CS764 Project Proposal

Brian Kroth

2014-02-06

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 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.). The Moodle portion of the service 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]

Dynamic user data such as PDF files and quiz question banks are split between an NFS store and a MySQL database server. The MySQL server has binary logging enabled in order to asynchronously replicate changes to a slave server so that consistent backups can be taken of the database without locking all tables or interrupting service (mixed student and instructor usage means that the service does not follow a typical diurnal pattern). No reads are currently directed at the slave database server.

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 serialized data structure, 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 consistency requirements on it are not as great 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 [3], 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.

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 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 on the row object that isn't

[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.

[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] http://bugs.mysql.com/bug.php?id=2917

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 [4], Moodle makes heavy use of caching to attempt to improve performance. By default, and as it is currently configured, it caches minified JavaScript, unified CSS, localized strings, LaTeX rendered equation images, and more 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 not necessarily built with multiple backends servers in mind, inconsistencies and program errors [5] 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 Project Goals

3.1 Session Data

In addition to the main database session store provider described above, the Moodle code base now also has support for Memcached and MongoDB as session data targets. 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.

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

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

[4] Indeed most things in computer science.

[5] This one occurred just recently, for instance: https://tracker.moodle.org/browse/MDL-40569

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 [6] of requests as we vary the number of active clients in our tests.

3.2 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, MongoDB, and NFS as cache targets.

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

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

4 Test Methods

To test each of these alternative configurations we propose to

1. Implement a memcached cluster. There are questions of sizing and authentication here, to name a few options, that we will need to explore.
2. Implement a MongoDB database. There are questions of write sync policy, master-slave replication, and authentication here, to name a few option, that we will need to explore.
3. Create a test environment with 1000 test accounts and at least one course and quiz.
4. Construct a Selenium [7] test suite to simulate a user taking the quiz.
5. Conduct test runs of 50, 100, 250, 500, and 1000 simultaneous quiz attempts from our roughly 100 Linux lab machines.
6. Measure the response time for each page request.

We intend to report our results in terms of 95 percentiles.

As this test involves multiple pages, many database writes, cache and session interactions, and varying levels of concurrency, we think it is a reasonably characteristic representation of our workload.

[6] eg: 95 percentile

[7] http://docs.seleniumhq.org/

5 Related and Future Work

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.

The NoBench [1] microbenchmark provides a means for directly measuring specific query types for a given NoSQL system. However, it currently lacks the ability to measure concurrency. Furthermore, as we do not currently know the exact workload types that Moodle may impart on each of these candidate systems, NoBench 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.

There isn't a great wealth of Moodle application specific performance analyses that were initially able to find, so further research will also be required.

References

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