CS764 Project Update

Brian Kroth

2014-03-15

1 Problem Description

The Computer Aided Engineering center at the College of Engineering of UW-Madison manages a platform of web, database, storage, and other related servers for a campus wide Moodle service (a learning management system). Due to the large scope and supported user base of this web application it encounters some interesting systems and database related problems. The main focus of this project is to evaluate and analyze some of the different database and related (eg: NoSQL) configurations for user session and application cache data that Moodle currently supports (or can be made to) with respect to their impact on scalability, reliability, security, and performance.

2 Background

The current Moodle service server platform is based off of a distributed LAMP stack, and is a subtype of a larger general purpose website server platform that currently handles over 800 vhosts of various flavors (eg: HTML only, PHP5, prepackaged Wordpress, etc.). [31] has more details on the general system. It largely resembles the architecture of a scalable MOOC as described in [42], except that it is currently all homegrown and running locally rather than using any cloud services.

The Moodle portion of the service currently consists of two production vhosts as well as a number of development and testing vhosts and has a pair of frontend Apache servers running `mod_proxy_balance` (and optionally `mod_disk_cache` to save certain responses) to distribute client requests to a pool of backend Apache `mod_php5` application servers that are specifically tuned for Moodle vhosts. [1] Each backend server has a local copy of the Moodle PHP code base, synchronized with an authoritative NFS server store using a homegrown 2 phase commit `rsync` based protocol, in order to avoid the overhead of `stat` calls for code that doesn't often change during runtime as well as provide relatively atomic code updates between all of the backends when code does need to change so that clients don't receive different and inconsistent responses. [2]

For each site, dynamic user data such as PDF documents and quiz question banks are split between an NFS store and a MySQL 5.6.15 database server, respectively. There is a separate database server VM for each of two production sites, as well as a third for the development sites. [3] The MySQL servers each have binary logging enabled in order to asynchronously replicate changes to their own slave server so that consistent application level backups can be taken of the database without locking all tables or interrupting service. [4] No reads are currently directed at the slave database server. Data is stored in InnoDB storage format tables for ACID compliance.

2.1 Session Data

To provide an interactive user interface, and overcome limitations in browser cookie size, the Moodle code currently stores per-user session data as a seri-

[1] At the time of this writing the main campus Moodle site, courses.moodle.wisc.edu, has 5 backend VMs assigned to it, each with 4 vCPUs and 6G of RAM, though that number may change throughout the year according to load demands in a manner originally developed in [30] that resembles that of [28], whereas the other production site, innovate.moodle.wisc.edu, has 2 backend VMs assigned to it. The backend assignment systems attempts to avoid overlapping "large" vhosts whenever possible. [30]

[2] The `stat` calls are a function of the PHP opcode cache subsystem (APC) and can be disabled at the price of potentially inconsistent responses due to lack of awareness of code base changes, so it is therefore usually not done. Additionally, once the code files are local, the `stat` calls tend to hit the OS inode cache, which has a much lower overhead when compared with an NFS call.

[3] Each database server VM's resources are tuned to the dataset size of the particular site that it serves. Currently the two production servers have 4 vCPUs and 16G of RAM each with separate disk LUNs backed by storage arrays for data and logging volumes.

[4] Mixed student and instructor usage means that the service does not follow a typical diurnal pattern.

alized data structure [5], one row per active session, in a database table (`mdl_sessions`).

It is valuable for this data to survive a system restart so that users in the middle of a task (eg: a quiz essay question) need not start over from scratch, however the durability requirements on it are not as stringent as other parts of the database. For example, the session data need not even be backed up, let alone replicated.

Since MySQL binary logging cannot currently be disabled for a single table [12], during periods of high activity, the amount of churn on the `mdl_sessions` alone can result in upwards of 5x as much binary logging traffic as there is total database data. [6].

Additionally, for educational research purposes, all user activities such as a page view or button click are logged to another table (`mdl_log`), which can also generate a large amount of database traffic. To keep it a manageable size [7] we currently rotate the table data out to long term storage at least once a year.

To make sure that the serialized session data remains consistent, each Moodle page load (excepting certain mostly static theme resources like images and CSS) requests a named lock (`GET_LOCK()`) on the row object that isn't released until just before the page is returned to the user. This has the unfortunate effect of serializing a single user's requests and potentially tying up a large number of database resources in the process, especially in the case of errant client browser behavior (eg: redirect loop). This also means that every page load involves at least two database writes.

2.2 Cache Data

Like many web applications [8], Moodle makes heavy use of caching to attempt to improve performance. Moodle's Universal Cache (MUC) [24] system separates this cache into 2 distinct pieces:

1. User specific session cache, and
2. Module specific application cache that applies to all users (eg: JavaScript, unified CSS, localized strings, LaTeX rendered equation images, etc.).

By default, and as it is currently configured, these caches are stored in a file system hierarchy local to the given backend server. It attempts to make sure that files (often identified by a hash) that are missing or look out of date are regenerated as necessary, so that inconsistencies do not arise. However, as Moodle is a distributed development effort, not all of it is necessarily built with multiple backend servers in mind, so inconsistencies and program errors [9] do still occur.

Moodle also does some in database cache materialization.

Note that in both cases, the cache does not require persistence as it can be rebuilt from its original sources as necessary.

3 Evaluation

3.1 Project Goals

3.1.1 Session Data

In addition to the main database session store provider described above, the Moodle code base now also has support for memcached. [10]

Additionally, in the past we have explored other options such as using a separate MySQL database that did not have binary logging enabled as the target for the `mdl_sessions` data, though our initial implementation suffered from poor integration with the standard database driver. We believe this should be easier to accomplish in the current version of Moodle (2.6 vs. 2.4). With such a system additional session data specific optimizations such as reduced concurrency levels would be possible (eg: there are no range searches, so phantoms are not an issue).

With standard memcached servers, if the server is restarted, even gracefully for maintenance operations, all data within it is lost. If that data is session data, it will effectively log out the user. As noted above, this may be undesirable from a long running user's point of view if they're in the middle of some work, but not as big of a problem for a very recently logged in user who hasn't had a chance to start anything yet.

The version of MySQL we are running also provides an InnoDB table backed memcached interface. [2] This provides us a drop in replacement with the

[5] Maximum size for a record is currently 4M, though it is usually less than 50K.

[6] The largest we have seen the `mdl_sessions` table, including it's indices, is roughly 500M. The entire database is roughly 20G, whereas there is currently over 75G of log data for the past day.

[7] The table currently has more than 16M rows comprising 5.2G for the 2013-2014 academic calendar year.

[8] Indeed most things in computer science.

[9] [14] occurred just recently, for instance.

[10] In both cases, it is merely a wrapper around the standard PHP session handler.

ability to define more granular durability modes by specifying that MySQL's memcache operations need only flush periodically, for instance. Couchbase [1] and MemcacheDB [13] would be other such options, though with some missing features especially with respect to expiration of values and some additional ones such replication.

In this project we would like to explore each of these options as possible alternatives to our current configuration.

Because we operate in a shared environment, in such an evaluation it is also important for us to understand the reliability and security offered by each system as well as the performance and scalability.

Though the former will likely be a qualitative analysis, we can aim for a more quantitative analysis in the latter two by measuring the response times [11] of requests as we vary the number of active clients in our tests.

3.1.2 Cache Data

In addition to the per-backend file system based cache system described above, the Moodle code base now also has support for memcached [12] and MongoDB as cache targets.

While at first glance a local file system based cache and its associated OS page cache would appear to have far less overhead than any of the network based cache systems, this hypothesis deserves testing.

It is also possible that a local file system based cache on the web server scales only so long as the number of clients, and therefore Apache processes, is small enough to allow for the OS page cache to take the hit instead of the disk and that given sufficient load the memory contention on the web server makes a network round trip to a dedicated memory cache service more attractive.

Furthermore, a shared cache should not suffer the same inconsistency problems that multiple "separate but equal" caches do, though this also deserves testing.

As with session data, we need to understand the security, resiliency, performance, and scalability of each option.

[11] eg: 95 percentile

[12] It actually supports both memcache and memcached PHP client library flavors, though we concentrate only on memcached for this project due to its extra features such as binary protocol support.

3.2 Test Methods

To test the performance each of these alternative configurations we plan to

- Implement two memcached services for use as cache and sessions targets.

  There are questions of sizing and authentication here, to name a few options, that we will need to explore.

- Implement two MySQL InnoDB memcached services for use as cache and sessions targets.

  There are questions of what the durability level should be here, in addition to those presented above.

- Implement a separate database connection session handler.

  There are questions of what the concurrency and durability levels should be here.

- Implement a MongoDB service for use as a cache target.

  There are questions of write flush policy, sharding, replication, and authentication and authorization here, to name a few options, that we will need to explore.

- Create a test Moodle environment.

  We have a choice here between an environment that includes all of the usual UW Moodle modules and themes, or a more standard off the shelf Moodle environment.

- Construct a repeatable load test suite.

  For instance a quiz attempt or a forum post and view.

  As each of these involves multiple pages, many database writes, cache and session interactions, and varying levels of concurrency, we think they are reasonably characteristic representations of our most intensive workloads. [13]

- Conduct test runs of 1, 10, 50, 100, 250, 500, 1000, and 2000 simultaneous users from our roughly 100 Linux lab machines.

- Measure the response time for each page request as well as the overall test run time.

[13] Historically we have had the most load during final exams periods.

- Report the results for each cache configuration and each session configuration in terms of min, max, average and 95 percentiles.

  Other useful metrics to gather may be things like CPU load, number of main database reads/writes, and number of successfully handled requests.

To test the resiliency characteristics of each of the alternative configurations we plan to repeat our tests used above, but part way through the test

1. Gracefully restart the service. This simulates an administrative action such as a service configuration change or patch that requires a restart.
2. Forcefully restart the service. This simulates a full server crash.

We want to know

1. In the case of session configurations, what happens to a given users session? For instance, do they need to login again?
2. How long does it take for the system to recover?
3. During that period is the system as a whole inoperable while the given cache or session service is unavailable or is it merely slower?

These tests are easiest done with a single client, though that may not show issues such as the thundering herb problem [11]. They're also most interesting when done with more than one server, though time constraints may prevent us from exploring such variations.

3.3 Implementation Discussion

3.3.1 memcached

To address the sizing of our memcached service we plan to begin with a single separate memcached virtual server running with 2 vCPUs and 2G of RAM. We feel this should be sufficient to test with as it is, even taking data structure overheads into account, still larger than the maximum current cache and session size.

If there's time, we intend to measure the effects of varying the number of servers from 1 to 3 as well as to understand the application behaviors as we make part of the service unavailable. It should be noted that in modern memcached clients, failover does not usually occur since it would cause invalid cache consistency between nodes. Instead, a form of consistent hashing of keys is used to distribute values to particular server, and clients either block or timeout if that server is unavailable. [3] For cache systems this is okay since the data can ultimately be reconstructed from the original source, but for sessions this may not be acceptable.

One interesting thing to note about this aspect of memcached is that not only can any client that can connect to the server explore and dirty the keyspace, but they can also flush it entirely. This makes a single memcached undesirable to service multiple Moodle instances since a key feature of the MUC is the ability to purge all of the cache's data for events like code upgrades or theme changes. Doing so would invalidate the data of all the Moodle instances, leading to another warm up period for all sites. This problem is further exacerbated in our shared hosting environment. Alternatively, if the same memcached server were used for both session data and cache, then even a single site's purge caches function could clear all actively logged in sessions.

Also note that memcachedb and the MySQL InnoDB and Couchbase memcached services also share this disadvantage since it is a deficiency in the protocol.

To address this and the authentication aspects of memcached, we could simply configure test systems both with and without using SASL authentication to see its performance effects. Even though the authentication mechanism currently only serves as a gateway barrier to the service as a whole, something a firewall could generally do, in our shared hosting environment, the shared credentials requirement provides an added layer of separation of competing needs between different sites and their use of one or more memcached servers as it represents a declaration of intent for the use of the service.

Another performance related option we could explore is the use of data compression with memcached. By default most memcached client libraries compress data exceeding a particular size (eg: 2000 bytes). The theory is that the time it takes to compress the data will be saved in network IO latency. This seems reasonable, so we chose to skip evaluating this particular setting. [3]

3.3.2 MySQL InnoDB memcached

There are a number of tunable settings that affect the MySQL InnoDB memcached service. Of most interest to us are those relating to durability and consistency. The main reason to use a MySQL provided

memcached is for the granularity that these options offer us, even though it's likely to add write performance overhead.

In this case, we think that one MySQL server instance devoted only to serving memcached clients (read: data that isn't required to survive a restart), setting `daemon_memcached_r_batch_size=50`, `daemon_memcached_w_batch_size=1000`, `innodb_flush_log_at_trx_commit=2`, `innodb_flush_log_at_timeout=1`, `innodb_doublewrite=0`, `innodb_api_bk_commit_interval=5`, `innodb_api_enable_binlog=0`, `daemon_memcached_enable_binlog=0` should provide a good balance between performance and semi-timely durability since data should make it to disk within 1 second after a smaller number of in memory operations and won't suffer from the binary logging overhead due to high churn.

3.3.3 MongoDB

MySQL's InnoDB Memcache API provides a way to get tunable persistence of memcache values, thereby limiting the amount of cold start and session availability issues that a service restart or outage might entail. However, they are server wide rather than based on an individual client's requested level of durability.

MongoDB on the other hand allows for the application to specify a given write's durability requirements through the use of write concern levels. [8].

For example in the case of cache writes, we may be perfectly happy with a write concern of "acknowledged" but not necessarily journalled (this is the current Moodle default for MongoDB caches), since in the case of a failure, the cache should simply not contain the data (in MongoDB all single document writes are in theory atomic), and the system can go about repopulating it from the original source.

On the other hand, if replication is being used, the above situation may get us into a discrepancy when a write to is acknowledged at the primary, but fails to reach the secondary, and then the primary dies leading to that secondary being elected as the new primary. Now, with automatic client failover, or client specified read preferences which allow them to read from the secondary, the secondary (now primary) may return stale cache rather than up to date or no cache data, which can lead to display or logic errors for the client.

MongoDB's replication feature is also an attractive feature for its potential to reduce system stalls, cold cache degradation, session loss, or outages through the use of automatic client failover. If time allows, we intend to test a MongoDB cache configuration with 2 secondary replicas as well. One concern we might have is that since replication effectively still uses a binary logging technique (MongoDB calls it oplog), then high churn on session cache data might cause the same issues that we've seen with MySQL. It appears that MongoDB bounds the size of the oplog log, but not the journal files at initial configuration time, so this may still be an issue. [5] [6]

Automatic sharding is another feature that MongoDB provides, that splits a namespace up amongst participating servers in order to distribute the load. However, unlike the memcached implementation which depends on the client to perform a hash to determine the server to communicate with, the MongoDB solution requires several more servers (mongos and config servers) to manage the shards. Given we think our current cache and session sizes can fit within 2G, we think this is an unnecessary extra set of overhead for our current environment and do not intend to test it. Clearly if we were to scale Moodle another order of magnitude or two to the MOOC scale, it might need reconsideration.

Unlike memcached which has a single flat authorization space per server, recent versions of MongoDB (2.4) allows administrators to assign user roles separately for each database. [7] This is another very attractive feature for our shared hosting environment.

3.3.4 Test Environment

We have a spectrum of choices for how to configure Moodle. On the one hand, we could take a barebones basic install with only the minimal configuration tweaks required to operate in our environment. On the other, we could include all of the local modules, themes, plugins, and customizations that the current UW Moodle service offers.

Given that some of the available load testing tools operate on more standard installations more easily, and that many of our current Moodle tweaks are for an older version of Moodle (2.4) than we're testing (2.6) so may not work immediately, we've opted for a setup that's further towards the former end of the spectrum.

3.3.5 Load Testing

There are a number of tools available for performing web site testing.

PhantomJS, Selenium, Tsung, Jmeter, Faban, `curl-loader`, `ab`, `httperf`, tcpcopy, and so on.

Of the few that we've listed, PhantomJS and Selenium are closer to the unit testing for correctness end of the spectrum. They operate by attempting to simulate a browser's rendering of a web page's content and allowing one to specify DOM selectors to script interactions with the results and match against expected return data. This can be useful in replicating a browser's tendency to open multiple connections to fetch extra page resources (eg: CSS, JS, etc.) as necessary and for timing the expected time for a user to perceive a page as ready, which incorporates aspects such as the order in which the extra page resources where requested in the original page, rather than simply a network client's ability to receive the bytes. However, they are generally meant for single client simulations.

On the other end of the spectrum, Apache Benchmark (`ab`) and `httperf` can simulate many concurrent connections, complete with keepalives and POST data, but only to a single URL. This can be useful to stress test a particularly popular page, however it is generally emblematic of typical user traffic.

`tcpcopy` is a network layer TCP replay tool that is capable of reusing actual client traffic to test development systems.

Tsung, Jmeter, Faban, and `curl-loader` are somewhere in between. They all have a limited amount of document rendering capabilities, but are more focused on replaying a stream of requests for a requested protocol, in this case HTTP(S). They each also have the ability to handle cookies and POST form data to simulate some limited user interactivity. They are also especially meant to scale these sorts of operations in order to simulate different users concurrently accessing different parts of the system. Included in each are useful graph and summary statistics generation tools as well as warm up period specification options.

Recently Moodle developers have produced an automated course, forum, and JMeter test suite generation tool [9] that others have used to create a semi-automated performance comparison tool [**?**].

We intend to use this tool, perhaps with some extensions to simulate users taking quizes and ensure that extra page resources are also being fetched, in order to conduct our tests.

4 Related and Future Work

4.1 Previous Performance Studies

4.1.1 Moodle

There isn't a great wealth of formal Moodle application specific performance analyses that we were able to find.

The Moodle developers post microbenchmark regression counts of the number of function calls, regex, db load/stores, etc. between major versions in the release notes, but they are not necessarily indicative of overall performance, nor do they include any cache or session subsystem details, so limited insight on different configurations can be gleaned from them. The Moodle forums have no shortage of comments and advice for performance recommendations, but very little with respect to MongoDB, nor is the suggestion of memcached often justified beyond anecdotes.

Work by Coelho et al [18] is the only paper on "Moodle performance" we were able to find that wasn't a study of student learning or instructor teaching achievements. In it, they simulate different load levels for different hardware, OS (Windows vs. Linux) and RDBMS (MySQL vs PostgreSQL) configurations based on some samples from a running system. However, the study was conducted with significantly smaller databases sizes than we have (400M was their "large" database), and on a much older version of Moodle (1.9 in 2008) prior to Moodle including memcached or MongoDB support.

For these reasons, we believe our work still merits attention.

4.1.2 Memcached

Numerous memcached specific benchmarks and comparisons have been done that focus on making memcached faster. [44, 27]

However, while nearly every current major web presence uses some form of an in memory key-value cache like memcached [**?**], there are relatively few studies that examine the general effect of introducing memcached to a typical web application platform consisting of one or more web/application servers and a database server or cluster.

[16] offers one such analysis. As expected it finds that memcached can certainly reduce the load on the database server and thereby improve the number of requests per second the web/application servers are

able to handle. However, they do not address concerns such as the use of memcached for data that should be, at least in part, persistent, nor the effect of cold cache or failures. It is interesting to note that, although they do not take failure scenarios into account, their experiments actually show a slightly higher overhead to using a clustered memcached architecture. We guess that this is due to the extra connection management and hash computations required by the web/application servers.

Another such analysis comes from Facebook. In [37] they introduce a number of changes to the standard memcached to accommodate the extra load and scale that Facebook has as well as handle their concerns with failures, consistency, and at least probabilistic persistence. For instance, a client (mcrouter) handles scheduling and batching requests from web servers to memcached servers helping to reduce the number of network connections and round trips required. They add replication to account for distributing the load for hot key requests as well as failures, and add an invalidation daemon (mcsqueal) on all database servers to handle authoritative updates to the memcached data in order to maintain consistency, again using batching to alleviate network constraints. The replication feature also allows for a simple cache warmup mechanism. In addition to a number of other daemon modifications, they also reworked the memory allocation strategy so that data in the cache could survive a software upgrade, though not a complete systems reboot, also reducing the possibility of a cold cache scenario. However, several of their extensions, not all of which have been open sourced, required modifications to their application code. For instance, for mcsqueal to be able to invalidate upstream caches upon writes to the database layer, queries needed to include the memcache lookup key. Although their work certainly advances the memcached technology, it does not seem directly applicable to such a comparatively small platform as ours.

4.1.3 MongoDB

There have been a number of studies on performance that have included MongoDB [39, 22, 47, 33, 45]. However, they tend to focus on whether or not MongoDB is a suitable replacement for a traditional RDBMS system, particularly for workloads such as document management, analytics, GIS data, etc. There appears to be very little data available on MongoDB when used as a cache system in front of an RDBMS.

4.2 Performance Analysis Techniques

4.2.1 Benchmarks

The NoBench [17] microbenchmark provides a means for directly measuring specific query types for a given NoSQL system. However, it currently lacks the ability to measure concurrency or scale out. [34] also proposes a combined NoSQL RDBMS system benchmark, currently developed for Windows, however the Yahoo! Cloud Serving Benchmark (YCSB) [19] is probably the current standard for such comparisons and supports scalability as well as simulated concurrency through the use of multiple threads on a single YCSB client.

Each of these tools is beneficial in comparing two or more products for serving the same task, but does not necessarily help us when determining their effect when used in combination with one another, each for a different subtask. Furthermore, as we do yet not currently know the exact workload types that Moodle may impart on each of these candidate systems, single system benchmark results would not offer us very much insight, though we could consider taking traces of some of our tests in order to help understand that relationship better.

Perhaps the closest benchmark for testing such an environment that was not Moodle application specific could be the now discontinued TCP-W benchmark [15]. However, the specification limits the tests to single systems since the focus is not on scalability. Moreover, its model is based on that of an e-commerce site that requires higher ACID compliance than the cache and session portions of our dataset that we are trying to alleviate load from.

4.2.2 Load Testing

Unlike benchmarking, which tends to stress test a particular component, load testing is focused on stress testing an actual end-to-end system. In our case a web application.

Since there are more variables at play there are many questions to consider when devising such a test. [20] provides a nice overview of a number of options relating to web application testing.

As mentioned in Section 3.3.5, tools like `ab` and `httperf` that only fetch a single URL, perhaps according to some probability distribution, do not adequately capture most client interactions with modern web applications, and so do not serve as useful load testing solutions.

[35, 40] each offer an argument for closed or partly-open loop models that include think time for user sessions in order to accurately reflect their interactions within a web application. As Moodle interactions are generally gated by a login step, this partly-open-loop model is a much better representation of our workload, and one which we believe our Jmeter tests implement well.

[35], which is web site load testing specific, also argues for reporting results in terms of number of abandoned user sessions due to long response times and lost revenue in addition to the typical page load response times as it varies over number of clients. Although the lost revenue figures do not directly apply to our scenario, we like the idea of incorporating the fact that users give up if a site seems to overwhelmed.

[21] also includes the idea that multiple different classes of clients, that interact with the site in different ways, may be interacting with the system concurrently. This is certainly true in our environment as some students may be merely browsing course content (read dominant) while others could be taking a quiz (write and CPU dominant) or interacting with a forum (a mixture or read and writes).

Characterizing each of those interactions can be a difficult task in any load testing framework. Work in [25] aims to help automate that task through log mining of previously recorded load tests, though there is no reason they couldn't also be applied to production logs as well, in order to weed out anomalies that may require manual intervention as well as allowing an operator to focus on dominant behavior patterns.

Cloudstone [41] includes a nice summary of the arguments for needing to take the entire web application stack and different user interaction streams into account when testing. Though their model is based around computing the optimal platform setup for a given cost per user and SLA (eg: 99th percentile response time) within a cloud hosted environment such as Amazon's, the techniques they employ and specific test cases they examine are very similar to those we have.

While the above focus on techniques for developing an general testing strategy that can be applied to specific instances, WikiBench [43] is an example of a specific implementation of such a test strategy that employs real data set and traffic replicas. Unfortunately, it is specific to the MediaWiki application, though can be used to test different platform configurations for it.

A subject related to load testing is correctness testing through the use of unit tests for instance, however we do not address this issue in our work.

4.3 Failure Models

As mentioned in in Section 3.2, we also would like to understand the impact of both brief component outages due to software maintenance vs. machine failures that may lead to full system outages or cold cache performance issues.

There are several other failure models we could have considered, such as network partitions or the Netflix Chaos Monkey.

However, given the comparatively small size of our system we think our simple tests are the far more likely scenario. Beyond that, the entire system is likely to be affected anyways.

Usually service outages are mitigated through replication and automatic failover and partitioning. MongoDB for instance includes this functionality natively. However, in a typical memcached cluster, there is no replication, only key space partitioning, so failures may reduce the scope of an outage partially.

[46] offers a brief analysis of cache consistency and failures when using memcached in a clustered environment. In it, they propose the use of consistent hashing combined with replication in order to tolerate failures, as well as Paxos [32] to achieve consistency between replicas. However, they do not address the performance implications of such an addition, which we expect could be substantial.

4.4 Security

In general, most NoSQL systems, including MongoDB and memcached are designed to be operated in a trusted environment. Authentication is usually not required by default, so granular authorization is not possible. Moreover, just as their SQL predecessors, they are often vulnerable to injection attacks from clients. [38]

In our general shared hosting platform we employ ModSecurity [4] to attempt to filter input for things like injection and DoS attacks before it is received by the application from the web server. However, due to the high expectation of false positives for computer science related coursework in particular, the precaution was requested to be disabled for the Moodle service.

Moreover, the filter does nothing to prevent an attack or abuse on the backend services, malicious or not, once a client is operating within the web service platform.

4.5 Additional Configurations and Technologies

In addition to MySQL, Moodle also supports PostgreSQL as a primary database. It is possible that it offers different performance characteristics than MySQL which may worth be exploring at some point. Unfortunately, there is little more than conjecture to this point on much of the moodle.org related sites mostly coming down to very old comments of what constitutes "properly tuned" and whether or not MySQL's InnoDB was available for production use. Our initial impression from the literature is that PostgreSQL has traditionally been better at ensuring proper data durability and full ACID compliance, which might lead to more overhead in write intensive scenarios such as ours.

There are also other persistent NoSQL choices such as Cassandra or CouchDB that Moodle could likely be adapted to support. Both of which list some compelling resiliency features similar to MongoDB's such as multi-master replication, no single point of failure, and per operation tunable durability.

As mentioned in Section 3.1.1, Couchbase is another alternative to memcached which provides protocol compatibility but adds replication and persistence capabilities to avoid cold cache restart and repartitioning issues, though comes at a licensing cost. memcachedb is an opensource alternative with similar advantages, but does not currently support expiration of key-values.

In general we note that not all data is equal and applications' need for a combination of technologies like fully ACID RDBMS, no durability memcached, and somewhere in the middle NoSQL solutions like MongoDB's write concern levels represent a need for "degrees of durability", similar in nature to Gray's degress of consistency work. [23]

The term "degrees of durability" also appears in work on the Oracle NoSQL Database [26] in which applications can specify per operation what level of durability and consistency they would like the system to provide. For instance, a write may be considered committed as soon as either a network acknowledgment from a replica or the local log disk write has been received, reducing the overall write operation's latency to whichever of the two systems has the lower latency. This is very similar to MongoDB's notion of write concerns.

In [36] Nance et al also offer a nice summary of the different arguments for a mix of both traditional RDBMS and NoSQL systems, as they are often designed for and solve different problems relating to both the data organization required as well as their consistency and durability requirements.

References

[1] Couchbase memcached. `http://www.couchbase.com/memcached`.

[2] Innodb integration with memcached. `https://dev.mysql.com/doc/refman/5.6/en/innodb-memcached.html`.

[3] Memcached: Client configuration. `http://code.google.com/p/memcached/wiki/NewConfiguringClient`.

[4] Modsecurity: Open source web application firewall. `http://www.modsecurity.org/`.

[5] Mongodb: Journaling mechanics. `http://docs.mongodb.org/manual/core/journaling/`.

[6] Mongodb: Replica set oplog. `http://docs.mongodb.org/manual/core/replica-set-oplog/`.

[7] Mongodb: User privilege roles. `http://docs.mongodb.org/manual/reference/user-privileges/`.

[8] Mongodb: Write concerns. `http://docs.mongodb.org/manual/reference/write-concern/`.

[9] Moodle: Jmeter test plan generator. `http://docs.moodle.org/26/en/JMeter_test_plan_generator`.

[10] Moodle performance comparison tool. `https://github.com/moodlehq/moodle-performance-comparison`.

[11] Thundering herd problem. `http://en.wikipedia.org/wiki/Thundering_herd_problem`.

[12] Mysql bug 2917. `http://bugs.mysql.com/bug.php?id=2917`, 2010.

[13] Memcache benchmarks. `http://symas.com/mdb/memcache/`, 2013.

[14] Moodle issue 40569. `https://tracker.moodle.org/browse/MDL-40569`, 2013.

[15] Tpc-w. `http://www.tpc.org/tpcw/`, 2013.

[16] K. A. Bakar, M. H. M. Shaharill, and M. Ahmed. Performance evaluation of a clustered memcache. In *Information and Communication Technology for the Muslim World (ICT4M), 2010 International Conference on*, pages E54–E60. IEEE, 2010.

[17] C. Chasseur, Y. Li, and J. M. Patel. Enabling json document stores in relational systems.

[18] J. Coelho and V. Rocio. A study on moodle's performance. 2008.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[20] G. A. Di Lucca and A. R. Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172–1186, 2006.

[21] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber. Realistic load testing of web applications. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 11–pp. IEEE, 2006.

[22] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang. Can the elephants handle the nosql onslaught? *Proceedings of the VLDB Endowment*, 5(12):1712–1723, 2012.

[23] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[24] S. Hemelryk. The moodle universal cache (muc). `http://docs.moodle.org/dev/The_Moodle_Universal_Cache_(MUC)`, 2013.

[25] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 307–316. IEEE, 2008.

[26] A. Joshi, S. Haradhvala, and C. Lamb. Oracle nosql database-scalable, transactional key-value store. In *IMMM 2012, The Second International Conference on Advances in Information Mining and Management*, pages 75–78, 2012.

[27] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 9. ACM, 2012.

[28] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *Proceedings of the 15th international conference on World Wide Web*, pages 595–604. ACM, 2006.

[29] J. Krizanic, A. Grguric, M. Mosmondor, and P. Lazarevski. Load testing and performance monitoring tools in use with ajax based web applications. In *MIPRO, 2010 Proceedings of the 33rd International Convention*, pages 428–434. IEEE, 2010.

[30] B. Kroth. Cs787 project: Bin packing: A survey and its applications to job assignment and machine allocation. 2013.

[31] B. Kroth. Cs787 project proposal. 2013.

[32] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[33] Y. Li and S. Manoharan. A performance comparison of sql and nosql databases. In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, pages 15–19. IEEE, 2013.

[34] I. LUNGU and B. G. TUDORICA. The development of a benchmark tool for nosql databases. *Database Systems Journal BOARD*, page 13.

[35] D. Menascé. Load testing of web sites. *Internet Computing, IEEE*, 6(4):70–74, 2002.

[36] C. Nance, T. Losser, R. Iype, and G. Harmon. Nosql vs rdbms-why there is room for both. In *Proceedings of the Southern Association for Information Systems Conference*, pages 111–116, 2013.

[37] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 385–398. USENIX Association, 2013.

[38] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov. Security issues in nosql databases. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 541–547. IEEE, 2011.

[39] Z. Parker, S. Poe, and S. V. Vrbsky. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*, page 5. ACM, 2013.

[40] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, volume 6, pages 18–18, 2006.

[41] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, volume 8, 2008.

[42] N. Sonwalkar. The first adaptive mooc: A case study on pedagogy framework and scalable cloud architecturepart i. In *MOOCs Forum*, volume 1, pages 22–29. Mary Ann Liebert, Inc. 140 Huguenot Street, 3rd Floor New Rochelle, NY 10801 USA, 2013.

[43] E.-J. van Baaren. Wikibench: A distributed, wikipedia based web application benchmark. *Master's thesis, VU University Amsterdam*, 2009.

[44] I. Voras, D. Basch, and M. Zagar. A high performance memory database for web application caches. In *Electrotechnical Conference, 2008. MELECON 2008. The 14th IEEE Mediterranean*, pages 163–168. IEEE, 2008.

[45] Z. Wei-ping, L. Ming-Xin, and C. Huan. Using mongodb to implement textbook management system instead of mysql. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 303–305. IEEE, 2011.

[46] P. Xiang, R. Hou, and Z. Zhou. Cache and consistency in nosql. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 6, pages 117–120. IEEE, 2010.

[47] J. Yang, W. Ping, L. Liu, and Q. Hu. Memcache and mongodb based gis web service. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 126–129. IEEE, 2012.