8-2012

Exploiting Data Locality in Dynamic Web Applications

Paul Talaga
Syracuse University

Abstract

The Internet has grown from a static document retrieval system to a dynamic medium where users are both consumers and producers of information. Users may experience above-average website latencies due to the physical distances information must travel. Because user satisfaction is related to a website's responsiveness, e-commerce may be hindered and prevent online businesses from reaching their full potential.

This dissertation analyzes how temporal and relational dependencies in web applications limit their ability to become distributed. Two contributions are made, the first showing the location of data inside a datacenter influences the web system's performance, and secondly, that relaxing strict consistency inside the web application at a fine-grained level can greatly lower latencies for geographically diverse users. Experiments are used to show when and how much these optimizations can benefit a dynamic web application.

Exploiting Data Locality in Dynamic Web Applications

by

Paul Talaga

B.S, St.Lawrence University, 2003

M.S, Syracuse University, 2006

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer & Information Science.

Syracuse University
August 2012

Contents

List of Figures **ix**

List of Tables **x**

1 Introduction **1**

1.1 Why Latency Matters 2

1.2 Sources of Web Latency 3

1.3 Data Dependencies 7

1.3.1 Temporal Dependencies 8

1.3.2 Relational Dependencies 8

1.3.3 Data Consistency 10

1.3.4 Relational Relaxation 11

1.4 Thesis & Contributions 12

1.5 Organization 13

2 Internet Architecture and Technologies **14**

2.1 Location-Aware Client Routing 15

2.2 Web Farm Architecture 17

2.3 Web Server Software Architecture 19

2.4 Summary 21

3 Location-Aware Memcache **22**

3.1 Introduction 22

3.2 Memcache Background 25

3.3 Memcache Performance Prediction Model 27
3.3.1 Assumed Network Topology 27
3.3.2 Model Constants and Calculation 28
3.4 MemcacheTach 32
3.5 Memcache Architectures 33
3.5.1 Standard Deployment Central - SDC 34
3.5.2 Standard Deployment Spread - SDS 34
3.5.3 Standard Deployment Replicated - SDR 35
3.5.4 Snooping Inspired - Snoop 35
3.5.5 Directory Inspired - Dir 36
3.6 Latency Estimation 37
3.7 Experimental Results 40
3.7.1 Latency 42
3.7.2 Network Load 43
3.7.3 Review 45
3.8 Discussion 46
3.8.1 Latency, Utilization, and Distributed Load 46
3.8.2 Multi-Datacenter Usage 46
3.8.3 Selective Replication 47
3.8.4 Object Expiration 48
3.8.5 User Space Caching 48
3.8.6 Overflow 48
3.8.7 System Management 49
3.9 Summary 49

4 Tentacle **50**

4.1 Background 51

4.1.1 Web Application Architecture 51

4.1.2 Relaxing Consistency 52

4.1.3 Database Replication 53

4.2 Tentacle 55

4.2.1 Architecture 57

4.2.2 Operation 59

4.3 Experimental Results 62

4.3.1 osCommerce 63

4.3.2 Application State 64

4.3.3 Simulated Traffic 64

4.3.4 Results 65

4.4 Discussion 68

4.4.1 Alternate Consistency Specifications 68

4.4.2 Database Considerations 69

4.4.3 Cache Pre-warming 69

4.4.4 Stampede Mitigation 69

4.4.5 Pattern-Based Homing 70

4.5 Summary 70

5 Experimental Results Details **71**

5.1 User Simulation & Response Time 71

5.2 Web Server Configurations 72

5.3 Database Configuration 74

5.4 Memcache Performance Measurement 74

5.4.1 Data Capture 75
5.4.2 Data Analysis 76
5.5 Network Utilization Measurement 78
5.6 Memcache Evaluation Progression 79
5.7 Tentacle Evaluation Progression 80
5.8 Summary 82
6 Related Work **84**
6.1 Caching & Distributed Computation 84
6.2 Distributed Datastores 87
6.2.1 Keystores 87
6.2.2 Distributed Filesystems 90
6.2.3 Distributed Databases 91
6.3 Other Systems 93
6.4 Database Middleware Systems 94
6.4.1 DBProxy 94
6.4.2 Ganymed 95
6.4.3 GlobeCBC 96
6.4.4 GlobeDB 97
6.4.5 GlobeTP 98
6.5 Asynchronous Database Writes 98
6.6 Content Delivery Networks 100
6.7 Summary 103
7 Conclusion & Future Work **104**
7.1 Conclusions 104
7.2 Summary of Contributions 105

7.3 Future Work 106
7.3.1 Performance-Aware Caching 106
7.3.2 Rich Application-Datastore Interactions 106
Appendix A Memcache Latency Formula 108
Appendix B SDR: Mem_dup.php 110
Appendix C Mem_RackAware.php 114
Appendix D Snoop: Mem_snoop.php 116
Appendix E Dir: Mem_dir.php 119
Appendix F MemcacheTach: memcache-logging.php 122
Appendix G MemcacheTach: analyse.php 125
Appendix H Tentacle: tentacle.php 133
Appendix I Tentacle: loop.php 139
Appendix J Tentacle: tentacle-daemon.php 140
Appendix K Tentacle: remoteaccess.php 143
References 145

List of Figures

1.1 WAN activity to request web content 3

1.2 Worldwide Latencies on Verizon's Network, Feb 2012 7

2.1 Webserver architecture with PHP, Database, and Memcache . . 19

3.1 Assumed Network Topology 28

3.2 Latency Estimation Calculation 29

3.3 MediaWiki profile under different switch speeds and *ps* values. . 39

3.4 MediaWiki profile under different *ps* values. 39

3.5 Varying read/write ratio and *ps* values. 40

3.6 Varying read/write ratio and *ps* values with an East and West coast DC 47

4.1 Database Replication & Query Routing 58

4.2 HTTP Request Processing Using Tentacle 60

4.3 Evaluation Hardware Configuration 62

4.4 Average Per Page Response Times (ms) 67

5.1 Memcache Data Capture 80

5.2 Memcache Data Processing 81

6.1 Categories of Database Middleware Systems [85] 95

List of Tables

3.1 Network Performance Variables 30
3.2 Web Application Profile Variables 30
3.3 19 monitored Memcache commands 31
3.4 MediaWiki usage values (full caching) 33
3.5 MemcacheTach Overhead 33
3.6 MediaWiki usage for each configuration 42
3.7 Expected and Measured Memcache Latency 43
3.8 Expected and Measured Network Load 44

4.1 osCommerce Page Access Proportions 65
4.2 Aggregate osCommerce Latency with 100ms WAN 66

Chapter 1

Introduction

The Internet has grown from an academic and military research endeavor into a system for commerce and information transmission. What began as a static document delivery and navigation system has transformed into a rich user-centric medium powered by dynamic web applications.

As the geographic reach of the Internet grows throughout the globe, user-perceived performance in the form of page response time has increased due to the larger size. Because the speed of signal propagation is limited, websites hosted farther away take longer to load than nearby ones.

Between 2009 and 2010 wordwide e-commerce revenue grew at an estimated 18.9% to $680 billion, and within the US e-commerce continues to take a larger share of retail purchases[75]. In 2009, 13.7% of all US advertising was done online. This demonstrates an increasing trend of commerce moving online. Due to the worldwide reach of the Internet, a business can now greatly expand into new markets by going online. Unfortunately, as will be described in detail later, a website's response time is directly related to user satisfaction and the customer's willingness to conduct business. Lower response times result in increased visitation, participation, and sales.

Lowering latency for web content is a large academic and business area. In this work we explore how data locality, where data is stored and manipulated, can impact a web application's performance for geographically distributed users.

1.1 Why Latency Matters

End-user latency, or the elapsed time to completely load a web page, has been directly linked to user satisfaction. A study by Google in 2009 showed when page generation slowed from 400ms to 900ms in order to return 30 search results rather than 10, visitors reduced their searches by 25%. A similar Google study showed when page results were delayed by 2% due to additional content, searches per user droped an equivalent 2% [56].

From a revenue perspective, Shopzilla reported at Velocity 2009 that, when their site was redesigned, their page latency dropped from 7 to 2 seconds [34]. After deployment of the new site, page views increased by 25% compared to their old high-latency version and revenue increased between a 7% and 12%. The increased speed also reduced their server load, allowing a 50% reduction in hardware while still able to provide the lower latency for the high traffic volume. Simply by lowering latency they significantly lowered their costs while raising revenue.

User expectation of page latency has also changed. A study done in 2003 found that users would tollerate a load time of 8.6 seconds [46], while another study done in 2006 found a customer would leave an e-commerce site after only a 4 second delay [2]. A survey done in 1984, concerning users reactions to computer responsiveness, found no single value for maximum latency, but in many cases users would become frustrated or think there was an error if the

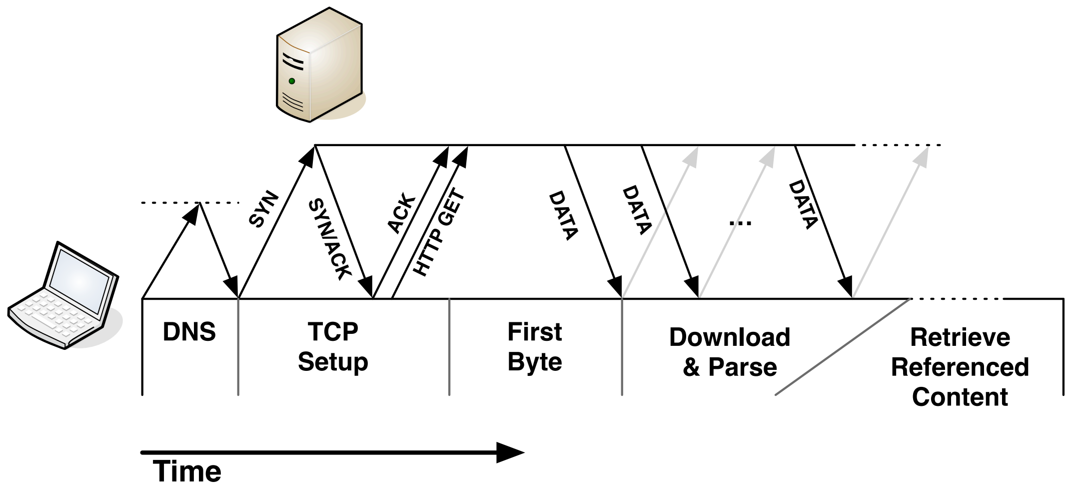

Figure 1.1: WAN activity to request web content

delay was significant. They found a delay of more than twice the average delay without progress notification would concern users. Thus, as high bandwidth Internet connections become more prevalent, average page latency will decrease and lower a users toleration time.

1.2 Sources of Web Latency

Web page latency occurs due to a sequence of time-consuming events. A user request is issed by the user entering a URL, or clicking an existing link, followed by the retrieval of the content. This process can be broken down into five steps: a DNS query, connection establishment, time to get the first byte, download time & page parsing, and downloading of embedded content [38, 19, 29, 76], detailed below. Figure 1.1 shows a timeline of the process.

1. **DNS:** If the URL does not contain an IP address, a DNS query must be issued to resolve the name. DNS entries are heavily cached throughout the web as well as on a user's computer, making queries fast but still

measurable. A detailed study done in 2002 found non-locally cached DNS latencies ranging from 950ms to 2.31s [53].

2. **TCP Connection:** Once the IP address is found, a TCP link is established with the remote HTTP server. Due to the three-way-hanshake to set up a TCP connection, this will take at least one round-trip to the server. Latency for this step will depend on proximity to the server, network traffic, network devices, and server utilization.

3. **Time to First Byte:** Immediately following the last acknowledgement (`ACK`) to the server of the TCP three-way handshake, the HTTP request can be sent. The server then processes the request and replies with data. Latency involves one round-trip plus any server processing time.

4. **Download and Parsing:** Depending on the size of the returned page, multiple packets may be sent each with appropriate verification `ACK` responses or retransmissions. Browsers may begin parsing the HTML page in the middle of transmission, speeding the page display to the user, and lowering perceived latency. Prior work has noted that users accommodate longer page latencies if some sort of progress feedback is displayed [38], supporting immediate parsing technology to minimize user dissatisfaction. Latency is dependent on the server's page production speed, network conditions, and the bandwidth between the client and server.

5. **Embedded Content:** The HTML page downloaded from the server may contain references to other documents needed for page display, such as style sheets, Javascript, or images. During HTML parsing, these refer-

ences are found and the same download process described here is used for each.

Once the original HTML page is downloaded and parsed and all referenced content is downloaded and displayed, the user can view the complete page. The time from the initial user click to the completed page is the page latency time.

The absolute minimal latency time for a web page is thus the time of two round-trips to the HTTP server.

Referenced content can be retrieved faster using `HTTP Keep-Alive` (introduced in HTTP 1.1 and unofficially added to HTTP 1.0), which allows a TCP connection to remain open after the HTTP transaction has completed. Subsequent requests to the same server can use this already open TCP link rather than suffer the additional latency of the three-way startup of a new connection. Another feature in HTTP 1.1, `HTTP Pipelining`, allows a browser to send multiple HTTP requests simultaneously over a TCP link to a server to download content in parallel. Most browsers only support 8 simultaneous connections per domain. The average page in the Top-100 contain 40 separate download requests per page, leading to additional latencies because of serialization [14, 72]. Even with these two optimizations in place, the distance to web server still heavily dictates page latencies [15].

A study in 1998 found the TCP connection stage took at least 25% of the total page latency time for non-persistent (non-`HTTP Keep-Alive`) requests for actual web traffic comprising 1 million requests [54]. This demonstrates latencies due to network conditions contribute significantly to overall page latency.

A popular method for lowering latency addresses the referenced content in an HTML page. Static or seldom-changed content is placed in a content delivery network (CDN), which replicates the data throughout the globe allowing fast retrieval for the end-user[43, 73, 104]. Such services also provide DNS routing capabilities to direct requests to the nearest CDN edge node. CDNs identify a key optimization, locality of data, for reducing web page latency.

For an example of how physical distance influences latency, consider a user in India requesting a web page from a server in New York, a distance of 12,500 kilometers (km). Based on optimal signal propagation in optical fiber (60% speed of light), one round-trip would take 125ms on the surface of the earth. Therefore, the best latency possible for the requested page would be 250ms (two round-trips), ignoring all server processing, network congestion, bandwidth limitations, and referenced content download latency. This best-case page time presents a physical limit to web page latency that can't be overcome with advances in technology.

Figure 1.2 shows current latencies between select cities on the Verizon network in February 2012 [103]. Due to a non-direct path, network utilization, and equipment-introduced delays, the current latency between India and New York is 267ms, more than twice the theoretical minimum, resulting in 534ms just to set up the TCP connection. Note the high latencies in Asia and India. Users in these locations, where the Internet is growing rapidly, will experience high loading times for US web content.

A report by Google similarly claims web page load times are linearly related to the round trip time (RTT) to the web server [15]. The effect of increased bandwidth is investigated and found to have little effect on page download

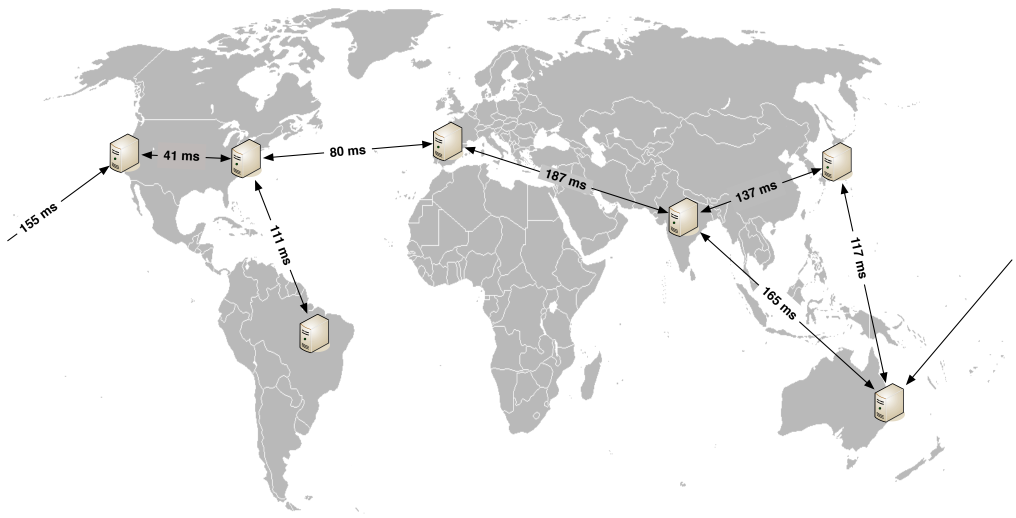

Figure 1.2: Worldwide Latencies on Verizon's Network, Feb 2012

time. Reducing the RTT is far more productive than increasing bandwidth. An alternate approach is to reduce the number of round-trips necessary for a page load, which is a goal of their SPDY project [72].

Each of the five sources of page latency offers opportunity to optimize. As mentioned before, CDNs handle static referenced page content well, as does caching of DNS data to speed DNS lookup. The remaining three sources of latency can benefit from two optimizations: (1) moving the HTML generation closer to the user to reduce distance latencies and (2) producing the HTML faster. By optimizing these first crucial steps, pages will load faster and allow referenced content to be retrieved that much sooner.

1.3 Data Dependencies

User modifiable data is vital to a dynamic web application. Mutable state allows a user's actions to affect others or the real world, through (for example) forum posts, webmail messages, video responses, product orders, or financial transactions. Interactability requires some user-initiated application state change, which can be difficult to process reliably for large sites or in a distributed web application due to the inevitability of hardware failure and data duplication[82]. Dependencies inside the web application and its data increase the complexity of building a large distributed web application.

1.3.1 Temporal Dependencies

Computer programs rely on causality of actions. If a command changes the application's state, all subsequent commands must observe this change. Causality is trivial when state is stored in a single location, but causality requires additional resources when state is replicated. In the case when multiple simultaneous users operate on shared data, much effort must go into assuring data is read and written in order so that errors do not occur.

1.3.2 Relational Dependencies

Web applications may contain data that require the storage of relationships between internal data. For example, consider an e-commerce site that allows items to be purchased. An order consists of multiple items to be shipped to the same user. Each order therefore must contain links to the items, as well as the user who placed the order, so that the shipping department can process the order. Using a Relational Database Management System (RDBMS) allows storage of this interrelated data and let complex queries access it. Retrieving all items ordered by a specific user on a specific day requires the access of multiple data sets (database tables) as well as knowledge of their dependencies.

Other queries accessing the RDBMS, such as listing all the current users, or information about a specific product, do not require access to all the data. These queries may only access one or two data sets or tables. Even though relational dependencies exist in the database, they are not always used for every query.

Relational Database Management Systems (RDBMS) were developed as a way of storing and retrieving relational data. To assure reliable operation in a multiuser environment, RDBMS implement a set of guaranteed properties. Such guarantees are needed for many mission-critical applications such as banking, airline, insurance, and other businesses where a computing error may result in loss of life or financial ruin. The four properties, known as the ACID properties, are:

1. **Atomicity:** Commands sent to the database can be grouped into transactions. If during the execution of a transaction a command fails or the transaction is aborted, all previous commands in the transaction must be undone, with no effect on the data. This property gives transactions an all-or-nothing capability.

2. **Consistency:** Data in a database must never be allowed to be in an ambiguous state. Relational and temporal dependencies are obeyed.

3. **Isolation:** Operations in a transaction must only observe state changes initiated within the same transaction, even if other queries are being simultaneously applied. This property assures temporal dependencies are obeyed inside a transaction.

4. **Durability:** Successful transactions are permanent, and all subsequent actions will observe the changes. This property assures hardware or network failures do not effect the result of completed transactions. This property assures temporal dependencies.

By obeying these ACID properties, a RDBMS can be used reliably by multiple simultaneous users. The widespread use of RDBMS before and during the development of the web made it a natural data-storage system for web applications, even though it was never intended to be used in that area [91]. Relaxing any of the ACID properties, or the dependencies above, can increase performance. The recent NoSQL class of storage technologies sacrifices consistency for increased performance [65]. Next, consistency relaxation is discussed.

1.3.3 Data Consistency

Data consistency assures that all operations performed on a datastore are viewable by all users thereafter, and at no point in time the data is corrupt. This is assuring temporal dependencies, and a strongly consisten data-store respectively. There exists many different ways consistency can be relaxed. Any policy that assures all users view the actions of previous operations *at some point* thereafter are considered weakly consistent. Vogels identifies several variants of weakly consistent policies [106]:

Eventual Consistency: After a change to the data, there is some calculable time in the future when all users will observe the change.

Causal Consistency: A stronger form of eventual consistency where, if one client updates an item and notifies another client of the action, the new client will observe the change and further writes by each will follow the original update. Clients with no knowledge of the update will follow eventual consistency.

Read-your-writes Consistency: If a client makes an update, subsequent accesses by that client will reflect the change. Other clients will follow eventual consistency.

Session Consistency: An isolated version of read-your-writes where a single client has read-your-writes consistency for specific session-related data. Data is not durable and must be regenerated in the event of an error.

Monotonic Read Consistency: After a client reads a particular piece of data, subsequent accesses by that client will not return older versions of the same data.

Monotonic Write Consistency: A policy where all modifications done by a client are serialized with respect to that, and only that client.

Because any consistency relaxation can cause non-typical application behavior, the developer must be aware of the dangers and be sure the application preforms as expected. Any type of consistency relaxation is a form of temporal dependency relaxation. Next we look at relational dependency relaxation.

1.3.4 Relational Relaxation

As discussed previously, RDBMS's provide a storage and retrieval system for related data. Some queries may be complex, by accessing multiple data sets, while others are simple, by only accessing one or two sets. In order for the database system to be prepared for any query, all data must be available.

Relaxing dependencies inside the data can be done by selective storage of data. If a dataset or some relationship between data will never be queried or required then its storage is not useful.

1.4 Thesis & Contributions

This dissertation's thesis states:

> **The locality of data in a dynamic web application can influence and improve the performance for geographically diverse users by exploiting temporal and relational dependencies inside the application.**

We present two contributions supporting our thesis:

1. **Location-aware Memcache Architectures:[92]** This work develops a set of location-aware caching strategies for the popular object caching system Memcache. Based on multi-CPU cache architectures, we show how, for certain uses, these strategies can lower latency, decrease network usage, and increase availability for object caches that support HTML generation. This work shows that in the physical distances of a LAN in a datacenter, latency reductions can be made, showing larger promise for geographically larger installations. To predict application performance under these and other Memcache configurations, a network and application model is developed. Predictive measurements are made and then compared to analytical results obtained from a mock-datacenter running a popular open-source web application. Results support the models and usefulness of our presented Memcache architectures.

2. **Tentacle:** This developed software layer enables a generic database-backed web application written in PHP to easily transition from a central serving system to a global application providing lower average latency

to any user worldwide. This latency reduction is accomplished via five complementing components that allow placement of web servers near the end user while obeying consistency constraints. Key to this work is the observation that state need not be consistent for all database queries in the entire application. As an example, we applied our layer and modifications to a popular e-commerce application and found that many of the database queries and pages could allow stale information. By exploiting relaxed consistency requirements, the resulting latency was better on average than the original application deployed for local or geographically distributed users. This work shows that relaxing temporal and relational dependencies for a web application, when the application allows, can lower latency for geographically distributed users.

1.5 Organization

The remainder of this dissertation is organized as follows. Chapter 2 discussed relevant Internet technologies and architectures necessary for our contributions. Chapter 3 details our location-aware Memcache architectures, performance prediction models, and performance evaluation. Chapter 4 discusses our Tentacle system, its application to an e-commerce application, and resulting performance. Chapter 6 discussed in-depth the tools and configurations of all experimental data, as well as the use of our developed software. Chapter 7 contains future work and concluding remarks. Appendices A through K exhibit formula and code for our contributions.

Chapter 2

Internet Architecture and Technologies

These contributions are most applicable in a web system setting, though they could be used anywhere distributed data is needed quickly in a high latency environment. This chapter discusses background Internet and software architectures in which the contributions are best suited, plus a discussion of relevant web technologies.

The Internet is made up of many parts, but this work concentrates on supplying the end-user with the requested HTML document. More specifically, this work concentrates on the transmission of HTTP requests from an end-user to a webserver, the generation of the HTML response, and the return of the HTTP response to the end-user. Many technologies support this interaction. Below, three important advanced web technologies necessary for our work are discussed: routing requests to the closest serving location, web farm architecture for serving high web-request load, and the web server's internal software architecture for satisfying dynamic requests.

2.1 Location-Aware Client Routing

Due to distance-induced latencies discussed in Section 1.2, a website can lower the end-user response time by satisfying a request close in network proximity to the requestor. This section discusses techniques for routing an HTTP request to a local server with no additional user action. In all cases it is assumed the website has servers spread throughout the Internet and users request a common URL. These distributed servers are called edge servers as they exist at the edge of the Internet, very close to the users.

One of these methods is necessary for the Tentacle contribution in Chapter 4 to be effective. Below are four options for implementing client routing, with more details available by Barbier et al.[13]:

1. **Application Implemented:** Clients can be forwarded to a local server after first contacting a central server. Here, each web server or webfarm has a unique dns subdomain, such as `us.mydomain.com`, `uk.mydomain.com`, `au.mydomain.com`. Requests to `mydomain.com` go to a single server where the source address is used to calculate the closest local server or subdomain. Alternate client location methods based on IP address have been investigated including by autonomous system (AS) number[63] or others[66]. An HTTP redirect is then issued and all further communication is done with the local subdomain. The first page request, such as to `mydomain.com`, will be slow due to the further network distance, but subsequent requests with location-specific subdomains will be fast.

2. **DNS:** Due to DNS request resolution at an owner-controlled facility, custom responses can be given using specialized DNS software. Based on

the IP address and location of the request, the DNS server can provide a custom response which resolves the DNS entry to a local location. Unfortunately, this method can provide erroneous results due to DNS resolution not always being location specific[108].

3. **Anycast DNS:** Anycast provides a way for a single IP address to be used for multiple devices on the Internet. Internet routing techniques then route requests for that IP to the nearest location to the requestor. As Internet routing changes, so will the routing for that IP, making connection-oriented communication difficult. DNS uses UDP, which does not rely on a TCP connection. In Anycast DNS, multiple DNS servers are used world-wide, all with the same IP, but responding with different name-to-IP mappings. This allows webservers or webfarms throught the word to have unique IPs. In this way, the same URL will resolve to different IP address wherever the requestor is, and optimally to the closest web server to the user.

4. **Anycast IP:** Similar to Anycast DNS, this configuration uses multiple servers with identical IP address spread over the Internet. Rather than serving connection-less DNS requests, this method uses anycast for the HTTP web traffic. All DNS entries are identical. Again, requests will be routed to the closest server, but due to HTTP's longer-lived TCP connection over DNS's UDP, the possibility exists a TCP connection will be interrupted due to a routing change. Nonetheless, the speed and flexibility of this configuration has made it popular for CDNs.

Barbir et. al. discuss these routing techniques in detail.

2.2 Web Farm Architecture

Multiple webservers may be needed if traffic is high or the computational load of serving requests overburdens a single webserver. A web farm is a single location containing more than one webserver serving identical content. Applications using local data storage may require modification to be used in a web farm setting by using central data storage.

A load balancing mechanism is needed in order to optimally utilize all web serving resources. Balancing the web load over all webservers in a web farm can be done in one of three ways[40]:

1. **DNS:** The DNS system allows for multiple prioritized A entries, which can be reported in any order. Thus, different clients will resolve the same URL to different IP addresses. These different IP addresses belong to different webservers inside the the web farm, and thus load is distributed. This method does not allow balancing based on server utilization, or allow easy removal or addition of webservers as new DNS entries must be created and distributed.

2. **Level 4 Loadbalancer:** This method operates on level 4 of the OSI model and deals with TCP. The DNS resolves to a single IP address belonging to the load balancing device in the web farm. The load balancer's purpose is to accept incoming TCP connections and route them to one of the available web servers. Internal state is stored in the load balancer to route future packets from the same TCP session to the samer webserver. Routing for new connections can be done in many ways, such as round-robin, weighted round-robin, lowest utilization, or the lowest number of

current TCP connections to name a few.

3. **Level 7 Loadbalancer:** Operating at a higher level allows inspection of the HTTP request itself. URL parameters, HTTP header information, cookie values, or any other information at level 7 or lower can help the load balancer route traffic to the best host. Routing based on URL is a popular method of keeping web cache hit-rates high[40].

   Sticky session support is an important feature of this type of load balancer. This feature routes requests in the same HTTP session to the same server or rack of servers. HTTP session data is needed in many dynamic web applications to keep track of a user's state in the application. Session information can be conveyed in the HTTP request either in the URL or as an attached cookie value. The load balancer's configuration must specify how sessions are identified in the specific application. Storage of session data can either be in a centralized database, local file, or other location. A truly scalable web application would not require local data storage, but some applications may require it. The location-aware Memcache contribution in Chapter 3 requires sticky sessions and exploits data-usage patterns.

Additional equipment is also needed in a web farm, such as routers, switches, file servers, and database servers. Chapter 4 discusses databases in a web farm setting in detail. The next section discusses the software architecture in a web server.

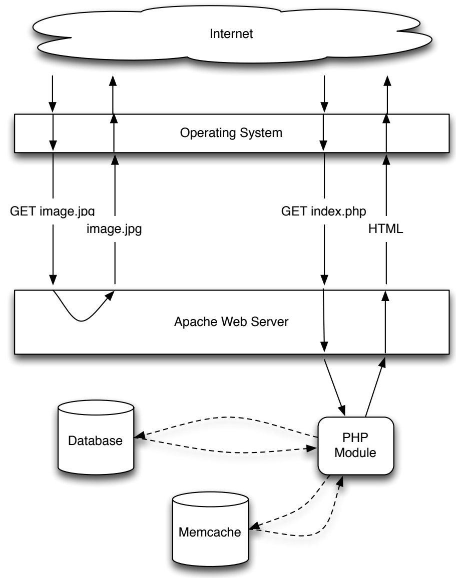

Figure 2.1: Webserver architecture with PHP, Database, and Memcache

2.3 Web Server Software Architecture

Any software able to respond to HTTP requests can be considered a web server. The web's original purpose was serving static content, but through the progression of web scripts, server gateway interfaces (SGI), and web scripting languages, custom HTML serving has become more popular. Due to the this progression, static web serving applications are still used, but with additional modules providing dynamic capabilities. This research uses the PHP module in the Apache web server to respond to PHP page requests, with the Apache server satisfying all static content. If needed, PHP can contact a database or any external resource, including a Memcache server. Figure 2.1 depicts how web requests are handled by the operating system, Apache, and possibly PHP.

PHP is an interpreted web scripting language developed for dynamic page generation. In practice, a PHP script is an HTML-like file with embedded code.

The PHP interpreter scans this file when requested and preforms the listed instructions, finally returning the generated HTML to the requestor. Since its beginning in 1994, PHP as gone through many modifications and now supports object-oriented programming and is a major scripting language used on the web[90].

Web applications written in PHP do not need to adhere to any software architecture rules. Depending on the need, a small amount of script code could be used on a single HTML page, or an entire web application could be developed. Built-in libraries allow database-interaction, image creation, email support, HTTP session handeling, PDF manipulation, as well as many other web-related features useful in building a dynamic web application[109]. The Model-View-Controller construction and architecture method is popular for large applications in order to allow many developers to contribute simultaneously.

Caching is a popular method of speeding dynamic web serving. During execution of the application, expensive or often-used data is cached to speed future execution. In a web farm setting where multiple webservers execute application code, a common cache can be beneficial. The next chapter discusses Memcached[1], a whole-datacenter caching system extensively used at Facebook, Wikipedia, Flickr, Twitter, and others. The web developer can store any string or serialize-able object in Memcache by key, retrievable by any Memcache client in the system. By only storing data in RAM, great speed is possible.

[1] http://memcached.org/

2.4 Summary

The Internet has seen many advances and changes since its original development in the 1980s. This, as well as the Introduction chapter, review necessary technologies in order to understand how these contributions fit into and advance a fully-functioning web serving system. Specific contributions are discussed next.

Chapter 3

Location-Aware Memcache

Caches, by design, speed processing by keeping frequently used data easily accessible. The principle of locality enables typical in-computer caches to increase the performance of the entire system. This chapter examines how a popular web caching system, Memcache, can be extended to increase performance by storing data close to where it may be used. Relational dependencies within the cached data must be handled by the application, allowing flexible cache designs such as these.

3.1 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 that provide a storage API to the web application over a network or local file socket connection. No persistence guarantees are made in the case of failure, and thus Memcache is best used to cache regenerable content. The storage location of data in a set of Memcache servers is determined via a hash function applied to the key. 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.

A key/value storage system such as Memcache contains no relational dependencies internally, thereby allowing data units to be stored and managed independently. This freedom allows Memcache's performance to grow linearly as resources are added.

As web farms and cloud services grow and use faster processors, the relative delay from network access increases because signal propagation is physically constrained by the speed of light and network technology. Developing methods for measuring and minimizing network latencies is necessary to continue to provide rich web experiences with fast, low latency interactions.

As an example, consider a webserver and Memcache server on opposite ends of a datacenter the size of a football field. The speed of light limits the fastest round-trip time (RTT) in fiber to about 1μs. Current network hardware claim a RTT for this example between 22μs and 3.7 ms [80], more than an order of magnitude slower than the physical minimum. Assuming Memcache uses optimal network transport, 100 Memcache requests would take between 2.2 ms and 370 ms, ignoring all processing time. This range is supported by multiple measurements of latencies in both Google App Engine and Amazon EC2 showing between 300μs and 2 ms RTT between two instances for a single message [61, 25]. Section 1.1 discusses why fast web responses are important. Any reduction in server processing time will benefit users and therefore the site.

Latency is related to network load. More network utilization will translate into more latency due to collisions and buffering, leading to costly network hardware to keep utilization low [80]. Reducing network load, especially over utilized links, will keep latency low.

This work explores how data locality can be exploited to benefit Memcache by reducing latency and core network utilization. Non-uniform cache usage-patterns allow these optimizations. While modern LAN networks in a data-center environment allow fast transmission, locating data close to where it is used can lower latency and distribute network usage, resulting in better system performance [107]. When looking at inter-datacenter communication, latency becomes more pronounced.

We present five Memcache architecture variants, the typical architecture, two natural extensions, and two novel to this area based on prior multi-processor caching schemes. A network and usage model is developed to predict the performance of all variants under different configurations. This model allows a mathematical derivation of best and worst case situations, as well as the ability to predict performance.

This work makes five significant contributions:

1. A model for predicting Memcache performance

2. A tool for gathering detailed Memcache usage statistics

3. Two novel Memcache architectures based on multi-CPU caching methods

4. Mathematical comparison between five Memcache architectures

5. Experimental comparison of five Memcache architectures using MediaWiki

3.2 Memcache Background

As previously mentioned, Memcache is built using a client-server architecture. Clients, in the form of an instance of a web application, store 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. A per-entry 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. Data is only stored in memory (RAM) rather than permanent media for speed.

The location of a Memcache daemon can vary. Small deployments may use a single daemon 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 processing.

The Memcache daemon recognizes 16 commands as of version 1.4.13 (2/2/2012), categorized into storage, retrieval, and status commands. Various APIs written in many languages communicate with a Memcache daemon 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;
}
```

These functions cache data in Memcache, rather than query the database on every function call. 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`, triggering a subsequent database query when the function is called next. Thus, at most once a minute the database will be queried, with the function returning a value 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. This will store separate data per user/session. Periodically, or on logout, another function clears the cached data. During an active session the `last_login` function will only access the current session's data with no other user needing the data. Thus, if sticky load balancing (requests from the same session are routed to the same web 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. It may make sense for a local copy to be stored, rather than each webserver requesting commonly used data over the network. Caching data locally, when possible, is the basis for the proposed architectures.

3.3 Memcache Performance Prediction Model

When evaluating different Memcache architectures it is useful to be able to predict performance of any proposed system. Using the constants and formula developed here allows a rough estimation of latency and network utilization for a Memcache system.

To simplify our model we assume the following traits:

1. **Linear Network Latency:** Network latency is linearly related to network device traversal count [80].

2. **Sticky Sessions:** A web request in the same session can be routed to a specific rack or webserver.

The result of applying the model to a specific application, network, and Memcache configuration is an average per-Memcache-command latency.

3.3.1 Assumed Network Topology

Network topology greatly influences the performance of any large interconnected system. A physical hierarchical star network topology is assumed consisting of multiple web, database, and Memcache servers.

Web servers, running application code, are grouped into racks. Racks can describe a group of physical 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.

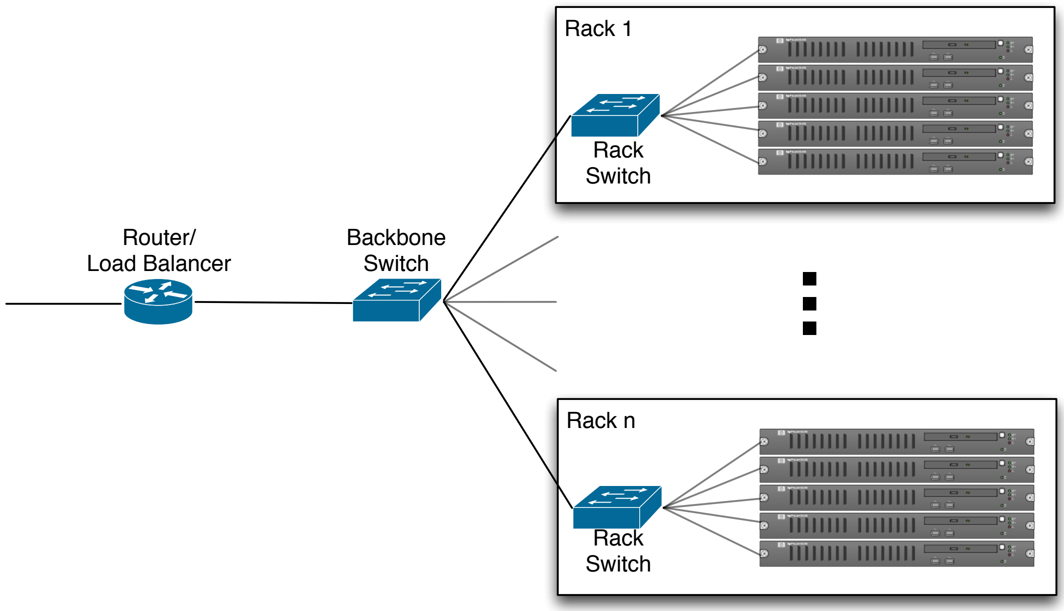

Figure 3.1: Assumed Network Topology

Racks are connected through a backbone link. Thus, for one webserver in a rack to communicate to another in a different rack at least 4 network segments connected with 3 switches must be traversed.

Figure 3.1 depicts this configuration.

3.3.2 Model Constants and Calculation

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 round-trip-time (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 times described later.

Similarly l_2 represents a request traversing 2 switches, such as from a rack

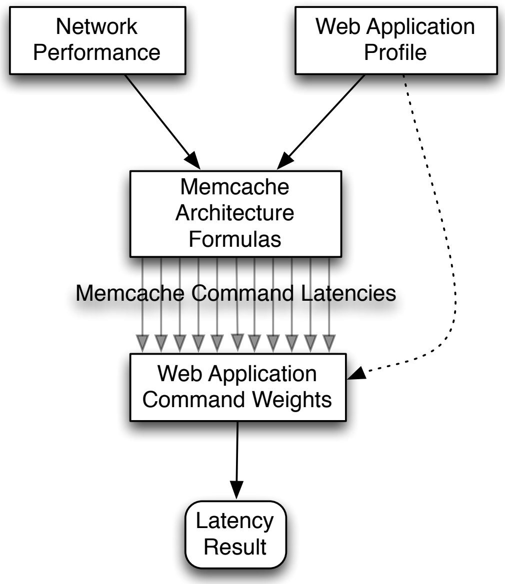

Figure 3.2: Latency Estimation Calculation

to a node on the backbone and back. l_1 represents traversing a single switch and l_{local} is used to represent this closest possible network distance. In some cases $l_{\text{localhost}}$ is used to represent l_0, where no physical network layer is reached. This metric differs from hop count as every device a packet must pass through is counted, rather than only routers.

Figure 3.2 depicts the data-flow through the model to calculate a final latency estimate.

Each block represents a set of variables or formula in the model. Table 3.1 describes variables in the `Network Performance` block.

The `Web Application Profile` contains variables inherent to the web application, such as the size of objects, key distribution throughout the web application, percent of Memcache commands which are reads, Memcache commands used, and command distribution. The command distribution dictates

$l_{\text{localhost}}$ or l_0	Average RTT (ms) to localhost for the smallest size packet possible
l_{local} or l_1	Average RTT (ms) to a nearby node through one network device
l_n	Average RTT (ms) traversing n devices
r	Number of racks used in system
k	Data replication value used when possible
$size_{note}$	Average size of data location note (bytes)
$switch$	Delay time (ms) per switch or network device traversed
$base$	Constant OS, network delay, and Memcache overhead (ms)
bw	Minimum network bandwidth in system (Mbps)

Table 3.1: Network Performance Variables

ps	Proportion of Memcache commands used on the average page that are only referenced by a single HTTP session [0-1]
rw_{cmd}	Percent of Memcache commands which are reads [0-1]
rw_{net}	Percent of network traffic based on data transfer which are reads [0-1]
$size_{object}$	Average size of typical on-wire object (bytes)
$usage_m$	For a specific application usage scenario, proportion of each Memcache command used plus duplicates for command failures (Table 3.3) [0-1] where $m = 0$ to 19 and $\sum_{m=0}^{19} usage_m = 1$

Table 3.2: Web Application Profile Variables

the `Command Weights`, used later. Table 3.2 describes each variable in detail.

The ps value is central to our approaches. It gives a measure of how many session specific Memcache requests are used per web page. If an application only stores session information in Memcache $ps = 1$. If half the Memcache requests on a page are session specific and half used by other users/sessions, then $ps = 0.5$. For example, if an application used the above `get_num_users()` and `last_login($id)` in Section 3.2 once per page then for 100 user sessions Memcache would store 101 data items, of which 100 are session specific. Even

Set	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

Table 3.3: 19 monitored Memcache commands

though session data will consume 99% of all stored data, our *ps* value will still be 0.5.

The $usage_m$ variable captures how often each of the Memcache commands are used. Only commands that manipulate data are tracked, of which there are 13. The `append` and `prepend` commands are combined. Of these 13 Memcache commands, 7 have different latency performance if the the command was successful or not, thus we break these into a `Hit` or `Miss` variants. Table 3.3 lists all 19 tracked Memcache commands and variants. Note `CAS` commands are not fully tracked at this time.

The `Memcache Architecture Formulas` block contains 19 formula, each using the `Network Performance` and `Web Application Profile` variables to estimate the latency for a specific Memcache command. These formula are specific to the Memcache architecture being used. Section 3.5 discusses standard and proposed architectures, with a mathematical comparison using this model in Section 3.6. Appendix A lists all formula.

The final `Latency Result` value is calculated by obtaining latency values for each of the `Memcache Architecture Formulas` and weighting each using $usage_m$ `Command Weights` and summing.

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

3.4 MemcacheTach

Predicting Memcache usage is not easy. User demand, network usage, and network design all can influence the 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? Which Memcache servers are responding slowly?

To answer these and other questions we developed MemcacheTach, a Memcache client wrapper which intercepts and logs all requests. Source code is listed in Appendix F and G, with usage and implementation details in Section 5.4. 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. Values for the `Network Performance` and `Web Application Profile` model variables above are provided via MemcacheTach analysis, plus the ratio of the 19 Memcache request types, and other useful information about a set of Memcache requests. Figure 3.4 shows the measured values for MediaWiki from a single run in our mock datacenter with full caching enabled for 100 users requesting 96 pages each. See Section 3.7 for further run details. Section 5.4 provides a detailed discussion of MemcacheTach and its use.

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

switch (ms) :	0.21 (ms)	ps :	0.56	Avg. data size (Kbytes):	3.3
base (ms) :	3.0 (ms)	rw_{cmd} :	0.51	Mem. requests per page:	16.7
Avg. net size (bytes):	869				
Set hit:	24%	Replace hit :	0%	Inc hit :	0%
Set miss :	0%	Replace miss :	0%	Inc miss :	21%
Add hit :	0%	Delete hit :	2%	Dec hit :	0%
Add miss :	0%	Delete miss :	0%	Dec miss :	0%
Get hit :	44%	CAS1 :	0%	App/Prepend hit :	0%
Get miss :	7%	CAS2 :	0%	App/Prepend miss :	0%
Flush :	0%	CAS3 :	0%		

Table 3.4: 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

Table 3.5: MemcacheTach Overhead

56% of keys used per page were used by a single webserver ($ps = 0.56$), 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 3.5. `Off` is our Memcache baseline, while `Logging` used MemcacheTech and saved data on each Memcache call. See Section 3.7 for implementation details. On average MemcacheTech had an overhead of 36 ms.

3.5 Memcache Architectures

Described and compared here are Memcache architectures currently in use, two natural extensions, and our two proposed versions. All configurations are implemented on the web server(s) via wrappers around existing Memcache clients, thus requiring no change to the Memcache server. Source code is available in Appendix B, D, E, with Appendix C providing common functions to Snoop and Dir.

Estimation formula for network usage of the central switch and space usage efficiency for all variants are given using the variables defined in Section 3.3.2. An in-depth discussion of latency is given in Section 3.6.

3.5.1 Standard Deployment Central - SDC

The typical deployment consists of a dedicated set of *memcached* instances 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%

3.5.2 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). The key dictates the data storage location, which could be in any rack, not only the local. This architecture requires no code changes compared to SDC, but distributes Memcache servers into each rack.

With some portion of the data local, the central switch will experience less traffic:

Network Usage: $\frac{r-1}{r} * 100\%$

All available space is used for object storage:

Space Efficiency: 100%

3.5.3 Standard Deployment Replicated - SDR

Durability is added by storing k copies of the data on different Memcache daemons, preferably on a different machine or rack. While solutions do exist for data duplication in the server (repcached [47]), duplication is done here on the client and uses a locality metric to read from the closest resource possible. This can be implemented either through multiple storage pools or in our case by modifying the key in a known way, essentially creating a hash chain, to choose a different servers or racks[7, 102]. A write must be applied to all replicas, but a read contacts the closest replica first, reducing latency and core network load. Appendix B contains PHP code for this variant.

Reading locally can lower central switch usage over pure duplication:

Network Usage: $rw_{net} \times (1 - \frac{k}{r}) + (1 - rw_{net}) \times (k - \frac{k}{r}) * 100\%$

The replication value lowers space efficiency:

Space Efficiency: $100/k\%$

3.5.4 Snooping Inspired - Snoop

Based on multi-CPU cache snooping ideas, this architecture places Memcache server(s) in each rack allowing fast local reads [41, 51]. Writes are done locally with a data location note 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 [93]. An update overwrites all notes and data with the same key. To avoid race conditions deleting data, notes are first stored in parallel, followed by the actual data. Thus, in the worst case multiple copies could exist, rather than none. A retrieval request first queries the local Memcache 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 is not stored in the system. Appendix D contains PHP code for this variant.

The broadcast nature of a set could be more efficient if UDP was used with network broadcast or multicast. Shared memory systems have investigated using a broadcast medium, though none in the web arena [94].

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: $(rw_{net} \times (1 - ps) + \frac{(1-rw_{net}) \times size_{message} \times r}{size_{object}}) * 100\%$

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

Space Efficiency: $\frac{size_{object} * 100}{size_{message} \times (r-1) + size_{object}}\%$

3.5.5 Directory Inspired - Dir

An alternate multi-CPU caching system uses a central directory to store location information [41, 51]. Here, 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 contact 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 with its current location. Appendix E contains PHP code for this variant.

Rather than sending many notes on the central switch per write as with Snoop, Dir is able to operate with three central requests, one to retrieve the current note, the second to clear the old data, and the last to `set` the new note, no matter how many racks are used. This allows Dir to stress the central switch the least especially when many racks are used.

Network Usage: $(rw_{net} \times (1-ps) \times \frac{size_{message}+size_{object}}{size_{object}} + \frac{(1-rw_{net})\times size_{message}\times 3}{size_{object}}) * 100\%$

Likewise with Snoop, message size dictates storage efficiency:

Space Efficiency: $(rw_{net} \times (1-ps) \times \frac{size_{message}+size_{object}}{size_{object}} + \frac{(1-rw_{net})\times size_{message}\times 3}{size_{object}}) * 100\%$

3.6 Latency Estimation

The above architecture options are evaluated by estimating latency using the model described in Section 3.3.2. The `Memcache Architecture Formulas` were derived for each architecture variant, provided in Appendix A.

In general, each formula estimates the latency for a specific Memcache command using the following components:

- **Bandwidth:** Time the average sized object will require to traverse the network given the provided bandwidth, if data is transferred.
- **Switching:** Time needed to traverse the network given a specific network distance.
- **Architecture:** Time for additional communication due to the architecture.
- **Read/Write:** Time weighting for the proportion of reads to writes.
- **Location:** Additional time if data is not stored locally.

As an example, the `set hit` command would have a latency (ms) of $l_2 + \frac{size_{object}}{bw \times conv}$, where $conv = 1024 * 1024/8/1000$, under SDC, factoring in distance and bandwidth. SDS would have $\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + \frac{size_{object}}{bw \times conv}$ for a `set hit` because some portion of the keys would be local ($\frac{1}{r}$), and thus faster, while the rest ($\frac{r-1}{r}$) would need to travel farther. The same bandwidth calculation applies. SDR would take $l_3 + \frac{size_{object}}{bw \times conv}$ because multiple `set`s can be issued simultaneously. Snoop would need $l_{local} + l_3 + \frac{size_{object} + size_{message}}{bw \times conv}$ with data being sent to the local rack and messages sent to all others in parallel.

Using a specific `Web Application Profile` and `Network Performance` statistics, here from a run of MediaWiki[58], we can vary individual parameters to gain an understanding of the performance space under different environments. A bandwidth (bw) value of 100 Mbps was used for all.

We first look at how network switch speed can effect performance. Recall 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 4.4ms

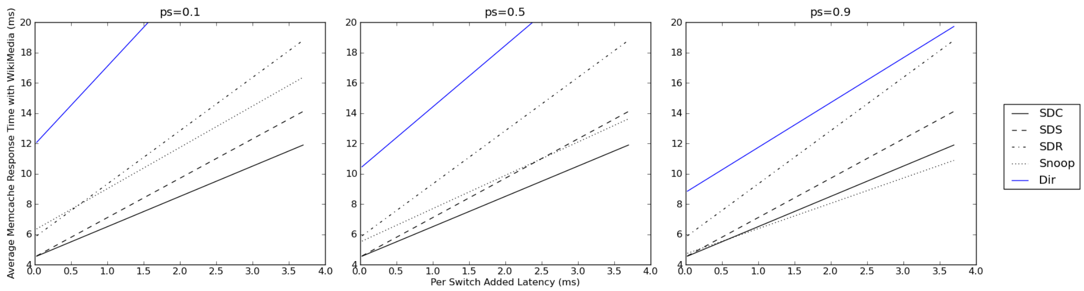

Figure 3.3: MediaWiki profile under different switch speeds and ps values.

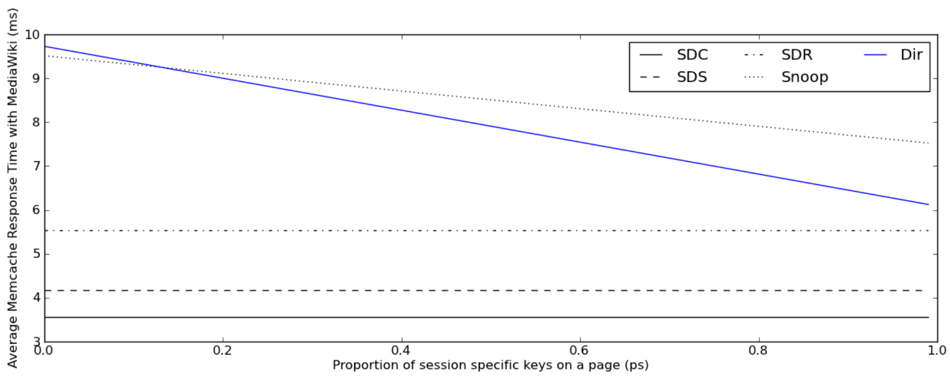

Figure 3.4: MediaWiki profile under different ps values.

OS delay, in the Fig. 3.3 plots. 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 Fig. 3.3, as ps increases the latency for the location-aware schemes improve. 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 Fig. 3.4 using a fixed switch latency of $1.0ms$ and our MediaWiki usage profile.

Predictably all 3 location-averse schemes (SDC, SDS, and SDR) exhibit no change in performance as ps increases. Snoop and Dir improve as ps increases, but are not able to out-perform the other architectures in this situation.

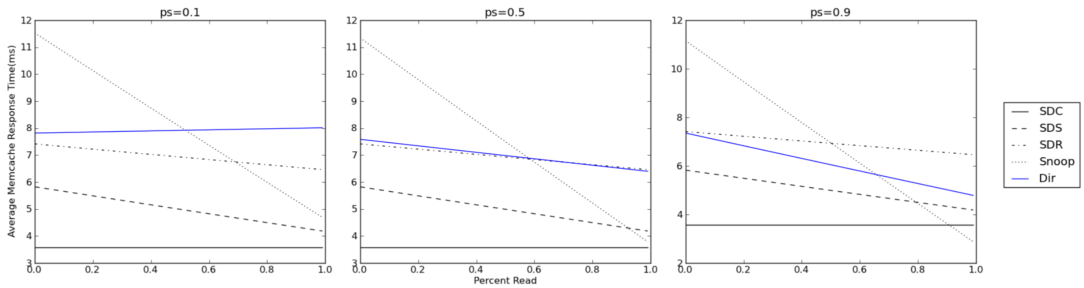

Figure 3.5: Varying read/write ratio and ps values.

So far we've analyzed performance using MediaWiki's usage profile. Next, the more general case where the 19 commands are split 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 3.5 varies the read/write ratio while looking at three ps values.

With high read/write ratios, Snoop is able to outperform SDC, here when $switch = 1.0$ ms and $ps = 0.9$ at $rw = 0.9$.

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 next, its low network usage is beneficial.

3.7 Experimental Results

To validate our model and performance estimation formula, we implemented our alternate Memcache schemes and ran a real-world web application, MediaWiki[58], with real hardware and simulated user traffic. MediaWiki allows users to create, edit, search, and view content through a web interface. MediaWiki was chosen due to its built-in Memcache support, freely available source code, and mature codebase which powers Wikipedia.

Multiple caching configurations are possible depending on the intended installation and use. 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 containing links to other pages, browsing 20 random pages, 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 and 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

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%

Parameter	Full	Limited	Session
ps	.56	.59	1
rw_{cmd}	.51	.49	.5
rw_{net}	.61	.78	.54
Avg. net size (bytes)	870	973	301

Table 3.6: MediaWiki usage for each configuration

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. Section 5.5 discusses network traffic measurement in detail. 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.

MediaWiki was measured in all configurations using MemcacheTach with results presented in Table 3.6. Only non-zero Memcache commands are listed for brevity.

3.7.1 Latency

To predict latency we require two measurements of network performance, *switch* & *base*. These were found using an SDS run and calculating the relative time difference between Memcache commands in-rack (l_{local}) and a neighboring rack (l_3) and subtracting the delay from limited bandwidth. For the 100Mbps network, $switch = 0.21$ms and $base = 3.0$ms. The 10Mbps network had, $switch = 0.22$ms and $base = 4.0$ms. The resulting predicted and observed per Memcache command latences are given in Table 3.7.

The resulting predicted and observed per Memcache command latences are

	Predicted Latency (ms)			Observed Latency			Predicted Latency			Observed Latency		
Scheme	Full	Limited	Session	Full	Limited	Session	Full	Limited	Session	Full	Limited	Session
	100Mb/s Central Switch						10Mb/s Central Switch					
SDC	3.5	3.5	3.4	3.5	3.9	3.2	5.2	5.2	4.7	12.1	13.9	3.5
SDS	4.2	3.6	3.6	3.8	4.1	3.6	5.3	5.4	4.8	20.1	20.6	4.6
SDR	5.7	5.0	3.5	5.1	5.4	5.3	7.0	7.4	4.8	35.6	29.5	9.2
Snoop	5.9	5.2	5.1	6.1	6.6	7.3	6.3	7.0	6.9	15.6	17.0	10.9
Dir	8.4	7.5	8.5	5.5	5.8	5.8	9.2	9.5	11.6	9.3	9.9	6.3

Table 3.7: Expected and Measured Memcache Latency

given in Figure 3.7. Each configuration was run 4 times and averaged.

In the case of fast switching, SDC was the best predicted and observed performer. The location-aware schemes, Dir and Snoop, both don't fit the expected values as close as the others. This is likely due to the interpreted nature of the architecture logic in PHP. Future work will explore native C implementations.

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

3.7.2 Network Load

Using the formula developed in Section 3.5, 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 Table 3.8 with 4 samples per configuration.

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

Table 3.8: Expected and Measured Network Load

Notice SDR'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. Future work measuring network utilization for Memcache using UDP would be a good next step, as has been investigated by Facebook[81, 99].

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.

3.7.3 Review

These results show that the model, application profile, and performance estimation formulas 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 3.6, 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.

3.8 Discussion

3.8.1 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 SDR'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.

3.8.2 Multi-Datacenter Usage

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. Assume an l_3 value of 40ms, a best case CA to NY latency, with l_1 = 5ms inside the datacenter. For Dir's directory we assume it spans both datacenters like SDS. See Fig. 3.6 for the plotted comparison.

Here the difference between locality aware and averse is more pronounced.

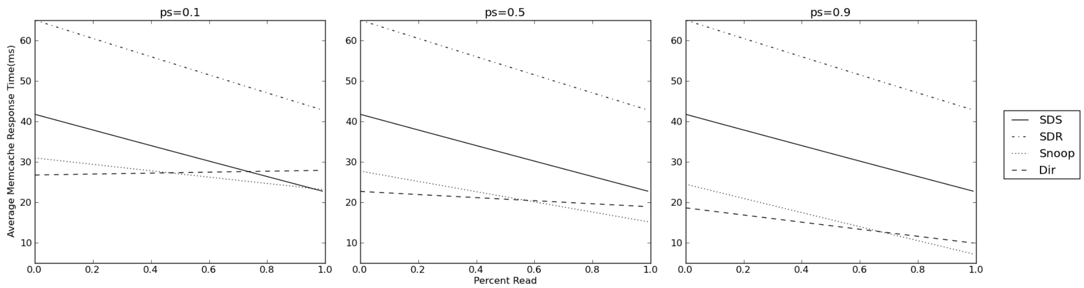

Figure 3.6: Varying read/write ratio and ps values with an East and West coast DC

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.

3.8.3 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 selectively replicating frequently used data could save space while increasing speed. Snoop and Dir could be easily 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.

3.8.4 Object Expiration

In Memcache, objects are generally removed in a least-recently-used manner if the allocated memory is consumed. In a standard deployment this works well, but in our case where meta information is separate from data, the 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.

3.8.5 User Space Caching

As mentioned in Section 3.2, 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, using the standard Memcache protocol. Either way, transportation costs and protocol overhead are incurred. By moving the Memcache server, emulating it, or allowing shared memory within the same process space as the client, the extra costs could be eliminated. Object serialization overhead may also be reduced. Research done in 2009 using memory mapping between virtual machine (VM) hosts in a cloud environment demonstrated that an 86% reduction in Memcache latency for read operations is possible, supporting these methods [110].

3.8.6 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. 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 Memcache balancing system.

3.8.7 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 based on the number of network devices traversed. This can be easily computed automatically in some cases. 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.

3.9 Summary

Placing cache data close to where it will be used, as demonstrated, can lower latency and reduce network utilization inside a web datacenter. This leads to faster page response times, higher capacity, and more satisfied users.

Data dependencies can be identified for a running application using MemcacheTach and exploited, leading to better cache performance. Using application usage data, as well as the developed model and formula, the developer can estimate performance for alternate Memcache caching strategies.

Chapter 4

Tentacle

This chapter explores the limits of data locality by augmenting web applications to generate as much content near the web user as possible. Processing and data storage are distributed to edge servers in close proximity to the end users. Temporal and relational dependencies limit the extent to which an application can be distributed. Relaxing these dependencies, when possible, can allow a non-distributed web application to be used in a distributed way. The goal is to lower end-user page latencies as much as possible by serving dynamic content locally.

As discussed in Section 1.3, dependencies inside a web application can limit its ability to be distributed. Some applications, such as a webmail client, do not have any dependencies preventing it from being distributed. In this case, temporal dependencies exist, but relational ones do not: a user's email data is not used by anyone else in the system. Other applications, such as a bulletin board system, contains both temporal and relational dependencies that limit their ability to grow. This chapter concentrates on the harder latter case where data is needed by many users.

Pivotal to the work presented here is the exploitation of allowable consistency relaxation inside a web application, otherwise known as temporal dependencies, freshness of data, or bounded staleness[23, 98]. This chapter presents a solution that attempts to lower user latencies by distributing computation geographically for a web application which was not developed to do so. More specifically, it is designed to benefit standalone web applications using a single DBMS for all storage. Large web businesses such as Google, Amazon, and Facebook have designed their web systems with global reach and scalability in mind. This chapter demonstrates that significant performance gains can be made without application redesign by selectively relaxing consistency requirements.

4.1 Background

Presented here are topics and current practices which are necessary for this system. Web application architecture, consistency relaxation in web applications, and database replication techniques are covered. Similar, more general, topics are covered in Chapter 2.

4.1.1 Web Application Architecture

The structure of a web application and how it uses data dictates the application's temporal and relational dependencies. Numerous web architectures are possible, including Model-View-Controller (MVC) [50], Service Oriented [79], or the layered approach [40].

This chapter is aimed at applications using the layered approach to application design. Three layers are typical, with the bottom layer storing state in a RDBMS. Business logic implementing a general application actions sits above the RDBMS. On top is the view layer which generates the HTML and uses the business logic layer to interact with the application. Each layer could be executed on separate machines, but typically the business and view layers are contained within the web server itself. Thus, even though a three layer architecture is used, there are only two parts: a stateless web server and a database to store data. In this configuration the computationally intensive application logic can be distributed over many servers.

4.1.2 Relaxing Consistency

Data dependencies in web applications must not always be strictly followed. As detailed in Section 1.3, variations in weakly consistent policies may be allowable.

The recent NoSQL storage movement uses eventual consistency resulting in scalability and performance gains. Rather than the typical SQL language interface, the NoSQL interface uses simpler *set* or *get* methods following a key/value structure. Due to the limited interface and lack of built-in comparison mechanisms, complex data queries such as the SQL's `JOIN` command must be done in the application, or use some other storage system. By relaxing temporal and relational dependencies, as well as removing data computation facilities inside the database, scalability and global reach can be accomplished. Unfortunately, using a NoSQL-type system requires drastic changes to the application or a complete rewrite [65].

This chapter explores how simple consistency relaxing modifications to a RDBMS-backed web application can yield a more distributed system. Tentacle exploits consistency at the database query level. As an example, consider three read queries in an e-commerce web application:

1. Retrieve the user's current shopping cart.

2. Retrieve a random banner advertisement.

3. Show all available products.

For a specific application, displaying a stale banner advertisement may be acceptable, or a slightly outdated product list. Conversely, the user will quickly notice an error in their shopping cart.

Each of the above queries can and are typically retrieved using a single SQL query. This demonstrates an opportunity to expose consistency relaxation at the individual SQL query level. For an in-depth discussion on how freshness varies between applications, see the LazyBase work[23].

4.1.3 Database Replication

Due to relational dependencies in a web application's database, scaling can be difficult [82]. Scalability is the ability for a system to continually increase performance as computing or storage resources are added. Scaling vertically, by using a faster machine, has an upper limit on performance based on hardware technology. Scaling horizontally, by adding more machines, is optimal but not always possible due to the dependencies discussed in Section 1.3.3. Horizontally scaling a database is difficult because any read query may require a `JOIN` between any table, requiring all data to be stored on the same machine. Similarly, modifications may require the entire dataset to be available and queries serialized so temporal and relational dependencies are upheld.

Scaling RDBMS's horizontally is a rich research area, but typically performance, ACID properties, or relational dependencies are degraded for size. Using database replication, additional performance can be gained without changing the application significantly.

Database replication separates read queries from write queries, and directs each to a separate database system. Reads query a copy of the write's database. Many duplicate databases can exist allowing read queries to be distributed over all. Database writes can be handled in a synchronous or asynchronous manner. A synchronous system will block a modification query until all replicas acknowledge the change, and an asynchronous will not [5]. Modification conflicts between separate databases are caught before a query is committed in synchronous system whereas conflicts must be repaired after-the-fact in non-synchronous systems. A common way to mitigate conflicts (consistency violations) in non-synchronous systems is to modify only a single database, thereby serialized the modification queries [82]. All replica databases can then be used for read-only queries.

When replication is used in a non-synchronous replication scheme, care must be taken to avoid temporal dependency violations. A delay exists, called replication lag, between when a write query is performed and when a subsequent read on a replica will reflect the change. The exact lag time depends on server load, network conditions, or distance induced network latencies.

This chapter concentrates on database replication using MySQL, a commonly used database for web applications. Two general types of replication configurations are possible in MySQL: Master/Slave and Master/Master [59].

1. **Master/Slave:** This scheme directs all database writes to a single master database. Temporal dependencies are assured for all writes. Master changes are sent to one or more slave databases by using some replication technique [82]. Read requests can be sent to any of the slave databases,

increasing performance by distributing the read load. All slave databases contain all data and each must be able to handle the entire write load of the system.

2. **Master/Master:** Here two databases act as a master and a slave to each other. Writes can be directed to either, and subsequently replicated to the other. To prevent a write from being applied multiple times, a source database identifier is sent with each replication command. Commands containing an identical source identifier to the database's own are not applied or forwarded. More than two databases can be used if placed in a ring topology. Reads can be sent to any database. Temporal dependencies may be violated due to duplication lag. Configurations having higher lag times will be more prone to such violations.

4.2 Tentacle

Tentacle is a database middleware client, forwarding agent, and caching system which uses per-query consistency information. Its goal is to minimize page latency for end users while assuring consistency requirements.

The Tentacle system lowers end-user latencies by using local data whenever possible. Edge datacenter nodes are placed throughout the Internet, each datacenter containing database and web servers.

Of all datacenter nodes, one is selected to be the master with all database writes applied to it. All other installation's databases become slaves to this master using MySQL Master/Slave database replication. Database reads are directed to the local database when possible, otherwise routed to the master database, though rarely as discussed later.

Consistency requirements are conveyed by modifying the application's query template calls. A query template is a string which contains the majority of the query, with important information inserted at runtime. The templates are custom in nature to their purpose, such as retrieving a shopping cart, or buying an item. Thus, this presents a good location to specify consistency requirements based on the template and surrounding application logic.

To quantify the consistency requirements per query template, Tentacle uses a time measurement in positive or zero seconds. For read queries (`SELECT` statements) this value signifies how stale the result can be. For example, a shopping cart query would have a value of 0, to show the current state, whereas a product query might use 30 seconds. Thus, if a change is made to the product listing all clients will see the modification in 30 seconds or less. Choosing a consistency requirement time is arbitrary and dependent on the desired application's performance. Care should be taken to assure user-induced write actions follow read-your-own-writes consistency. If consistency requirements are in doubt, a value of 0 can be used.

Similarly, write queries (`UPDATE`, `INSERT`, and `DELETE` statements) are labeled, but signify the maximum delay time allowable until the master database is updated. Any time greater than 0 is considered an asynchronous update query which will not return any response code. In many cases this is possible, such as incrementing the view count of a specific banner advertisement, or updating the number of online users. In other cases, such as final order placement, no lag can occur.

Because Tentacle aims to lower response times for diverse users, it must avoid any possibility of slowing the application over a centralized approach.

The central master database must handle those queries which do not allow any consistency relaxation (0 consistency time value), possibly requiring multiple round-trips from the edge per page. A forwarding feature is used to send the entire HTTP request to the central node for all processing in this situation. Thus, the maximal time a page request will be returned is no greater than a centralized approach.

Session data, originally stored in the database, exhibits no relational dependencies with other data and can be removed. Storing sessions locally at the edge allows faster access and does not hinder the application as long as the session is consistently routed to that edge location. A local database or object storage system such as Memcache can be used for session handling. If an HTTP request requires forwarding to the central node, the local session information is sent as well, with any session update returned with the response. Forwarding code should be inserted in the application before any data is sent to the user so that new HTTP headers may be inserted correctly.

4.2.1 Architecture

A sample database replication and query routing scheme overlaid on possible locations are shown in Figure 4.1. Edge nodes should be placed at maximal network distance from each other to minimize latency for the most users.

Tentacle consists of 5 components which work together to provide a distributed web application. The 5 components are listed and described below.

1. **Local Session Storage:** Sessions should be stored in the edge location, if not already done so. This can be done with a local database or a Memcache session system. Sessions and application data do not mix within the

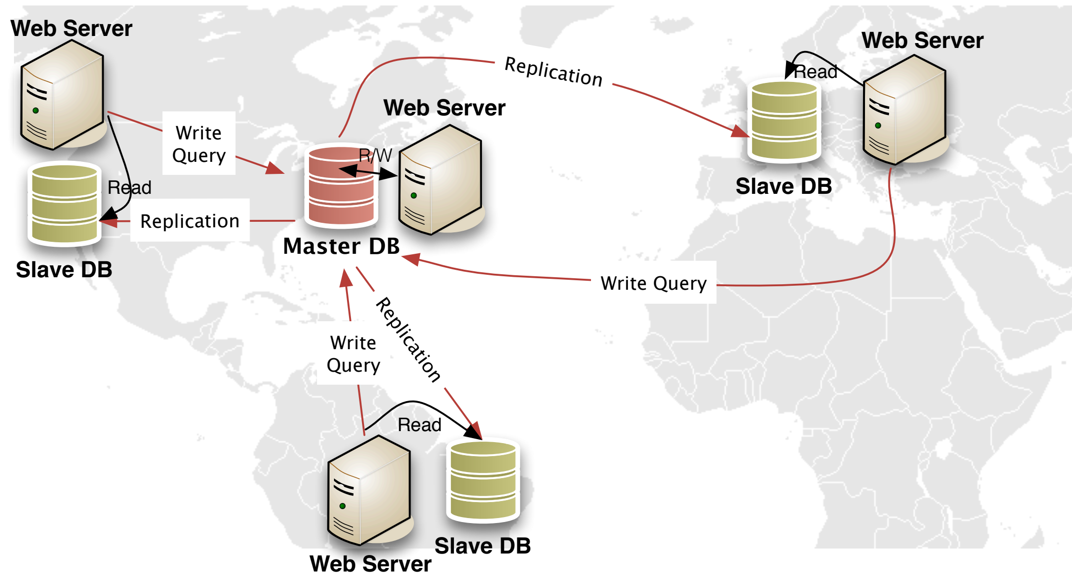

Figure 4.1: Database Replication & Query Routing

database allowing easy separation.

2. **Database Replication with Local Reads:** By replicating the master database, read requests with a time larger than the slave lag time can query locally.

3. **Local Database Query Caching:** To speed local reads, query/response pairs are cached in Memcache locally with an expire time equal to the allowed consistency time minus the slave lag time.

4. **Asynchronous Writes:** Write queries with consistency times larger than the WAN latency to the master location are queued locally and periodically applied to the master database. No return status code is provided to the application.

5. **HTTP Forwarding:** Queries with low (or 0 time) consistency times must be sent directly to the master node. To prevent multiple database round-

trips to the master node, any page containing strict consistency times are forwarded before any data is sent to the user. A list of URLs and minimal consistency times are kept in each web server for this purpose. Updating the list can be done by Tentacle at runtime or by an administrator.

The current version of Tentacle consists of four files `tentacle.php`, `loop.php`, `tentacle-daemon.php`, and `remoteaccess.php`, listed in Appendix H, I, J and K respectively. The PHP class `Tentacle` is defined in `tentacle.php` and contains the majority of the functionality. This class is called by the web application and preforms the HTTP forwarding and database access. The `remoteaccess.php` file is used on the central HTTP servers to accept HTTP forwarding requests and responds with the page result. The two remaining files, `loop.php` and `tentacle-daemon.php` support background SQL update commands. When Tentacle allows, SQL update queries are saved to a temporary file(s) on the local webserver. These two files periodically check the temporary file(s) for updates and send the query to the central server's database for immediate application.

4.2.2 Operation

The flowchart for a single page request using Tentacle is given in Figure 4.2. A walkthrough of the process follows.

An HTTP request enters the edge node and the forwarding list is queried for any URL matches. If so, the stored lowest latency time value is compared with the current replication and write queue lag times. If current system performance can not satisfy the requirement then the HTTP request is bundled with the current session state and sent to the central node for processing. When the

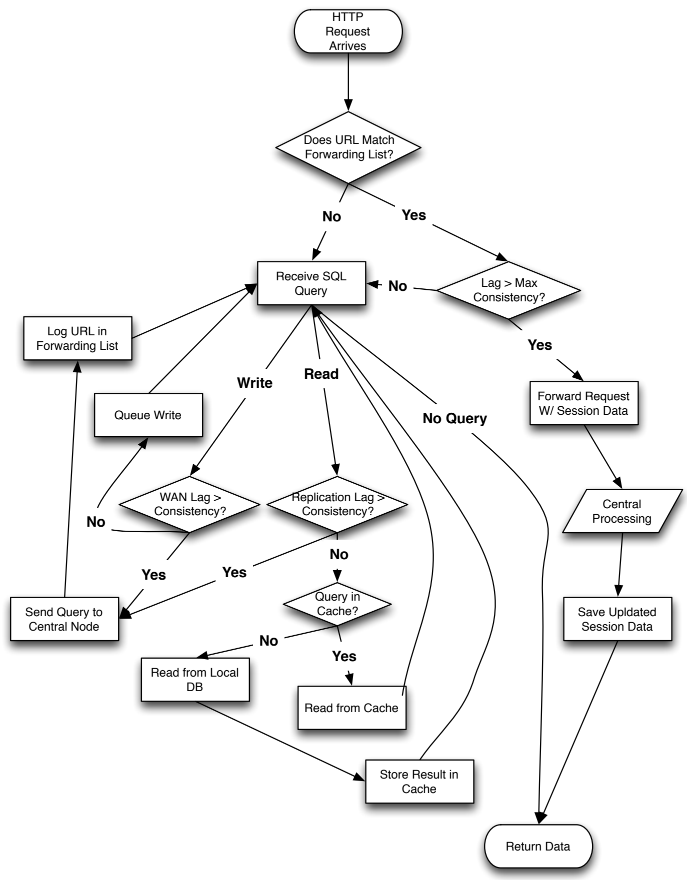

Figure 4.2: HTTP Request Processing Using Tentacle

response returns, the updated session state is unbundled and saved locally, followed by returning the page data to the user.

If system performance is better than the required consistency time, the request is handled locally, just as URL's which did not match the forwarding list. The application code is executed at the node and Tentacle waits for SQL queries. Queries are separated into read and write as they arrive. Those read queries which can be handled locally, having a consistency query time higher than the current replication lag time, will first query the cache. If a match is found it will be returned, otherwise it will query the local database followed by storage within the cache. To assure correct consistency relaxation, a cache expiration time is used for each item in the cache equal to the query's consistency time minus the current replication lag. Thus, if a cached query result is found it must be still valid according to its allowed lag time.

A read query not able to be handled locally are sent to the central node's database, followed by logging of the URL and consistency time in the forwarding list. The next time the same URL arrives the entire HTTP request will be sent to the central node rather than suffer possible multiple round-trips to satisfy database requests.

Local write queries having a consistency time of zero are considered synchronous and will be sent to the central database for processing. Queries having a consistency time greater than zero but less than the current WAN latency must also be sent to the central database. The remaining write queries are buffered locally and applied to the master database at a later time. These buffered queries are considered asynchronous and will not return any value. SQL queries are repeatedly handled until the page is complete.

All edge nodes except for the central node preform the process outlined above for all web requests. The central node uses Tentacle's query caching system only, with all remaining database requests handled by the database.

The central node must also handle forwarding requests from edge nodes. These requests contain HTTP requests for which the edge node could not satisfy given their consistency requirements. Session information, stored locally at the edge nodes, must also be sent to the central node in order to it properly handle the request. Any session changes are sent back to the edge node with the page response. Back at the edge node, local session data is stored and the page data

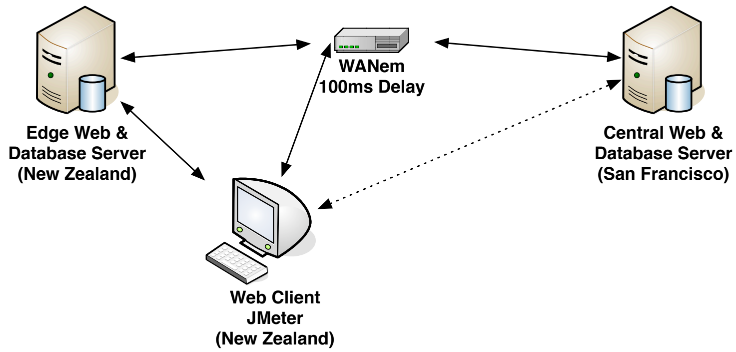

Figure 4.3: Evaluation Hardware Configuration

is sent to the user.

The forwarding of HTTP requests from an edge location is a common method of accelerating dynamic web applications[113, 12, 71]. As mentioned in Section 1.2, establishing the TCP link can take one round trip. By keeping a persistent TCP connection open to an origin server, tunneling can reduce latencies. Future work on Tentacle will implement persistent connections.

4.3 Experimental Results

Tentacle was evaluated on a simulated Wide Area Network (WAN) environment using a modified e-commerce web application and timed automated browsing sessions. Simulation was done using multiple VMware virtual machines communicating over a simulated network. Figure 4.3 shows network topology for the simulation.

Two server machines were emulated, each able to serve web-content via PHP, provide a MySQL database, and handle Memcache traffic. See Section 5.7 for details.

Communication was delayed 100ms between the two via a WANem[45, 60] installation which approximates the latency seen across the Pacific on Verizon's network [103]. Bandwidth was unlimited. For simplicity, these are labeled as NZ (New Zealand server node) and SF (SanFrancisco server node).

Browsing sessions were directed to the NZ host to simulate an edge serving location, through a delayed connection to the SF host, or directly to SF to simulate minimal network latency for comparison. The SF location served as the central web and database location.

4.3.1 osCommerce

The popular open source e-commerce PHP application osCommerce [32] was chosen as an example application. A business can present their sellable items online, manage their shop, and sell to users using many online payment schemes using this MySQL backed application. osCommerce was chosen for its mixture of strict and loose consistency requirements. System requirements for running osCommerce are low, only requiring PHP and a PHP supported database. osCommerce 2.3.1, downloaded in January 2012, was used.

The application consists of two parts, a user and admin codebase folders. This allows easy folder-based security as well removal of administration code from exterior facing servers if desired. User code was concentrated on in this chapter. Modification of admin code is possible, but would have no impact on the customer's latency.

osCommerce's session handling uses a database, which needed modification. The session table is not accessed by any other module besides the session, nor will any session require access to another session. This allows a distributed session storage system to be used without effecting the operation of the application. The session module was modified to use a local Memcache server for all session handling.

Database access throughout the application uses functions declared in a single file, `database.php`, allowing other database engines to be retrofitted easily. Tentacle's database querying hooks are placed here. The old query function, `tep_db_query($query)`, was replaced with a new function requiring a staleness value, `tep_db_query_stale($consistency_time,$query)`. The entire codebase was then searched for references to the old function and replaced with the new after choosing a staleness value.

To enable forwarding of requests, the Tentacle class is instantiated early in the application before any data is sent to the client, and the `doForward()` function called. This function searches the forwarding list for matches against the current URL and preforms forwarding if necessary.

4.3.2 Application State

osCommerce is an e-commerce application with the majority of the content supplied by the shop owner, such as product categories and all details of products. Evaluation was done from an end user's perspective, so a prefilled shop was used. 15 categories were created each containing between 1 and 20 items. Items contained a textual description to represent a non-trivial database load.

4.3.3 Simulated Traffic

Traffic emulated a typical visitor browsing categories, items, and finally checking out. 50 users were simulated using Apache JMeter 2.6 with at most 10 simultaneous sessions. Each session navigated to the home page, selected a category, and looked at an item description, followed by viewing other items, categories, or the home page. One tenth of the items viewed were put in the shopping

Page	Proportion	Page	Proportion
Enter Category	38%	New Account Form	1%
View Product	38%	Login Page	1%
Home Page	7%	Attempt Login	1%
Add to Cart	4%	Checkout Shipping	1%
Check Reviews	4%	Click Next	1%
Check Shopping Cart	1%	Select Payment	1%
Checkout & Login	1%	Confirm Order	1%
Create New Account	1%		

Table 4.1: osCommerce Page Access Proportions

cart, averaging 5 purchased items. The session then checked out by creating a new account, selecting a shipping method, supplying payment information, and finally finalizing the order. Each session viewed 131 pages on average. The dwell delay between pages is set to a Gaussian distribution with average of 1 second and standard deviation of 300ms. HTTP `Keep-Alive` was enabled to reduce reconnection delays between the simulated browser and the webserver. Table 4.1 shows the proportion of pages viewed per session.

4.3.4 Results

Multiple configurations were tested using the above layout, shop state, and user load. Table 4.2 presents the aggregate results. Five configurations were chosen. For comparison the `LAN No Optimization` is our baseline best-case configuration where the user, web server, and database all exist in SF with no introduced latency. The `All WAN` configuration moves the user 100ms away to NZ with the datacenter in SF. `Syncronized Writes` uses an edge server in NZ where all reads are local, but all database writes communicate with the central database in SF 100ms away. Multiple round-trips may be introduced per page

Configuration	Latency (ms)	Median	90%	St.Dev	Samples
WAN w/Tentacle	212	131	413	320	6560
Tunnel All	937	890	1051	322	6535
Sync. Writes	1732	1750	1947	476	6534
All WAN	681	670	791	97	6585
LAN No Optimization	257	222	383	277	6556

Table 4.2: Aggregate osCommerce Latency with 100ms WAN

if multiple write queries are issued. `Tunnel All` encapsulates all HTTP traffic at the edge and tunnels it to the central datacenter in SF for all processing. Lastly `WAN W/Tentacle` uses an edge server in NZ, central datacenter in SF, and uses all five features of Tentale.

The results show that even in a WAN environment Tentacle can decrease latency over a local configuration. This is due to practically non-existant delay for the majority of write queries and the caching of read queries. The speedup is significant enough to offset the forwarding cost for some of the pages as shown. Note the near 400ms slowdown between `LAN` and `All WAN`,when all trafic is sent over the 100ms WAN. This highlights the inefficiencies of TCP in short-lived connections over high-latency links. Tunneling has the possibility of reducing the overhead using persistent TCP connections, though our `Tunnel All` configuration did not use persistent connections from the edge to central servers. This may explain the slower `Tunnel All` configuration compared to the `All WAN` where HTTP `Keep-Alive` was used. Adding persistent connections and optimizing the session encapsulation/decapsulation would lower the tunneled latency for better `Tunnel All` and `WAN w/Tentacle` configurations.

Sending all query writes to the central database is the worst performing option as every page request has at least one write. Using HTTP forwarding

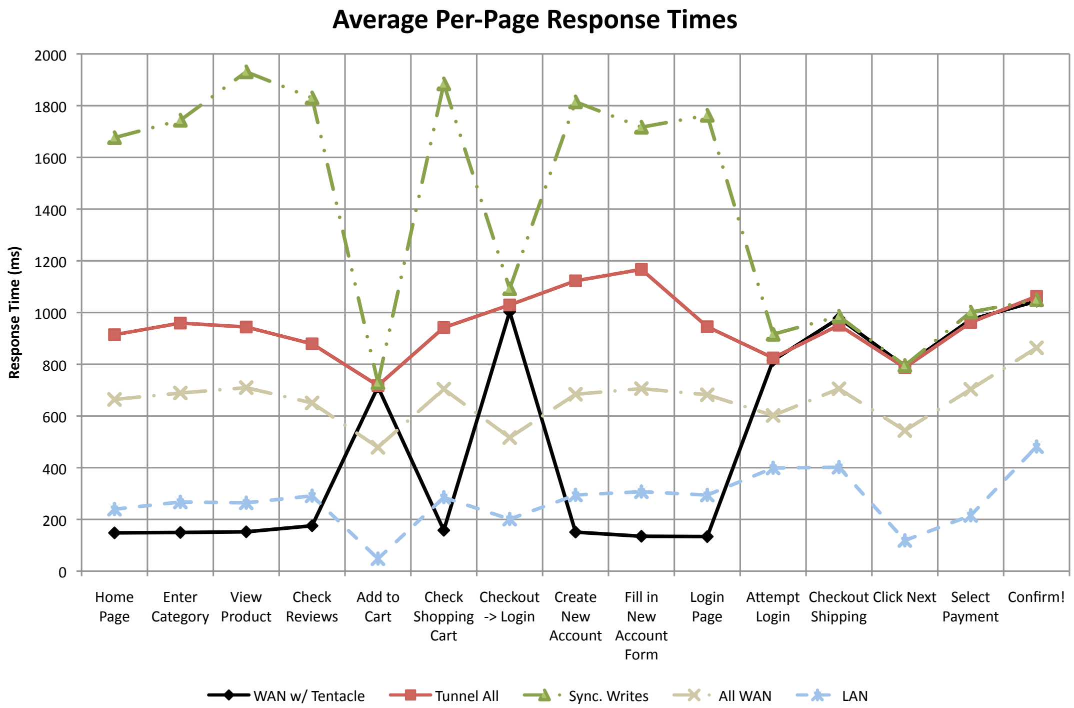

Figure 4.4: Average Per Page Response Times (ms)

to mitigate this write latency is unique to this work. In this application, few of the common write queries are time sensitive, updating banner advertisement counts or removing old sessions, allowing Tentacle to queue these and respond to the query call much faster. Figure 4.4 shows per-page average response times. Due to Tentacle's superior performance for the frequently accessed `Home Page`, `Enter Category`, `View Product`, and `Check Review` pages the aggregate average response time is lower than that of any other configuration. Many of the other pages, including `Add to Cart` and the checkout procedure, are slower with Tentacle than the user directly contacting the central datacenter due to the session and request encapsulation.

These results show latency can be lowered for geographically diverse users for some pages in a dynamic web application by selectively relaxing consistency constraints. Other configurations and consistency relaxation schemes are possible, discussed next.

4.4 Discussion

The five components of Tentacle listed in Section 4.2.1 describe a single solution to lowering latency. Alternate consistency rules, different database configurations, cache pre-warming, stampede mitigation, and dynamic central database movement could be used to increase performance in specific applications. Each application may need features enabled or disabled for optimal performance.

4.4.1 Alternate Consistency Specifications

Tentacle relaxes temporal consistency for individual query templates in the form of a relaxation time. This requires some knowledge of the query calling pattern in the application in order for read-your-own-writes consistency to be enforced. Because adding an item to your cart shows the shopping cart page, all read queries must use strict consistency, otherwise the new product will not be shown. Had the user just navigated to the `View Cart` page otherwise, the request will still be serviced by the central datacenter and suffer the increased latency.

An alternate scheme could specify which previous update queries are required to be reflected in the local database before a local read query can be used. In the case of the `View Cart` page, local data could be used, if possible, to prevent an unneeded central database query or forward. A combination dependency and time-related consistency scheme would support this.

4.4.2 Database Considerations

As described, Tentacle uses entire-database replication. If scalability and high performance are needed, typical database sharding techniques could be used with Tentacle [82, 40]. By moving unrelated database tables into separate databases, write load can be split between the two increasing capacity. This exploits relational dependencies in the database's schema. Tentacle would need to be modified to route queries to the correct database.

4.4.3 Cache Pre-warming

Most caches store data after the first miss. Subsequently, the cache will provide the data until it is invalidated or expires. In this case the cache can be application-logic independent. Optimally, all read queries will be handled by the fast and scalable caching layer. Tentacle already has knowledge of page URLs, read queries they contain, individual query consistency values, and a database query time for each query. If a graph of accessible pages in the application could be provided to Tentacle, it could predict future read queries and their cost, if not already cached. Tentacle could pre-issue read queries, filling the cache, to minimize user latency based on past application use.

4.4.4 Stampede Mitigation

Tentacle presents a database middleware solution that should provide consistent performance. If computationally expensive read queries exist on a heavily used website, a possibility exists for periodic high-latency events due to the caching layer. Cache entries, representing database read queries, will expire after a set time derived from the query's consistency requirement. Once expiration occurs subsequent queries will be satisfied by the database until the cache is refilled. While the database is queried, other cache misses could occur for the same query and issue identical database reads. The result is a deluge of identical read queries which persists until the first cache miss query can fill the cache. Long-running queries are more susceptible to this behavior, known in the industry as a dog pile, thundering herd, or stampede [42]. Mitigating this behavior would beneficial to Tentacle.

4.4.5 Pattern-Based Homing

For consistent-strict pages, encapsulation will direct the query to the central datacenter. To minimize the average response time for all users, the central datacenter location could be dynamically moved. Due to full replication, each edge and central location must be able to handle the full write load of the system. Thus, any should be able to be the central datacenter. As user load shifts, perhaps hourly or daily, the central datacenter could shift in response to minimize average user latencies.

4.5 Summary

Tentacle lowers response time for geographically disperse users by serving dynamic content locally when possible. Placing all data at edge locations is not sufficient to reduce latency due to consistency constraints. By exploiting allowed consistency relaxations for individual query templates, local stale data can be used safely. Using HTTP forwarding, Tentacle is able to limit the page latency to one round-trip time to the central datacenter or less for all page requests.

Chapter 5

Experimental Results Details

This chapter presents tools, technologies, and techniques used to evaluate the contributions above. Chapters 3 and 4 provide a discussion of the contributions, but this chapter describes in detail how they fit into a complete web-serving system, and how the results were measured.

A working knowledge of web technologies is assumed, such as TCP/IP, HTTP, HTML, and the client-server model of the web. Tools which simulate web users are discussed first, followed by the serving components, and ending with specific contribution measurement methods.

5.1 User Simulation & Response Time

Reducing end-user page latency is our goal. User behavior varies, as does their computing resources, making realistic simulation and measurement a difficult task. Both contributions rely an users visiting an HTML page and clicking a link to another page. As discussed in Section 1.2, the process of viewing a complete page typically requires multiple resources in addition to the original HTML. For simplification, only the page HTML is requested and the response time of that HTTP request is measured. In our case all additional resources were static and could be serviced using an alternative system such as a CDN.

Latency for static content falls outside the scope of our work.

Simulating users for both contributions was done using Apache JMeter[1]. By specifying a browsing sequence using this tool, any number of simultaneous web clients can be simulated. Browsing sequences are described in Sections 3.7 and 4.3.3. Many features are supported such as sessions, page validation, page content searching, conditional statements, and delays. JMeter monitors per-request response times, aggregates and graphs the results, and allows saving to an external CSV datafile. Our page latency statistics were generated by a custom Perl script using this datafile.

JMeter was used on a separate workstation than the serving system. This configuration assured JMeter's overhead was not inadvertently measured, while also including realistic network overhead in the latency measurements.

To simulate users on a WAN for Tentacle, a network delay device, WANem[45, 60], was inserted between JMeter and the serving system, as described in Section 4.3.4. This approximated users from New Zealand browsing a site located in San Francisco; about a 100ms round-trip time as reported on the Verizon network in March 2012[103].

5.2 Web Server Configurations

All web content was served using minimal CentOS 5.x operating systems, configured for optimal webserver performance. All packages and services not used for web serving or administration were removed. Apache httpd 2.x responded to HTTP requests, with PHP 5.3.x handeling all dynamic code as an Apache module. The APC 3.1.x PHP module was installed to speed PHP by caching

[1]Apache JMeter - http://jmeter.apache.org/

PHP bytecode for later use, leading to a 30% reduction in page generation latency, as suggested by [101]. A MySQL database was accessed in php using the php-mysql 5.3.x module.

NTP[1] was used to keep server time correct, vital to merging multiple log files. NTP's behavior was not consistent when used in a virtualized environment due to multi-hour time offsets when pausing and resuming virtual machines. The `tinker panic 0` directive was inserted into the `/etc/ntp.conf` file to allow NTP to synchronize time in this configuration[105].

To simulate a web-farm for our Memcache evaluation, a set of 23 identical Dell Poweredge 350 servers servers was used, with specifications given in Section 3.7. Installation and configuration of all machines were automated using PXE boot[2] and a CentOS kickstart[3] file, customized per machine based on its MAC address. This allowed static IP addresses and hostnames to be assigned with all other configurations identical. After operating system installation, a shell script using password-less SSH was used to run commands on all servers simultaneously. Files were pushed to all webservers via a looped `rsync` UNIX command.

Of the 23 Dell servers, one was chosen as the web load-balancer. HAProxy[4] 1.4.18 was installed and forwarded web requests in a round-robin fashion to the web servers. Sticky sessions were enabled, allowing sessions to be routed to the same webserver for the duration of the session. Sticky sessions was enabled using HTTP cookie detection in HAProxy's configuration file.

Tentacle was evaluated using the same web serving software configuration

[1] http://www.ntp.org/
[2] http://wiki.centos.org/HowTos/PXE/PXE_Setup
[3] http://wiki.centos.org/TipsAndTricks/KickStart
[4] http://haproxy.1wt.eu/

above, but on two VMware virtual machine instances with 4GB ram and four processing cores each. Both contained web and MySQL database servers, with one as master and the other slave. A Dell Precision 690 with dual Xenon processors and 12GB RAM was used as a host machine, running Fedora Core 16 and VMware Workstation 7.1.5.

5.3 Database Configuration

A fast server was used for the database, moving the typical bottleneck from the database to the webservers. MySQL 5.1.x was used, running on an CentOS 6 server with a dual-core AMD Opteron 248 processor and 756MB RAM. A 200MB RAM disk was used to store the database's data files to increase performance as much as possible. During all trials the CPU utilization stayed low, showing the database was not the bottleneck.

For each trial run the database was restored to an identical state. This was done by by halting the database server daemon, replacing the current database data files with a previous state, restarting the database daemon, and finally running the SQL command `OPTIMIZE ALL TABLES`. As a positive site effect, all persistent TCP database connections from the web applications were terminated and forced to reconnect on the next run. Without reseting the TCP connections the per-page latency results would not have been consistent.

5.4 Memcache Performance Measurement

Section 3.4 discussed MemcacheTach's purpose and use briefly. This section goes into more detail about its use and implementation.

MemcacheTach's purpose is to capture and analyse the performance of a

Memcache system. It consists of two parts: (1)Data capture, and (2) Data analysis.

5.4.1 Data Capture

The `Memcache_Logging` PHP class (memcache-logging.php in Appendix F) is a wrapper around a standard php memcache module. As written, it uses the PECL `Memcache` PHP module rather than the `Memcached` module. To use this logging class, import the class and use it like the PECL `Memcache` PHP module. By default the log file is saved to `/tmp/stress_validate_log.txt` on the local filesystem, changeable by using the `log_file($filename)` function in the class.

All Memcache commands which would modify the cache are logged. For reliability, a log entry line is appended to the file after every command, with a single writer mutex attached to the file-handle. Care should be taken in a heavily parallel web serving system that a single log file mutex does not become a bottleneck. To mitigate such a possibility, per-thread log files could be used, or some other high performance logging service such as Ganglia[1]. In our case mutex contention was not an issue.

The log file consists of one Memcache request per line, containing ten fields each, stored as a serialized PHP array. Only commands which modify the cache are logged. Below are descriptions of each field in order of their appearance:

1. **Timestamp:** Server time in tenth of seconds when response was received from the Memcache server and log was written.
2. **Session Identifier:** First two characters of the hashed cookie value for the current page.

[1]http://ganglia.sourceforge.net/

3. **Webserver IP Address:** Dotted decimal string representation of webserver's IP.
4. **Memcache Server IP:** Dotted decimal string representation of the Memcache server contacted for this request.
5. **Command:** Memcache command sent (=get=, `set`, ...).
6. **Key:** Key used for storage or retrieval command.
7. **Data Length:** Number of characters of data portion stored or retrieved.
8. **Found:** 0/1 if data was found, or storage succeeded.
9. **Parameters:** Serialized array of extra Memcache command arguments provided.
10. **Delay:** Elapsed seconds for Memcache command to return.
11. **Conversation ID:** Five alphanumeric characters common to all Memcache commands issued from a single page request.

Due to the log file's simple textual serialized PHP format, custom analysis programs can easily be written for the specific need. Multiple log files can be combined by simple concatenation. If a time-ordered log is needed the Unix `sort` command can be used. Each log entry line begins with identical characters, followed by the Unix time of the event with tenths of seconds, allowing `sort` to be used. An analysis script extracting the useful statistics for this work was written and is documented next.

5.4.2 Data Analysis

The `analysis.php` file in Appendix G is the analysis script used in our Memcache work. The script is written to read a single log file described above, specified as the first script argument. The script reports the following statistics for a given log file/run:

1. **Run Time, Start, Stop:** Tenth of seconds since Epoch
2. **All Webserver-to-Memcache Server Network Stats:** For every webserver to Memcache server daemon pairing, average data size, average response time, number of Memcache requests, estimated bandwidth, and estimated latency. Bandwidth and latency based on linear regression of data size and response time.
3. **Total Message Sent/Received:** Number of messages sent from each webserver and received by each Memcache server daemon.
4. **Individual Rack Statistics:** Based on rack organization specified in top of script, average Memcache response times for requests inside and outside a rack, with variance, for each rack.
5. **Aggregate Rack Statistics:** Averaged intra and inter-rack statistics.
6. **Conversation Statistics:** Per-page Memcache statistics such as number of pages served with Memcache requests, average number of Memcache requests per page, and total time elapsed waiting for Memcache requests per page.
7. **Key Usage:** Number of unique keys seen, key usage distribution between all webservers.
8. *ps* **Value:** Of all Memcache keys used on the average web page, the average proportion of session only keys. [0-1]
9. **Aggregate Request Statistics:** Total Memcache requests sent, average

number of Memcache bytes sent per request, proportion of read requests to write requests, number of bytes for read and write requests.

10. **Object Storage Statistics:** Number of bytes stored in cache, number of objects, and average size object.
11. **Memcache Command Usage:** Number and proportion of each Memcache command seen.

All results are printed to the command line with textual descriptions. If a second command line argument is provided to the script, the computed results in CSV form are exported to this file. See source file for column labels.

5.5 Network Utilization Measurement

The Memcache evaluation required accurate measurement of network traffic. This was done using a secondary Memcache-only network. The second network interface on each server was placed into a separate subnet and connected to servers within the same simulated rack via a small 10/100 switch. Each rack switch was then connected to each other via a managed 10/100 switch. SNMP was enabled in the managed switch with each port's traffic queried once per minute via MRTG[1] 2.9.17, and logged in RRDtool[2]. MRTG was modified to provide minute resolution of SNMP data rather than the default 5 minute. A Perl script was used to query the RRD database for total per-port data transfer based on the beginning and ending times listed in the MemcacheTach log file. Together with the MemcacheTach report, these values were useful in measuring the proportion of backbone Memcache traffic to inter-rack Memcache traffic for

[1] http://oss.oetiker.ch/mrtg/
[2] http://oss.oetiker.ch/rrdtool/

each configuration.

5.6 Memcache Evaluation Progression

Each of the above tools provide some insight into the performance of a running Memcache system. The results in Chapter 3 combine each and report on the aggregate. This section describes the way all data was captured and processed.

Each Memcache configuration was evaluated using different application configurations, different switching speeds, and repeated four times for a larger sample size. In total 120 data runs was captured: 5 architectures x 3 application configurations x 2 switch speeds, x 4 = 120. See Section 3.7 for details of the configurations.

The log data flow for each run is depicted in Figure 5.1. A single bash script, `RunAll.sh` resets the MySQL database for each run and starts requesting pages through JMeter. Next, JMeter issues HTTP requests to the load balancer according to the specified run file and logs individual page data to a log file. The load balancer routes HTTP sessions evenly over all webservers where MemcacheTach creates a log file on each webserver. On completion of the run, all MemcacheTach log files are concatenated, sorted, and saved as an aggregate log file. As the run progressed the backbone network switch was queried every minute for per-port byte counts and saved to a database. When a run is complete, the `RunAll.sh` script saves all logs using a specific run name, changes a configuration file in all webservers to reflect the next run, and the process repeats for all 120 runs.

Once all runs were complete, `analyse.php` is used to extract relevant run data from the MemcacheTach logs, JMeter logs, and the network usage database,

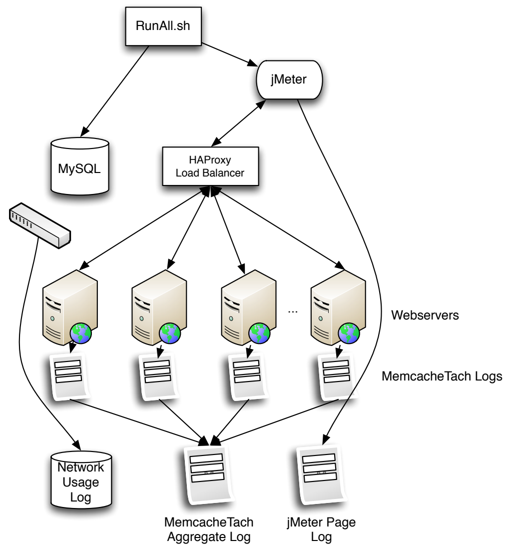

Figure 5.1: Memcache Data Capture

process the data, and save the result to a single CSV file. Afterwards, MS Excel is used to analyse this CSV file and report the final results. Figure 5.2 depicts this process.

5.7 Tentacle Evaluation Progression

Tentacle's performance was evaluated using page latency statistics from various simulated WAN configurations of MediaWiki. Apache JMeter simulated users interacting with the site, described in detail in Section 4.3.3.

Four computers were used for evaluation, three virtual and the fourth the host system. Figure 4.3 depicts the information flow between the machines. One minimal virtual machine (512MB RAM, single core) with dual network cards ran the WANem[45, 60] operating system and forwarded packets between interfaces while emulating a desired WAN performance. The other two virtual machines both ran Apache web servers and MySQL servers with specifics in

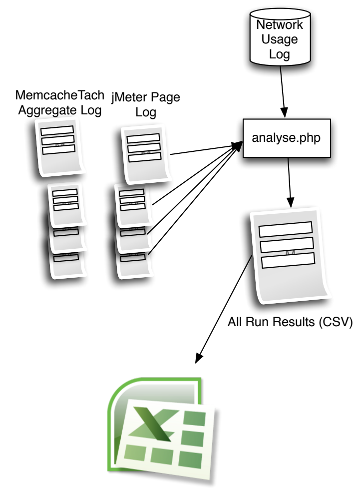

Figure 5.2: Memcache Data Processing

Sections 5.2 and 5.3 above. All machines were placed in the same network subnet. When a WAN delay was needed, the ARP routing tables were changed to funnel traffic through the WANem machine. A 100ms delay, 0ms jitter, was applied for packets in both directions. This delay approximates a transoceanic New Zealand to San Francisco traversal observed on the Verizon network in March 2012[103].

Five configurations were tested, with results give in Section 4.3.4. Each configuration changed either Tentacle features or the simulated network configuration. Each of the five configurations are described below.

1. **WAN w/Tentacle:** A user in New Zealand is simulated by directing the JMeter traffic to the NZ server with no delay. Communication is delayed between the NZ and SF servers. All Tentacle features are enabled, with the central database server residing in SF.

2. **Tunnel All:** The same New Zealand user is simulated, with appropriate

delay between the NZ and SF servers. The NZ server receives all web traffic, but tunnels all data to SF with no other features enabled. This measures the tunneling delay.

3. **Sync. Writes:** The same New Zealand user is simulated, with appropriate delay between the NZ and SF servers. All Tentacle features are enabled, but database writes are not buffered and thus the database query call delays the page until the write returns from the distant database server. This measures the effect of write buffering.

4. **All WAN:** A New Zealand user is simulated communicating with the SF server over the WAN link. No Tentacle features are enabled. This measures the long-distance TCP WAN delay.

5. **LAN No Optimization:** A user is simulated with no delay to the web and central database server. This measures the best-case latency scenario.

Each of the five configurations were run three times. JMeter was configured to output a log showing per-request information such as page requested, bytes returned, request latency, request time, and if the request returned a successful HTTP response (200). A Perl script was used to analyze the log file and gather statistics on per-page response times, and overall average response times. Plots were generated using Microsoft Excel. Results and plots are given in Sections 4.3.3 and 4.3.4.

5.8 Summary

The tools and measurement techniques described in this chapter allow reproduction of our contributions, or evaluation of any similar web system. MemcacheTach allows detailed Memcache statistics to be gathered and calculated for a running Memcache system.

Chapter 6

Related Work

Accelerating web applications has been heavily researched in academia as well as industry ever since the introduction of the web. This work is based on a wide range of computing topics, many predating the Internet. This chapter starts by addressing data-locality with respect to caching in computing, with an emphasis on multi-processor systems. A brief overview of distributed keystores, filesystems, databases, and other systems are discussed next. Database-middleware systems are discussed next, along with databases which employ asynchronous database writes, similar to our Tentacle system. Content delivery networks providing dynamic application support, which Tentacle can be considered as being, are discussed last.

6.1 Caching & Distributed Computation

Data-locality, or where data is stored, has impacted CPU and computer designs since the beginning of computing. The IBM System/360 upgrade from model 67 to 85 in the early 1970s was the first implementation to incorporate a CPU buffer[27, 52, 28]. This was needed due to memory and processing occurring in separate racks. The CPU buffer reduced the memory cycle by a third to one quarter of non-buffered models and increased processor performance. Typical memory access times of between 10ns and 250ns were typical, fast enough for distance-induced latency to be significant. The IBM System/360 used write-through caching/buffering. Buffering data close to the cpu, in this case in the same cabinet, rather than across the room or building, offered significant speed improvements.

When multiple processing units use a common memory, there exists multiple caching possibilities. Smith[89] identifies four cache types in the literature: (1) Shared Cache, (2) Broadcast Writes, (3) Software Control, and (4) Directory Methods.

1. **Shared Cache:** This design, such as in the UNIVAC 1100/80 where two CPUs share a single cache[16], is functionally equivalent to the typical Memcache architecture where all processors (clients) operate on the same single cache.

2. **Broadcast Writes:** This design keeps caches consistent by informing all other caches of any update operation and either invalidating the data, or removing it. This broadcast idea inspired the `Snoop` Memcache architecture variant where all caches are kept consistent by informing all caches of every write operation. This is typically done with a common bus architecture, but our implementation sends individual messages. The current `Snoop` implementation does not allow duplication of data, but using consistency methods in the caching area may allow such an extension. Extra book-keeping information would be needed for duplication, analogous to the dirty, presence, modified, shared, invalid, owned, exclusive, or other consistency bits used in cache coherency protocols[67, 8, 36, 21].

3. **Software Controlled:** This type allows the operating-system to manage the location of stored data in the caches to keep consistent[57]. A Memcache architecture of this nature would be possible, but knowledge of the specific usage pattern of data would be needed by the programmer and specified to Memcache. In practice, multiple caches could serve the same purpose, such as session storage only, or those shared by all.

4. **Directory:** This method uses a central location for cache state information[44, 95]. Our `Dir` Memcache architecture is based on a simplified version of this idea. In our case, when a write occurs all copies of data are updated, removing the need for additional status information in the directory. An interesting modification of the directory scheme, suggested by Drimak et al.[35], called broadcast search, requests data from all caches and the directory on a cache read miss. If the data is in any cache, it is retrieved faster than waiting for the directory and then contacting main memory. In the Memcache case such a method would always work because all data resides somewhere in the various racks and so would return faster than a directory lookup and subsequent retrieval. The network overhead of such a design may be prohibitive, suggesting a random-subset search may be beneficial.

As mentioned above, multi-cpu caching research has greatly influenced our Memcache architectures. Ironically, the issue of distance-induced latencies in mainframes spanning entire rooms or buildings in the 1960s and 1970s is still relevant in modern web datacenters, which is why this work is based on the research and lessons learned from that era.

6.2 Distributed Datastores

Accessing a common data-source with multiple clients is very common in computing. Many names exist to describe functionally similar storage systems. Keystores, or key/value storage systems, store pieces of data retrievable via a key. Similarly, filesystems store data accessible via a filename, like a key, but support a hierarchical naming scheme. The naming scheme and meta-file data is the only relational data stored in such systems. Databases allow more complex relationships between stored data and provide methods of querying data based on the relations. Additionally, manipulations are allowed in databases, which process subsets of data into a more useful form. Common to all these systems are data and how it is accessed, stored, and manipulated. Thus, these systems are all related to this work and those which exhibit some data-locality component are of interest.

The remainder of this section discusses datastores ordered from the least relational, to the most, followed by other storage systems.

6.2.1 Keystores

Keystores, or storage systems which reference data though a key-value only, contain no relational aspect other than the key/value mapping, and are the simplest storage system related to our work. This type of storage system has seen a recent explosion of growth, known as the NoSQL movement, due to their ability to store large amounts of data and scale well compared to other storage systems. Performance is gained from the lack of internal data-dependencies and a relaxation of consistency.

Both contributions are heavily influenced by advances in this type of storage system. Discussed below are keystore systems and papers which are similar to our work.

Dynamo[33] is a proprietary key/value datastore developed and used by Amazon.com to run their online web business. The goal is to assure high-availability for data spread throughout the world. Assuring high-availability not only applies to access to data, but how fast it can be accessed. When serving their online web business, strict latency requirements are made and any data access not able to meet them is not considered available. By relaxing consistency, high-availability is achieved, similar to our own Tentacle work. While Dynamo allows the developer to specify consistency parameters at the database level, Tentacle extends the granularity to each operation allowing much finer control. Dynamo does not write data to the closest storage location, as our Memcache architectures do, but can contact the storage server which responded fastest to the last read request. This leads to fast local reads and the possibility for local writes. The Riak system[96, 48] duplicates many of the design decisions from Dynamo.

The COPS[55] key/value system proposes a new version of consistency called *causal+*, where not only are causal events in the same thread of execution ordered, but those within the same datacenter as well. COPS is designed to operate in a multi-datacenter configuration where values are asynchronously replicated, similar to our Tentacle system, but version numbering assures causal consistency. Their view of consistency is convenient for programmers as causality assures correct ordering of commands in each web request call. Tentacle's time-based consistency view is weaker than causality+ and can such make programming difficult, but for the majority of datastore calls a time-based view is sufficient and provides superior performance. Future work will investigate either an explicit causality parameter, allowing the programmer to force causality between a set of operations, or a transparent one using source-code analysis. Unlike our Tentacle system that allows full SQL queries, COPS is purely a key/-value system capable of only `put` and `get` calls. To port a typical SQL-backed web application to use COPS would require a significant application rewrite, or complete rewrite, whereas the Tentacle system would require fewer changes.

Voldemort [78, 107] is an open-source persistent key-value datastore which handles replication and locality parameters using zones. Zones can group servers within a rack, or servers within a datacenter such that replication occurs in separate zones for reliability and availability. Replication is done using a chain replication method[7, 102], like our SDR Memcache architecture. Data-locality is exploited for read requests by accessing a local replica if possible. Writes are based on hashing, unlike our `Snoop` and `Dir` Memcache architectures, and thus may place data far away. Storage in this fashion is analogous to our SDR architecture which has the advantage of no data-location overhead compared to `Snoop` and `Dir`.

Microsoft's AppFabric [62] provides an object caching system across many computers using key/value lookup . Key hashing is not used to locate data, rather, a routing table is used to locate data requiring some organized management. Replication is supported via failover, and thus a read can not access the closest replica. A local cache can be maintained providing fast access for future reads. Durability can be added by configuring backup nodes to replicate data, though failover is used, rather than allowing the closest replica to be read. Due to the local cache, successive reads will be satisfied by the local cache, but any write will be sent over the network.

6.2.2 Distributed Filesystems

Distributed filesystems provide a hierarchical structure for organized storage and retrieval of data for a large number of users. Filesystems which use a central server are not covered here, but those with a distributed, non-centralized structure are. These systems spread data throughout the clients and therefore allow data-locality to influence performance. Filesystems contain more data dependencies than keystores.

The Hadoop Distributed File System [17, 18] is designed to serve distributed applications over extremely large data sets while using commodity hardware. High availability is attained through replication, which allows fast access by reading from the closest replica, similar to our Memcache architectures. Replicas are placed specifically to improve reliability, availability, and lower network utilization. Primary data placement is dependent on a master node, which does not take the writer's location into account at this time. Thus, writes are not location-aware, but reads take advantage of nearby replica placement. Network utilization will be localized to the read replica while a write will effect the central network.

Peer-to-peer (p2p) distributed filesystems are related to our Memcache work due to their data-locality aspects, such as Napster, Gnutella, Freenet[24] , and CFS[30]. These decentralized systems allow locally stored data (files) to be found and retrieved by any member participating in the system. Just like the Memcache Snoop and Dir architectures, data is stored locally, but available to all. The main advantage of such p2p filesystems is their efficient file query schemes. Unlike Snoop, file-location information is not replication on all nodes, nor is it centrally stored like Dir. Instead, these p2p systems query a subset (typically $O(logN)$) of the N servers to obtain the location of the stored data. Incorporating such a system into a new Memcache architecture would be possible, but may not be faster than the typical architecture due to the multiple hops to locate data. Web applications with a high ps value, or installations with high server turnover, may be a good fit for such a caching system.

6.2.3 Distributed Databases

Increasing the data dependencies further leads to databases. These systems allow complex interdependent data to be stored, manipulated, and queried. Database systems are typically non-distributed due to the interdependencies within the data. Database replication, discussed in Section 4.1.3, allows a centrally located database to increase read capacity by replicating the database allowing duplicates to satisfy read queries. The Tentacle system uses database replication. In this section we discuss related database systems which are more distributed than the replication approach. More specifically, systems which incorporate dependency relaxing techniques or scalable writes (non-centralized).

The LazyBase database system[23] provides high throughput by allowing each read query to specify how fresh the result will be, similar to our Tentacle system. Less recent data will have lower query latency. Unlike the Tentale system, LazyBase is designed for high-volume, low-update batch-oriented queries. Data flows through LazyBase in a pipeline fashion, with older data at then end. The LazyBase system is not designed for a multi-cite configuration, but rather a high performance single-site. Their results show per-query freshness specification can increase performance, supporting our Tentacle system.

Cassandra [11] allows the storage and manipulation of large amounts of structured data through an eventually consistent storage policy. Stored data can contain common traits (columns) and queried on such traits, but full relational semantics are not supported. The absence of strong internal relational dependencies allows this system to scale to thousands of nodes. Replicas are used for durability and accessibility. Reads can either query the closest replica, or query all replicas and return when a quorum is reached. Likewise, writes can return after the master location plus any number of replicas have been written to. The master location of data is dictated by a hashing strategy with no locality information. Replicas of data can follow 'Rack Unaware', 'Rack Aware', or 'Datacenter Aware' policies thereby allowing fast local reads. Due to the hashing master location scheme, writes may require costly distant communication. Network utilization will be localized to the read replica while a write will effect the central network.

The TACT system[112, 111] similarly identifies per-query consistency as an opportunity to increase database performance. Their work centers around many geographically distributed databases with each using local reads and writes, unlike Tentacle which centralizes writes. To keep the probability of a write conflict low, a per-query consistency value is used (a `conit`) incorporating allowed numerical, order, and temporal errors. The `conit` is a more complex specification than the time-only consistency value in Tentacle. The advantage of TACT is the ability to write locally and quickly read the result thereafter, increasing performance for applications with high *ps* values.

6.3 Other Systems

Other systems and works which do not fit into the previous sections are discussed here.

Time-based consistency, as used in our Tentacle work, has been investigated in other areas besides web applications, such as in concurrent programming in the Beehive system[83], and in distributed objects[100].

EHCACHE Terracotta [97] 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. BigMemory implements a key/value store using key hashing for Java objects. A local cache can be used in a multi-layered configuration. Due to the local cache, successive reads will be satisfied by the local cache, but any write will be sent over the network. Our proposed Memcache architectures could prove beneficial for this type of system.

The HOC system [4] is designed as a distributed object caching system for Apache, specifically to enable separate Apache processes to access others' caches. Of note is their use of local caches to speed subsequent requests and a broadcast-like remove function. A write will distribute the object to all nodes in the system, like the `Snoop` Memcache architecture, with reads subsequently reading locally.

6.4 Database Middleware Systems

Many database middleware systems have been proposed which aim to speed query resolution. Tentacle can be considered such a system. Database middleware systems provide a database interface to an application, identical to standard RDBMS interfaces, yet provide increased performance by use of caches, multiple databases, or some other method. Applications require no or minimal changes to switch from a standard RDBMS to such a system.

Database middleware systems can be broken down into two types: query caching and database replication, though many exhibit features of both [85, 86]. Query caches store the results of queries where database replication replicates the database. Further differentiation can occur by breaking query caching into content-aware and content-blind, and database replication into full and partial replication. Each have their benefits to application performance by increasing scalability in some way, thereby lowering end-user latency. Not all are targeted toward distributed deployment, though many can be used in that way. Tentacle is a full database replication system along with a local content-blind query cache. Middleware systems which relax temporal or relational dependencies or can be distributed are of particular interest. Figure 6.1 breaks down the relevant systems and their classifications.

6.4.1 DBProxy

DBProxy [6] presents an intermediate database layer exhibiting content-aware query caching and partial data replication. This layer intercepts all SQL queries and either services them locally or forwards them to a central database. For read queries a local query cache is kept, as well as a database containing enough data

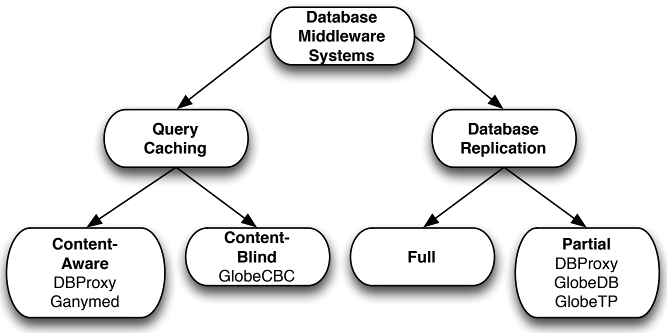

Figure 6.1: Categories of Database Middleware Systems [85]

to satisfy the cached requests. As new read queries are observed, a decision is made weather the current database state can satisfy the request. If so, the local database is queried. If not, a request is made to the central database to return enough data to satisfy the request. All update queries are passed to the central database, incurring significant delays if geographically distant. To mitigate delays for writes, Tentacle locally buffers writes when possible. Changes to the central database are pushed to all edge databases. Tentacle replaces this system's query analysis feature with a time-consistency value, allowing fast routing of queries.

This scheme exploits relational dependencies by only replicating enough data to satisfy common queries locally. In doing so it minimizes replication traffic. When deployed in a distributed environment this scheme would work well for read heavy applications, but suffer whenever a write occurs.

6.4.2 Ganymed

Ganymed [69] presents a database driver to an application which differentiates between read and write requests, but routes queries to a set of databases to maintain master and replica databases. It can be categorized as a full data replication strategy which is query-content aware. Unlike database replication discussed in Section 4.1.3, Ganymed assures strict consistency by analyzing each query and directing it to a replica with the most recent data pertinent to the query. Relational dependencies are exploited by routing queries to specific databases based on the query's content and the database schema.

A distributed application would be undesirable because all queries are directed through a central manager. Depending on the application, database schema, and manager complexity, the manager could be replicated and placed on edge nodes.

6.4.3 GlobeCBC

GlobeCBC [85] is a content-blind query caching system with relaxed consistency. By using database query templates a developer can specify which update templates would invalidate cached read templates, thus keeping results consistent. This exploits temporal dependencies at the query level. To allow weak consistency per template, a developer can specify TTI (Time To Invalidate), or U, the number of updates a query can ignore before being invalidated. This allows consistency to be relaxed whenever possible, allowing the system to use the cache as much as possible. Fine-grained invalidation is implemented so an update template does not invalidate all cached queries for an entire table. All write queries are sent to the database.

They report when consistency is lowered (TTI from 0 to 1 minute on all read queries) latency drops between 30% and 200% depending on load attributed to internal latencies. This clearly shows that relaxing consistency, when possible, can have large benefits, supporting our work. Unfortunately they do not report on the performance resulting from custom TTI per template.

In a distributed environment GlobeCBC will suffer due to the synchronous database calls on a cache miss. If replicated databases were placed at the edge nodes, read queries would have similar performance to our proposed system. Write queries would suffer due to the high latency round-trip to the central database. The Tentacle system described in Chapter 4 is based on GlobeCBC, augmented with features to keep latencies low in a distributed environment.

6.4.4 GlobeDB

GlobeDB [84] is a partial replicating middleware database layer. To minimize replication traffic, local databases hold a subset of the entire database differentiated by the table's primary key. They note that 80% of read queries are simple, which would result in a unique record having been found by primary key. A replication strategy dictates what and when records are copied locally. Queries which are not simple, or any which modify the database, are sent to a master database and modifications pushed down to database replicas. The strategy is similar to a multi-layer cache system. Future work with Tentacle may incorporate such a partial database replication stratgegy.

Relational dependencies are relaxed for data in specific databases. Appropriate queries are then directed toward these minimal databases to alleviate load on the central database. Consistency is not relaxed.

Due to a centralized database this system may not perform better in a distributed environment over a non-distributed one. For pages with simple (those satisfy-able by the local database) queries only, the scheme could work, but if any write queries or complex read queries exist, several round-trip delays would be incurred. Tentacle handles high write latency by buffering localy when possible.

6.4.5 GlobeTP

GlobeTP [37] is a partial replicating middleware database which takes advantage of templated queries in typical web applications. By knowing a priori the types of queries that will be issued, and therefore the relational dependencies, it can break up one large database containing many tables into a set of databases. A query router detects which table(s) are used in a query and directs it to a suitable database. By breaking up the database into smaller pieces the write capacity over a single database is increased. Based on query execution time, WAN latencies, and other metrics, individual tables can be replicated further or even replicated at edge nodes. This flexibility of duplication and locality allows optimization of resources.

6.5 Asynchronous Database Writes

Allowing asynchronous database access relaxes temporal dependencies in an application. Asynchronous database writes are not assured to be applied in the actual order of queries. In some cases this may be allowable, but in others a conflict may occur. The advantage of asynchronous database access is by allowing an application to issue a database query and continue execution. The systems described below all reply back to the application when the query has completed, unlike our Tentacle system. Asynchronous database access increases parallelism in the system thereby lowering end-user latency.

Use of a asynchronous database interface typically requires the use of a callback or other event subsystem to notify the application the result of the query is complete. No work could be found which uses asynchronous database connections in web applications, probably due to the overhead of a callback system. Tentacle removes the overhead of a callback system by not returning any value.

The Adobe Flex system allows local databases to be accessed in either synchronous or asynchronous modes [1]. In asynchronous mode queries are sent to the database and the application continues. At a later time the database will notify the application that the operation has completed. In our web environment where pages are served in hundreds of milliseconds, waiting for a database reply may slow the application. perl-mysql-async is another asynchronous database driver which allows Perl scripts to send a query and specify a callback function to be triggered when complete [68]. A similar Java driver exists as well [39]. Asynchronous connections to Microsoft SqlServer have also been proposed in C# by creating new connections in threads [26]. Due to the short-lived nature of serving pages in a web server the creation of threads per-request may outweigh the benefit. PostgreSQL contains Application Programming Interface (API) functions to interact asynchronously, though only one query can be running at a time [64, 70]. To obtain parallelism multiple connections must be made. Also, the caller will block until Postgres acknowledges the request, incurring distance latencies.

Using asynchronous database access can also speed multiple queries by sending many in parallel [10]. This method allows an application to issue a list of database queries to execute. Rather than using a single database connection and preforming them serially, this scheme creates multiple connections and issues all queries at once. In a WAN environment, where the client and database are separated by a significant distance, this technique can remove multiple round-trips. The Tentacle system uses a similar technique by bundling multiple queries together to be executed on the central database. Since write queries are only allowed, no return value is sent to the application.

6.6 Content Delivery Networks

Content delivery networks, as discussed in Section 1.2, provide distributed web serving capabilities throughout the world[43, 73, 104, 87]. Static or streaming content can currently be served by these services. This section discusses advances in CDN technology to add dynamic generation of content at their edge locations. Such services would allow non-distributed websites to be deployed worldwide, much like the goal of Tentacle.

When the web application is solely read-only, such as the case when web pages are generated from a database and no user-interaction can affect it, worldwide replication is straightforward[88]. The DUP system[22] builds a dependency graph between generated pages and data in a database. Thus, when the database is changed, the appropriate pages can be regenerated and cached until the next update is detected. In this manner the entire web application can be distributed and only database updates must be propagated. Further research has investigated the tradeoff between precomputing each page on a database change, or generation on-demand[49]. While individual page caching is not a feature of Tentacle, it could be used to serve read-only content at the edges and forward all interactive content to a central location by setting the consistency time to zero for all `insert` or `update` database calls. Database query caching would also speed page construction.

The vMatrix system[9] proposes encapsulating web server instances inside virtual machines and distributing them throughout the world. The vMatrix system then provides a virtual private network to all running virtual machines so communication over great distances is transparent, though still high in latency. Applications providing static or seldom changed content work well, but frequent communication with a central database slow this system down. Performance will be analogous to the sync. wites Tentacle configuration in Section 4.3.4. Many of the current cloud computing services such as Amazon EC2 and Microsoft Azure provide near identical features.

Rather than encapsulate an entire server, the system outlined by Rabinovich et al.[74] replicates CGI handlers over many webservers. Each CGI handler is capable of completely processing a single HTTP request, no other support software is available. This design allows fast scale-up and down of applications as only CGI files need to be changed, rather than an entire operating system. In practice this technique would work for dynamic content generation in a CDN environment, but high-latency database connections would negate the speed gain for most complex applications. Tentacle is able to reuse database connections between HTTP requests, which would not be possible in a mobile CGI environment.

Akamai Technologies has a similar feature to the previous Rabinovich system called EdgeComputing which allows a customer to deploy applications to edge nodes[31]. Applications can be Java (J2EE) or Microsoft .NET applications, with future work to bring C, PHP, and Perl support. Only small pieces of dynamic information can be stored at the edge, with session information most suitable. Large non-changing data can be deployed bundled with the application. Thus, splitting of an application is neccessary to isolate stand-alone parts and those which require dynamic database access. Similar drawbacks to the Rabinovich system apply, with local session storage a convenient feature.

A similar system to Tentacle's forwarding feature is ActiveCache[20]. This system uses a worldwide collection of intelligent proxy caches, capable of executing small java applets on each cache hit. A proxy cache miss will request the web content from a central server which will respond with the content and a small java applet. Future proxy cache hits will execute this applet and either modify the returned page, or preform some other action. The proxy is able to selectively reply with a cached version, or forward the request to the central server, just like our Tentacle system. ActiveCache can't be used for a complete database-backed web application as remote database access would negate all advantages, as shown in our Sync. Writes Tentacle configuration in Section 4.3.4.

A new business area for dynamic web serving are products offering dynamic site acceleration[77, 3]. Rather than generating dynamic content at the edge, as some of the above works and Tentacle does, dynamic site acceleration provides faster transport of content to an origin server. As mentioned in Section 1.2, establishing the TCP link can take one round trip. These services speed client access by redirecting users to an edge location where the request is compressed and tunneled over an already-connected TCP link to the origin server. This removes the initial full-distance round trip to set up the TCP link. Tentacle employs compression on tunneled data and future work will investigate persistent connections.

6.7 Summary

Tentacle and our proposed Memcache architectures use techniques and advances in computing from a wide range of computing, not just from the web area. Problems observed in mainframe systems due to distance latencies in the 1960's and 1970's are still problematic in today's web serving datacenter, with many past solutions applicable today. Tentacle is able to provide low latency service for geographically distant users by combining many web technologies into a simple database middleware and caching system.

Chapter 7

Conclusion & Future Work

The staggering growth of the Internet has transformed our society. Communication has become near-instantaneous, ideas can be shared easily between people all over the world, and now the barrier for international business has been lowered. As the number of users have grown, the physical reach of the Internet has also grown. Despite all our technological advances in communication, we are still bound and must adapt to physical limits. Providing fast responses to web requests in an environment where users are geographically distributed is investigated.

This chapter summarizes the findings in this work. Section 7.1 discusses the conclusions that can be drawn, Section 7.2 reviews the supporting contributions, and Section 7.3 suggests future directions for research in this area.

7.1 Conclusions

We have shown that non-trivial distance-induced latencies exist both in a datacenter and on the Internet. Reducing latency in these situations has been addressed by academia and industry, with this work offering a new perspective. Data within a dynamic web application is constantly modified and must obey temporal and relational dependencies, but not all data or queries of such data must follow these dependencies all the time.

Identifying locations within a web application where dependency relaxation can occur allows flexibility in data caching and storage location. Locating data close to where it can be used most effectively, due to the flexibility of storage, lowers end-user latencies and reduces overall network utilization.

7.2 Summary of Contributions

Supporting our thesis are the following two contributions:

1. By recognizing usage patters in Memcache requests using MemcacheTach, alternate locality-aware storage policies can be used to decrease cache latency and lower network utilization. A model is developed to predict cache performance using a specific application's usage profile under various configurations. Experimental data supports the developed model and alternate Memcache storage policies.

2. The design and implementation of Tentacle, a database middleware system using application provided consistency constraints to quickly provide an application with database access. When fast database access is not possible, the HTTP request is forwarded to a location capable of a fast response. Tentacle allows an application using a single SQL database to be deployed worldwide using edge nodes placed close to the users, while conforming to consistency constraints.

These contributions show that exploiting temporal and relational dependencies allow location-aware storage to be used, lowering end user latencies, reducing network utilization, all while obeying consistency requirements.

7.3 Future Work

The steady geographical growth of the Internet and the insatiable need for low-latency web interactions by users will drive academic and industry research in this area. New computational theories, storage technologies, programming styles, and businesses will undoubtedly emerge to satisfy the need. The performance of computers will only increase, while physical laws limiting communication will always be the same. Developing distributed systems able to quickly communicate with distributed users is therefore necessary to continue the advancement of the Internet.

Discussed below are possible research directions based on our work.

7.3.1 Performance-Aware Caching

Chapter 3 used relational dependencies and cache usage behavior to place data close to where it is used, thereby reducing latency and network utilization. Multiple schemes were discussed, each with unique performance characteristics. Rather than apply a single scheme to a cache system, a dynamic approach could be taken to optimize the cache configuration during runtime to maximize performance. Per-data configurations could be used to best optimize the system.

7.3.2 Rich Application-Datastore Interactions

The recent popularity of NoSQL storage systems has shown that one-size-fit-all storage systems are not always the best solution. Application specific performance has driven NoSQL's popularity with their myriad features, configurations, and architectures. Each type of storage system represents a set of features which are provided by an implementation following physical and logical constraints. Physical laws, such as the speed of light, force this wide array of storage systems.

Each application requires specific data storage requirements. An ACID compliant datastore allows applications to be written with predictable results. As shown in Chapter 4, allowing the developer to provide consistency relaxation can improve performance. The more information the storage system can obtain about how the data will be used, what constraints can be relaxed, and the access patterns of clients, the better a storage system can optimize itself for best performance. Data storage systems should be developed which can use these and other parameters to increase performance while balancing constraints.

Appendix A

Memcache Latency Formula

Where $bw_m = \frac{2 \times size_{message}}{bw \times 1024 * 1024/8/1000}$

$bw_o = \frac{size_{message} + size_{object}}{bw \times 1024 * 1024/8/1000}$

	SDC	SDS	SDR
Set	$l_2 + bw_o$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_o$	$l_3 + bw_o$
Add hit	$l_2 + bw_o$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_o$	$2 \times l_3 + bw_o$
Add miss	$l_2 + bw_o$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_o$	$2 \times l_3 + bw_o$
Replace hit	$l_2 + bw_o$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_o$	$2 \times l_3 + bw_o$
Replace miss	$l_2 + bw_o$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_o$	$2 \times l_3 + bw_o$
CAS hit match	$l_2 + bw_o$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_o$	$2 \times l_3 + bw_o$
CAS hit no	$l_2 + bw_o$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_o$	$l_3 + bw_o$
CAS miss	$l_2 + bw_o$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_o$	$l_3 + bw_o$
Delete hit	$l_2 + bw_m$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_m$	$k \times l_3 + bw_m$
Delete miss	$l_2 + bw_m$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_m$	$k \times l_3 + bw_m$
Inc hit	$l_2 + bw_m$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_m$	$k \times l_3 + bw_m$
Inc miss	$l_2 + bw_m$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_m$	$l_3 + bw_m$
Dec hit	$l_2 + bw_m$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_m$	$k \times l_3 + bw_m$
Dec miss	$l_2 + bw_m$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_m$	$l_3 + bw_m$
Flush	$l_2 + bw_m$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_m$	$l_3 + bw_m$
Get hit	$l_2 + bw_o$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_o$	$\frac{k}{r} \times l_{local} + \frac{r-k}{r} \times l_3 + bw_o$
Get miss	$l_2 + bw_m$	$\frac{1}{r} \times l_{local} + \frac{r-1}{r} \times l_3 + bw_m$	$k \times l_3 + bw_m$

	Snoop	Dir
Set	$l_3 + bw_o$	$2 \times l_2 + l_3 + bw_o$
Add hit	$l_{local} + bw_o$	$l_2 + bw_o$
Add miss	$l_{local} + l_3 + bw_o$	$l_2 + l_{local} + bw_o$
Replace hit	$l_{local} + ps \times l_{local} + (1 - ps) \times l_3 + bw_o$	$l_2 + ps \times l_{local} + (1 - ps) \times l_3 + bw_o$
Replace miss	$l_{local} + bw_o$	$l_2 + bw_o$
CAS hit match	$l_{local} + ps \times l_{local} + (1 - ps) \times l_3 + bw_o$	$2 \times l_2 + l_3 + bw_o$
CAS hit no	$l_{local} + ps \times l_{local} + (1 - ps) \times l_3 + bw_o$	$l_2 + bw_o$
CAS miss	$l_{local} + bw_o$	$l_2 + bw_o$
Delete hit	$l_3 + bw_m$	$2 \times l_2 + ps \times l_{local} + (1 - ps)l_3 + bw_m$
Delete miss	$l_3 + bw_m$	$l_2 + bw_m$
Inc hit	$l_{local} + ps \times l_{local} + (1 - ps) \times l_3 + bw_m$	$ps \times l_{local} + (1 - ps)(l_2 + l_3) + bw_m$
Inc miss	$l_{local} + bw_m$	$l_2 + bw_m$
Dec hit	$l_3 + bw_m$	$ps \times l_{local} + (1 - ps)(l_2 + l_3) + bw_m$
Dec miss	$l_{local} + bw_m$	$l_{local} + l_2 + bw_m$
Flush	$l_3 + bw_m$	$l_3 + bw_m$
Get hit	$ps \times l_{local} + (1 - ps)l_3 + bw_m$	$ps \times l_{local} + (1 - ps)(l_2 + l_3) + bw_o$
Get miss	$l_{local} + bw_o$	$l_{local} + l_2 + bw_m$

Appendix B

SDR: Mem_dup.php

```
<?php
    /*
     Copyright (c) 2012, Paul G Talaga
     All rights reserved.
     Redistribution and use in source and binary forms, with or without
     modification, are permitted provided that the following conditions are met:
     * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
     * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
     ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED. IN NO EVENT SHALL Paul G Talaga BE LIABLE FOR ANY
     DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
     (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
     ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
     (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
     SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
     */
    // Wrapper class to PECL memcache, provides duplication with no locality
    //  advantage (yet).
    // We use the findServer function of the PECL Memcache client to detect the
    //  storage server.  Thus, to duplicate data we append data to the key
    //  until a different server is chosen.  If your duplication value is equal
    //  to or greater than the number of servers we'll just store on all.
    // A GET will choose randomly from the duplicated keys until all return
    //  false, thus, a read miss will query your duplication
    //  level number of servers until it fails.  Consider limiting?  Thus this
    //  is is like RAID 1.
    // NOTE: Due to a read possibly taking a long time (timeout x duplication),
    //  consider reducing the timeout.
    class Memcache_Dup {
        // constructor funciton
        function __construct(){
            $this->num_dups = 2;
            $this->num_servers = 0;
            $this->mem = new MemcachePool();
            //$this->mem = new Memcache();
            //echo 'Dup';
        }
        function __destruct(){
            unset($this->mem);
        }
        function setDups($num){
            $this->num_dups = $num;
        }
        function getDups(){
            return $this->num_dups;
        }
        function setCompressThreshold($val){
            return $this->mem->setCompressThreshold($val);
        }
 /**************************************************************/
        function getKeys(&$key, $set = 0){
            // Returns an array of keys to store (or get) based on the
            //  duplication factor and # of servers.
            // First in returned array should be in local rack
```

```
            // NOTE: findServer returns ip:port, so this will NOT garuntee
            //  duplicates on different machines!
            $slist = array(0 => $this->mem->findServer($key));
            $ret = array(0 => $key);
            $append = 0;
            while(count($slist) < $this->num_dups && count($slist) < $this->num_servers){
                $append++;
                $new_key = $key . '#' . $append;
                $s = $this->mem->findServer($new_key);
                if(!in_array($s,$slist,TRUE)){
                    $slist[] = $s;
                    $ret[] = $new_key;
                }
            }
            // Sort in increasing order, then rotate so next largest ip is first.
            // Assuming your IP's are given in order through all racks,
            // next largest memcache server IP may be in same rack
            // NOT ELEGANT!
            $ret2 = array();
            $sv = explode('.',$_SERVER['SERVER_ADDR']);
            array_multisort($slist,$ret);
            $m = explode('.',$slist[0]);
            $m = explode(':',$m[3]);
            $i = 0;
            while($sv[3] > $m[0] && $i < 50){
                $s = array_shift($slist);
                array_push($slist,$s);
                $s = array_shift($ret);
                array_push($ret,$s);
                $m = explode('.',$slist[0]);
                $m = explode(':',$m[3]);
                $i++;
            }
            return $ret;
        }
/*************************/
        function addServer($host, $tcp_port = 11211, $udp_port = 0, $persistent = true,
             $weight = 1, $timeout = 1, $retry_interval = 15, $status = true, $rack = -1){
            // Add a server.  Rack doesn't matter.  Future releases will use!
            $this->num_servers++;
            $ret = $this->mem->addServer($host, $tcp_port, $udp_port, $persistent, $weight,
                  $timeout, $retry_interval);
            return $ret;
        }
/*************************/
        function connect($host, $tcp_port = 11211, $udp_port = 0, $persistent = true,
             $weight = 1, $timeout = 1, $retry_interval = 15, $rack = -1){
            return $this->mem->connect($host, $tcp_port, $udp_port, $persistent, $weight,
                 $timeout, $retry_interval);
        }
 /***********************************************************/
        function add(&$key,&$value, $flag = 0,$expire = 0){
            // Add item with key
            // Return true if ANY call returns true, and will set failed adds
            //  via set so they all match
            // TODO: look at uses for add and decide if this is the best policy!
            $ret = FALSE;
            $repair = array();
            foreach($this->getKeys($key) as $k){
                if($this->mem->add($k,$value,$flag,$expire) === TRUE){
                    $ret = TRUE;
                }else{
                    $repair[] = $k;
                }
            }
            if($ret && count($repair) > 0){
                foreach($repair as $k){
                    $this->mem->set($k,$value,$flag,$expire);
                }
            }
            return $ret;
            // WHOA! Need to deal with $key beeing an array already!
        }
/*************************/
        function delete($key,$timeout = 0){
            // Delete data with key.
            $keys = $this->getKeys($key);
            return $this->mem->delete($keys,$timeout);
        }
/*************************/
        function flush(){
            // flush
            return $this->mem->flush();
        }
```

```
/*************************/
        function get(&$key, &$flags = 0, &$cas = ''){
            // Randomly pick a key to retrieve, and keep picking if it fails
            foreach($this->getKeys($key) as $k){
                $ret = $this->mem->get($k);
                if($ret !== FALSE){
                    return $ret;
                }
            }
            return FALSE;
        }
/*************************/
        function replace($key, $value, $flag = 0, $expire = 0){
            // Replace should return false if the item doesn't already exist.
            // Return false if all return false
            $ret = FALSE;
            $repair = array();
            foreach($this->getKeys($key) as $k){
                if($this->mem->replace($k,$value,$flag,$expire) === TRUE){
                    $ret = TRUE;
                }else{// if any succeded, set the broken ones to match
                    $repair[] = $k;
                }
            }
            if($ret == TRUE && count($repair) > 0){
                foreach($repair as $k){
                    $this->mem->set($k,$value,$flag,$expire);
                }
            }
            return $ret;
            $keys = $this->getKeys($key);
            return $this->mem->replace(array_combine($keys,array_fill(0,count($keys),$value
                 )),$flag,$expire);
        }
/*************************/
        function decrement($key, $amt = 1){ // TODO: deal with defval and exptime!!!!
            // Decrement
            // Will return decremented value if any return a value (and all same),
            //  otherwise false
            // Those which return false (does not exist), will be set with the
            //  value
             $ret = FALSE;
            $repair = array();
            foreach($this->getKeys($key) as $k){
                $val = $this->mem->decrement($k,$amt);
                if($ret == FALSE && $val !== FALSE){ // First time around
                    $ret = $val;
                }else if($ret !== FALSE && $val !== FALSE && $ret != $val){
                    // Got something earlier, and different now, fix
                    $repair[] = $k;
                }
            }
            if($ret !== FALSE && count($repair) > 0){
                foreach($repair as $k){
                    $this->mem->set($k,$ret);
                }
            }
            return $ret;
        }
/*************************/
        function increment(&$key, $amt = 1){
            // Increment
            // Will return incremented value if all are equal, otherwise false
            $ret = FALSE;
            $repair = array();
            foreach($this->getKeys($key) as $k){
                $val = $this->mem->increment($k,$amt);
                if($ret == FALSE && $val !== FALSE){ // First time around
                    $ret = $val;
                }else if($ret !== FALSE && $val !== FALSE && $ret != $val){
                    // Got something earlier, and different now, fix
                    $repair[] = $k;
                }
            }
            if($ret !== FALSE && count($repair) > 0){
                foreach($repair as $k){
                    $this->mem->set($k,$ret);
                }
            }
            return $ret;
        }
/*************************/
        function set(&$key,&$value,$flag = 0,$expire = 0){
            // Set for all keys
```

```
            // Can use setMulti!
            $keys = $this->getKeys($key,1);
            $ret = $this->mem->set(array_combine($keys,array_fill(0,count($keys),$value)),
                null,$flag,$expire);
            return $ret;
        }
/*************************/
        function getStats($type, $slabid, $limit = 100){
            return $this->mem->getStats($type,$slabid,$limig);
        }
/*************************/
        function findServer($key){
            return $this->mem->findServer($key);
        }
/*************************/
        function setLocalRack($rack){
            // Dummy function to ease testing of other configs
        }
    }
?>
```

Appendix C

Mem_RackAware.php

```
<?php
    /*
      Copyright (c) 2012, Paul G Talaga
      All rights reserved.
      Redistribution and use in source and binary forms, with or without
      modification, are permitted provided that the following conditions are met:
      * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
      * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
      ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
      WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
      DISCLAIMED. IN NO EVENT SHALL Paul G Talaga BE LIABLE FOR ANY
      DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
      (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
      LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
      ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
      (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
      SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
      */
    // This class declares common functionality for many of the rack-aware memcache clients
  class Memcache_RackAware{
        // constructor function
        function rackConstruct($options = 0){
            $this->racks = array(); // Where we'll store each memcache instance, one per
                rack
            $this->local = -1; // The local rack number
            $this->options = $options;
        }
        function __destruct(){
            // Call destruct on all memcache instances
            foreach($this->racks as $k=>&$r){
                // uses unset
               unset($this->racks[$k]);
            }
        }
        function addServer($host, $tcp_port = 11211, $udp_port = 0, $persistent = true,
             $weight = 1, $timeout = 1, $retry_interval = 15, $status = true, $rack = -1){
             //PECL Memcache
        //function addServer($host,$port, $weight = 1, $rack = -1){ //PECL Memcached
            // Add a memcache server to use. There are numerous ways to
            //  differentiate which 'rack' a server should be added too, the
            //  easiest being a parameter, rack.
            // To make the interface consistent with other implementations, we
            //  optionally differentiate via modulo the last octet of the server
            //  address plus the port.
            // The local rack is the first added, unless explicitly specified
            //  with setLocalRack
            // Host is a string, so convert last characters to integer
            //  (must be in dotted address form).
            if($rack == -1){
                $parts = explode('.',$host);
                $rack = ($parts[3] + $tcp_port) % 50;
            }
            if(count($this->racks) == 0 && $this->local == -1){
                // first addition, save as local!
                $this->local = $rack;
            }
```

```
            if(!isset($this->racks[$rack])){
                // rack pool not set up yet, start!
                //$this->racks[$rack] = new Memcached;
                    // Required by Memcache_Dir (Mem_dir.php) & Memcache_Dup
                    // (Mem_dup.php) for append and getServerByKey
                //$this->racks[$rack] = new Memcache;
                // OK for Memcache_Snoop (Memcache_snoop.php)
                $this->racks[$rack] = new MemcachePool();
            }
            // Add to that rack pool
            $ret = $this->racks[$rack]->addServer($host, $tcp_port, $udp_port, $persistent,
                 $weight, $timeout, $retry_interval);
            return $ret;
        }
        /*************************/
        function setLocalRack($rack){
            $this->local = $rack;
        }
}
?>
```

Appendix D

Snoop: Mem_snoop.php

```
<?php
    /*    a
     Copyright (c) 2012, Paul G Talaga
     All rights reserved.
     Redistribution and use in source and binary forms, with or without
     modification, are permitted provided that the following conditions are met:
     * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
     * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
     ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED. IN NO EVENT SHALL Paul G Talaga BE LIABLE FOR ANY
     DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
     (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
     ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
     (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
     SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
     */
    // Wrapper class to memcache, provides snooping based cache locality using
    //  notes distributed to all other memcache clusters.  No data redundancy!
    // Due to this being in PHP, we can't issue simultaneous requests to all
    //  racks at once.  Thus, we'll just have to deal with the slowdown until a
    //  C implementation is done.
    include_once('Mem_RackAware.php');
    class Memcache_Snoop extends Memcache_RackAware{
        // constructor function
        function __construct(){
            // A note is stored under the appropriate key, with data
            //  '<secret><rack id>', where rack id is where to find the
            //  actual data.
            $this->secret = 'S!';  // Should be complex enough to prevent
                                   //collisions with acutal data.
            $this->rackConstruct(); // constructor for RackAware,
                    // provides $this->local which is the rack number this
                    // machine should use, and $this->racks which
                    // is an array of Memcache instances indexed by rack number.
                    // So $this->racks[$this->local] would be the closest
                    // memcache instance.
                                   //echo 'Snoop';
        }
        // addServer      provided by Memcache_RackAware
 /**********************************************************/
        function add($key,$value,$flag = 0,$expire = 0){
            // Add item with key
            // Add note to all other racks
            $return = $this->racks[$this->local]->add($key,$value,$flag,$expire);
            if(!$return){return false;} // Fail if data or note already there.
            foreach($this->racks as $k => &$r){
                if($k != $this->local){ // Don't overwrite actual data!
                    $r->set($key,$this->secret . $this->local,$flag,$expire);
                        // We do a set just to be sure we don't fail somewhere else.
                }
            }
            return true;
        }
/*************************/
        function connect($host,$port,$timeout = 1){
```

```
            return self::addServer($host,$port,TRUE,1,$timeout);
        }
/*************************/
        function delete($key,$timeout = 0){
            // Delete data with key.  We'll only return failure if local failed
            foreach($this->racks as $k => &$r){
                if($k != $this->local){
                    $r->delete($key,$timeout);
                }
            }
            return $this->racks[$this->local]->delete($key,$timeout);
        }
/*************************/
        function flush(){
            // flush all racks!
           foreach($this->racks as &$r){
                $r->flush();
            }
        }
/*************************/
        function get($key,$flags = ''){
            // Get local, and if note found go get actual.
            $ret = $this->racks[$this->local]->get($key,$flags);
            $len = strlen($this->secret);
            if($ret != FALSE && is_string($ret) && strlen($ret) > $len && substr($ret,0,
                 strlen($this->secret)) == $this->secret){
                $ret = $this->racks[substr($ret,$len)]->get($key,$flags);
                if($ret === FALSE){
                    // Data not where it should be! remove all notes!
                    foreach($this->racks as $k => &$r){
                            $r->delete($key);
                    }
                }
            }
            return $ret;
        }
/*************************/
        function replace($key, $value, $flags = '',$expire = 0){
            // Replace should return false if the item doesn't already exist.
            // But, so that we don't error to early (new rack being brought up),
            //  we'll only error if the local copy doesn't exist
            // and use sets otherwise
            $return =  $this->racks[$this->local]->replace($key,$value,$flags,$expire);
            if(!$return){return false;}
            // unfortunatly we must update the expire times on all notes
            foreach($this->racks as $k => &$r){
                if($k != $this->local){ // Don't update twice!
                    $r->set($key,$this->secret . $this->local,$flags,$expire);
                }
            }
            return true;
        }
/*************************/
        function decrement($key, $amt = 1){
            // Decrement wherever it is, if local re-set all notes
            // Return value should be result after decrement, but possibility
            //  exists for inconsistent result.
            $ret = $this->racks[$this->local]->get($key);
            $len = strlen($this->secret);
            if($ret != FALSE && is_string($ret) && strlen($ret) > $len && substr($ret,0,
                 strlen($this->secret)) == $this->secret){
                //its somewhere else! decrement that!
                $ret = $this->racks[substr($ret,$len)]->decrement($key,$flags);
                if($ret === FALSE){
                    // Data not where it should be! remove all notes!
                    foreach($this->racks as $k => &$r){
                            $r->delete($key);
                    }
                }
            }else if($ret != FALSE){
                // must be local, try to decrement
                $ret = $this->racks[$this->local]->decrement($key,$amt);
                $flags = 0;
                $expire = 0; // PGT FIX
                // Touch notes to premote in LRU
                foreach($this->racks as $k => &$r){
                    if($k != $this->local){
                        $r->set($key,$this->secret . $this->local,$flags,$expire);
                    }
                }
            }
            return $ret;
        }
/*************************/
```

```
        function increment($key, $amt = 1){
            // Increment on all racks
            // Return value should be result after increment, but possibility
            //  exists for inconsistent result.
            // Here we return the local value, though we could also return
            //  false if ANY returned a non-consistent value.
            $ret = $this->racks[$this->local]->get($key);
            $len = strlen($this->secret);
            if($ret != FALSE && is_string($ret) && strlen($ret) > $len && substr($ret,0,
                strlen($this->secret)) == $this->secret){
                //its somewhere else! decrement that!
                $ret = $this->racks[substr($ret,$len)]->increment($key,$flags);
                if($ret === FALSE){
                    // Data not where it should be! remove all notes!
                    foreach($this->racks as $k => &$r){
                            $r->delete($key);
                    }
                }
            }else if($ret != FALSE){
                // must be local, try to decrement
                $ret = $this->racks[$this->local]->increment($key,$amt);
                $flags = 0;
                $expire = 0; // PGT FIX
                // Touch notes to premote in LRU
                foreach($this->racks as $k => &$r){
                    if($k != $this->local){
                        $r->set($key,$this->secret . $this->local,$flags,$expire);
                    }
                }
            }
            return $ret;
        }
/*************************/
        function set($key,&$value,$flag = 0,$expire = 0){
            // Set data on local, and note on all others
            $return = $this->racks[$this->local]->set($key,$value,$flag,$expire);
            if($return == FALSE){return FALSE;}
            foreach($this->racks as $k => $r){
                if($k != $this->local){ // Don't update twice!
                    $this->racks[$k]->set($key,$this->secret . $this->local,$flag,$expire);
                }
            }
            return true;
        }
/*************************/
        function setCompressThreshold($limit){
            // Set limit in all racks
            foreach($this->racks as &$r){
                $r->setCompressThreshold($limit);
            }
        }
/*************************/
        function getStats($type, $slabid, $limit = 100){
            return $this->racks[$this->local]->getStats($type,$slabid,$limig);
        }
/*************************/
        function findServer($key){
            return $this->racks[$this->local]->findServer($key);
        }
}
?>
```

Appendix E

Dir: Mem_dir.php

```
<?php
    /*
     Copyright (c) 2012, Paul G Talaga
     All rights reserved.
     Redistribution and use in source and binary forms, with or without
     modification, are permitted provided that the following conditions are met:
     * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
     * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
     ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED. IN NO EVENT SHALL Paul G Talaga BE LIABLE FOR ANY
     DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
     (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
     ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
     (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
     SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
     */
    // Wrapper class to memcache, provides directory based cache locality using
    //  a centralized memcache cluster for note storage which keeps track of where
    //  all actual data is stored or replicated.  This centralized cluster is
    //  specified with rack 0, and the Local rack can't be 0!  Hard coded is a
    //  replication
    //  value so local copies don't take over everying.  This makes deletion a
    //  little easier and also allows clients to choose the 'closest' local copy
    //  to use.
    //  The first listed cluster is where the truth is, so if any error occurs,
    //  replace the value from there.
    // Due to this being in PHP, we can't issue simultaneous requests to all
    //  racks at once (cross pools).  Thus, we'll just have to deal with the
    //  slowdown until a C implementation is done.
    // Compression is turned OFF for centralized note storage so append works
    //  as desired.
    // For atomic centralized note update we need append, thus
    //  we need PECL Memcache > 3.0.0.
    // TODO: Think about what expire time to use on local copy.
    //  Actual expire value not available, maybe store in note?
    include_once('Mem_RackAware.php');
    class Memcache_Dir extends Memcache_RackAware{
        // constructor function
        function __construct(){
            $this->dupLimit = 2;
            // A note is of the form <rack>,<rack>,...,<rack> up to dupLimit.
            $this->rackConstruct();
            // constructor for RackAware, provides $this->local which is the rack
            // number this machine should use, and $this->racks which
            // is an array of Memcache instances indexed by rack number.
            //  So $this->racks[$this->local] would be the closest memcache
            // instance.
                                //echo "Dir\n";
        }
        // addServer      provided by Memcache_RackAware
/***********************************************************/
        function add($key,$value,$flags = 0,$expire = 0){
            // Add item with key
            // First check to see if its in local by doing an append of ''.
            // This will return true if it exists (and won't touch it, or
```

```
            //  change expire)
            //  or false if it doesn't exist.
            if($this->racks[$this->local]->append($key,'') === TRUE){
                return false;   // exists in local cache, so must exist in
                                // centralized cluster as well
            }
            // now add note to centralized, return false if it fails (stored elsewhere)
            if(!$this->racks[0]->add($key,$this->local,$flags,$expire)){
                // Just store this rack number, replicated racks will use comma
                return false; // exists in centralized cluster, so return false
            }
            // Now we've got to put the data in local, so use a set to be sure
            $this->racks[$this->local]->set($key,$value,$flags,$expire);
            return true;
        }
/**************************/
        function connect($host,$port,$timeout = 1){
            return self::addServer($host,$port,TRUE,1,$timeout);
        }
/**************************/
        function delete($key,$timeout = 0){
            // Delete data with key.  We'll only return failure if
            //  centralized failed
            // Get duplication info
            $dup = $this->racks[0]->get($key);
            if($dup === FALSE){
                return false;   // doesn't exists in centralized cluster,
                                //  so return false
            }
            // Delete centralized and duplicates
           $this->racks[0]->delete($key,$timeout);
           foreach(explode(',',$dup) as $d){
                $this->racks[$d]->delete($key,$timeout);
           }
           return TRUE;
        }
/**************************/
        function flush(){
            // flush all racks!
           foreach($this->racks as $r){
                $r->flush();
            }
        }
/**************************/
        function get($key, &$flags = 0, &$cas = ''){
            // Validate all these flags and cas stuff works
            // Get local, otherwise get note and get it
            $ret = $this->racks[$this->local]->get($key,$flags,$cas);
            if($ret !== FALSE){
                return $ret;
            }
            // go get note
            $note = $this->racks[0]->get($key);
            if($note === FALSE){
                return FALSE;
            }
            $rep = explode(',',$note);
            $order = range(0,count($rep) - 1);
            // here we could optimize by choosing a close copy
            shuffle($order);
            $succ = 0;
            foreach($order as $o){
                $ret = $this->racks[$rep[$o]]->get($key,$flags,$cas);
                if($ret !== FALSE){$succ = 1; break;}
            }
            if($succ == 0){return FALSE;}
            // move local?
            if(count($rep) < $this->dupLimit){
                $this->racks[0]->append($key,',' . $this->local);
                $this->racks[$this->local]->set($key,$ret); // never expire
            }
           return $ret;
        }
/**************************/
        function replace($key, $value,$flag = 0,$expire = 0){
            // Replace should return false if the item doesn't already exist.
            // So check to see if a note exists on the central, and if so just
            //  do a set so it always works
            if($this->racks[0]->get($key) !== FALSE){
                return $this->set($key,$value,$flag,$expire);
            }
            return false;
        }
/**************************/
```

```
	function decrement($key, $amt = 1, $exptime = 0){
		// First get note, if it exists do a decrement on all, and if the
		//  others don't match the first, set them to match
		// Optionaly we could remove all dups and let gets re-distribute
		$flags = 0;
		$note = $this->racks[0]->get($key,$flags);
		if($note === FALSE){
			return FALSE;
		}
		$rep = explode(',',$note);
		$correct = $this->racks[$rep[0]]->decr($key,$amt,$exptime);
		foreach($rep as $r){
			if($r =! $rep[0]){
				$n = $this->racks[$r]->decr($key,$amt,$exptime);
				if($n =! $correct){
					$this->racks[$r]->set($key,$correct,$exptime);
				}
			}
		}
		return $correct;
	}
/*************************/
	function increment($key, $amt = 1, $exptime = 0){
		// First get note, if it exists do a increment on all, and if the
		//  others don't match the first, set them to match
		// Optionaly we could remove all dups and let gets re-distribute
		$flags = 0;
		$note = $this->racks[0]->get($key,$flags);
		if($note === FALSE){
			return FALSE;
		}
		$rep = explode(',',$note);
		$correct = $this->racks[$rep[0]]->incr($key,$amt,$exptime);
		foreach($rep as $r){
			if($r =! $rep[0]){
				$n = $this->racks[$r]->incr($key,$amt,$exptime);
				if($n =! $correct){
					$this->racks[$r]->set($key,$correct,$exptime);
				}
			}
		}
		return $correct;
	}
/*************************/
	function set($key,$value,$flags = 0,$expire = 0){
		// a Set clears out duplicates!
		// we must get the note before overwriting it so to delete
		$note = $this->racks[0]->get($key);
		if($note !== FALSE){
			$rep = explode(',',$note);
			foreach($rep as $r){
				$this->racks[$r]->delete($key);
			}
		}
		// now add note to centralized
		if(!$this->racks[0]->set($key,$this->local,$flags,$expire)){
			// Just store this rack number, replicated racks will use comma
			return FALSE;
		}
		// Now we've got to put the data in local
		if(!$this->racks[$this->local]->set($key,$value,$flags,$expire)){
			// hm, local failed, so remove central note
			$this->racks[0]->delete($key);
			return FALSE;
		}
		//we'll return false if any returned false, though you won't know
		//  which rack failed.
		return TRUE;
	}
	function findServer($key){
		return $this->racks[$this->local]->findServer($key);
	}

}
?>
```

Appendix F

MemcacheTach: memcache-logging.php

```
<?php
    /*
     Copyright (c) 2012, Paul G Talaga
     All rights reserved.
     Redistribution and use in source and binary forms, with or without
     modification, are permitted provided that the following conditions are met:
     * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
     * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
     ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED. IN NO EVENT SHALL Paul G Talaga BE LIABLE FOR ANY
     DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
     (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
     ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
     (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
     SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
     */
    // Wrapper class to PECL memcache, which logs all memcache commands
    error_reporting(E_ALL);
    class Memcache_Logging extends Memcache{
        // constructor funciton
        function __construct(){
            $this->log = '';
            $this->unique = rand();
            $this->save_log = '/tmp/stress_validate_log.txt';
            $this->servers = array();
            $this->counter = 0;
        }
        function __destruct(){
            $this->save_log();
        }
        function save_log(){
            $fp = fopen($this->save_log,'a');
            flock($fp, LOCK_EX);
            fwrite($fp,$this->log); // Plain text output.  Consider compressing
            //  for space.  Compressing each line isn't very efficient.
            flock($fp, LOCK_UN);
            fclose($fp);
            $this->log = '';
        }
        function log($cmd,$key,$out_len,$found,$params,$delay){
            $this->counter++;
            $cookie = var_export($_COOKIE,true);
            // 1/10 second accuracy for time
            if($key !== ''){$dest_srv = parent::findServer($key);}else{$dest_srv='all';}
            list($usec, $sec) = explode(" ", microtime());
            $time = $sec * 10 + round($usec * 10);
            $data = serialize(array($time,substr(md5($cookie),0,2), isset($_SERVER['
                SERVER_ADDR'])?$_SERVER['SERVER_ADDR']:'0.0.0.0',$dest_srv,$cmd,$key,
                $out_len,$found,$params,$delay,substr(md5($_SERVER["REQUEST_TIME"] .
                $_SERVER['SERVER_ADDR'] . $_SERVER["REMOTE_ADDR"]),0,5)));
            $this->log .= $data . "\n" ;
```

```
}
function log_file($log){
    if($log){
        $this->save_log = $log;
    }
    return $this->save_log;
}
function add(&$key,&$value,$flags = 0,$expire = 0){
    $now = microtime(true);
    $result = parent::add($key,$value,$flags,$expire);
    self::log('add',$key, self::getNetlength($value), $result,serialize(array(
        $flags,$expire)),microtime(true)-$now);
    return $result;
}
function delete(&$key,$expire = 0){// TODO: Handle array
    $now = microtime(true);
    $data = parent::delete($key,$expire);
    self::log('delete',$key,0,$data,serialize(array($expire)),microtime(true)-$now)
        ;
    return $data;
}
function flush(){ // We should really find out how many memcache servers
                  // there are to predict total network bytes sent
    $now = microtime(true);
    $return = parent::flush();
    self::log('flush','',0,'',serialize(array()),microtime(true)-$now);
    return $return;
}
function get(&$key,&$flags = null,&$cas = null){
    if(is_array($key)){ // multiget - we save everything as one entry,
                        // with overall time recorded
        $succ = array();     // the keys get saved as indexes to the
                             // destination server
        $lengths = array();
        $dest = array();
        $now = microtime(true);
        $data = parent::get($key,$flags,$cas);
        $i = 0;
        foreach ($data as $k => $v){
            $key2[$i] = $k;
            if($v === FALSE){$succ[$k] = FALSE;
            }else{$succ[$i] = TRUE;}
            $lengths[$i] = self::getNetlength($v);
            $i++;
        }
        self::log('multiget',serialize($key2),serialize($lengths),serialize($succ)
            ,0,microtime(true)-$now);
    }else{ //single get
        $now = microtime(true);
        $data = parent::get($key,$flags,$cas);
        self::log('get',$key,self::getNetlength($data),!($data === FALSE),serialize
            (array($flags,$cas)),microtime(true)-$now);
    }
    return $data;
}
function replace(&$key, &$value, $flags = 0,$expire = 0){
    $now = microtime(true);
    $result = parent::replace($key,$value,$flags,$expire);
    self::log('replace',$key,self::getNetlength($value), $result,serialize(array(
        $flags,$expire)),microtime(true)-$now);
    return $result;
}
function decrement(&$key, $amt = 1){ // TODO: Handle array
    $now = microtime(true);
    $result = parent::decrement($key,$amt);
    self::log('decr',$key,self::getNetlength($amt) + self::getNetlength($result),
        $result,serialize(array()),microtime(true)-$now);
    return $result;
}
function increment(&$key, $amt = 1){ // TODO: Handle array
    $now = microtime(true);
    $result = parent::increment($key,$amt);
    self::log('incr',$key,self::getNetlength($amt) + self::getNetlength($result),
        $result,serialize(array()),microtime(true)-$now);
    return $result;
}
function set(&$key,&$value,$flags = 0,$expire = 0){
    $now = microtime(true);
    $result = parent::set($key,$value,$flags,$expire);
    if($result == FALSE){trigger_error("Set fail on $key");};
    self::log('set',$key,self::getNetlength($value), $result,serialize(array($flags
        ,$expire)),microtime(true)-$now);
    return $result;
```

```
        }
        function append(&$key,&$value, $flags = 0,$expire = 0){
            $now = microtime(true);
            $result = parent::append($key, $value, $flags, $expire);
            self::log('append',$key, self::getNetlength($value), $result, serialize(array(
                $flags, $expire)), microtime(true)-$now);
            return $result;
        }
        function prepend(&$key,&$value, $flags = 0,$expire = 0){
            $now = microtime(true);
            $result = parent::prepend($key, $value, $flags, $expire);
            self::log('prepend',$key, self::getNetlength($value), $result, serialize(array(
                $flags, $expire)), microtime(true)-$now);
            return $result;
        }
        function getNetlength(&$value){
            // returns the number of bytes which should be put on the wire for
            //  this object
            if(is_string($value)){
                return strlen($value);
            }else if(is_int($value)){
                return strlen(strval($value)) + 2; // for flags
            }else if(is_float($value)){
                return strlen(sprintf('%.14g',$value)) + 3; // for flags
            }else if(is_bool($value)){
                return 1;  // 0 or 1
            }else{
                return strlen(serialize($value));
            }
        }
    }
?>
```

Appendix G

MemcacheTach: analyse.php

```
#!/usr/bin/php
<?php
 /*
     Copyright (c) 2012, Paul G Talaga
     All rights reserved.
     Redistribution and use in source and binary forms, with or without
     modification, are permitted provided that the following conditions are met:
     * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
     * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
     ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED. IN NO EVENT SHALL Paul G Talaga BE LIABLE FOR ANY
     DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
     (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
      LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
     ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
     (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
     SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/
/*
     analyse.php - Command-line PHP script to analyse MemcacheTACH log output.
     Usage (typical):     php analyse.php log.txt              - Analyze log.txt
           (CSV export): php analyse.php log.txt out.csv      - Analyze log.txt and export
                result (append) to out.csv
     For correct rack analysis be sure to fill in below.
*/
// Rack config for rack latency analysis
/* $rack is an associative array which maps an IP address to a rack number.  The IP can be
      of the webserver or the
   memcache server.  For memcache be sure to include the port number as well.  The rack
        number can be any integer, but
   memcache servers use a negative of the rack in which it belongs.
*/
$pre = '128.230.109.';  // Helpful prefix values
$pre2 = '192.168.1.';
$rack = array();
$rack[$pre . 49] = 1;
$rack[$pre . 50] = 1;
$rack[$pre . 51] = 1;
$rack[$pre . 52] = 1;
$rack[$pre2 . 52 . ':11211'] = -1;
$rack[$pre . 70] = 2;
$rack[$pre . 53] = 2;
$rack[$pre . 54] = 2;
$rack[$pre . 55] = 2;
$rack[$pre . 56] = 2;
$rack[$pre2 . 56 . ':11211'] = -2;
$rack[$pre . 69] = 3;
$rack[$pre . 57] = 3;
$rack[$pre . 58] = 3;
$rack[$pre . 59] = 3;
$rack[$pre . 60] = 3;
$rack[$pre2 . 60 . ':11211'] = -3;
$rack[$pre . 61] = 4;
$rack[$pre . 62] = 4;
$rack[$pre . 63] = 4;
$rack[$pre . 64] = 4;
```

```
$rack[$pre2 . 64 . ':11211'] = -4;
$rack[$pre . 65] = 5;
$rack[$pre . 66] = 5;
$rack[$pre . 67] = 5;
$rack[$pre . 68] = 5;
$rack[$pre2 . 68 . ':11211'] = -5;
$rack[$pre2 . 71 . ':11211'] = -6;
// ****************************************
    if(isset($argv[1])){
        $log_file = $argv[1];
        echo "Analyzing $log_file ...\n\n";
    }else{
        echo "Enter log file to analyze as command line arguement.\n";
        die;
    }
    if(isset($argv[2])){
        $csv_file = $argv[2];
        echo "Using $csv_file for row output.";
    }else{
        $csv_file = FALSE;
    }
    $sim_file = FALSE;
$wnf = FALSE; // Warn when an item was found by memcache, but not seen previously in our
    log file
//$fudge_skip = array(0,421177, 421267); // Fill with line number to skip
$fudge_skip = array();
// Network latency/bandwidth calculation
//$stats = array('hits_log' => 0, 'hits_sim' => 0, 'miss_log' => 0, 'miss_sim' => 0, '
    total_time_log' => 0, 'total_time_sim' => 0,'sets' => 0, 'gets' => 0, 'st' => 0);
$tspan = array('start' => 0, 'end' => 0, 'last' => 0);
$plots = array(); //hit_real, hit_sim, hit_diff, miss_real, miss_sim, miss_dif,
    http_num_requests, http_latency, http
$conv = array();
$xmean = array();
$ymean = array();
$changecnt = 0;
$num_samples = array();
$num_samples_multiget = 0; // multiget for now does not keep track of the server the
    requests go to, so sum separately
$sum_top = array();
$sum_bottom = array();
$sum_objects = 0;
$num_object_transfer = 0;
$i = 0;
$mymiss = 0;
$mysetmiss = 0;
$singlecount = 0;
$writebytes = 0;
$readbytes = 0;
$lastnet = 0;
$size = array(); // store how large objects are for use in later gets
/*b1 =    [ (xi - xb)(yi - yb) ] /    [ (xi - xb)2]
b1 = r * (sy / sx)
b0 = yb - b1 * xb
 three passes: 1:means, 2:regression, 3: Residual calculation
 also do hit and miss calculation
*/
$fh = fopen($log_file,'r'); // assume file is in time order!  You can cat each log file
    together then do sort
while($line = fgets($fh)){
  $i++;
  if( strlen($line) < 10 || in_array($i,$fudge_skip))continue;
    $data =  unserialize($line);
    if($data === FALSE){echo "Unserialize error on line $i - ",strlen($line),"!!!\n\n$data\
        n\n";continue;}
    list($time,$cookie,$sv,$msvo,$cmd,$key,$out_len,$found,$params,$delay,$conversation) =
        $data;
    //if($msvo == ''){echo "Time: $time\nCookie: $cookie\nFrom Server: $sv\nMemcache Server
        : $msvo\nCommand: $cmd\nKey: $key\nLength: $out_len\nFound? $found\n Params:--\
        nDelay: $delay\nConv: $conversation\n";}
    // Make sure time goes forward!
    if($tspan['start'] == 0 || $tspan['start'] > $time){$tspan['start'] = $time;}
    //if($tspan['end'] < $time && $sv != '128.230.109.69'){$tspan['end'] = $time;}     //
        PGT 69 removal!!!!!
    if($tspan['last'] > $time){echo "Log error! Not sorted! Continuing anyway.\n"; }
    $tspan['last'] = $time;
    // conversation tracking
    if(!isset($conv[$conversation])){$conv[$conversation] = array('num_actions' => 0, '
        tot_time' => 0, 'end_time' => 0, 'hits' => 0, 'misses' => 0,'tot_sim_time' => 0);}
    $conv[$conversation]['num_actions']++;
    $conv[$conversation]['tot_time'] += $delay;
    $conv[$conversation]['end_time'] = $time; // last will stick
    //
```

```
// $out_len needs to be more complicated so to measure the entire length on network
// also, a get of a found key needs to transfer data, so keep track
$msv = $sv . '->' . $msvo; // track each webserver to each memcache server
if(!isset($xmean[$msv])){$xmean[$msv] = 0; $ymean[$msv] = 0; $num_samples[$msv] = 0;
    $sum_top[$msv] = 0; $sum_bottom[$msv] = 0;}; // initialize to zero
$lastnet = $xmean[$msv]; // for read/write byte counting
if($cmd == 'set'){
  $xmean[$msv] += $out_len +  strlen($key) + strlen($out_len) +13 + 8;  //8 is response
  $size[$key] = $out_len;
  $sum_objects += $out_len;
  $num_object_transfer += 1;
  $changecnt += 1; // count the number of 'sets' issued
  if($found){$conv[$conversation]['hits']++;}else{$conv[$conversation]['misses']++;}
  if($found == FALSE){
    echo "Set bad: $key $mysetmiss $sv $msvo Len:$out_len\n";
    $mysetmiss++;
  }else if($out_len == 0){
    echo "Zero length!!\n";
  }
}else if($cmd == 'add'){
  if($found){
    $xmean[$msv] += $out_len + strlen($key) + strlen($out_len) + 21;
  }else{
    $xmean[$msv] += $out_len + strlen($key) + strlen($out_len) + 25;
  }
  $size[$key] = $out_len;
  $sum_objects += $out_len;
  $num_object_transfer += 1;
  $changecnt += 1; // count the number of 'sets' issued
  if($found){$conv[$conversation]['hits']++;}else{$conv[$conversation]['misses']++;}
}else if($cmd == 'replace'){
  if($found){
    $xmean[$msv] += $out_len + strlen($key) + strlen($out_len) + 25;
  }else{
    $xmean[$msv] += $out_len + strlen($key) + strlen($out_len) + 29;
  }
  $size[$key] = $out_len;
  $sum_objects += $out_len;
  $num_object_transfer += 1;
  $changecnt += 1; // count the number of 'sets' issued
  if($found){$conv[$conversation]['hits']++;}else{$conv[$conversation]['misses']++;}
}else if($cmd == 'append'){
  if($found){
    $xmean[$msv] += $out_len + strlen($key) + strlen($out_len) + 24;
  }else{
    $xmean[$msv] += $out_len + strlen($key) + strlen($out_len) + 28;
  }
  $size[$key] = +$out_len;
  //$sum_objects += $out_len;
  //$num_object_transfer += 1;
  $changecnt += 1; // count the number of 'sets' issued
  if($found){$conv[$conversation]['hits']++;}else{$conv[$conversation]['misses']++;}
}else if($cmd == 'prepend'){
  if($found){
    $xmean[$msv] += $out_len + strlen($key) + strlen($out_len) + 25;
  }else{
    $xmean[$msv] += $out_len + strlen($key) + strlen($out_len) + 29;
  }
  $size[$key] += $out_len;
  //$sum_objects += $out_len;
  //$num_object_transfer += 1;
  $changecnt += 1; // count the number of 'sets' issued
  if($found){$conv[$conversation]['hits']++;}else{$conv[$conversation]['misses']++;}
}else if($cmd == 'get'){
  if($found && isset($size[$key])){
    $xmean[$msv] += strlen($key)  + $size[$key] + strlen($key) + strlen($size[$key]) +
         30; // includes 4 digit cas
    $sum_objects += $size[$key];
    $num_object_transfer += 1;
  }else if(isset($size[$key])){
    if($found && $wnf){echo "$key not defined - get\n";}
    $xmean[$msv] += strlen($key) + 12;
  }else{
    if($found && $wnf){echo "$key not defined - get\n";}
    $xmean[$msv] += strlen($key) + 12;
  }
  if($found){$conv[$conversation]['hits']++;}else{$conv[$conversation]['misses']++;}
}else if($cmd == 'multiget'){
  $keys = unserialize($key);
  $lengths = unserialize($out_len);
  $founds = unserialize($found);
  $num_object_transfer += 1;
  foreach($keys as $i => $k){
      if($founds[$i] && isset($size[$k])){
```

```
            $xmean[$msv] += strlen($size[$k]) + $size[$k] + $lengths[$i];
            $sum_objects += $size[$k];
          }else{
            if($founds[$i] && $wnf){echo "$key not defined - multiget\n";}
            $xmean[$msv] += 4 + strlen($k) + 2 + 5;
          }
          if($founds[$i]){$conv[$conversation]['hits']++;}else{$conv[$conversation]['hits'
              ]++;}
      }
      //$num_samples_multiget++;
     }else if($cmd == 'delete'){
        if($found){
            $xmean[$msv] += strlen($key) + 18;
        }else{
            $xmean[$msv] += strlen($key) + 20;
        }
        if(isset($size[$key])){
          //unset($size[$key]); // Don't delete, compute stats later
        }
        $changecnt += 1; // count the number of 'sets' issued
        if($found){$conv[$conversation]['hits']++;}else{$conv[$conversation]['misses']++;}
    }else if($cmd == 'incr'){
        if($found){$xmean[$msv] += strlen($key) + $out_len + 6;
        }else{$xmean[$msv] +=  strlen($key) + $out_len + 17;};
        $changecnt += 1; // count the number of 'sets' issued
        if($found){$conv[$conversation]['hits']++;}else{$conv[$conversation]['misses']++;}
    }else if($cmd == 'decr'){
        if($found){$xmean[$msv] += strlen($key) + $out_len + 6;
        }else{$xmean[$msv] +=  strlen($key) + $out_len + 17;};
        $changecnt += 1; // count the number of 'sets' issued
        if($found){$conv[$conversation]['hits']++;}else{$conv[$conversation]['misses']++;}
    }else if($cmd == 'flush'){
        $xmean[$msv] += 15;
        $changecnt += 1; // count the number of 'sets' issued
    }else{
      echo "Did not understand $cmd!!\ on line $i n";
      die;
    }
    $ymean[$msv] += $delay;
    $num_samples[$msv]++;
    // Read/write data counting
    if($cmd == 'get'){ // read attemp!
        $readbytes += ($xmean[$msv] - $lastnet);
    }else{ // write attemp
        $writebytes += ($xmean[$msv] - $lastnet);
    }
}
foreach($xmean as $k => $d){
    $xmean[$k] = $xmean[$k] / $num_samples[$k];
    $ymean[$k] = $ymean[$k] / $num_samples[$k];
}
rewind($fh);
$i = 0;
while($line = fgets($fh)){
    $i++;
    if( strlen($line) < 10 || in_array($i,$fudge_skip))continue;
    $data =  unserialize($line);
    if($data === FALSE){echo "Unserialize error on line $i!!!\n";continue;}
    list($time,$cookie,$sv,$msvo,$cmd,$key,$out_len,$found,$params,$delay,$conversation) =
        $data;
    $msv = $sv . '->' . $msvo; // track each webserver to each memcache server
    $sum_top[$msv] += ($out_len - $xmean[$msv])*($delay - $ymean[$msv]);
    $sum_bottom[$msv] += pow($out_len - $xmean[$msv],2);
    // fill in sets and gets in plot
    $pi = round(($time - $tspan['start'])/($tspan['end'] - $tspan['start']) * $pimax);
    if($cmd == 'set' || $cmd == 'add'){$plots[$pi]['writes']++;} // deal with delete
    if($cmd == 'get'){$plots[$pi]['reads']++;}
    if($cmd == 'multiget'){$keys = unserialize($key); foreach ($keys as $key){$plots[$pi]['
        reads']++;};};
    // Sim stuff
    if($sim_file){
        if(isset($sim[substr(md5($line),0,10)])){$s = $sim[substr(md5($line),0,10)];}else{
            echo "sim not found!\n";}
        if($cmd == 'get' && $s['h'])$plots[$pi]['hit_sim']++;
        if($cmd == 'get' && !$s['h'])$plots[$pi]['miss_sim']++;
    }
}
echo "Time start (x10): ",$tspan['start']," end: ",$tspan['end']," total seconds: ",($tspan
    ['end'] - $tspan['start'])/10,"\n\n";
echo "Network analysis for all paths based on latency and packet size\n";
$csv_line .= ($tspan['end'] - $tspan['start'])/10 . ",";
$latency_predict = array(); // used later
```

```php
$server_memserver_num_requests = array();
$latency_rack = array();
// prefill in zeros for rack analysis
foreach($rack as $rn){
    if($rn < 0){$rn = -$rn;}
    $latency_rack[$rn] = array('intersum' => 0, 'intersum2' => 0, 'intercnt' => 0, 'intrasum' => 0, 'intrasum2' => 0, 'intracnt' => 0);
}
foreach($xmean as $k => $d){
    $slope = $sum_top[$k]/$sum_bottom[$k];
    $int = $ymean[$k] - $slope * $xmean[$k];
    if($num_samples[$k] == 1 || $slope == 0){// Can't do regression on a single point!
        echo "$k slope: --(single sample) mbps latency: ", round($int * 1000000) / 1000," (ms) \n    size_mean: ",round($xmean[$k] / 1024 * 1000)/1000,"(kB) time_mean: ", round($ymean[$k] * 1000000) / 1000," (ms) num_requests*: ",$num_samples[$k],"\n\n";
    }else{
        echo "$k slope: ",round(1/$slope /1024 / 1024 * 8 * 10) / 10 ," mbps latency: ", round($int * 1000000) / 1000," (ms) \n    size_mean: ",round($xmean[$k] / 1024 * 1000)/1000,"(kB) time_mean: ",round($ymean[$k] * 1000000) / 1000," (ms) num_requests*: ",$num_samples[$k],"\n\n";
    }
    // analyze racks                    weight with # samples, notice square fix below
    $use_this_latency = $int * 1000 * $num_samples[$k]; // $int is predicted based on slope (0 crossing), $ymean[$k] is average of all latencies
    list($from,$to) = explode('->',$k);
    // handle from and to counting
    if(!array_key_exists($from,$server_memserver_num_requests)){
        $server_memserver_num_requests[$from] = 0;}
    if(!array_key_exists($to,$server_memserver_num_requests)){
        $server_memserver_num_requests[$to] = 0;}
    $server_memserver_num_requests[$from] += $num_samples[$k];
    $server_memserver_num_requests[$to] += $num_samples[$k];
    //update rack counts
    if(!(array_key_exists($from,$rack)) || !array_key_exists($to,$rack)){echo "What rack is $from or $to?!?\n"; continue;}
    if($rack[$from] == -$rack[$to]){// same rack
        //echo "Same Rack\n";
        $latency_rack[$rack[$from]]['intrasum'] += $use_this_latency;
        $latency_rack[$rack[$from]]['intrasum2'] += $use_this_latency * $use_this_latency / $num_samples[$k]; // Used for variance calculation
        $latency_rack[$rack[$from]]['intracnt']+= $num_samples[$k];
    }else{
        $latency_rack[$rack[$from]]['intersum'] += $use_this_latency;
        $latency_rack[$rack[$from]]['intersum2'] += $use_this_latency * $use_this_latency / $num_samples[$k];
        $latency_rack[$rack[$from]]['intercnt']+= $num_samples[$k];
    }
}
echo "\nMemcache Client & Server messages passed\n";
ksort($server_memserver_num_requests);
foreach($server_memserver_num_requests as $k => $d){
    echo "     $k   $d   messages (sent/received)\n";
}
echo "\n\nRack Latency Stats, based on accumulated source/dests\n";
// latency clustering and averaging
// Intra rack mean/varaince
// Inter rack mean/variance
$ltotals = array('intercnt' => 0, 'intersum' =>0, 'intracnt' => 0, 'intrasum' =>0);
$intercnt = 0;
$intracnt = 0;
foreach($latency_rack as $r => $d){
    if( $d['intracnt'] == 0) {$d['intracnt'] = 1;}
    if( $d['intercnt'] == 0) {$d['intercnt'] = 1;}
    $intramean = $d['intrasum'] / $d['intracnt'];
    $intermean = $d['intersum'] / $d['intercnt'];
    echo "Rack $r, within ", round(1000 * $intramean)/1000, ' (ms) var ', round(1000 * ($d['intrasum2']/$d['intracnt'] - $intramean * $intramean))/1000,"\n";
    echo "       leaving ", round(1000 * $intermean)/1000, ' (ms) var ', round(1000 * ($d['intersum2']/$d['intercnt'] - $intermean * $intermean))/1000,"\n";
    echo "       intracnt: ", $d['intracnt']," intercnt: ", $d['intercnt'],"\n\n";
    $ltotals['intercnt'] += $d['intercnt'];
    $ltotals['intracnt'] += $d['intracnt'];
    $ltotals['intersum'] += $intermean * $d['intercnt'];
    $ltotals['intrasum'] += $intramean * $d['intracnt'];
}
echo "Averaged over all racks (count weighted), within is ",round(1000* $ltotals['intrasum'] / $ltotals['intracnt'])/1000, ', leaving is ',round(1000* $ltotals['intersum'] / $ltotals['intercnt'])/1000,"\n\n";
$csv_line .= (1000* $ltotals['intrasum'] / $ltotals['intracnt'])/1000 . ',';
$csv_line .= (1000* $ltotals['intersum'] / $ltotals['intercnt'])/1000 . ',';
// Do the same, but look at each datapoint
```

G. MEMCACHETACH: ANALYSE.PHP

```
$intra_inter_rack = array('intersum' => 0, 'intersum2' => 0, 'intercnt' => 0,  'intrasum'
    => 0, 'intrasum2' => 0, 'intracnt' => 0);
rewind($fh);
$i = 0;
while($line = fgets($fh)){
    $i++;
    if( strlen($line) < 10 || in_array($i,$fudge_skip))continue;
    $data =  unserialize($line);
    if($data === FALSE){echo "Unserialize error on line $i!!!\n";continue;}
    list($time,$cookie,$sv,$msvo,$cmd,$key,$out_len,$found,$params,$delay,$conversation) =
        $data;
    $from = $sv;
    $to = $msvo;
    if(!(array_key_exists($from,$rack)) || !array_key_exists($to,$rack)){echo "What rack is
        $from or $to?!?\n"; continue;}
    $delay = $delay * 1000;
    if($rack[$from] == -$rack[$to]){// same rack
        $intra_inter_rack['intrasum'] += $delay ;
        $intra_inter_rack['intrasum2'] += $delay * $delay; // Used for variance calculation
        $intra_inter_rack['intracnt']+= 1;
    }else{
        $intra_inter_rack['intersum'] += $delay;
        $intra_inter_rack['intersum2'] +=$delay * $delay; // Used for variance calculation
        $intra_inter_rack['intercnt']+= 1;
    }
}
if($intra_inter_rack['intracnt'] == 0){$intra_inter_rack['intracnt'] = 1;};
$intramean = $intra_inter_rack['intrasum'] / $intra_inter_rack['intracnt'];
$intermean = $intra_inter_rack['intersum'] / $intra_inter_rack['intercnt'];
echo "Averaged over all messages, within a rack is ",round(1000* $intra_inter_rack['
    intrasum'] / $intra_inter_rack['intracnt'])/1000, ' (',$intra_inter_rack['intrasum2'] /
     $intra_inter_rack['intracnt'] - $intramean * $intramean ,' var, ',$intra_inter_rack['
    intracnt'],' samples), leaving is ',round(1000* $intra_inter_rack['intersum'] /
    $intra_inter_rack['intercnt'])/1000,' (',$intra_inter_rack['intersum2'] /
    $intra_inter_rack['intercnt'] - $intermean * $intermean, " var, ",$intra_inter_rack['
    intercnt']," samples)\n\n";
$csv_line .= $intra_inter_rack['intrasum'] / $intra_inter_rack['intracnt'] . ',';
$csv_line .= $intra_inter_rack['intrasum2'] / $intra_inter_rack['intracnt'] - $intramean *
    $intramean . ',';
$csv_line .= $intra_inter_rack['intracnt'] . ',';
$csv_line .= $intra_inter_rack['intersum'] / $intra_inter_rack['intercnt'] . ',';
$csv_line .= $intra_inter_rack['intersum2'] / $intra_inter_rack['intercnt'] - $intermean *
    $intermean . ',';
$csv_line .= $intra_inter_rack['intercnt'] . ',';
// Request analysis
// calc averages
$num_actions = 0;
$delays = 0;
$i = 0;
foreach($conv as $c){
    $num_actions += $c['num_actions'];
    $delays += $c['tot_time'];
    $i++;
}
$num_actions = $num_actions / $i;
$delays = $delays / $i;
echo "$i conversations, with $num_actions memcache requests average, taking ",round($delays
     * 1000000) / 1000," (ms) per conversation.\n\n";
$csv_line .= $i . ',';
$csv_line .= $num_actions . ',';
$csv_line .= $delays * 1000 . ',';
// Key tracking.  What proportion of keys are only used by one server?
    *****************************************************************
$key_track = array();
rewind($fh);
$i = 0;
while($line = fgets($fh)){
    $i++;
    if( strlen($line) < 10 || in_array($i,$fudge_skip))continue;
    $data =  unserialize($line);
    if($data === FALSE){echo "Unserialize error on line $i!!!\n";continue;}
    list($time,$cookie,$sv,$msvo,$cmd,$key,$out_len,$found,$params,$delay,$conversation) =
        $data;
    if(isset($key_track[$key])){ // Seen before, so set server
        $key_track[$key][$sv] = 1;
    }else{
        // New key seen, add
        $key_track[$key] = array($sv => 1);
    }
}
$key_counts = array();
foreach($key_track as $on){
    if(!isset($key_counts[count($on)])){$key_counts[count($on)] = 0;}
```

```
    $key_counts[count($on)]++;
}
echo "Of ",count($key_track)," keys mentioned, \n";
$csv_line .= count($key_track) . ',';
foreach($key_counts as $k => $c){
    echo "        $c were used by $k server(s)\n";
}
// More Key tracking. Of all keys mentioned durring a conversation, how many were unique
     to one server?****************************
$conv_track = array();
rewind($fh);
$i = 0;
while($line = fgets($fh)){
    $i++;
    if( strlen($line) < 10 || in_array($i,$fudge_skip))continue;
    $data =  unserialize($line);
    if($data === FALSE){echo "Unserialize error on line $i!!!\n";continue;}
    list($time,$cookie,$sv,$msvo,$cmd,$key,$out_len,$found,$params,$delay,$conversation) =
         $data;
    if(!isset($conv_track[$conversation])){ // Seen before, so set server
        $conv_track[$conversation] = array('single' => 0, 'all' => 0);
    }
    if(count($key_track[$key]) == 1){$conv_track[$conversation]['single'] ++;}
    $conv_track[$conversation]['all'] ++;
}
$num_conv = 0;
$sum_s_conv = 0;
foreach($conv_track as $on){
    $sum_s_conv += $on['single'] / $on['all'];
    //echo $on['single'], ' -> ', $on['all'], '   ', $num_conv,"\n";
    $num_conv++;
}
echo "\nGiving a ps value of ",round($sum_s_conv / $num_conv * 100),"%. (Counts all keys,
     even a get miss)\n";
echo "         (Per Session ps, so of all keys mentioned for a page request\n          what'
     s the percentage that are only used on one client)\n\n";
$csv_line .= $sum_s_conv / $num_conv . ',';
// Object size calculation
     *****************************************************************
// Average object size, total 'writes'
$total_object_bytes = 0;
$total_requests = 0;
foreach($xmean as $k => $d){
    $total_object_bytes += $d * $num_samples[$k];
    $total_requests += $num_samples[$k];
}
echo "$total_requests requests sent, with an average network size of ", $total_object_bytes
     / $total_requests, " bytes per request, and ", $changecnt," writes,\n";
echo "or ", round(($total_requests - $changecnt)/$total_requests * 100), "% reads\n";
echo "Total network bytes sent: $total_object_bytes\n";
echo "\n\n Readbytes: $readbytes Writebytes: $writebytes\n";
$csv_line .= $readbytes / ($readbytes + $writebytes) . ',';
$csv_line .= $total_object_bytes / $total_requests . ',';
$csv_line .= $changecnt . ',';
$csv_line .= $total_object_bytes . ',';
$sum_object_sizes = 0;
foreach($size as $k => $s){
    $sum_object_sizes += $s;
}
echo "In cache, $sum_object_sizes bytes stored, ",count($size)," objects, avg size ",
     $sum_object_sizes / count($size), " bytes\n\n";
echo "Counting only requests with objects transfer ($num_object_transfer), $sum_objects
     bytes sent on network, or ", $sum_objects/ $num_object_transfer," bytes (object) per
     request. \n";
echo "\n";
$csv_line .= $sum_object_sizes . ',';
$csv_line .= count($size) . ',';
// Command usage
rewind($fh);
$i = 0;
$cmd_usage = array(); // We'll be lazy and just do indexes
for($i = 0;$i<=19;$i++){$cmd_usage[$i] = 0;}
$i = 0;
$reqs = 0;
while($line = fgets($fh)){
    if( strlen($line) < 10 || in_array($i,$fudge_skip))continue;
    $i++;
    $reqs++;
    $data =  unserialize($line);
    if($data === FALSE){echo "Unserialize error on line $i!!!\n";continue;}
    list($time,$cookie,$sv,$msvo,$cmd,$key,$out_len,$found,$params,$delay,$conversation) =
         $data;
    if($cmd == 'set'&& $found){$cmd_usage[0]++;
    }else if($cmd == 'set' && !$found){$cmd_usage[19]++;
```

```
    }else if($cmd == 'add' && $found){$cmd_usage[1]++;
    }else if($cmd == 'add' && !$found){$cmd_usage[2]++;
    }else if($cmd == 'replace' && $found){$cmd_usage[3]++;
    }else if($cmd == 'replace' && !$found){$cmd_usage[4]++;
    }else if($cmd == 'caaaaaaaas'){ // todo: fix!!
    }else if($cmd == 'delete' && $found){$cmd_usage[8]++;
    }else if($cmd == 'delete' && !$found){$cmd_usage[9]++;
    }else if($cmd == 'incr' && $found){$cmd_usage[10]++;
    }else if($cmd == 'incr' && ! $found){$cmd_usage[11]++;
    }else if($cmd == 'decr' && $found){$cmd_usage[12]++;
    }else if($cmd == 'decr' && !$found){$cmd_usage[13]++;
    }else if($cmd == 'flush'){$cmd_usage[14]++;
    }else if($cmd == 'get' && $found){$cmd_usage[15]++;
    }else if($cmd == 'get' && !$found){$cmd_usage[16]++;
    }else if($cmd == 'multiget'){   // we treat a multiget as multiple single gets
        $founds = unserialize($found);
        $hit = 0;
        foreach($founds as $f){
            if($f){$hit++;}
        }
        if($hit > count($founds)/2){$cmd_usage[15]++;
        }else{$cmd_usage[16]++;}
    }else if($cmd == 'append' && $found){$cmd_usage[17]++;  // Combine append/prepend since
        they have the same behavior
    }else if($cmd == 'append' && !$found){$cmd_usage[18]++; // from our point of view.
    }else if($cmd == 'prepend' && $found){$cmd_usage[17]++;
    }else if($cmd == 'prepend' && !$found){$cmd_usage[18]++;
    }else{ echo $cmd, " not known!!!!!!\n";
    }
}
    echo "Command Usage out of $reqs requests (multiget treated as single get, with
        majority hit/miss):\n";
    $csv_line .= $reqs . ',';
    echo "Set Hit     (0): ",$cmd_usage[0]," or ",round($cmd_usage[0] / $reqs * 100),"%\n";
    echo "Set Miss   (19): ",$cmd_usage[19]," or ",round($cmd_usage[19]/ $reqs * 100),"%\n";
    echo "Add Hit     (1): ",$cmd_usage[1]," or ", round($cmd_usage[1] / $reqs * 100),"%\n";
    echo "Add Miss    (2): ",$cmd_usage[2]," or ",round($cmd_usage[2] / $reqs * 100),"%\n";
    echo "Replace Hit(3): ",$cmd_usage[3]," or ",round($cmd_usage[3] / $reqs * 100),"%\n";
    echo "ReplaceMiss(4): ",$cmd_usage[4]," or ",round($cmd_usage[4] / $reqs * 100),"%\n";
    echo "CAAAA1      (5): ",$cmd_usage[5]," or ",round($cmd_usage[5] / $reqs * 100),"%\n";
    echo "CAAAA2      (6): ",$cmd_usage[6]," or ",round($cmd_usage[6] / $reqs * 100),"%\n";
    echo "CAAAA3      (7): ",$cmd_usage[7]," or ",round($cmd_usage[7] / $reqs * 100),"%\n";
    echo "Delete Hit (8): ",$cmd_usage[8]," or ",round($cmd_usage[8] / $reqs * 100),"%\n";
    echo "Delete Miss(9): ",$cmd_usage[9]," or ",round($cmd_usage[9] / $reqs * 100),"%\n";
    echo "Inc Hit    (10): ",$cmd_usage[10]," or ",round($cmd_usage[10]/ $reqs * 100),"%\n";
    echo "Inc Miss   (11): ",$cmd_usage[11]," or ",round($cmd_usage[11]/ $reqs * 100),"%\n";
    echo "Dec Hit    (12): ",$cmd_usage[12]," or ",round($cmd_usage[12]/ $reqs * 100),"%\n";
    echo "Dec Miss   (13): ",$cmd_usage[13]," or ",round($cmd_usage[13]/ $reqs * 100),"%\n";
    echo "Flush      (14): ",$cmd_usage[14]," or ",round($cmd_usage[14]/ $reqs * 100),"%\n";
    echo "Get Hit    (15): ",$cmd_usage[15]," or ",round($cmd_usage[15]/ $reqs * 100),"%\n";
    echo "Get Miss   (16): ",$cmd_usage[16]," or ",round($cmd_usage[16]/ $reqs * 100),"%\n";
    echo "Ap/Pre Hit(17): ",$cmd_usage[17]," or ",round($cmd_usage[17]/ $reqs * 100),"%\n";
    echo "Ap/PreMiss(18): ",$cmd_usage[18]," or ",round($cmd_usage[18]/ $reqs * 100),"%\n";
for($i = 0;$i<=19;$i++){$csv_line .= $cmd_usage[$i] . ',';}
// CSV
if($csv_file){
    $f = fopen($csv_file,'w');
    $csv_line .="0\n";
    fwrite( $f,$csv_line);
    fclose($f);
    echo "\n\n$csv_file csv file written\n";
}
```

Appendix H

Tentacle: tentacle.php

```
<?php
    /*
     Copyright (c) 2012, Paul G Talaga
     All rights reserved.
     Redistribution and use in source and binary forms, with or without
     modification, are permitted provided that the following conditions are met:
     * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
     * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
     ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED. IN NO EVENT SHALL Paul G Talaga BE LIABLE FOR ANY
     DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
     (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
     ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
     (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
     SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
     */
    /*
     tentacle.php - PHP Class implementing a MySQL middleware client, forwarding agent
        and caching system.
     Developed to enable a non-distributed web application to be distributed, it relys
     on all database query calls also providing a consistency time value.  This value
     represents how stale the data can be if a read request, or how much time can be
     allowed to elapse before the write query is applied.
     Using the consistency time value, tentacle can direct read queries to a local
     database or query cache.  If some page is known to contain low consistency values,
     not satisfyable locally, the entire HTTP query is forwarded with session information
     to the central webserving/database location where it is handled.  In this manner all
     HTTP queries are garuneed to travel at most one round trip to the central server.
     Configurables in this class during initialization:
     1.  Central HTTP IP
     2.  Local MySQL database IP
     3.  Central MySQL database IP
     4.  Database name
     5.  Databaes username
     6.  Database password
     7.  Persistent DB connection
     8.  Local Memcache server IP
     9.  Local temp file prefix (for queing write queries)
     10. Local executable to flush write queries
     Notes: APC is required for fast local storage.
            See below for required constants.
     */
define('TENT_CACHE_READS',true);  // Cache read queries?
define('TENT_QUEUE_WRITES',true); // Queue writes if time > 0?
define('TENT_PROXY_ALL',false);   // Should all HTTP requests be forwarded?
define('DB_QUERY_SAVE_FILE','/tmp/db.txt');
define('FORWARD_LIST_FILE','/tmp/forward.txt');
class Tentacle{
  private $init = false;
  // IP or Domain to send important requests to
  private $central_http = null;
  // DB info
  private $localDB = null;
  private $localDBhost = null;
  public $centralDB = null;
  private $centralDB_host = null;
```

```
private $db_user = null;
private $db_password = null;
private $db_name = null;
private $use_pconnect = null;
// Memcache
public $memcache = null;
private $memcache_servers = null;
// Local exe info
private $exe_location = null;
private $local_file = null;
// Forwarding list
private $forwarding_list = null;
  // Can prewarm with an array of strings on line 121
private $central = false;
// Debug
private $query_count = 0;
function __construct($central_http, $local_db, $central_db, $db_name, $db_user,
     $db_password, $db_pconnect, $memcache_servers, $local_temp_file_prefix, $exe){
  // We require APC for fast local storage of state
  if($this->init)return; // bail if already initialized
  if(!extension_loaded('apc'))die('APC Module required for Tentacle, not found');
  $this->db_user = $db_user;
  $this->db_password = $db_password;
  $this->db_name = $db_name;
  $this->central_http = $centralDB;
  $this->centralDB_host = $central_db;
  if($central_http == $_SERVER['SERVER_ADDR']){
    $this->localDBhost = $central_db;
  }else{
    $this->localDBhost = $local_db;
  }
  $this->use_pconnect = $db_pconnect;
  $this->local_file = $local_temp_file_prefix;
  $this->exe_location = $exe;
  $this->central_http = $central_http;
  // If we're the central server don't worry about forwarding list
  if($central_http == $_SERVER['SERVER_ADDR']){
    $this->central = true;
  }else{
    // get forwarding list from apc
    if(apc_exists('forwarding_list')){
      $this->forwarding_list = apc_fetch('forwarding_list');
    }else if(file_exists(FORWARD_LIST_FILE)){// We really should pull the list from a
         memcache, but this will be easier
      $file = fopen(FORWARD_LIST_FILE,'r');    // NOT TESTED
      $contents = fread($file,filesize(FORWARD_LIST_FILE));
      fclose($file);
      $this->forwarding_list = unserialize($contents);
      apc_store('forwarding_list',$this->forwarding_list);
    }else{// nothing stored, start from scratch
      $this->forwarding_list = array(); // will be hash map
      apc_store('forwarding_list',$this->forwarding_list);
    }
  }
  // connect memcache
  $return = true;
  $this->memcache_servers = $memcache_servers;
  $this->connectMemcache($this->memcache_servers);
  $this->init = true;
}
function __destruct(){
  // Close connections to DB's TODO
  //$this->close();
}

function doForward(){
  // This function should be called in the application before ANY response is sent to the
       client.
  // Otherwise custom HTTP headers can't be sent!
  // if in forward list, forward request to central
  if($this->central){return;} // don't forward to yourself!
  // do forward?
  // to speed matching we take the requested URI and see if
  // one of the stored entries in the forwarding lists is a substring
  // if so, we forward.
  $uri = $_SERVER['REQUEST_URI'];
  $trip = false;
  foreach($this->forwarding_list as $find){
      if(strpos($uri,$find) !== FALSE)$trip = true;
  }
  if($trip == false && !TENT_PROXY_ALL)return;
  // get session & headers
  $headers = apache_request_headers();
  $sessname = tep_session_name();
  $sessid = tep_session_id();
```

```
    // get the session info from behind PHP's back!
    $sessData = $this->memcache->get(tep_session_id());
    // what kind of request was it?
    $rtype = $_SERVER['REQUEST_METHOD'];
    $postvars = $_POST;
    // package up the request including headers, type, and session info
    $tosend = array($uri,$headers,$rtype,$postvars,array(array('key' => tep_session_id(),'
        value' => $sessData)));
    $tosend = serialize($tosend);
    $tosend = gzcompress($tosend);
    // Forward packaged request to central /remoteaccess.php
    // Eventually security should be added!
    $response = $this->url_request('POST','http://' . $this->central_http . '/remoteaccess.
        php',array('r' => $tosend));
    $response = $response['content'];
    // Unpackage response
    list($response,$responseheaders, $newsession) = unserialize(gzuncompress($response));
    // Update session state
    foreach ($_SESSION as $key => $value){
        unset($_SESSION[$key]);
    }
    foreach($newsession as $s){
            session_decode($s['value']);
    }
    // Set reponse headers
    $h = explode("\n",$responseheaders);
    foreach($h as $i){
      if(strpos($i,'chunked') === FALSE)
        header($i);
    }
    // Send the response!
    echo $response;
    tep_exit();
  }
  function getLocalResource(){
    if($this->localDB != null)return $this->localDB;
    if($this->centralDB != null)return $this->centralDB;
    $this->connectLocalDB();
    return $this->localDB;
  }
  function query($delay_time, $query){
    // All MySQL queries in the application should call this instead, with a
    // delay time (in seconds) set.
    $pageId = $_SERVER['REQUEST_URI'];
    $this->printQuery($delay_time, $query); // Logs query if needed
    if($delay_time <= 0 && !$this->central){ // need immediate, go to central!
      // TODO: need to dynamically change the above  based on replication lag
          $this->forwarding_list[] = $pageId;
          $this->updateForwardingList();
    }
    // Decide if read or write
    if(strpos($query,'select') === FALSE){ // write
      if(TENT_QUEUE_WRITES && !$this->central){
        $handle = @fopen('/tmp/toUse.txt', "r");  // the write query log is rotated
        $myi = rtrim(fgets($handle, 4096));
        fclose($handle);
        $handle = @fopen('/tmp/queries' . $myi . '.txt', "a");
        fwrite($handle,serialize(array('now' => time(), 'query' => $query)) . "\n");
        fclose($handle);
        $result = 1;
      }else{
        if($this->centralDB == null)$this->connectCentralDB();
        $result = mysql_query($query, $this->centralDB);
      }
    }else{// read
      if(TENT_CACHE_READS && $delay_time > 1){
        $result = $this->getReadQueryWCache($delay_time,$query);
      }else{
        if($this->localDB == null)$this->connectLocalDB();
        $result = mysql_query($query, $this->localDB);
      }
    }
    return $result;
  }
  function getReadQueryWCache($delay_time,$query){
    // To mitigate the thundering herd problem, we embed the expire time
    // in the cached data so we know how much time is left and refresh the data
    // During the last  minute we probabilistically fail (1/(sec time left + 1))
    // so heavily used keys will get refreshed at most a second early, but hopefully
    // by only by 1 client.  OR, on a refresh fail, cause the background DB write daemon
    // to do the call TODO
    global $db_pointer;
    global $db_cache;
```

```
  global $cache_debug;
  $result = $this->memcache->get($this->hashMe($query));
  if($result === FALSE){
    if($this->localDB == null)$this->connectLocalDB();
    $r = mysql_query($query, $this->localDB);
    $result_array = $this->getArrayfromResource($r);
    $this->memcache->set($this->hashMe($query),array($delay_time + time(),$result_array)
        ,0,$delay_time);
    $cache_debug['miss']++;
    return $r;
  }
  $cache_debug['hit']++;
  // found!
  $expire_time = $result[0];
  $returned = $result[1];
  //var_dump($returned);
  //echo "<br><br>\n";
  $left = $expire_time - time();
  if($left < 60 && 0){  // Disabled pending verification
    // preempt update!
    if($left < 0 || (1000 / ($left + 1) > rand(1,1000))){
      $r = mysql_query($query, $this->localDB);
      $result_array = $this->getArrayfromResource($r);
      $this->memcache->set('q'.md5($query),serialize(array($query,$delay_time + time(),
          $result_array)),0,$delay_time);
      return $r;
    }
  }
  $db_pointer = 0;
  $db_cache = $returned;
  return array('pointer' => 0, 'rows' => 0);
}
function hashMe($pre){
  return md5($pre).md5($pre . $pre);
}
function getArrayfromResource($r){
  $ret = array();
  $i = 0;
  while($row = mysql_fetch_array($r,MYSQL_ASSOC)){
    $ret[$i] = $row;
    $i++;
  }
  if($i > 0)mysql_data_seek($r,0);
  return $ret;
}
function connectCentralDB(){
  // we don't automatically connect to the central database because usually we don't have
      to, ever
  // Once running all 0 timed queries are eliminated and the HTTP request is sent to
      central anyway
  if ($this->use_pconnect == 'true') {
    $this->centralDB = mysql_pconnect($this->centralDB_host, $this->db_user, $this->
        db_password);
  } else {
    $this->centralDB = mysql_connect($this->centralDB_host, $this->db_user, $this->
        db_password);
  }
  if(!$this->centralDB){die("Can't connect to (central) ". $this->centralDB_host ."!\n")
      ;}
  mysql_select_db($this->db_name,$this->centralDB);
}
  function connectLocalDB(){
  // we don't automatically connect to the central database because usually we don't have
      to, ever
  // Once running all 0 timed queries are eliminated and the HTTP request is sent to
      central anyway
  if ($this->use_pconnect == 'true') {
    $this->localDB = mysql_pconnect($this->localDBhost, $this->db_user, $this->
        db_password);
  } else {
    $this->localDB = mysql_connect($this->localDBhost, $this->db_user, $this->db_password
        );
  }
  if(!$this->localDB){die("Can't connect to (local) ". $this->localDBhost ."!\n");}
  mysql_select_db($this->db_name,$this->localDB);
}
function connectMemcache($memcache_servers){
  $this->memcache = new MemcachePool();
  if(is_array($memcache_servers)){
    foreach($memcache_servers as $s){
      list($l1,$l2,$l3,$l4,$l5,$l6) = $s;
      $this->memcache->addServer($l1,$l2,$l3,$l4,$l5,$l6);
    }
```

```
  }else{
    list($l1,$l2,$l3,$l4,$l5,$l6) = $memcache_servers;
    $this->memcache->addServer($l1,$l2,$l3,$l4,$l5,$l6);
  }
}
function updateForwardingList(){
  apc_store('forwarding_list',$this->forwarding_list);
  // todo: convert to memcache
  $content = serialize($this->forwarding_list);
  flock($f, LOCK_EX);
  $file = fopen(FORWARD_LIST_FILE,'w');
  fwrite($file,$content);
  fclose($file);
  flock($f, LOCK_UN);
}
// Debug functions
function printQuery($time, $query){
  if($time > 20)return;
  $f = fopen(DB_QUERY_SAVE_FILE,'a');
  if($f == FALSE){die('file open error DB_QUERY_SAVE_FILE');};
  $query = $query;
  $dSmall = substr($query,0,40);
  $output = $time . ':' . ' '. $_SERVER['REQUEST_URI'] . ' - ' . $dSmall . "\n";
  flock($f, LOCK_EX);
  fwrite($f, $output);
  flock($f, LOCK_UN);
  fclose($f);
}
function url_request($type,$url, $data, $headers = '') {
    // Convert the data array into URL Parameters like a=b&foo=bar etc.
    $data = http_build_query($data);
    // parse the given URL
    $url = parse_url($url);
    if ($url['scheme'] != 'http') {
        die('Error: Only HTTP request are supported !');
    }
    // extract host and path:
    $host = $url['host'];
    $path = $url['path'] ;
    // open a socket connection on port 80 - timeout: 30 sec
    $fp = fsockopen($host, 80, $errno, $errstr, 30);
    if ($fp){
        // send the request headers:
        if($type == 'POST'){
          fputs($fp, "POST $path HTTP/1.0\r\n");
        }else{
          fputs($fp, "GET $path HTTP/1.0\r\n");
        }
        fputs($fp, "Host: $host\r\n");
        if ($headers != ''){
            foreach($headers as $k => $d){
              if($k != 'Content-Length'){
              fputs($fp, "$k: $d\r\n");}
            }
        }
        if($type == 'POST'){
          fputs($fp, "Content-type: application/x-www-form-urlencoded\r\n");
          fputs($fp, "Content-length: ". strlen($data) ."\r\n");
          fputs($fp, "Connection: close\r\n\r\n");
          fputs($fp, $data);
      }
        $result = '';
        while(!feof($fp)) {
            // receive the results of the request
            $result .= fgets($fp, 1024); // 128
        }
    }
    else {
  fclose($fp);
        return array(
            'status' => 'err',
            'error' => "$errstr ($errno)"
        );
    }
    // close the socket connection:
    fclose($fp);
    // split the result header from the content
    $result = explode("\r\n\r\n", $result, 2);
    $header = isset($result[0]) ? $result[0] : '';
    $content = isset($result[1]) ? $result[1] : '';
    // return as structured array:
    return array(
```

```
            'status' => 'ok',
            'header' => $header,
            'content' => $content
        );
    }
}
?>
```

Appendix I

Tentacle: loop.php

```
<?php
    /*
     Copyright (c) 2012, Paul G Talaga
     All rights reserved.
     Redistribution and use in source and binary forms, with or without
     modification, are permitted provided that the following conditions are met:
     * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
     * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
     ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED. IN NO EVENT SHALL Paul G Talaga BE LIABLE FOR ANY
     DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
     (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
     ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
     (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
     SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
     */
while(1){
    $fh = fopen('http://localhost/tentacle-daemon.php','r');
        while (($buffer = fgets($fh, 4096)) !== false) {
            echo $buffer;
        }
    fclose($fh);
    sleep(1);
}
?>
```

Appendix J

Tentacle: tentacle-daemon.php

```
<?php
    /*
     Copyright (c) 2012, Paul G Talaga
     All rights reserved.
     Redistribution and use in source and binary forms, with or without
     modification, are permitted provided that the following conditions are met:
     * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
     * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
     ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED. IN NO EVENT SHALL Paul G Talaga BE LIABLE FOR ANY
     DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
     (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
     ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
     (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
     SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
     */
    /*
        Background Database Caller
        Given a queue of database statements saved via file, send them out every second.
        APC is not used as there is no way to start a script and garuntee it exists in the
            same process as the caller
        and thus see the same data.  Additionally memory is not shared between processes/
            threads.
        We use 3 files in /tmp , toUse.txt which contains a single digit, and queries<digit
            >.txt.  toUse acts as a switch so clients can write
        to the file while the cleaner executes the queries in the other digit files.  On
            completion it changes toUse.txt.
    */
    include('includes/configure.php');
    global $link;
    $toUse = '/tmp/toUse.txt';
    $queryprefix = '/tmp/queries';
    // connect to database
    $link = mysql_pconnect(DB_SERVER, DB_SERVER_USERNAME, DB_SERVER_PASSWORD);
    if(!$link){die("Can't connect to $server !\n");}
    if ($link) mysql_select_db(DB_DATABASE, $link);
    $queries = array();
    //
    $num_files = 3;
    $i = 0; // we always clean i-1, so many query files can exist and we clear the oldest
    while(1){
        $to_clean = ($i + $num_files + 1) % $num_files;
        echo "i: $i toclean: $to_clean\n";
        $clean = fopen($toUse, 'w');
        fwrite($clean,$i);
        fclose($clean);
        $min_write = PHP_INT_MAX;
        // do queries
        $handle = @fopen($queryprefix . $i . '.txt', "r");
        if ($handle) {
            while (($buffer = fgets($handle, 4096)) !== false) {
                $q = unserialize(rtrim($buffer));
                $query = $q['query'];
                if($q['now'] < $min_write)$min_write = $q['now']; // keep track of the
                    oldest query time
```

```
                if(strlen($query) > 0)$queries[] = $query;
            }
            if(count($queries) > 0)executeQueries($queries,$min_write);
            $queries = array();
            if (!feof($handle)) {
                echo "Error: unexpected fgets() fail\n";
            }
            fclose($handle);
        }
        // clear file
        $handle = @fopen($queryprefix . $i . '.txt', "w");
        fclose($handle);
        $i++;
        $i = $i % $num_files;
        usleep(200000);
    }
function executeQueries(&$queries,$min_write){
    $tosend = serialize(array('min_write' => $min_write, 'queries' => $queries,'client' =>
        TENT_CLIENT_ID));
    $tosend = gzcompress($tosend);
    // send it!
    echo "Sent \n";
    $response = url_request('POST','http://' . TENT_CENTRAL_HTTP . '/remote-sql-access.php'
        ,array('r' => $tosend));
    echo $response['content'];
    echo "Back\n";
}
function executeQuery($query){
    global $link;
    $return = mysql_query($query, $link) or tep_db_error($query . ' (write request)',
        mysql_errno(), mysql_error());
    echo "Returned $return\n";
    return $return;
}
  function url_request($type,$url, $data, $headers = '') {
      // Convert the data array into URL Parameters like a=b&foo=bar etc.
      $data = http_build_query($data);
      // parse the given URL
      $url = parse_url($url);
      if ($url['scheme'] != 'http') {
          die('Error: Only HTTP request are supported !');
      }
      // extract host and path:
      $host = $url['host'];
      $path = $url['path'] ;
      // open a socket connection on port 80 - timeout: 30 sec
      $fp = fsockopen($host, 80, $errno, $errstr, 30);
      if ($fp){
          // send the request headers:
          if($type == 'POST'){
            fputs($fp, "POST $path HTTP/1.1\r\n");
          }else{
            fputs($fp, "GET $path HTTP/1.1\r\n");
          }
          fputs($fp, "Host: $host\r\n");
          if ($headers != ''){
              foreach($headers as $k => $d){
                if($k != 'Content-Length'){
                fputs($fp, "$k: $d\r\n");}
              }
          }
          if($type == 'POST'){
            fputs($fp, "Content-type: application/x-www-form-urlencoded\r\n");
            fputs($fp, "Content-length: ". strlen($data) ."\r\n");
            fputs($fp, "Connection: close\r\n\r\n");
            fputs($fp, $data);
          }
          $result = '';
          while(!feof($fp)) {
              // receive the results of the request
              $result .= fgets($fp, 128);
          }
      }
      else {
          return array(
              'status' => 'err',
              'error' => "$errstr ($errno)"
          );
      }
      // close the socket connection:
      fclose($fp);
```

```
        // split the result header from the content
        $result = explode("\r\n\r\n", $result, 2);
        $header = isset($result[0]) ? $result[0] : '';
        $content = isset($result[1]) ? $result[1] : '';
        // return as structured array:
        return array(
            'status' => 'ok',
            'header' => $header,
            'content' => $content
        );
    }
?>
```

Appendix K

Tentacle: remoteaccess.php

```php
<?php
    /*
      Copyright (c) 2012, Paul G Talaga
      All rights reserved.
      Redistribution and use in source and binary forms, with or without
      modification, are permitted provided that the following conditions are met:
      * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
      * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
      * Neither the name of the <organization> nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.
      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
      ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
      WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
      DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY
      DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
      (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
      LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
      ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
      (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
      SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
      */
  include('includes/configure.php');
  // responds to requests from edge servers
  // TODO: add authentication!
  //phpinfo();
  if(!isset($_POST['r']))die;
  $request = $_POST['r'];
  $request = gzuncompress($request);
  list($rUri,$headers,$rtype,$postvars,$session) = unserialize($request);
  // Connect to Memcache and set session info
  $return = true;
  $servers = explode(',', MEMCACHE_SERVERS);
  $memcache = new MemcachePool();
  foreach($servers as $sv){
     list($s,$p) = explode(':',$sv);
     $ret = $memcache->addServer($s,$p);
      $return = $return && $ret;
            //echo "Server $s, port $p ok?$ret\n";
   }
   if(!$return){error_log('could not connect to memcache');die('Could not connect to
       memcache');}
   foreach($session as $s){
     session_decode($s['value']);
   }
  //
  $cookie = $headers['Cookie'];
  if($rtype == 'GET'){
    $opts = array(
          'http' =>array(
            'method' =>$rtype,
            'header' => "Cookie: $cookie\r\n"));
    $context = stream_context_create($opts);
    $reply['content'] = file_get_contents('http://' .$_SERVER['SERVER_ADDR'] . $rUri, false
        ,$context);
    $reply['header'] = '';
  }else{
```

```
    $reply = url_request($rtype,'http://' .$_SERVER['SERVER_ADDR'] . $rUri,$postvars,
        $headers);
  }
  $newsession = array();
  $debug = '';
  foreach($session as $s){
     $newsession[] = array('key' => $s['key'], 'value' => $memcache->get($s['key']));
  }
   $return = gzcompress(serialize(array($reply['content'],  $reply['header'] ,$newsession))
       );
  echo $return;
  exit();
  ///////////////////////////////////////////////////////////////////////////////
  function url_request($type,$url, $data, $headers = '') {
      // Convert the data array into URL Parameters like a=b&foo=bar etc.
      $data = http_build_query($data);
      // parse the given URL
      $url = parse_url($url);
      if ($url['scheme'] != 'http') {
          die('Error: Only HTTP request are supported !');
      }
      // extract host and path:
      $host = $url['host'];
      if(isset($url['query'])){
        $path = $url['path'] . '?' . $url['query'];
      }else{
        $path = $url['path'];
      }
      // open a socket connection on port 80 - timeout: 30 sec
      $fp = fsockopen($host, 80, $errno, $errstr, 30);
      if ($fp){
          // send the request headers:
          if($type == 'POST'){
            fputs($fp, "POST $path HTTP/1.1\r\n");
          }else{
            fputs($fp, "GET $path HTTP/1.1\r\n");
          }
          fputs($fp, "Host: $host\r\n");
          if ($headers != ''){
              foreach($headers as $k => $d){
                if($k != 'Content-Length'){
                fputs($fp, "$k: $d\r\n");}
              }
          }
          if($type == 'POST'){
            fputs($fp, "Content-type: application/x-www-form-urlencoded\r\n");
            fputs($fp, "Content-length: ". strlen($data) ."\r\n");
            fputs($fp, "Connection: close\r\n\r\n");
            fputs($fp, $data);
          }
          $result = '';
          while(!feof($fp)) {
              // receive the results of the request
              $result .= fgets($fp, 128);
          }
      }
      else {
          return array(
              'status' => 'err',
              'error' => "$errstr ($errno)"
          );
      }
      // close the socket connection:
      fclose($fp);
      // split the result header from the content
      $result = explode("\r\n\r\n", $result, 2);
      $header = isset($result[0]) ? $result[0] : '';
      $content = isset($result[1]) ? $result[1] : '';
      // return as structured array:
      return array(
          'status' => 'ok',
          'header' => $header,
          'content' => $content
      );
  }
?>
```

References

[1] Adobe. Flex 3 - adobe flex 3, http://livedocs.adobe.com/flex/3/html/help.html?content=SQL_14.html, 2012.

[2] Akamai. Retail web site performance - customer reaction to a poor online shopping experience, http://www.akamai.com/dl/reports/Site_Abandonment_Final_Report.pdf, 2006.

[3] Akamai. Dynamic site accelerator, http://www.akamai.com/dl/brochures/Product_Brief_Aqua_DSA.pdf, 2012.

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

[5] Y. Amir, C. Danilov, M. Miskin-amir, J. Stanton, and C. Tutu. On the performance of wide area synchronous database replication. Technical report, Johns Hopkins University, 2003.

[6] K. Amiri, S. Park, and R. Tewari. Dbproxy: A dynamic data cache for web applications. In *In Proc. ICDE*, pages 821–831, 2003.

[7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: a fast array of wimpy nodes. In *Proceedings of the*

ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[8] J. Archibald and J.-L. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, Sept. 1986.

[9] A. Awadallah and M. Rosenblum. The vmatrix: A network of virtual machine monitors for dynamic content distribution. In *In 7th International Workshop on Web Content Caching and Distribution*, 2002.

[10] G. Ayuso. Speed up php scripts with asynchronous database queries, http://gonzalo123.wordpress.com/2010/10/11/speed-up-php-scripts-with-asynchronous-database-queries/, 2009.

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

[12] A. Bakre and B. Badrinath. I-tcp: indirect tcp for mobile hosts. In *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*, pages 136 –143, may-2 jun 1995.

[13] A. Barbir, B. Cain, R. Nair, and O. Spatscheck. *Known Content Network (CN) Request-Routing Mechanisms*. RFC Editor, United States, 2003.

[14] M. Belshe. A client-side argument for changing tcp slow start, https://docs.google.com/viewer?a=v&pid=sites&srcid=Y2hyb21pdW0ub3JnfGRldnxneDo0NDEyNDM3MzRiZDk4YTE4, 2010.

[15] M. Belshe. More bandwidth doesn't matter (much), https://docs.google.com/viewer?a=v&pid=sites&srcid=Y2hyb21pdW0ub3JnfGRldnxneDoxMzcyOWI1N2I4YzI3NzE2, 2010.

[16] B. R. Borgerson, M. D. Godfrey, P. E. Hagerty, and T. R. Rykken. The architecture of the sperry univac 1100 series systems. In *Proceedings of the 6th annual symposium on Computer architecture*, ISCA '79, pages 137–146, New York, NY, USA, 1979. ACM.

[17] D. Borthakur. Hdfs architecture guide, http://hadoop.apache.org/common/docs/current/hdfs_design.html, 2012.

[18] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[19] J. Brutlag. Speed matters for google web search, http://code.google.com/speed/files/delayexp.pdf, 2009.

[20] P. Cao, J. Zhang, and K. Beach. Active cache: caching dynamic contents on the web. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 373–388, London, UK, UK, 1998. Springer-Verlag.

[21] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27:1112–1118, 1978.

[22] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE*, 1999.

[23] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, III, C. A. Soules, and A. Veitch. Lazybase: trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 169–182, New York, NY, USA, 2012. ACM.

[24] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *INTERNATIONAL WORKSHOP ON DESIGNING PRIVACY ENHANCING TECHNOLOGIES: DESIGN ISSUES IN ANONYMITY AND UNOBSERVABILITY*, pages 46–66. Springer-Verlag New York, Inc., 2001.

[25] Cloudkick. Visual evidence of amazon ec2 network issues, https://www.cloudkick.com/blog/2010/jan/12/visual-ec2-latency/, 2011.

[26] M. Cochran. Multi-threaded asynchronous programming in c#. async database calls. part iii., http://www.c-sharpcorner.com/UploadFile/rmcochran/multithreadedasyncdb05132007005938AM/multithreadedasyncdb.aspx, 2007.

[27] C. J. Conti, D. H. Gibson, and S. H. Pitkowsky. Structural aspects of the system/360 model 85 i: General organization. *IBM Systems Journal*, 7(1):2–14, 1968.

[28] I. B. M. Corporation. *IBM System/360 Model 85, Functional Characteristics*. IBM systems reference library. IBM, 1968.

[29] Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[30] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 202–215, New York, NY, USA, 2001. ACM.

[31] A. Davis, J. Parikh, and W. E. Weihl. Edgecomputing: extending enterprise applications to the edge of the internet. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, WWW Alt. '04, pages 180–187, New York, NY, USA, 2004. ACM.

[32] H. P. de Leon. oscommerce, open source online shop e-commerce solutions, http://www.oscommerce.com/, 2012.

[33] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[34] P. Dixon. Shopzilla's site redo - you get what you measure: Velocity 2009 - o'reilly conferences, june 22 - 24, 2009, san

jose, ca, http://velocityconf.com/velocity2009/public/schedule/detail/7709, 2009.

[35] E. G. DRIMAK, P. F. DUTTON, and W. R. SITLER. Attached processor simultaneous data searching and transfer via main storage controls and intercache transfer controls. *IBM Tech. Disclosure Bull.*, 24:26–27, 1981.

[36] S. Frank. Tightly coupled multiprocessor system speeds memory-access times. *Electronics*, 57:164–169, 1984.

[37] T. Groothuyse, S. Sivasubramanian, and G. Pierre. GlobeTP: Template-based database replication for scalable web applications. In *Proceedings of the 16th International World Wide Web Conference*, Banff, Canada, May 2007. http://www.globule.org/publi/GTBDRSWA_www2007.html.

[38] M. A. Habib and M. Abrams. Analysis of sources of latency in downloading web pages. In *PROCEEDINGS OF WEBNET 2000*, 2000.

[39] M. Heath. Asynchronous database connectivity in java, http://code.google.com/p/adbcj/, 2012.

[40] C. Henderson. *Building Scalable Web Sites: Building, Scaling, and Optimizing the Next Generation of Web Applications*. O'Reilly Media, Inc., 2006.

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

[42] T. Hoff. High scalability - strategy: Break up the memcache dog pile, http://highscalability.com/strategy-break-memcache-dog-pile, 2009.

[43] S. Hull. *Content Delivery Networks: Web Switching for Security, Availability, and Speed*. McGraw-Hill, Inc., New York, NY, USA, 2002.

[44] J. D. JONES and D. M. JUNOD. Cache address directory invalidation scheme for multiprocessing system. *IBM Tech. Disclosure Bull.*, 20:295–296, 1997.

[45] H. K. Kalitay and M. K. Nambiarz. Designing wanem : A wide area network emulator tool. In *COMSNETS*, pages 1–4, 2011.

[46] A. King and J. Nielsen. *Speed Up Your Site: Web Site Optimization*. Pearson Education, 2003.

[47] KLab. repcached - add data replication feature to memcached, http://repcached.lab.klab.org/, 2011.

[48] R. Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[49] A. Labrinidis and N. Roussopoulos. Webview materialization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 367–378, New York, NY, USA, 2000. ACM.

[50] A. Leff and J. Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Com-*

puting Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International, pages 118 –127, 2001.

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

[52] J. S. Liptay. Structural aspects of the system/360 model 85 ii: The cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[53] R. Liston, S. Srinivasan, and E. Zegura. Diversity in dns performance measures. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, pages 19–31, New York, NY, USA, 2002. ACM.

[54] B. Liu and E. A. Fox. Web traffic latency: Characteristics and implications. In *In WebNet98*. Online]. Available: citeseer.ist.psu.edu/liu98web.html, 1998.

[55] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[56] M. Mayer. Keynote: Velocity 2009 - o'reilly conferences, june 22 - 24, 2009, san jose, ca, http://velocityconf.com/velocity2009/public/schedule/detail/8913, 2009.

[57] G. Mazare. A few examples of how to use a symmetrical multi-micro-processor. *SIGARCH Comput. Archit. News*, 5(7):57–62, Mar. 1977.

[58] mediawiki. Mediawiki, http://www.mediawiki.org/wiki/MediaWiki, 2011.

[59] MySQL. Mysql :: Mysql 5.0 reference manual :: 15 replication, http://dev.mysql.com/doc/refman/5.0/en/replication.html, 2012.

[60] M. Nambiar, H. K. Kalita, D. Mishra, and S. Rane. Wanem - wide area network emulator, http://wanem.sourceforge.net/, 2009.

[61] S. Newman. Three latency anomalies, http://amistrongeryet.blogspot.com/2010/04/three-latency-anomalies.html, 2011.

[62] 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.

[63] J. Oberheide, M. Karir, and D. Blazakis. Vast: visualizing autonomous system topology. In *Proceedings of the 3rd international workshop on Visualization for computer security*, VizSEC '06, pages 71–80, New York, NY, USA, 2006. ACM.

[64] oldmoe. oldmoe: Faster io for ruby with postgres, http://oldmoe.blogspot.com/2008/07/faster-io-for-ruby-with-postgres.html, 2008.

[65] K. Orend. Analysis and classification of nosql databases and evaluation of their ability to replace an object-relational persistence layer. *Architecture*, page 100, 2010.

[66] V. N. Padmanabhan and L. Subramanian. An investigation of geographic mapping techniques for internet hosts. *SIGCOMM Comput. Commun. Rev.*, 31(4):173–185, Aug. 2001.

[67] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, Jan. 1984.

[68] perl-mysql async. Pure perl asynchronous mysql driver, http://code.google.com/p/perl-mysql-async/, 2012.

[69] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *In Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference*, pages 155–174, 2004.

[70] PostgreSQL. Postgresql: Documentation: Manuals: Asynchronous command processing, http://www.postgresql.org/docs/8.3/static/libpq-async.html, 2012.

[71] A. Prof and D. Duchamp. Abstract analytical characterization of the throughput of a split tcp connection, http://www.cs.northwestern.edu/~ais/ms_thesis.pdf, 2001.

[72] G. S. Project. Spdy: An experimental protocol for a faster web, http://dev.chromium.org/spdy/spdy-whitepaper, 2012.

[73] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Addison-Wesley, 2001.

[74] M. Rabinovich, Z. Xiao, and A. Aggarwal. Computing on the edge: A platform for replicating internet applications. In *In International Workshop on Web Caching and Content Distribution (WCW*, pages 57–77, 2003.

[75] L. Rao. J.p. morgan: Global e-commerce revenue to grow by 19 percent in 2011 to $680b — techcrunch, http://techcrunch.com/2011/01/03/j-p-morgan-global-e-commerce-revenue-to-grow-by-19-percent-in-2011-to-680b/, 2011.

[76] J. Ravi, Z. Yu, and W. Shi. A survey on dynamic web content generation and delivery techniques. *J. Netw. Comput. Appl.*, 32(5):943–960, Sept. 2009.

[77] D. Rayburn. How dynamic site acceleration works, what akamai and cotendo offer, http://blog.streamingmedia.com/the_business_of_online_vi/2010/10/how-dynamic-site-acceleration-works-what-akamai-and-cotendo-offer.html, 2010.

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

[79] R. Ruggaber. Internet of services sap research vision. *Enabling Technologies, IEEE International Workshops on*, 0:3, 2007.

[80] RuggedCom. Latency on a switched ethernet network, http://www.ruggedcom.com/pdfs/application_notes/latency_on_a_switched_ethernet_network.pdf, 2011.

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

[82] B. Schwartz, P. Zaitsev, V. Tkachenko, J. D. Zawodny, A. Lentz, and D. J. Balling. *High Performance MySQL: Optimization, Backups, Replication, and Load-Balancing*. O'Reilly Media, 2.a. edition, 2008.

[83] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in beehive. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 211–220, New York, NY, USA, 1997. ACM.

[84] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. Globedb: autonomic data replication for web applications. In A. Ellis and T. Hagino, editors, *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pages 33–42. ACM, 2005.

[85] S. Sivasubramanian, G. Pierre, and M. V. Steen. Globecbc: Content-blind result caching for dynamic web applications. Technical report, Vrije University, 2006.

[86] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. Analysis of caching and replication strategies for web applications. *IEEE INTERNET COMPUTING*, 11:60–66, 2007.

[87] S. Sivasubramanian, M. Szymaniak, G. Pierre, and M. van Steen. Web replica hosting systems. Technical report, Vrije Universiteit, 2003.

[88] S. Sivasubramanian, M. . Szymaniak, G. Pierre, and M. V. Steen. Replication for web hosting systems. *ACM Computing Surveys*, 36:291–334, 2004.

[89] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, Sept. 1982.

[90] C. Snyder and M. Southwell. *Pro PHP Security (Pro).* Apress, Berkely, CA, USA, 2005.

[91] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[92] P. G. Talaga and S. J. Chapin. Exploring non-typical memcache architectures for decreased latency and distributed network usage. In *WEBIST*, pages 36–46, 2012.

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

[94] A. S. Tanenbaum, A. S. Tanenbaum, M. F. Kaashoek, M. F. Kaashoek, H. E. Bal, and H. E. Bal. Using broadcasting to implement distributed

shared memory efficiently. In *Readings in Distributed Computing Systems*, pages 387–408. IEEE Computer Society Press, 1994.

[95] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, AFIPS '76, pages 749–753, New York, NY, USA, 1976. ACM.

[96] B. Technologies. Basho: Welcome to the riak wiki, http://wiki.basho.com/Riak.html, 2012.

[97] Terracotta. Ehcache documentation cache-topologies, http://ehcache.org/documentation/distributed_caching.html, 2011.

[98] D. Terry. Replicated data consistency explained through baseball. Technical Report MSR-TR-2011-137, Microsoft Research, October 2011.

[99] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 1013–1020, New York, NY, USA, 2010. ACM.

[100] F. J. Torres-Rojas, M. Ahamad, and M. Raynal. Timed consistency for shared distributed objects. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, PODC '99, pages 163–172, New York, NY, USA, 1999. ACM.

[101] S. Trent, M. Tatsubori, T. Suzumura, A. Tozawa, and T. Onodera. Performance comparison of php and jsp as server-side scripting languages. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 164–182, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[102] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[103] Verizon. Ip latency statistics - verizon business, http://www.verizonbusiness.com/about/network/latency/, 2012.

[104] D. C. Verma. *Content Distribution Networks: An Engineering Approach*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[105] VMware. Vmware kb: Timekeeping best practices for linux guests, http://kb.vmware.com/kb/1006427, 2012.

[106] W. Vogels. Eventually consistent. *Commun. ACM*, 52:40–44, Jan. 2009.

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

[108] D. Warne. Why using google dns / opendns is a bad idea, http://apcmag.com/why-using-google-dns-opendns-is-a-bad-idea.htm, 2010.

[109] L. Welling and L. Thomson. *PHP and MySQL Web Development*. Sams, Indianapolis, IN, USA, 2003.

[110] A. Wolfe Gordon and P. Lu. Low-Latency Caching for Cloud-Based Web Applications. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB '11)*, Athens, Greece, 2011.

[111] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[112] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, Aug. 2002.

[113] Y. Zhang, N. Ansari, M. Wu, and H. Yu. On wide area network optimization. *Communications Surveys Tutorials, IEEE*, PP(99):1 –24, 2011.

Paul G. Talaga

Contact Information

130 Beech Grove Rd.
Honesdale, PA 18431 USA

| *Mobile:*+1 570-906-4071
| *E-mail:* pgtalaga@syr.edu
| *WWW:* www.fuzzpault.com

Research Interests

Data locality in geographically distributed dynamic web applications, distributed caching, assuring HTML compliance for dynamic web applications, functional programming for the web, evolutionary robotics, neural network systems, artificial life, chaos and fractals, super image resolution

Education

Syracuse University, Syracuse, NY

PhD, Computer and Information Science, August 2012

- Thesis: *Exploiting Data Locality in Dynamic Web Applications*
- Adviser: Professor Stephen Chapin
- GPA: 3.89/4

Master of Science, Computer Science, May 2006

- GPA: 3.91/4

St.Lawrence University, Canton, NY

B.S., Dual Majors: Computer Science & Math, May 2003

- CS Honors Project: *Raytracing as a Scene Rendering Technique*
- Minor: Physics
- GPA: 3.4/4

Conference Publications

[1] P Talaga, S Chapin. Exploring Non-typical Memcache Architectures for Decreased Latency and Distributed Network Usage. *Accepted for presentation & publication at WEBIST 2012*

[2] P Talaga, S Chapin. Guaranteeing Strong (X)HTML Compliance for Dynamic Web Applications WEBIST 2011, *Proceedings of the 7th International Conference on Web Information Systems and Technologies*, Noordwijkerhout, Netherlands, 6-9 May, 2011. Pages 71-79, SciTePress, 2011.

[3] P Talaga, J Oh. Combining AIMA and LEGO Mindstorms in an Artificial Intelligence Course to Build Real World Robots *Journal of Computing Sciences in Colleges*, Vol 24, Issue 3, Pages 56-64, January 2009

[4] K Jayaraman, G Lewandowski, P Talaga, S Chapin. Enforcing Request Integrity in Web Applications *In Proceedings of 24th Annual Working Conference on Data and Applications Security* 2010.

[5] K Jayaraman, P Talaga, G Lewandowski, S Chapin. Modeling User Interactions for (Fun and) Profit: Preventing Workflow-based Attacks in Web Applications *In Proceedings of the 16th Pattern Languages of Programs Conference* (Chicago, Aug 28-30, 2009.). PLoP 09.

Other Publications

[6] P Talaga, S Chapin. Towards a Guaranteed (X)HTML Compliant Dynamic Web Application WEBIST 2011 (Selected Papers), *Lecture Notes in Business Information Processing*, Springer 2012

Technical Reports

[7] P Talaga, S Chapin. Exploring Non-typical Memcache Architectures for Decreased Latency and Distributed Network Usage Syracuse University Technical Report SYR-EECS-2011-10, Sept. 15, 2011

[8] P Talaga, S Chapin. Strong (X)HTML Compliance with Haskells Flexible Type System Syracuse University Technical Report SYR-EECS-2010-04, Oct. 22, 2010

Teaching Experience

Syracuse University, Syracuse, NY

Instructor **Summer 2006**

CPS 181: Introduction to Computing
- Taught online course through Blackboard System
- Handled all aspects of course including online content creation, management, and grading

Teaching Assistant **September 2004 to May 2009**

CIS 275: Discrete Mathematics
- Autumn 2004 & Autumn 2005
- Responsible for two 45-minute recitation sessions per week, weekly office hours
- Grading of all student work excluding exams
- Blackboard course management

CIS 321: Probability and Statistics
- Spring 2005 & Spring 2007
- Responsible for two 45-minute recitation sessions per week, weekly office hours
- Solutions & grading of all student work excluding exams
- Blackboard course management

CIS 352: Programming Languages
- Spring 2006 & Spring 2008
- Responsible for two 45-minute recitation/lab sessions per week, weekly office hours
- Solutions & grading of all student work excluding exams
- Blackboard course management

CIS 467/667: Introduction to Artificial Intelligence
- Fall 2006 & Fall 2007
- Responsible for two 45-minute lab sessions per week, weekly office hours
- Solutions & grading of all student work excluding exams
- Management of LEGO robotics AI lab
- Creation & guideance of robotic labs
- Course website administration

CIS 252: Introduction to Computer Science
- Spring 2008
- Responsible for three 45-minute lab sessions per week, weekly office hours
- Solutions for HW
- Blackboard course management

CIS 453/454: Software Specification & Design
- Fall 2008, Spring 2009
- Assist with project selection & implementation
- Grading of all student work

Professional Experience

Fuzzpault Technologies, LLC, Honesdale, PA

Owner/Operator **2003 to present**

- Provide web design, maintenance, and hosting services to local businesses (client list available)
- Provide on-site system maintenance, repair, and management of computing, network, and printing systems
- Computer hardware and software repair

VA Hospital - Research Division, Syracuse, NY

Research Programmer **May 2008 to May 2009**

- Develop test suite for visual stimulus research using Matlab, OpenGL, & iView gaze capture system
- Integrate reliable iView remote control via ethernet from Matlab while satisfying real-time constraints

US Air Force Research Lab (AFRL), Rome, NY

Mathematics Technician **May 2005 to August 2005**

- Develop melodic sensing algorithms using wavelet theory for use in sensor systems

Himalayan Institute, Honesdale, PA

Assistant Network Administrator **May 2003 to August 2004**

- Provide tech support and manage 200+ MS Windows & Mac systems in two locations
- Assist with Cisco network enhancements & NORTEL PBX maintenance
- Upgrade and patch MS Windows Servers (2003) used for order processing and shipping

Segway LLC., Manchester, NH

Control Systems Intern **Summer 2001, Summer 2002**

- Test, capture, and analyze data from algorithm modifications to the Human Transporter (HT)
- Provide hardware support for software and controls departments, including HT repair and rebuild
- Develop Matlab test suites for efficiency, performance, and stability measurement

Professional Memberships

Association for Computing Machinery (ACM), Member, 2004–present

Awards

St. Lawrence University - 2003

- Pi Mu Epsilon - Mathematics Honor Society
- Society of Physics Students
- Kurt Douglas Award for Technical Theater

Hardware and Software Skills

Computing and Networking Systems:

- Desktop, server, and laptop repair/upgrade
- Installation/troubleshooting switches, routers, firewalls
- LAN management, printer installation/repair

Information/Internet Technologies:
- Networking (UDP, TCP, ARP, DNS, Dynamic routing), Services (Apache, SQL, MediaWiki, osCommerce, POP, IMAP, SMTP, application-specific daemon design)
- Installation/management of Linux based web servers
- IPTables firewall, MySQL administration
- PHP and Perl scripting

Computer Programming:
- C, C++, Objective C, Java, JavaScript, Perl (/TK), PHP, Lisp, Haskel, UNIX shell scripting, SQL, MySQL, Matlab, LaTeX, and others

Matlab skill set:
- Linear algebra, Fourier transforms, Wavelet analysis, Monte Carlo analysis, GPU computation, OpenGL visualization, polynomials, statistics, genetic algorithms, neural networks
- Toolboxes: genetic algorithms and neural networks

Productivity Applications:
- TeX (LaTeX, BibTeX), emacs, most common productivity packages (for Windows, OS X, and Linux platforms)
- MS Office: Word, Excel
- Photoshop CS5, Dia, OmniGraffle

Operating Systems:
- Microsoft Windows family, Apple OS, Apple iOS, Linux, and other UNIX variants

Expertise

Mathematics:
- Applied Mathematics, Graph Theory, Statistics, Combinatorics

Computer Science:
- Programming Languages, Artificial Intelligence, Computer Architecture, Operating Systems, Algorithms, Data Structures, Web Security, Artificial Life, Web Programming

Projects

- **Tentacle:**
  Database middleware system for PHP and MySQL web applications allowing geographical distribution of a non-distributed web application. Uses database replication, query caching, and per-SQL-template consistency specifications.

- **MemcacheTach:**
  Detailed Memcache logging and analysis tool. Provides data on key usage, key distribution, command usage, Memcache server response times, as well as many other parameters. *Published in WEBIST 2012*

- **Location-aware Memcache:**
  Developed and evaluated 2 new Memcache storage architectures providing decreased latency and network usage in specific applications. *Published in WEBIST 2012*

- **MindStorms Rubix Cube Solver:**
  AI class project in which I built a robot and managed software development. Used neural network-based vision system, external student-designed solver, and command playback. User-interface written in Perl/TK. YouTube video available.

- **InfBB:**
  Implementation of an infinite-depth web bulletin board system demonstrating secu-

rity patterns. *Published in PloP 2009.*

- **Evolutionary Robotic System:**
  Design and implementation of neural network-based evolutionary robotic software using C++, Bullet physics engine, and Lua on a grid system. Resulting networks were evaluated in hardware with LEGO Mindstorms running Lua.

- **Bayawak:**
  Implemented proof-of-concept URL rewriting system for web application security based on Perlbal. Continued work produced the Bayawak implementation published in *DBSec 2010.*

References Available to Contact

Dr. Stephen Chapin chapin@ecs.syr.edu (315) 443-4457
- Associate Professor, L.C. Smith College of Engineering and Computer Science, Syracuse University
- Ph.D. advisor

Dr. Kishan Mehrotra mehrotra@syr.edu (315) 443-2811
- Professor, L.C. Smith College of Engineering and Computer Science, Syracuse University
- Dr. Mehrotra taught many of the classes for which I was a TA.

Dr. Jae Oh jcoh@syr.edu (315) 443-4740
- Associate Professor, L.C. Smith College of Engineering and Computer Science, Syracuse University
- Dr. Oh taught AI, which I was a TA and robotics lab manager.

Dr. Brad C. Motter Brad.Motter@va.gov (315) 425-4873
- Research Health Scientist,
  Veterans Affairs Medical Center,
  Syracuse, NY
- I developed vision testing software to support Dr. Motter's research.

More Information

More information and auxiliary documents can be found at `http://www.fuzzpault.com`.