *ps* values.

With high read/write ratios Snoop is able to outperform SDC, here when switch=1.0ms at $rw = 0.75$.

These plots show when *ps* is near one and slow switches are used, Snoop is able to outperform all other configurations. In some situations, like session storage ($ps = 1$) across a large or heavily loaded datacenter, Snoop may make larger gains. From an estimated latency standpoint Dir does not preform well, though as we'll see in the next section its low network usage can overcome this.

6 EXPERIMENTAL RESULTS

To validate our model and performance estimation formula, we implemented our alternate Memcache schemes and ran a real-world web application, MediaWiki(mediawiki, 2011), with real hardware and simulated user traffic. Three MediaWiki configurations were used:

1. Full - All caching options were enabled and set to use Memcache.
2. Limited - Message and Parser caches were disabled, with all other caches using Memcache.
3. Session - Only session data was stored in Memcache.

The simulated traffic consisted of 100 users registering for an account, creating 20 pages each with text and links to other pages, browsing 20 random pages on the site, and finally logging out. Traffic was generated with jMeter 2.5 generating 9600 pages per run. The page request rate was tuned to stress Memcache the most, keeping all webservers busy, resulting in less-than optimal average page generation times. A run consisted of a specific MediaWiki configuration with a Memcache configuration.

The mock datacenter serving the content consisted of 23 Dell Poweredge 350 servers running CentOS 5.3, Apache 2.2.3 with PHP 5.3, APC 3.1, PECL Memcache 3.0, 800MHz processors, 1GB RAM, partitioned into 4 racks of 5 servers each. The remaining 3 servers were used for running the HAProxy load balancer, acting as a central Memcache server, and a MySQL server respectively. Four servers in each rack produced web pages, with the remaining acting as the rack's Memcache server.

To measure Memcache network traffic accurately the secondary network card in each server was placed in separate subnet for Memcache traffic only. This subnet was joined by one FastEthernet switch per rack, with each rack connected to a managed FastEthernet (10/100 Mb/s) central switch. Thus, we could measure intra-rack Memcache traffic using SNMP isolated from all other traffic. To explore how our configurations behaved under a more utilized network we reran all experiments with the central switch set to Ethernet (10 Mb/s) speed for Memcache traffic.

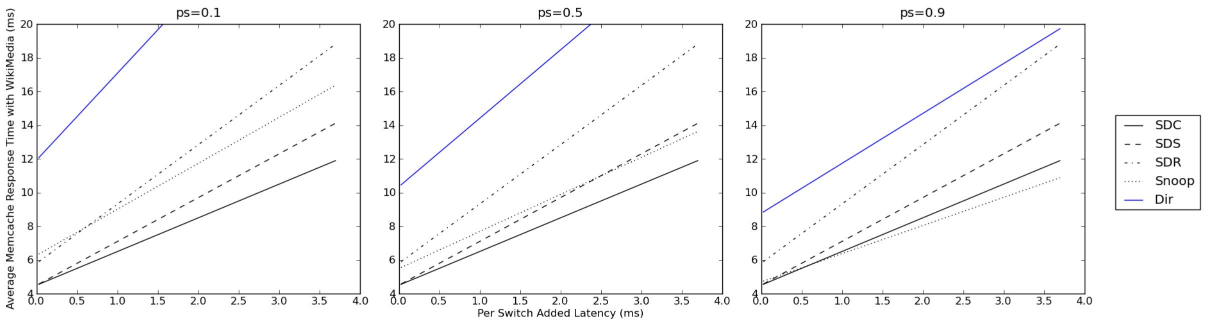

Figure 4: MediaWiki profile under different switch speeds and *ps* values.

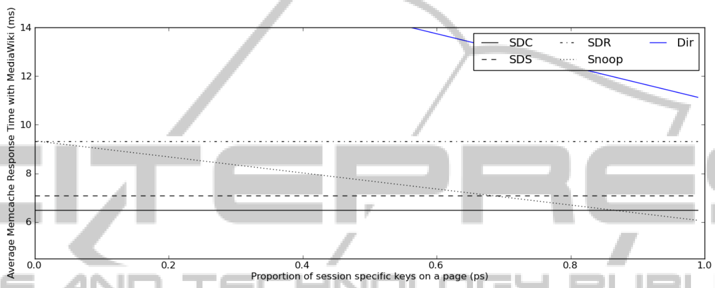

Figure 5: MediaWiki profile under different *ps* values.

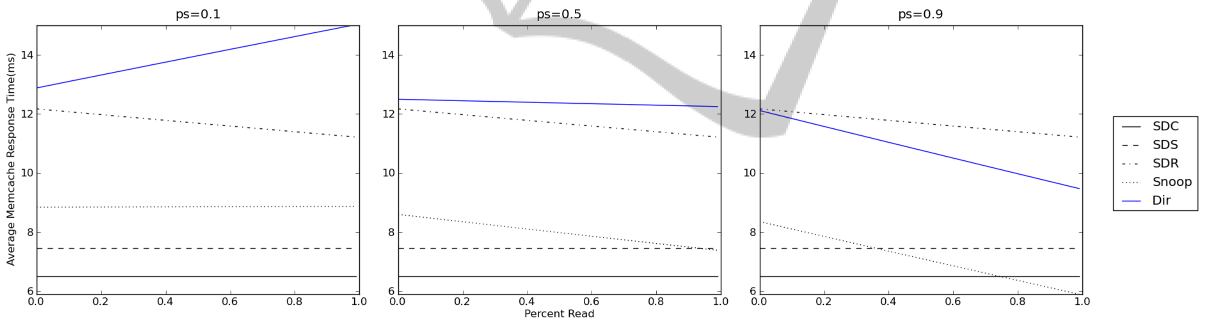

Figure 6: Varying read/write ratio and *ps* values.

Using MemcacheTach we measured MediaWiki in all configurations with results presented in Figure 7. Only non-zero Memcache commands are listed for brevity.

Parameter	Full	Limited	Session
Set	24%	23%	50%
Delete hit	2%	3%	0%
Inc miss	22%	24%	0%
Get hit	44%	42%	50%
Get miss	8%	8%	0%
ps	.56	.59	1
rw_{cmd}	.51	.49	.5
rw_{net}	.61	.78	.54
Avg. net size (bytes)	870	973	301

Figure 7: MediaWiki usage for each configuration.

	Switch (ms)	Base (ms)
100Mb/s	1.0	1.5
10Mb/s	1.3	1.7

Figure 8: Measured network performance.

6.1 Latency

To predict latency we require two measurements of network performance, *switch* & *base*. These were found using a SDS run and calculating the relative time difference between Memcache commands in-rack (l_{local}) and a neighboring rack (l_3). Results are given in Figure 8. Additional statistics available at http://fuzzpault.com/memcache.

The resulting predicted and observed per Memcache command latences are given in Figure 9.

In the case of fast switching SDC was the best predicted and observed performer. The location-aware

	Predicted Latency (ms)			Observed Latency (ms)		
Scheme	Full	Limited	Session	Full	Limited	Session
100Mb/s Central Switch						
SDC	3.5	3.5	3.5	3.5	3.9	3.2
SDS	4.1	4.1	4.1	3.8	4.1	3.6
Dir	7.1	7.0	7.2	5.5	5.8	5.8
Snoop	3.9	3.8	3.5	6.1	6.6	7.3
Dup	5.5	5.7	4.1	5.1	5.4	5.3
10Mb/s Central Switch						
SDC	4.3	4.3	4.3	12.1	13.9	3.5
SDS	5.0	5.0	5.0	20.1	20.6	4.6
Dir	6.5	6.6	8.8	9.3	9.9	6.3
Snoop	3.6	3.7	4.3	15.6	17.0	10.9
Dup	6.9	7.1	5.0	35.6	29.5	9.2

Figure 9: Expected and measured Memcache latency.

	Predicted Usage (%)			Observed Usage (%)		
Scheme	Full	Limited	Session	Full	Limited	Session
100Mb/s Central Switch						
SDC	100	100	100	100	100	100
SDS	80	80	80	80	81	73
Dir	38	39	30	35	34	69
Snoop	49	43	76	45	48	116
Dup	99	81	105	101	87	106
10Mb/s Central Switch						
SDC	100	100	100	100	100	100
SDS	80	80	80	83	82	81
Dir	38	39	30	37	35	71
Snoop	49	43	76	47	50	118
Dup	99	81	105	106	89	110

Figure 10: Expected and measured network load.

schemes, Dir and Snoop, both don't fit the expected values as close as the others. The *base* and *switch* values used to build the predicted latency were based on carrying full data messages whereas Dir and Snoop both used smaller messages which would take less time on average. Snoop's multi-set note sending may not be truly parallel showing a higher than expected latency.

When the central switch was slowed to 10Mb/s utilization increased and latency also increased. Here we see that Dir was able to outperform SDC in the Full and Limited caching cases due to the lower central switch utilization, as we'll see in the next section. Snoop still performed worse than expected, though still beating SDS and Dup in the Full caching case.

6.2 Network Load

Using the formula developed in Section 2.1 combined with the MediaWiki usage data we can compute the expected load on the central switch and compare it to our measured values. We used a $size_{message}$ value of 100 bytes, higher than the actual message to include IP and TCP overhead. The comparison is given in Figure 10.

Notice Dup's low network usage even though data is duplicated. This is a result of a location-aware strategy that writes to different racks and reads from the local rack if a duplicate is stored there. The low rack count, 5 in our configuration, assures that almost half the time data is local.

The actual central switch usage measurements match well with the predicted values. Note the location-aware rows. These show the largest skew due to the small message size and therefore the higher relative overhead of TCP/IP. This was validated by a packet dump during SDC/Full and the SDC/Session runs in which absolute bytes and Memcache bytes were measured. For SDC/Full, with an average network object size of 870 bytes, 86MB was transfered on the wire containing 61MB of Memcache communication, roughly a 30% overhead. SDC/Session transferred 9.8MB with 301 byte network objects, yet it contained 5.7MB of Memcache communication giving an overhead of 41%. Additional traces showed that for small messages, like the notes transferred for Dir and Snoop, 70% of the network bytes were TCP/IP overhead. This is shown by the higher than expected Session column when location-aware was used due to the smaller average object size. This shows that Memcache using TCP is not network efficient for small objects, with our location-aware schemes an excellent example of this. Future work measuring network utilization for Memcache using UDP would be a good next step, as has been investigated by Facebook(Saab, 2008).

If $size_{message}$ was 50 bytes, which may be possible using UDP, we should see Dir and Snoop use only 24% and 33% respectively as much as SDC on the central switch. Using the binary protocol may reduce message size further, showing less network usage.

6.3 Review

These results show that the model, application profile, and performance estimation formula do provide a good estimate for latency and network usage. While the actual Memcache latency values did not show an improvement over the typical configuration on our full speed hardware, they did support our model. In some cases, as shown by our slower network hardware configuration as well as described in Section 5, we'd expect locality-aware schemes to perform better than the typical. High rack densities and modern web-servers, even with modern network hardware, may increase network utilization to a point similar to our Ethernet speed runs and show increased latency under high load. Location-aware configurations lower core network utilization allowing more web and Memcache servers to run on the existing network.

Network usage proved difficult to predict due to additional TCP/IP overhead, but nonetheless the experimental data backed up the model with all architectures reducing core traffic, and the best reducing it to 34% of the typical SDC case.

7 DISCUSSION

7.1 Latency, Utilization, and Distributed Load

Through this work we assumed network latency and utilization are independent, but as we saw in the last section they are closely related. A heavily utilized shared-medium will experience higher latencies than an underutilized one. Thus, SDC, SDS, and Dup's latency when used on the slow network were much higher than predicted due to congestion. Unfortunately predicting the saturation point would require dozens of parameters such as link speeds, specific network devices, server throughput, as well as an estimation of other traffic on the network. At some point simulation and estimation outweigh actual implementation and testing.

7.2 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-agnostic 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 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 $l_1 = 5$ms inside the datacenter, we arrive at Figure 11 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 and Dir are able to outperform SDS when *ps* is above 0.5, especially for high read/write ratios. SDR preforms poorly due to consistency checks and multiple writes. Interestingly as more datacenters are added SDS becomes worse due to a higher proportion of data being farther away while the location aware architectures can keep it close when *ps* is high.

7.3 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? Snoop and Dir could be augmented to probabilistically copy data 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.

7.4 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 from data a possibility exists where meta expiration may cause orphaned data. The new Memcache command *touch*, which renews an object's expiration time, can be used to update the expiration of meta information reducing the chance of orphaned data, though the possibility does still exist. In a best-effort system such as Memcache such errors are allowed and should be handled by the client.

7.5 Overflow

The location-agnostic configurations (SDC, SDS, and SDR) all fill the available memory evenly over all servers due to the hashing of keys. Location aware configurations will not fill evenly, as is the case when some racks set more than others. In this case the data will be stored close to the sets, possibly overflowing the local Memcache server while others remain empty. Thus, it is important to evenly load all racks, or employ some balancing system.

7.6 System Management

Managing a Memcache cluster requires providing all clients a list of possible Memcache servers. Central to our location-aware strategies is some method for each Memcache client to prioritize Memcache servers

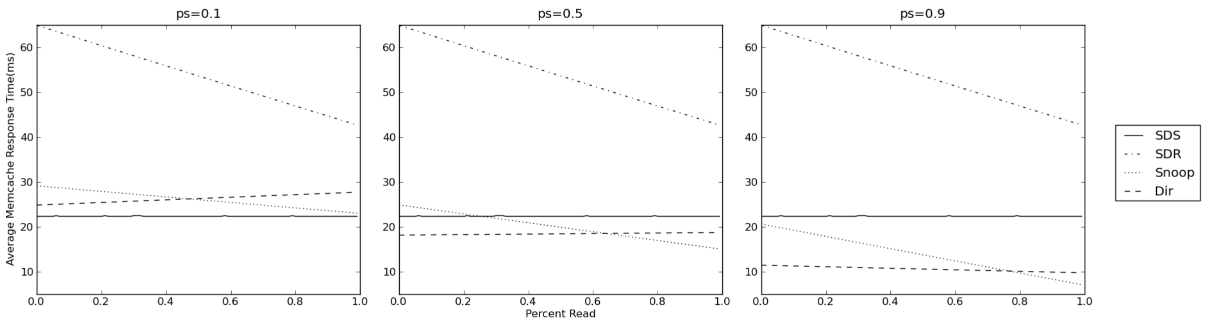

Figure 11: Varying read/write ratio and *ps* values with an East and West coast DC.

based on the number of network devices traversed. This can be easy to compute automatically in some cases. For example, in our configuration IP addresses were assigned systematically per rack. Thus, a client can calculate which Memcache servers were within the same rack and which were farther away based on its own IP address. Using this or similar method would minimize the added management necessary to implement a location-aware caching scheme.

8 RELATED WORK

Memcache is part of a larger key/value NoSQL data movement which provides enhanced performance over relational databases by relaxing ACID guarantees. Scalability is achieved by using a hashing system based on the key to uniquely locate an object's storage location. To achieve locale-aware caching we must modify the typical key/location mapping system. Here we discuss similar systems to Memcache and concentrate on those which have some locality component.

The Hadoop Distributed File System(Borthakur, 2011) 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 where data is small this mantra does not hold. File meta-data is stored and managed in a central location similar to our Dir architecture.

Cassandra can use a rack and datacenter-aware replication strategy(Bailey, 2011) 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(rsumbaly, 2011)(Voldemort, 2011). 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.

Microsoft's AppFabric provides very similar services to Memcache with object storage across many computers using key/value lookup and storage (Nithya Sampathkumar, 2009) in RAM. 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 permitting 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 is useful while ignoring typical Java GC issues. Essentially BigMemory implements a key/value store using key hashing for Java objects similar to Memcache. Additional configurations are possible.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) and configurable(Terracotta, 2011).

The HOC system is designed as a distributed object caching system for Apache, specifically to enable separate Apache processes to access others' caches(Aldinucci and Torquati, 2004). Of note is their use of local caches to speed subsequent requests and a broadcast-like remove function, similar to our SDS with duplication.

9 CONCLUSIONS

We've seen that when a web application has a high *ps* value, many reads per write, or slow a network, a location-aware Memcache architecture can lower latency and network usage without significantly reducing available space. The developed model showed where gains could be made in latency or network usage for each Memcache configuration under specific usage scenarios. Our tool, MemcacheTach, can be used to measure a web application's detailed Memcache usage to estimate performance with other architectures. Our example web application, MediaWiki, showed that the implemented Memcache architectures could reduce network traffic in our configuration by as much as 66%, and latency by 23%. 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-CPU cache systems for Memcache show better performance can be gained within the web datacenter under certain circumstances, with further gains found for more geographically distributed data centers over current techniques.

REFERENCES

Aldinucci, M. and Torquati, M. (2004). *Accelerating Apache Farms Through Ad-HOC Distributed Scalable Object Repository*, volume 3149 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Bailey, N. (2011). Frontpage - cassandra wiki.

Borthakur, D. (2011). Hdfs architecture guide.

Cloudkick (2011). Visual evidence of amazon ec2 network issues.

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

KLab (2011). repcached - add data replication feature to memcached.

Li, K. and Hudak, P. (1989). Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7:321–359.

mediawiki (2011). Mediawiki.

Newman, S. (2011). Three latency anomalies.

Nithya Sampathkumar, Muralidhar Krishnaprasad, A. N. (2009). Introduction to caching with windows server appfabric.

rsumbaly (2011). Voldemort topology awareness capability.

RuggedCom (2011). Latency on a switched ethernet network.

Saab, P. (2008). Scaling memcached at facebook.

Tanenbaum, A. S. and Steen, M. V. (2001). *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Tanenbaum, A. S., Tanenbaum, A. S., Kaashoek, M. F., Kaashoek, M. F., Bal, H. E., and Bal, H. E. (1994). Using broadcasting to implement distributed shared memory efficiently. In *Readings in Distributed Computing Systems*, pages 387–408. IEEE Computer Society Press.

Terracotta (2011). Ehcache documentation cache-topologies.

Voldemort, P. (2011). Project voldemort.