PaperSphere: A Social Network for Sharing Research Material

An Analysis of the Development Model, Scalability Testing and Availability Testing

CS290B: Scalable Internet Services (Fall 2013)

Hiranya Jayathilaka
Nevena Golubovic
Alex Pucher
Chris Horuk

1.0 Introduction

PaperSphere is an on-line social networking application that enables students and researchers to easily share research papers, technical articles and other published academic material. It is designed to make research and similar academic endeavors fun and easier by making it simple to locate interesting research material and share them with colleagues seamlessly. Main features of PaperSphere can be summarized as follows:

- Browse for published material in well-known databases like ACM Digital Library (http://dl.acm.org) and IEEE Explore (http://ieeexplore.ieee.org).
- Create reading lists consisting of research papers, articles and other published material.
- Create groups consisting of fellow researchers, teammates and colleagues.
- Share reading lists with groups.
- Rate articles, comment on them and take notes.

PaperSphere is implemented in Ruby-on-Rails and uses an RDBMS (e.g. MySQL) as the backend data store. The rest of this report discusses the experiences and lessons learnt during the development of PaperSphere, and illustrates the results obtained during the subsequent testing phase, which put the performance, scalability and availability of PaperSphere into trial.

2.0 Development Model

We followed an open and agile development methodology to implement PaperSphere. The initial development phase consisted of five sprints, with each sprint lasting for a week. A soft release and a demonstration were conducted at the end of each release to ensure forward progress. Our source code is available on GitHub (https://github.com/ethereal/papersphere).

We made extensive use of the branching and merging capabilities of Git during the development stage of the project. Each developer worked on a separate branch to avoid conflicts during the sprints. The entire development team met twice every week to merge all the development branches into the master branch, while addressing any potential conflicts as a team. This development model turned out to be both efficient and fun. All the soft releases and demonstrations were done from the master branch.

We did not follow a strict test-driven development approach, but we made sure all our code has ample test coverage. Each developer was responsible for adding unit tests and functional tests for every component (models, views and controllers) he/she implement. Having sufficient test coverage was considered a requirement for any component to be eligible to be merged into the master branch.

3.0 Scalability Testing and Optimization

After the five development sprints, we started testing PaperSphere on Amazon EC2. Following software packages were used to setup the initial production environment:

- Web server: Nginx
- Application server: Phusion Passenger
- Database: MySQL server

Capistrano was used to automatically push the latest code to the production environment, and set it up for testing.

Subsequent sections describe various iterations of our testing process, results observed and the optimizations performed.

3.1 Testing with a Moderately Large Dataset

To start with we created a test dataset that consists of 10,000 users where each user owns up to 5 reading lists. We populated the PaperSphere database with this test dataset, and manually tested the performance of the application by logging into the system and interacting with it using a regular web browser. Immediately we started noticing a number of performance issues in the application. Some of the most prominent issues are listed below:

- User home page takes over 5 seconds to load.
- A reading lists takes over 2 seconds to load.
- A paper takes over a second to load.

We debugged these issues using the built-in logging capabilities of Rails (especially the development log that specifies the SQL queries executed by the application, along with the time taken by each query). We noticed that our application makes a large number of unnecessary SQL queries, some of which are expensive joined range queries. We performed following optimizations where appropriate to avert the observed performance bottlenecks:

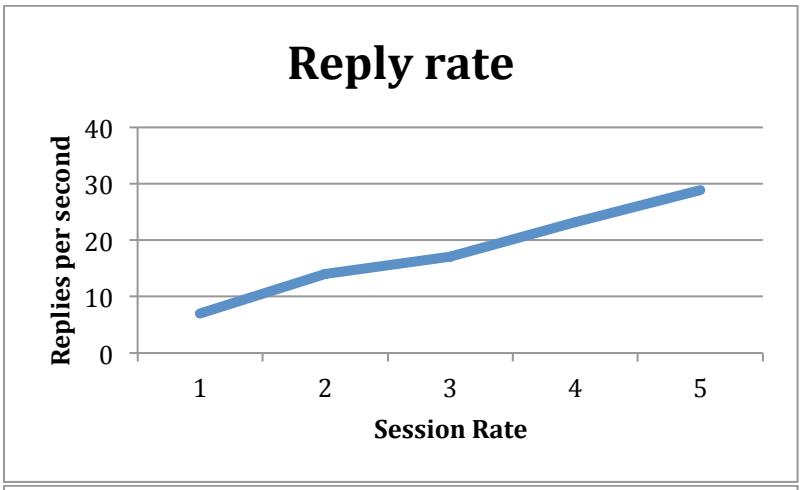
- Removing COUNT(*) queries by making the row count an attribute of the appropriate models (e.g. We added a paper_count attribute to the ReadingList model. This way, the number of papers in a reading list can be read without having to do a joined range query over the papers table.)
- Aggressively pre-fetching data from the tables, instead of loading one row at a time (avoiding iterative queries)
- Using more efficient custom SQL queries instead of the ones generated by the Rails ORM layer
- Using the delayed_jobs library to send e-mail notifications asynchronously in the background

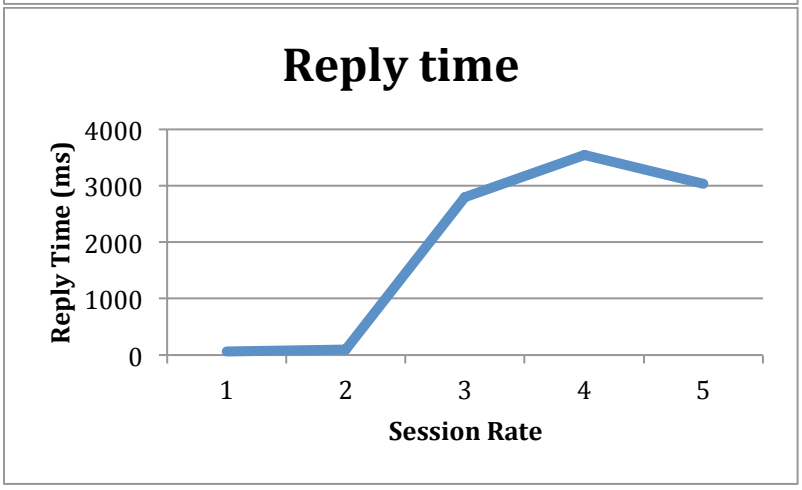
The performance gain obtained by these optimizations was significant. In case of loading the home page, we managed to reduce the latency from over 5 seconds to under 200 milliseconds. In general, the above optimizations helped us to get the loading time of every page in our application under 200 milliseconds.

3.2 Testing with a Very Large Dataset

As the next step, we created a very large dataset consisting of 100,000 users. We installed all components (web server, app server and database) of PaperSphere in a single m1.medium EC2 instance and conducted performance testing on it using httperf. The load generator was executed on a separate m1.xlarge instance. A larger instance was used to run the load test client, so that the client would not be inhibited by any resource shortages.

Following graphs show the behavior of PaperSphere as the session rate (no. of sessions initiated by httperf per second) is increased from 1 to 5. The particular workload used to test PaperSphere in this scenario is a read-only workload.





According to the above results, PaperSphere can serve up to a maximum of 17 replies per second without getting overloaded. This scenario occurs at a session rate of 3. But as seen from the reply time plot, the average latency has already increased past 2 seconds at this point, which is an indication that the server is nearing its capacity. Any attempts to further increase the session rate simply results in errors, as shown in the errors plot.

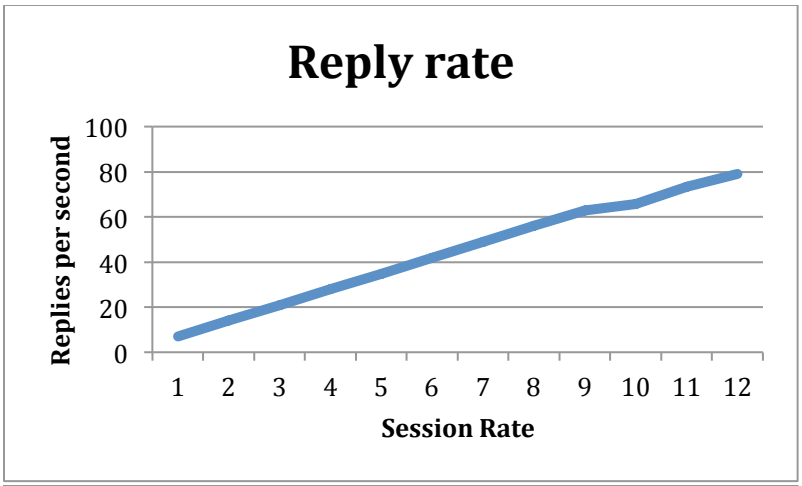
A read-write workload on an m1.medium instance yielded similar results with a maximum successful reply rate close to 16, and errors starting to surface as early as at session rate of 3.

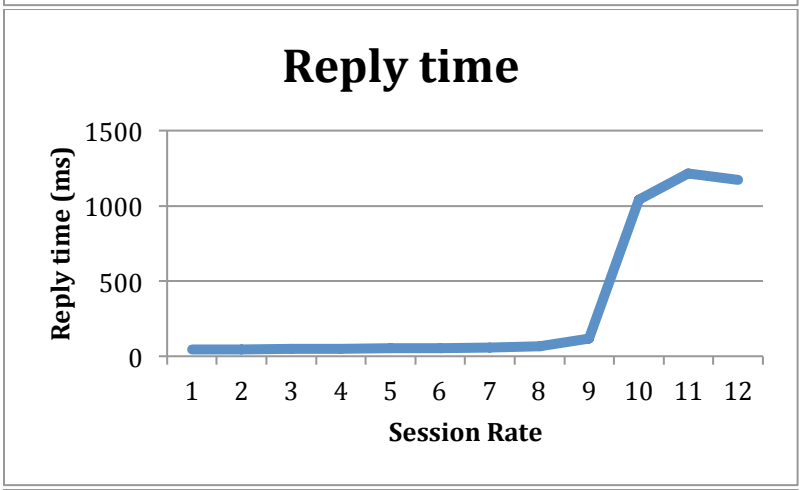
A closer investigation of the virtual machine hosting the PaperSphere application revealed that the CPU usage approaches 100% as we increase the session rate of httperf to 3. The requests get queued at Nginx awaiting processing, but at some point the queue starts to overflow. At this point, Nginx stops accepting new requests and simply responds to all new requests with HTTP 503 error. Since our application is somewhat CPU-bound, we concluded that a single core VM like an m1.medium instance is not suitable for further testing PaperSphere. We chose larger VMs with more CPU power for our subsequent load tests.

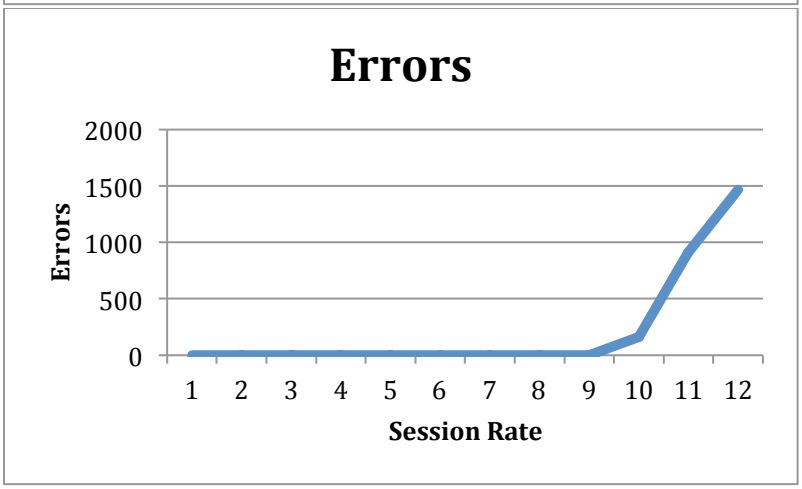
3.3 Scaling Up (Vertical Scaling)

Next we installed PaperSphere on a single m1.xlarge instance and ran the same read-only workload using httperf. The m1.xlarge instance type has 4 CPU cores and about 15GB of memory. This is almost a 4x increase in resources compared to m1.medium instances.

As expected, PaperSphere exhibited much better performance on this setup. The reply rate, reply time and error plots are displayed below.







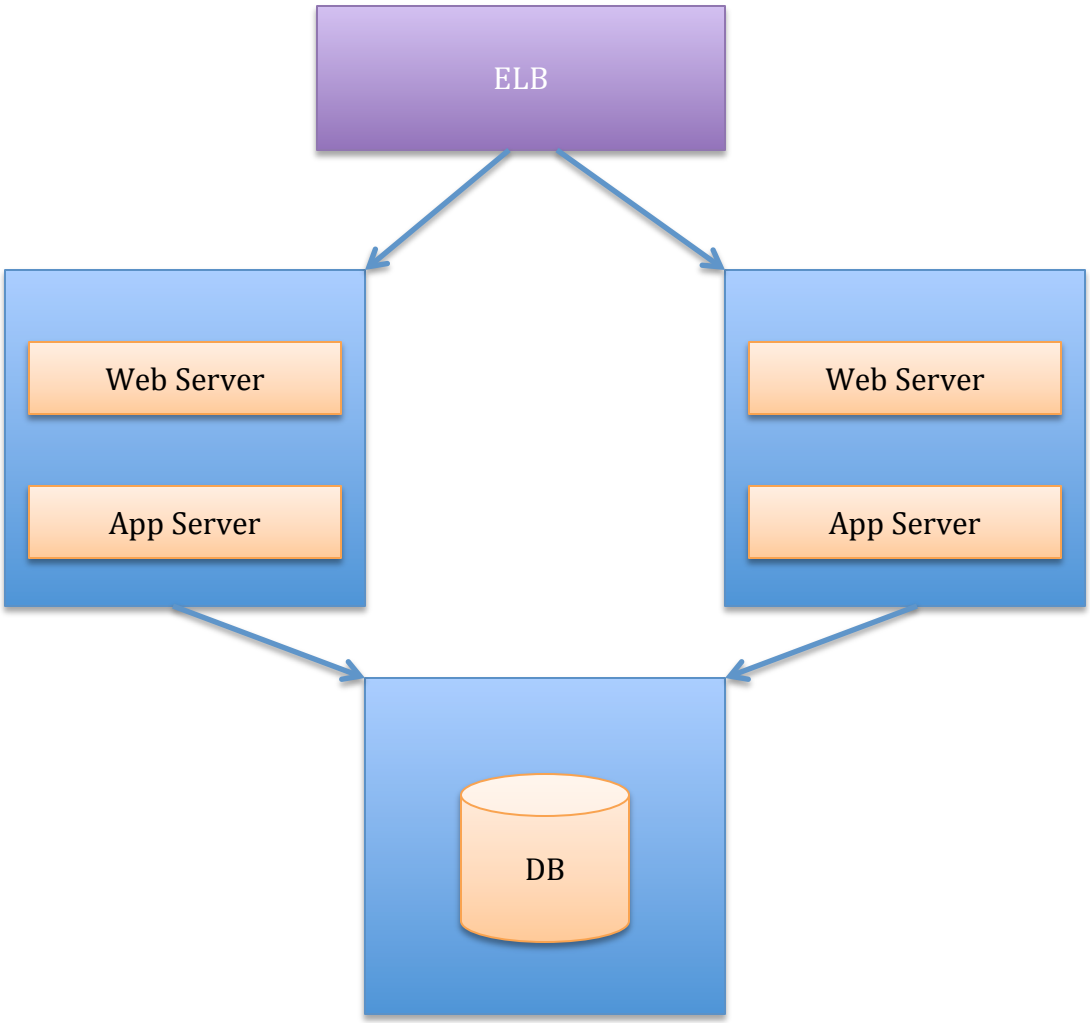
In this case we can increase the httperf session rate up to 9 without getting any errors. The reply rate observed at this point is around 63, an almost 4x improvement. The maximum latency observed during the test is also under 1300ms. These results indicate that PaperSphere can greatly benefit from vertical scaling.

A read-write workload on the same setup yielded very similar results. We achieved a maximum error-free reply rate close to 64 and errors started to appear at the session rate of 9.

As far as the usage patterns are concerned, we expect PaperSphere to be mostly read than written. Users will occasionally create new reading lists and groups, but will spend most of their time reading existing lists and papers. Therefore in production, we expect the workload to be mostly-read.

3.4 Scaling Out (Horizontal Scaling)

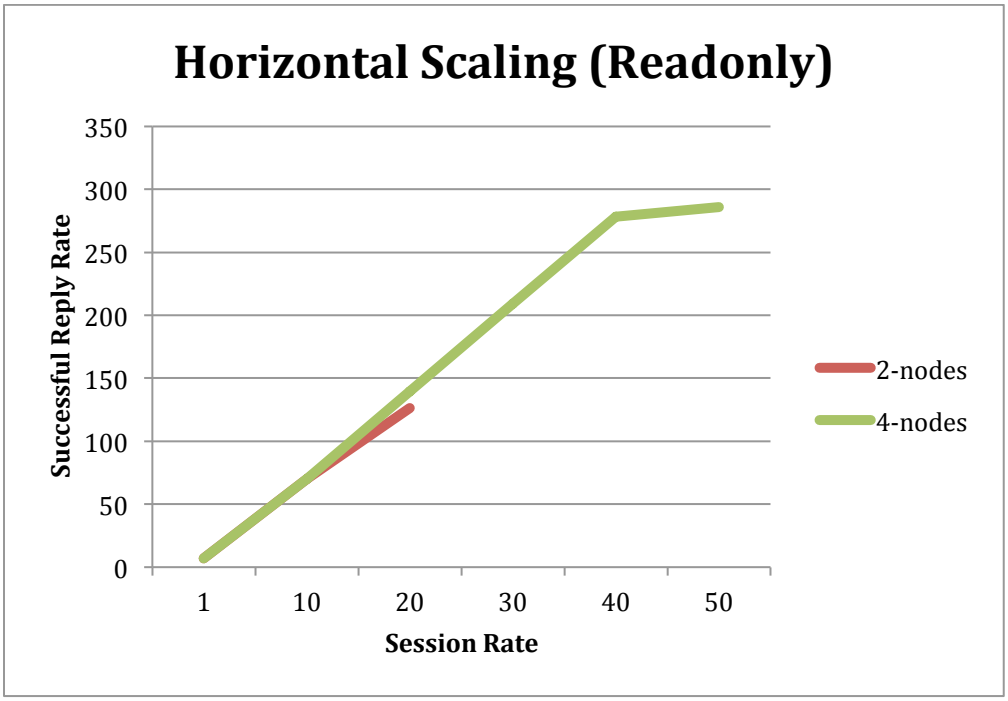
In order to test the horizontal scaling features of PaperSphere, we first separated its web server and app server components from the database. We deployed the database in a dedicated m1.xlarge EC2 instance. We acquired two additional m1.xlarge instances and installed a web server and an app server in each of them. We further acquired an elastic load balancer (ELB) instance and configured it to load balance the traffic among the two web servers. The resulting test environment is depicted below (each blue box is an m1.xlarge EC2 instance).

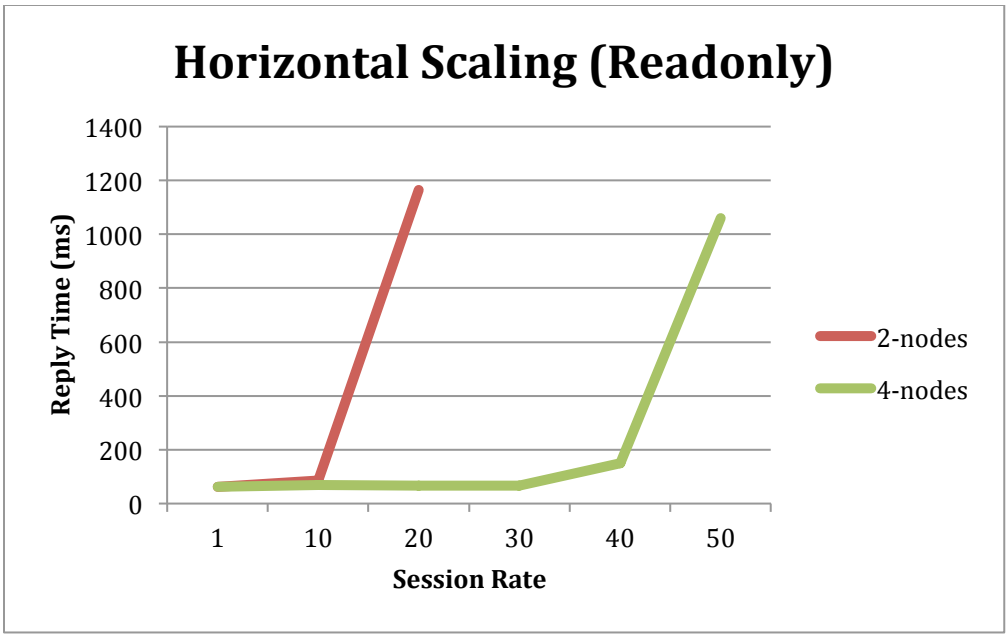


We tested the above setup by running httperf against the load balancer. We continued to test while increasing the number of web servers and app servers. That is, we acquired more m1.xlarge EC2 instances, installed web servers and app servers in them, and added them to the ELB target server pool.

We expected a near linear increase in performance with the number of servers added. But unfortunately, we initially failed to achieve the expected linear speedup. A closer look at our setup under load revealed that the database server is getting heavily loaded (very high load average on the EC2 instance hosting the database), when we increase the number of app servers. This resulted in sluggish SQL query performance all-around.

We again resorted to analyzing individual SQL query performance using the rails development logs, but this time under some heavy load. We noticed that when the traffic volume is high, some SQL queries take up to two seconds to complete. We wrote some parser scripts (using grep and awk), to detect the slowest queries under load. We analyzed each of these queries using the MySQL Explain tool and found almost all of them were conducting unnecessary full range scans on tables. We added four new indexes to the database to fix these limitations. With the new indexes in place, we managed to reduce the maximum time spent on a SQL query to under 50ms, even under heavy load. With that we managed to obtain the desired linear speedup. Results are shown below.





First, notice that with 2 app servers we can achieve a maximum successful reply rate close to 120. This is almost twice what we achieved in our all-in-one setup earlier. With 4 app servers the maximum reply rate doubles once again, thus indicating the ability of PaperSphere to speed up linearly with more app servers. The reply time plot also illustrates interesting results. With 2 app servers we can increase httperf session rate to around 10 before we see a significant degradation. With 4 app servers, we can go up to a session rate of 40 without any problems. The above results are for a read-only workload. We got very similar results for read-write workloads.

After the database indexes were added, the load average on the database node was kept at bay even with 4 app servers. The new limiting factor of performance in the latest setup was again the app server CPU utilization. Under heavy load, each app server node was running close to 100% CPU utilization.

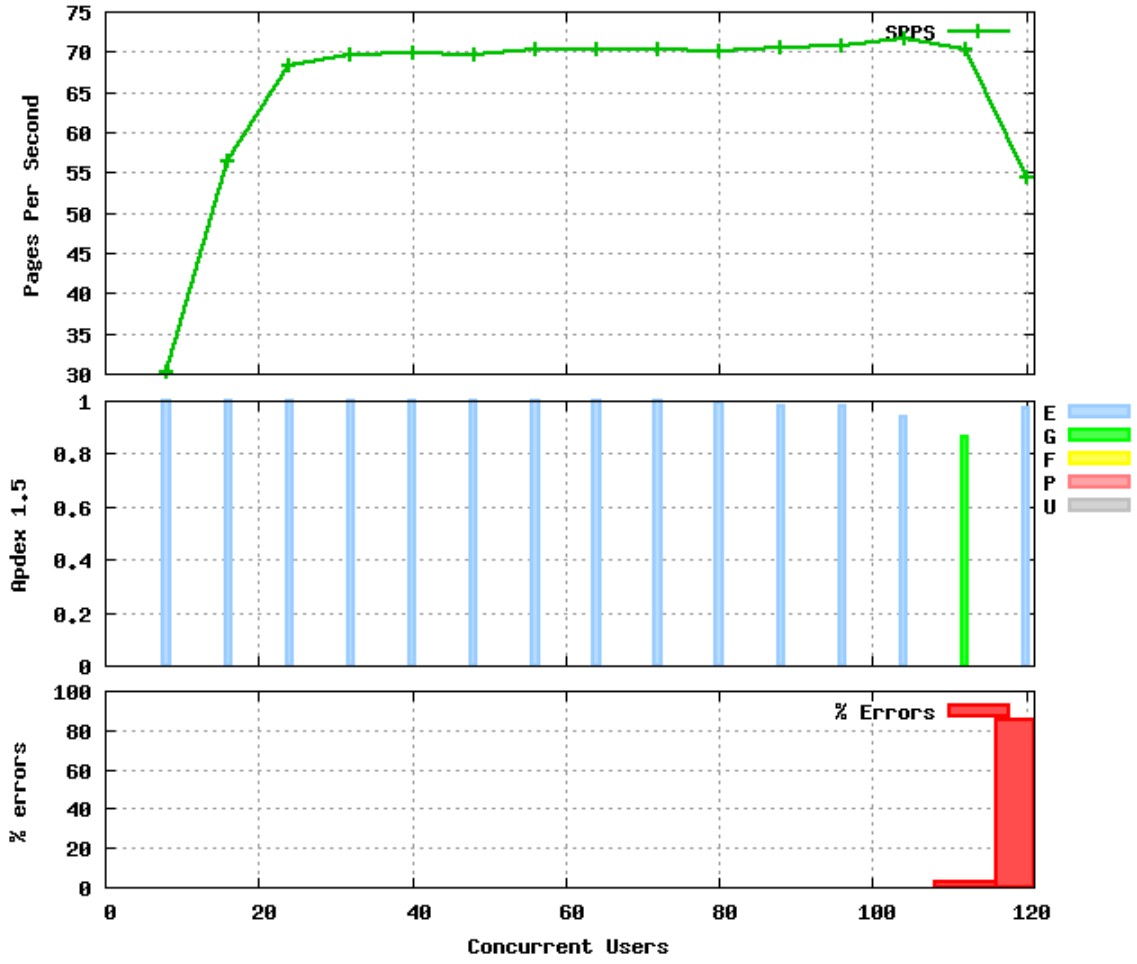
3.5 More Dynamic Tests with Funkload

In all our previous tests with httperf, the workloads were designed to interact with PaperSphere as a single specific user. As a result we could only exercise a very small portion of the overall test dataset. Simply put, httperf lacked the required level of flexibility and programmability to interact with PaperSphere as more than one user. Funkload, a Python-based load-testing tool gave us the required programmability to create more complex and dynamic workloads.

To begin with we created a read-only workload that logs into PaperSphere as a randomly chosen user, and then interacts with randomly chosen reading lists, papers and groups. This workload is way too dynamic to the point that it renders

any sort of caching performed by the database or the app server completely useless. Therefore, results obtained by running this workload can be considered a worst-case performance analysis of PaperSphere, since in real world, web applications have heavily skewed access patterns with some temporal and/or spatial locality.

We started testing PaperSphere using Funkload, in an all-in-one setup. That is, all the components of PaperSphere were installed in a single m1.xlarge EC2 instance. Funkload was executed from a separate m1.xlarge instance. We realized very early in the testing process that a single Funkload process is not sufficient to drive the server anywhere close to its capacity. Therefore we resorted to testing PaperSphere with multiple concurrent Funkload processes. We leveraged the built-in distributed testing support of Funkload to spawn eight Funkload processes on two m1.xlarge EC2 instances. The results obtained from the all-in-one server environment are shown below.



According to the above results, using a single m1.xlarge EC2 instance as the server, we can serve up to 70 pages per second, and 100 concurrent users without running into errors. However, the latency plot (not included) indicates that the average page load time degrades past 500ms once the number of concurrent users has been

increased beyond 50. But since this is the worst case performance analysis that doesn't benefit from caching, we believe the result to be acceptable for an all-in-one test environment.

We repeated this test with a read-write workload and achieved similar results, but with a slightly higher pages-per-second value (~100). Our read-write workload makes lot more requests compared to the read-only workload and we believe this discrepancy is due to that difference and it is not necessarily an artifact of the write performance of PaperSphere.

3.6 Horizontal Scalability Tests with Funkload

We repeated our Funkload tests while adding more app servers to the test setup fronted by an ELB. The database server was hosted on a separate dedicated m1.xlarge instance for these tests.

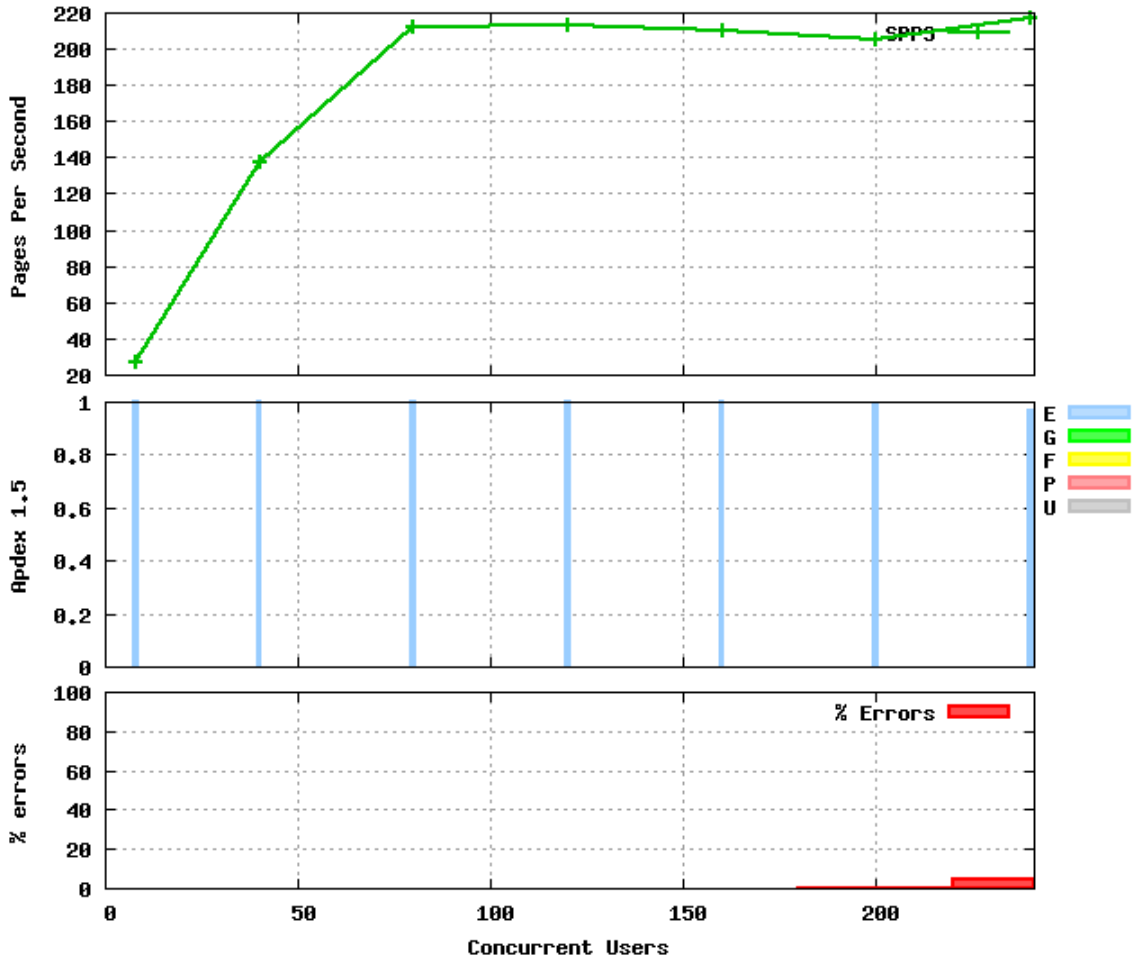
With 2 app servers we managed to get close to 140 pages per second, almost twice what we achieved in the previous all-in-one setup. We were also able to push the concurrency level up to 184 concurrent users without any errors. The page load time did not hit the 500ms mark until the concurrency level was increased to 100.

With 4 app servers PaperSphere served up to 275 pages per second without errors. This is again roughly twice the throughput achieved with 2 app servers. We could push it up to 352 concurrent users without observing any errors and the page load time reached 500ms only after the concurrent user count was increased past 200.

3.7 Vertical Scalability Tests with Funkload

A close look at our previous test environments showed that the app servers are running close to 100% CPU utilization most of the time. In other words, PaperSphere is mostly limited by the availability of CPU cores and CPU power. To test out this theory, we repeated our tests with the app servers hosted on c1.xlarge EC2 instances. These EC2 instances are compute optimized and contain 8 CPU cores. Interestingly they only have 7GB of memory (compared to 15GB in m1.xlarge), but our previous test results didn't indicate any contention for memory. Therefore this was a very sensible thing to try out.

Some plots obtained from this test are shown below (2 app servers setup). Clearly with c1.xlarge instances PaperSphere performs better. We were able to achieve over 200 pages per second as opposed to 140 observed with m1.xlarge. The page load time did not go past 500ms until we loaded the system with 150 concurrent users (compared to 100 users with m1.xlarge). Perhaps the most notable achievement was the reduction in maximum page load time, which was now under 900ms, as opposed to 1400ms observed with m1.xlarge instances.



3.8 Optimization via Configuration

At this point we started experimenting with some of the configuration settings of MySQL, Nginx and Passenger in the hope of getting better performance. Not all of the changes yielded a positive gain as expected. This section outlines some of the changes we did and the observed outcomes.

- Increasing MySQL buffer pool size:

  MySQL uses a buffer pool for caching data and indexes in memory. By default this is only 128MB in size. We increased this limit up to 1GB so that potentially MySQL can serve the entire database from the memory (our large dataset is about 750MB in size). We were able to comfortably accommodate this change into the test environment, since the m1.xlarge instance hosting the MySQL server had around 15GB of memory. This change did improve the performance of PaperSphere by slightly reducing the latency of most SQL queries.

- Database warm up:

  Before starting the load tests, we ran a special warm up script directly against MySQL, so that it would load most of its tables and indexes to the in-memory cache. The warm up script primarily comprised of a sequence of SELECT * queries and some other range queries. This also yielded some gain as it helped to improve the number of pages served per second by a small amount.

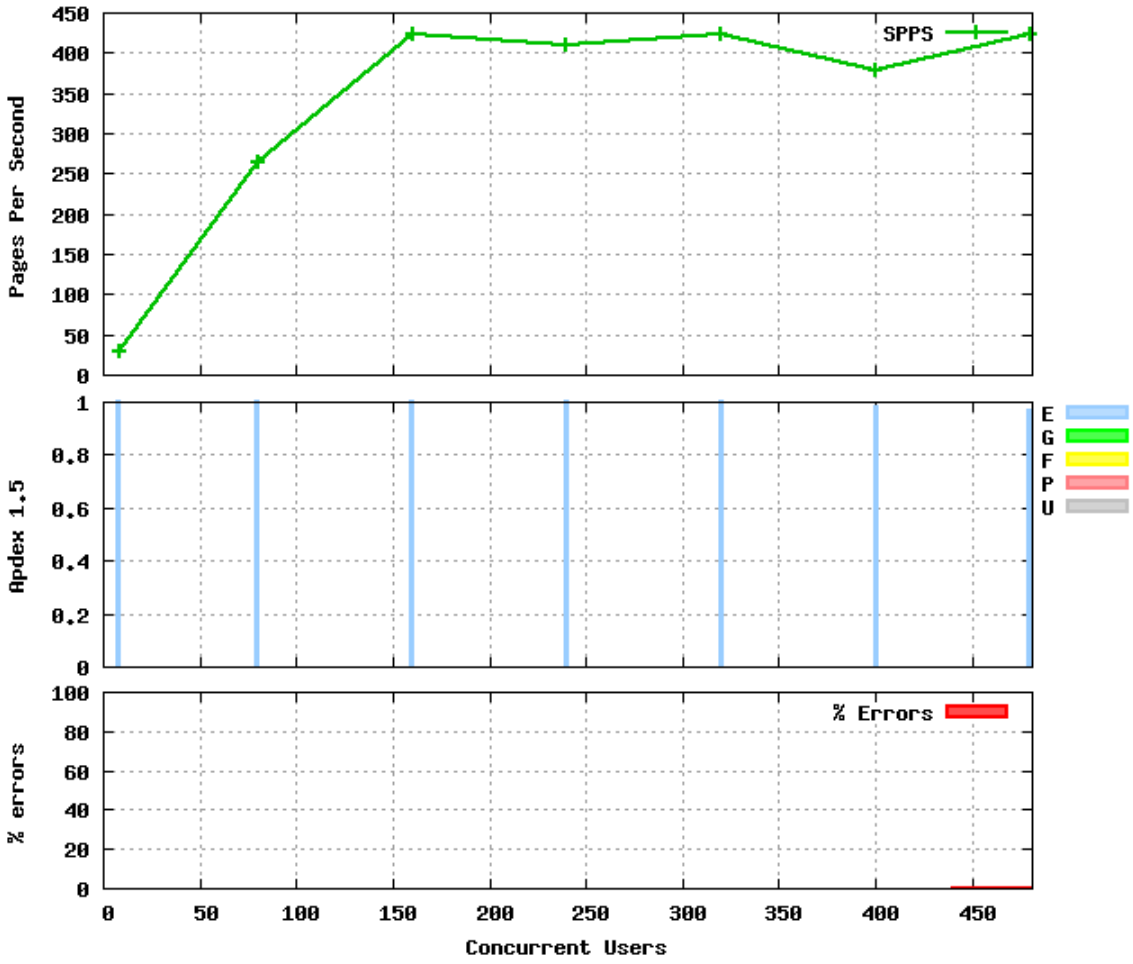
- Nginx optimizations:

  We experimented by changing the default number of Nginx workers, enabling response compression, disabling Nagle's algorithm and configuring various other low-level TCP settings. But none of these changes achieved any noticeable gain. It turned out that Nginx automatically chooses the optimal values for most of its configuration parameters on startup, and they yield the best possible performance level most of the time.

- Phusion Passenger Optimizations:

  At this point in time we were running our load tests with app servers hosted on c1.xlarge EC2 instances. By default each app server spawns up to 6 processes to handle the incoming requests. But c1.xlarge EC2 instances consist of 8 CPU cores, which meant we could run more worker processes on each VM. To make proper use of the CPU power available in c1.xlarge instances, we configured Passenger app server to spawn up to 16 worker processes (2 processes per core). This is another optimization that greatly paid off. Results from this test are shown below.

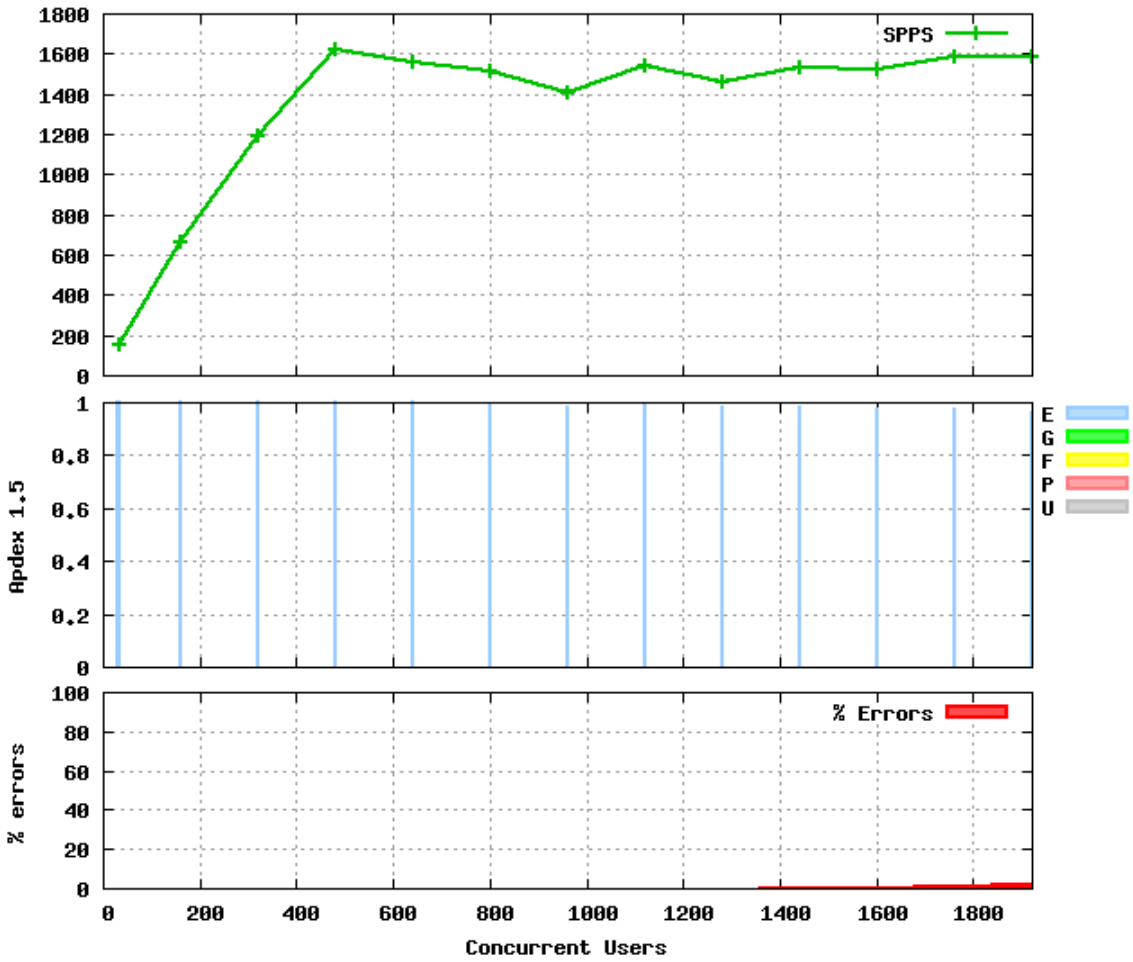
  With this modification in place, PaperSphere was able to serve over 400 pages per second (2 app servers setup). This is almost double the throughput achieved with default Passenger configuration (~210). We were also able to push the concurrency level to much higher limits with this optimization. Without this optimization, we started observing errors with around 180 concurrent users. But with the optimization in place we could push the system past 400 concurrent users without errors. To get the page load time to go beyond 500ms, we had to load PaperSphere up with over 300 concurrent users.



3.9 Testing with Even Larger Setups

With the intention of saturating our database node (which was still running on an m1.xlarge instance), we kept increasing the number of app servers in the test environment. We managed to get close to that point with 16 app servers behind ELB, all running on c1.xlarge EC2 instances. Once the number of app servers was increased past 8, we also had to add more Funkload processes to test the system, since 8 processes were not sufficient to drive the load. We ended up using a total of 16 Funkload processes on 4 m1.xlarge EC2 instances.

Results from the 16 app servers setup are shown below. PaperSphere serves around 1500 pages per second in this scenario (nearly 4x increase compared to the 4 node setup). The number of concurrent users has been pushed past 1200 without any errors. And the system keeps the page load time under 500ms as long as the number of concurrent users is less than 900.



We observed that the m1.xlarge instance hosting the database server was getting close to 100% CPU utilization under this test configuration. However, what was really limiting the database performance was the network throughput. EC2 cloud watch monitors indicated that the network links of the database node were getting saturated (~1Gbps).

4.0 Availability Plan and Testing

As mentioned earlier, PaperSphere consists of a web server component, an app server component and a database component. Out of these three, web server and app server components can be easily made highly available via replication. We can simply install multiple web server and app server instances on multiple physical or virtual machines, and set up a load balancer to distribute the traffic among them. In fact, this is exactly what we did during the scalability testing phase, where we set up multiple app server and web server instances, and used an ELB to load balance the incoming traffic.

Amazon ELB consists of a server health checker by which it is able to monitor the availability of backend servers, and automatically remove the failed servers from

the load balancing pool. In PaperSphere, we typically configure this health checker to ping the login page of PaperSphere once every 6 seconds. With a sufficiently redundant number of backend servers in place (distributed over multiple zones), this simple technique itself can improve the availability of PaperSphere by a great deal. However, this simple mechanism does not address the following availability limitations:

- Database failures: We use a single MySQL server instance as the backend data store. Any failures in this node will make PaperSphere unavailable, regardless of how many redundant web servers and app servers are present. Similarly an erroneous or accidental corruption of data can prove to be an availability nightmare.
- Datacenter (region) failures: A single ELB instance can only load balance the traffic among the servers in the same datacenter (e.g. us-east). Therefore a datacenter failure can potentially kill all the web server and app server instances making PaperSphere unavailable.
- Load balancer failures: Middle boxes like load balancers are just as susceptible to failure as regular backend servers. With just a single ELB to receive all the traffic, a load balancer failure will immediately make PaperSphere unavailable.

The rest of this section discusses potential remedies for the limitations identified above.

4.1 Improving Database Availability with a MySQL Cluster

Perhaps the simplest approach to make the PaperSphere database highly available is to use a cluster of database nodes instead of just one database server. MySQL supports several clustering configurations, which can be configured with moderate difficulty. We haven't tested this option yet, but we expect most database operations to slow down by some factor due to the cluster-wide replication and synchronization overhead that will be incurred. This is a case of trading in some of the performance for better availability.

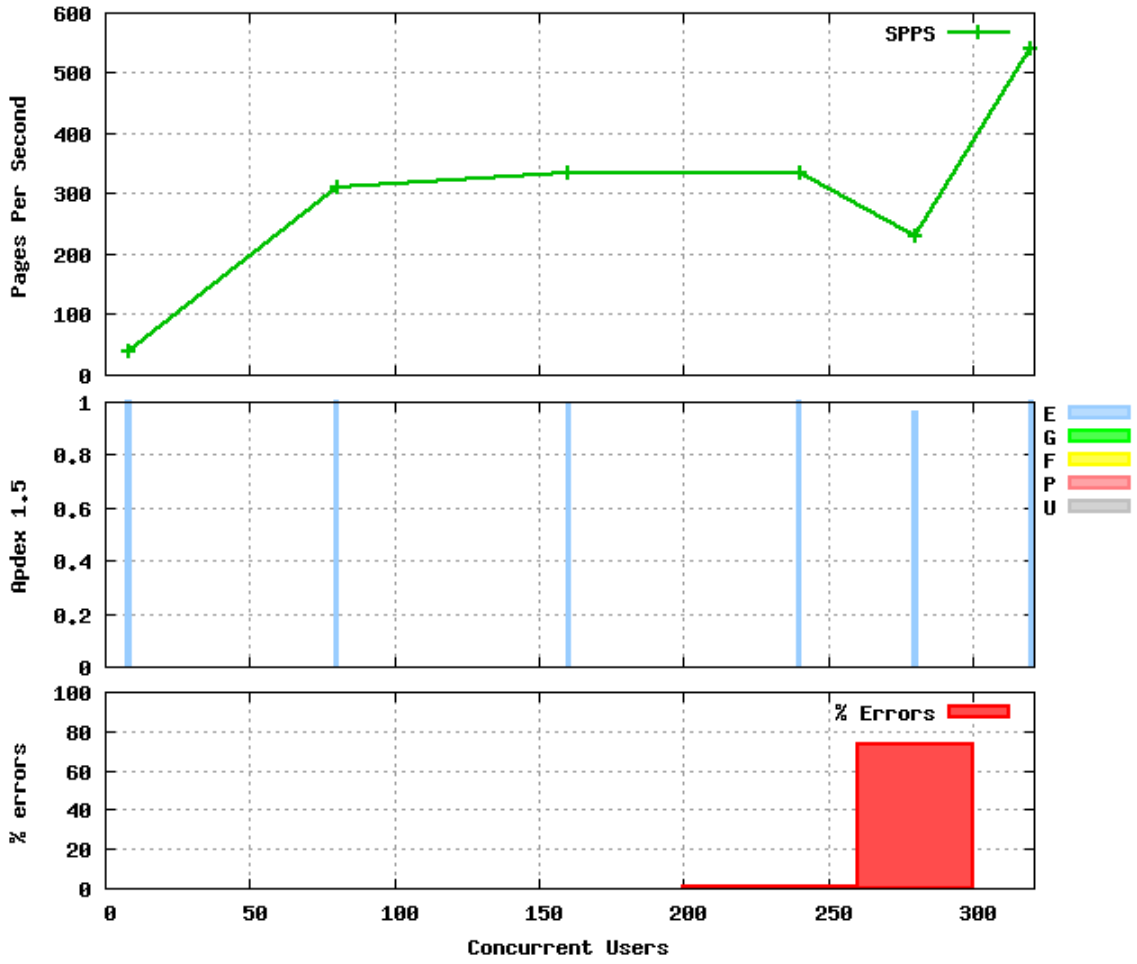
To further reduce the time-to-recover and increase availability, we can set up a job (e.g. cron), to periodically backup the database. We can use a tool like mysqldump for this purpose. This would allow us to restore the database quickly, after a total failure or an accidental corruption.

4.2 Improving Database Availability with Amazon RDS

RDS is a highly available Database-as-a-Service offering provided by Amazon. RDS provides a 99.95% availability guarantee, which equates to a downtime of 4.38 hours per year.

We managed to get PaperSphere working with RDS with minimal configuration changes. We used RDS to acquire a dbm1.xlarge MySQL server with 5GB storage capacity and multi-availability zone (multi-AZ) replication. With multi-AZ replication, RDS provisions PaperSphere with 2 database nodes, one acting as the primary. All writes are synchronously replicated to the secondary to ensure consistency. In the event of a long lasting failure, RDS automatically fails over to the secondary to keep the applications up and running. As per the documentation, the failover time is 3 to 6 minutes.

We ran a number of tests to evaluate the overhead of running PaperSphere on RDS with multi-AZ replication. Following plot is taken by running 2 app servers on c1.xlarge EC2 instances against RDS for a read-write workload. PaperSphere manages to serve over 300 pages per second in this setup without errors. This is only a marginal degradation in performance, compared to the previous results obtained by running a similar test without RDS. Therefore we conclude that using RDS with multi-AZ replication does not negatively impact the performance of PaperSphere.



The latency plots in the above test registers a maximum page loading time of 0.6ms, which further assures that using RDS does not cause a significant performance issue. Further, testing with 4 and 8 app servers against RDS showed that vertical scaling speed up is still near linear. With 4 app servers PaperSphere served over 500 pages per second and with 8 app servers it achieved a throughput over 1000 pages per second.

We also identify the following additional benefits of running PaperSphere with RDS:

- Automated periodic backups
- Ability to create snapshots, move them across regions (datacenters) and restore them within minutes
- Automatic security updates and patches
- Powerful monitoring features with RDS cloud watch

4.3 Cross Datacenter Failover with Route 53

In order to protect PaperSphere against ELB failures and some datacenter failures, we tested the possibility of running PaperSphere on multiple regions with DNS failover. Here we explain one of the experiments we conducted in detail. The test environment used is as follows:

US-East region (Virginia):

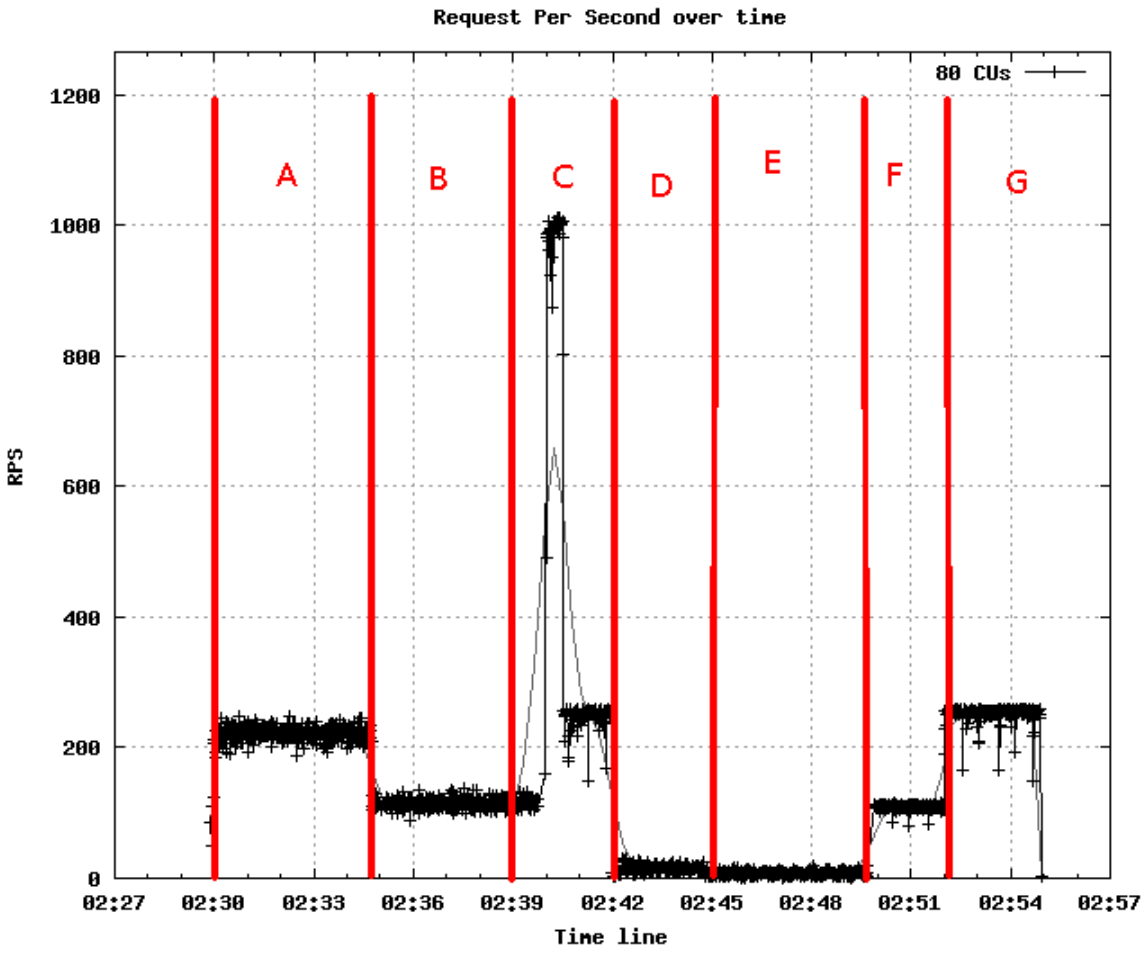
- 2 app servers on c1.xlarge EC2 instances fronted by an ELB

US-West region (CA):

- 2 app servers on c1.xlarge EC2 instances fronted by an ELB
- MySQL server on dbm1.xlarge RDS instance (app servers in both regions are configured to use this server as the data store)

Then we configured Route 53 DNS failover to use the US-West ELB instance as the primary endpoint and the US-East ELB as the secondary endpoint, thus effectively putting the two datacenters in an active-passive configuration. Both ELBs were configured to monitor backend server health, and Route 53 was configured to monitor the health of ELBs.

With the above test environment in place, we run Funkload against the system with a fixed number of concurrent users over a period of 30 minutes. Funkload was run from 2 EC2 instances in a completely different region (US-West/Oregon). Then we proceed to kill our app servers in 5-minute intervals (starting from the US-West region and moving on to US-East). We measure the number of requests served by PaperSphere per second and plot it over time. The final result is shown below. Note that the plot has been divided into 7 regions.



A. Traffic is handled by the 2 app servers in US-West.
B. Traffic is handled by a single app server in US-West. Throughput drops because there's half the server capacity to handle the same volume of traffic.
C. No app servers are available in US-West anymore. Route-53 failover kicks in which takes around 3 minutes. Errors are returned to the client in the meantime, which is registered as a spike in throughput.
D. Traffic is handled by the 2 app servers in US-East. Throughput is much less because the client-server (Oregon-Virginia) and server-DB (Virginia-CA) latency is high.
E. Traffic is handled by a single app server in US-East. Throughput drops even further.
F. No app servers are available in either region. Client starts receiving errors and the throughput increases.
G. Route-53 fails over again to the US-West region, which is closer to the client (Oregon). But still no servers are available in either region, hence the client continues to receive errors.

Based on our observations, we can conclude that we can perform a cross datacenter failover within 3 minutes, in the event an ELB or an entire datacenter goes down.

However we cannot handle all datacenter failures this way, since when the datacenter that hosts the RDS instances fail, PaperSphere will become unavailable. For now we rely on the high availability guarantee provided by RDS to deal with this type of failures. If we were to handle such outages ourselves, we would have to change the architecture of PaperSphere to use a high available data store that supports sharding or some form of cross datacenter data replication.

We also tested Route-53 failover with an active-active setup, but the results were not as clear and conclusive as in the case of active-passive setup. More tests and analysis is needed to do a proper comparison between these different failover configurations.

5.0 Summary and Conclusion

PaperSphere is a social networking application that enables users to browse and share published academic material. The system is implemented in Rails and uses an RDBMS as the backend data store. After subjecting PaperSphere to extensive scalability and availability testing we learnt the following:

- PaperSphere scales linearly as we add more servers (horizontal scaling).
- PaperSphere is mostly CPU bound, and hence benefits greatly from vertical scaling (especially from more CPU).
- Database optimizations go a long way. Adding indexes where appropriate, using more efficient SQL queries, using a larger cache and pre-warming the database have significant performance benefits in the long run.
- Rails ORM layer has several deficiencies such as making inefficient SQL queries, making too many COUNT(*) queries and making iterative SQL queries. Application code has to be optimized carefully to circumvent these issues.
- Ruby-on-Rails is very easy to develop with (greatly improves developer productivity). But it comes at the cost of resource utilization.
- Systems based on a stateless, shared-nothing architecture can be easily made highly scalable and highly available via simple replication of servers.
- DNS failover can be a simple yet very useful feature when it comes to dealing with middle box failures and certain datacenter failures.
- Handling all sorts of datacenter failures require a different data store architecture with sharding and cross datacenter replication (possibly with distributed consensus).