CS219

NOITCUA-Reverse Auction System

By:
Praveen Alevoor (304402148)
Shashank Kothapalli (504433926)
Naren Nagarajappa (004414529)
Akshay Vangari (004435621)

Motivation

Recently there has been a rise of “on demand” service oriented web applications. Take for example, Uber, which provides on demand rides from one place to another and is valued at $40 billion. By using location, rating, ride type, and many more factors, Uber is able to satisfy the needs of a consumer by matching him or her to an appropriate producer. Similarly, we noticed that college students are utilizing many social media platforms (Facebook, Google+, etc.) for requesting various services (tutoring, car fixing, plumbing, tech repair, etc.). However, these social media applications lack the necessary features needed in order to truly benefit the producer and consumer of these services. Our web application, Noitcua, is a web application which provides users the capability to request any kind of service they wish along with a preferred price. Businesses/individuals who are looking to provide these services can bid on these posts with their bid price and time frame with which they can get the job done. By viewing a user’s rating, location, and past work, the user can accept a bid which he or she feels most appropriate.

Introduction

Noitcua provides following main features:

- Request for services
- Bid on services
- A notification system, which tells users of any new bids that they received/won.
- A 5-star rating review system which allows users to review on service providers and service consumers.
- A real time chat system where users can chat with other and negotiate the contracts if necessary.
- A location based search system, which allows users to search for services near their location.

Our application was developed with Ruby on Rails and a SQL based database (MySQL) engine. The frontend of the application uses jquery and bootstrap for dynamic and appropriate user interface/experience for the user. The rest of the report focusses on the development of the application and how it was scaled to 100s of user requests per second.

Development

Pivotal tracker helped us follow agile development framework, where the tasks of the application were divided into sprints and each task was completed in those sprints. Tasks not completed in the specified sprint, were automatically moved into a backlog, to complete

in future sprints. This helped us manage our work smoothly and focus on tasks in the sprint rather than the application as a whole. However, we found pivotal tracker to be more complicated than other famous applications for agile like JIRA.

Rails being a test driven development framework helped us automatically create test cases for the classes we developed, which made sure that there was ample functional testing for the code. However, our team did not focus much on unit tests and rather focussed on developing as many features as possible for the system. Travis ci being a continuous integration service for building and testing projects, helped us in making sure that the code was bug free and all tests were passing every time we pushed new code.

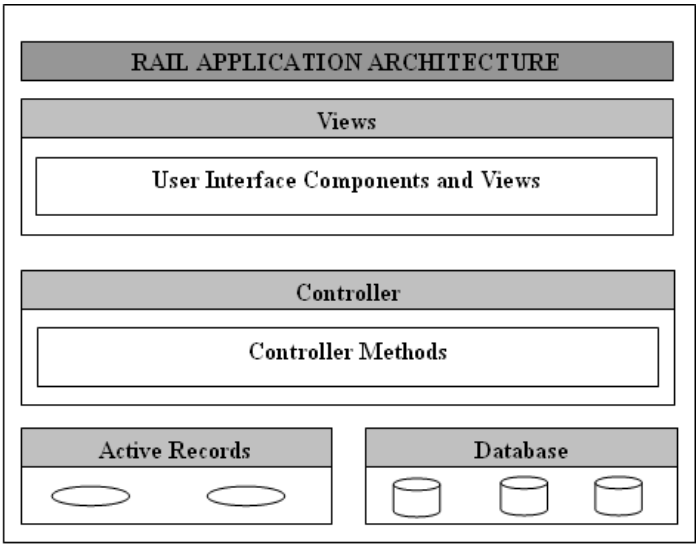
Source control management software Git is another major application which helped us in our development. Our application crashed many times in master and had to be reverted back to an older version, which is made possible by using Git. It also helped every developer of the application to focus on his own feature and helped us easily coordinate and merge our features to the application seamlessly.

Architecture

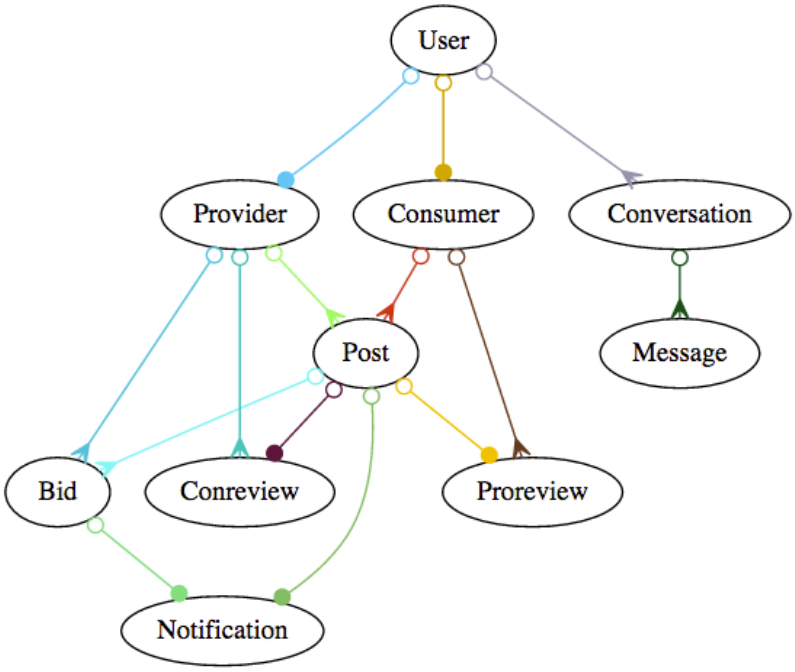
Rails helps us create easy MVC (Model View Controller) and Restful based applications, which helped us in easy management of the application. We tried to follow the BNF forms for databases as much as possible, but paved way at some places for easy development. Rails models helps a lot in easy fetching of the data from every table. The advantage of not writing queries and not worrying about sql injection is a big plus in Rails.

I: MVC Architecture

Model–view–controller (**MVC**) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, namely model view and controller. Model maintains the relationship between Object and Database and handles validation, association, transactions, and more. Controller is the facility within the application that directs traffic, on the one hand querying the models for specific data, and on the other hand organizing that data. And View presents the data in a particular format, triggered by a controller's decision to present the data.

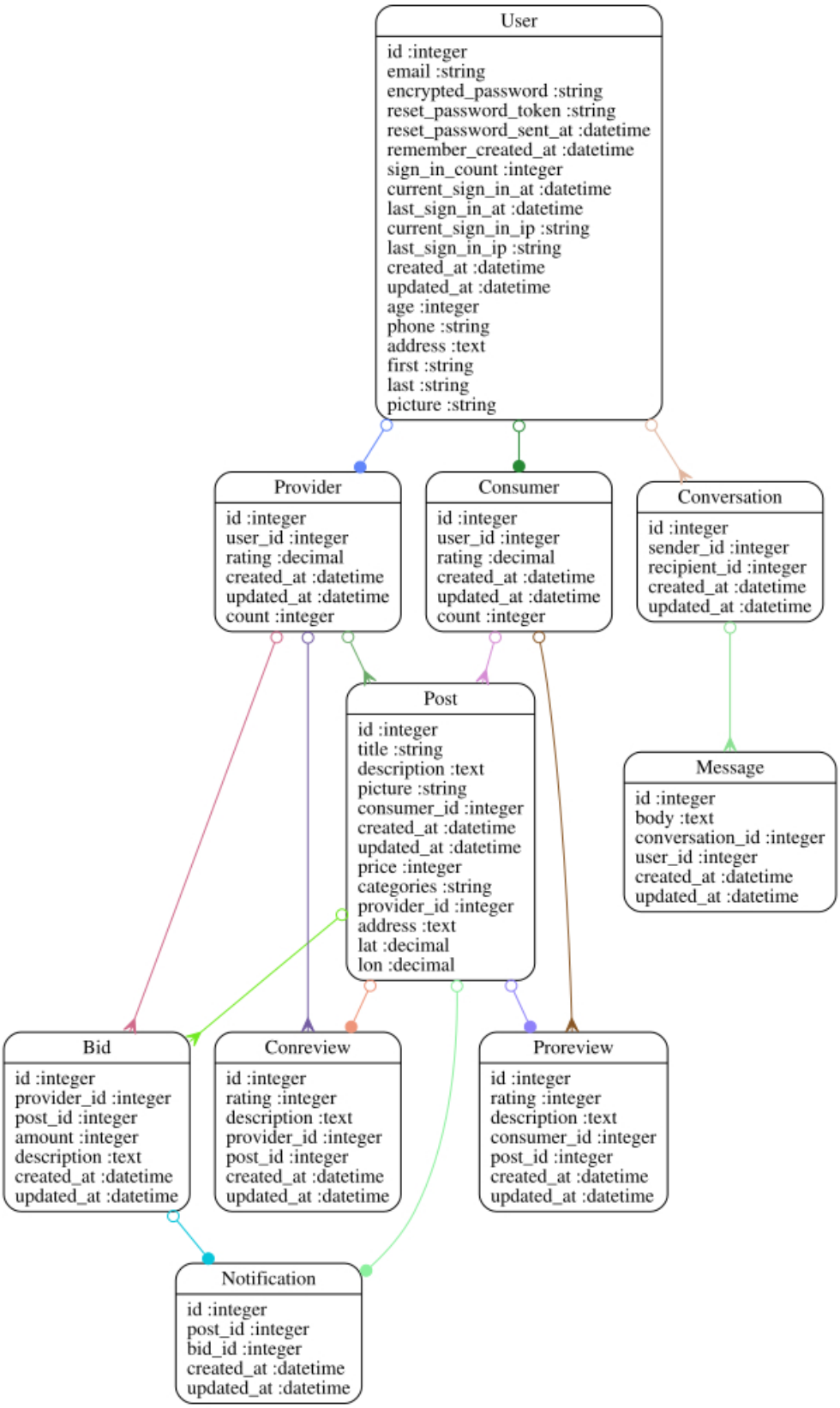


II: Database Architecture



Model	Description
User	This is the table which contains the basic profile information about the users like their name, age, email. Any of the fields from this table can be used to login.
Provider	This table extends the user table and acts like the junction table connecting users to their provider side features
Consumer	This table extends the user table and acts like the junction table connecting users to their consumer side features
Conversation/Message	Each user pair has a conversation between them and each conversation has many messages
Post	Users create multiple posts requesting the service
Bid	Each post has multiple bids, each one belonging to a single user
ConReview/ProReview	These 2 tables hold the reviews of the user, both as a consumer and a provider
Notification	This is used to notify the user when there is a new bid on his post or when the user wins the bidding on a post

Figure: Database UML Design



III: Important Gems

Carrierwave: Since a user may upload an image for his or her profile or posts, we had to make image upload as a feature. The most popular choice for file uploads is the Paperclip; however, for our needs, CarrierWave is more flexible. It is based on Rack, which means it works with Rails, Sinatra, and other Rack-based Ruby applications, and it supports a variety of ORMs including ActiveRecord, DataMapper, Mongoid. Another significant difference between CarrierWave and Paperclip is that CarrierWave keeps everything in a separate Uploader class so that all of the file attachments and the processing logic belongs in one place. This way file attachments don't end up mixed with the application's model classes.

Turbolinks: Turbolinks makes following links in your web application faster. Instead of letting the browser recompile the JavaScript and CSS between each page change, it keeps the current page instance alive and replaces only the body (or parts of) and the title in the head.

Private Pub with Faye and Thin Server: We wanted to enable the users to chat with each other in real time and hence needed a way to enable push notifications. We used a gem called Private_pub which is a Ruby gem for use with Rails to publish and subscribe to messages through Faye. Faye is a publish-subscribe messaging system based on the **Bayeux** protocol. It provides message servers for **Node.js** and **Ruby**, and clients for use on the server and in all major web browsers. Private_pub allows you to easily provide real-time updates through an open socket without tying up a Rails process. All channels are private so users can only listen to events you subscribe them to.

Sunspot Solr: Solr is a server based on Lucene which helps index the data in the server. It further provides different search interfaces which can be directly used in rails models. We used EdgeNGram Indexer which helped in partial searching. Solr also provided support for location based searching, which helped us to provide feature of location based searching with the use google maps.

Load Testing and Scaling

I: Initial Configuration

For load testing of the application we used Tsung. The purpose of Tsung is to simulate users in order to test the scalability and performance of IP based client/server applications. It can do load testing and stress testing of the application. Tsung load testing works based on a configuration file provided to it. Within the configuration file we mention the host name where the application is accessible, the number of users to simulate, and the actions the user would perform within the application. Tsung requires paths relative to the host name where different actions are performed. We identified crucial actions that a user would perform after logging into our application and then created a configuration file with the relative paths for the user to perform these actions in our application. We created multiple sessions of different types. For example in one of the session, a user would sign in, request for a new service, check all the services requested, visit a particular post of service and then bid for that service and log out.

The crucial actions that a user would perform include the following:

- User Sign up
- Log into the system
- See all the posts
- Access a post
- Create a new post
- Bid for a new Post
- Visiting different pages of posts

The arrival rate of the users in different phases are shown below. We use the same test file for all the tests performed in this project.

Phase	1	2	3	4	5	6	7	8	9	10	11	12
Users/sec	1	1.5	2	4	6	10	16	20	25	35	45	55

II: Vertical Scaling

Table: AWS Instances

Model	vCPU	Mem (GiB)	SSD Storage (GB)
m3.large	2	7.5	1 x 32
m3.xlarge	4	15	2 x 40
m3.2xlarge	8	30	2 x 80
t2.micro	1	1	EBS-Only

We hosted the application of Amazon Web Services (AWS). For the initial data in the database, we created a seed file of 1000 users and 4000 posts. This part of the scaling process was to just use a single server on AWS and scale it vertically i.e. to increase the capacity of the single server with better hardware. We ran the load test on 4 types of EC2 instances starting from t1.micro, m3.large, m3.xlarge and m3.2xlarge. The results of these experiments are shown below:

t1.micro: A t1.micro instance provides spiky CPU resources for workloads that have a differentiated CPU usage profile**.** They provide 1 Virtual CPU with 1GB of memory. This instance type is used for testing purposes and we started the load testing with this to make sure everything was working. As expected, the server couldn't handle the load at all.

The mean page access time was a high of 4.58s and the mean request time was 4.21s. As it can be seen from the mean transaction and page duration graph, the server starts breaking from the first phase itself when there is 1 user arriving every second.

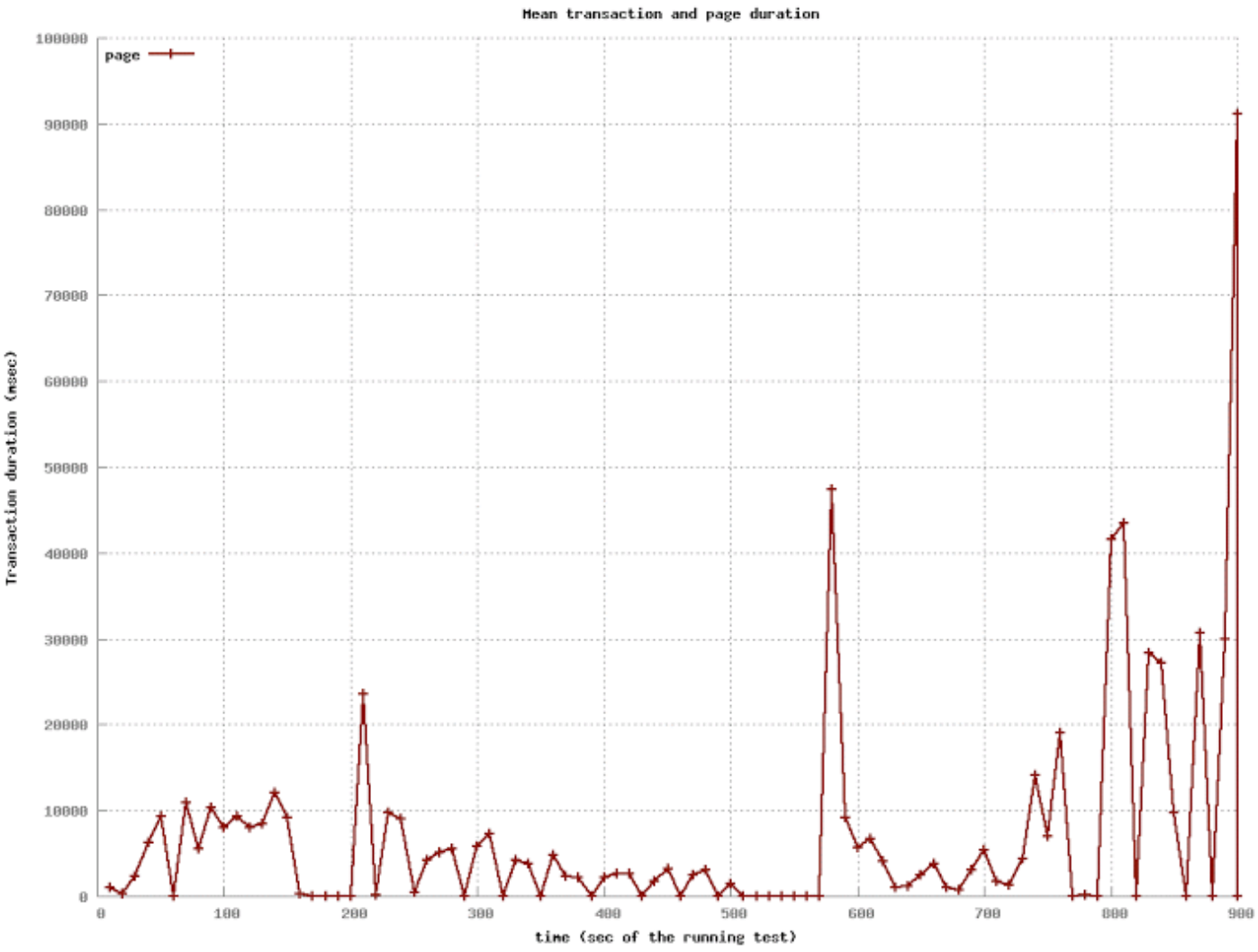


Figure: Transaction/Pages Response Time

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	1.9/sec	34

302	2/sec	143
404	1.2/sec	48
500	2.1/sec	51
502	185.8/sec	17671
503	326/sec	38515

- As can be seen from the table, there a lot of 502 and 503 responses as well. This shows that the simple micro instance couldn't handle the load from the start. The reason for the 404 responses are explained later in the section.

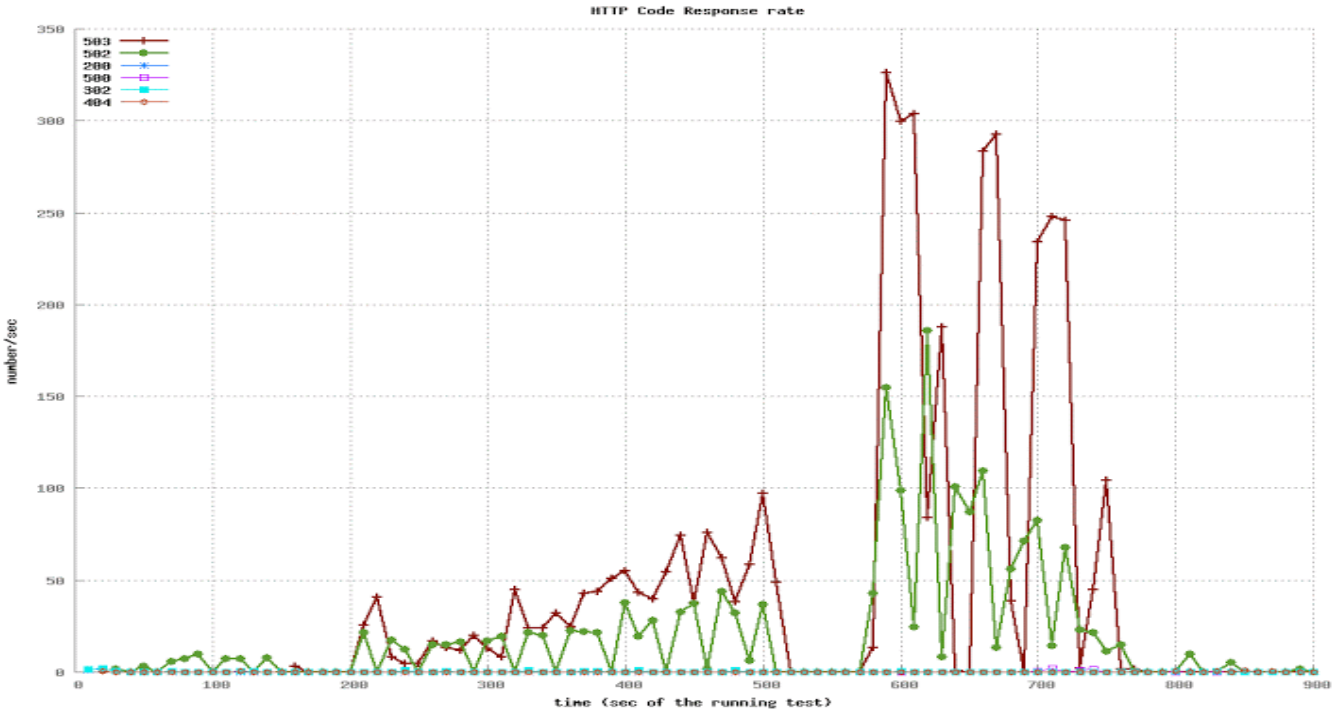


Figure: HTTP Return Code Status (Rate)

- The above graph shows the number of each type of HTTP response over time. 503 errors occur somewhere around 100s, when there are 1.5 users every second.

Next we use the M3 series, which has High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors, SSD-based instance storage for fast I/O performance, Balance of compute, memory, and network resources. They are better than t1/t2 instances.

m3.large : M3 series provides a balance of compute, memory, and network resources, and it is a good choice for many applications. The particular configuration of the instance type is shown in the table above. This performed a lot better than micro instance used before. The mean page access time was a high 1.30s and the mean request time was 1.19s. This was because most of them were 503.

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	2.8/sec	1272
302	10.5/sec	3966
404	5.3/sec	524
503	247.5/sec	50715

- Though the number of 503 errors has increased significantly, the 500 and 502 error have gone away completely. Also, the highest rate of 503 has reduced by a significant amount.

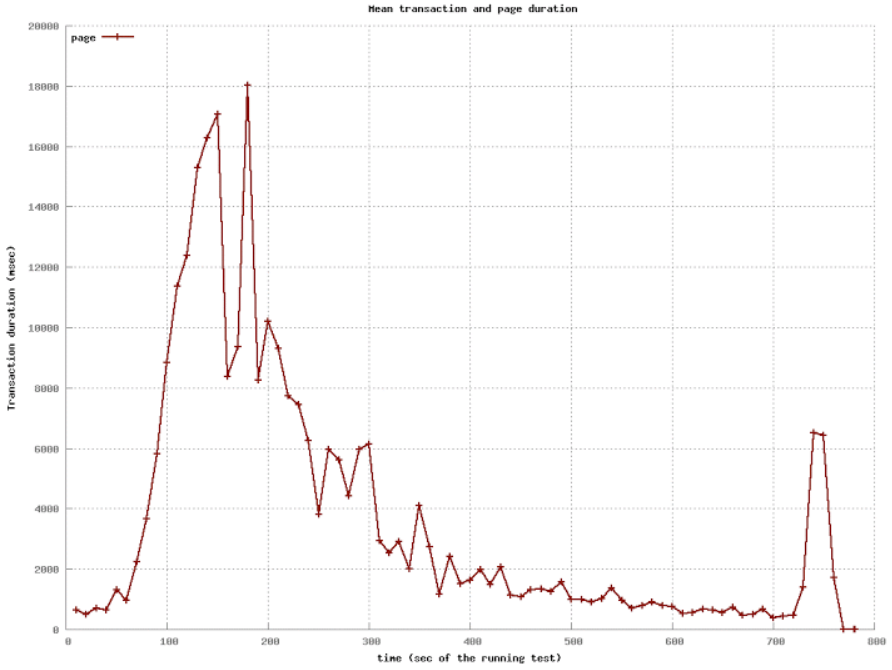


Figure: Transaction/Pages Response Time

This server is able to handle upto 1.5 users per second and starts deteriorating after that. After it reaches a maximum, the transaction time starts decreasing because most of the replies would 503s, as can be seen from the HTTP status code graph below.

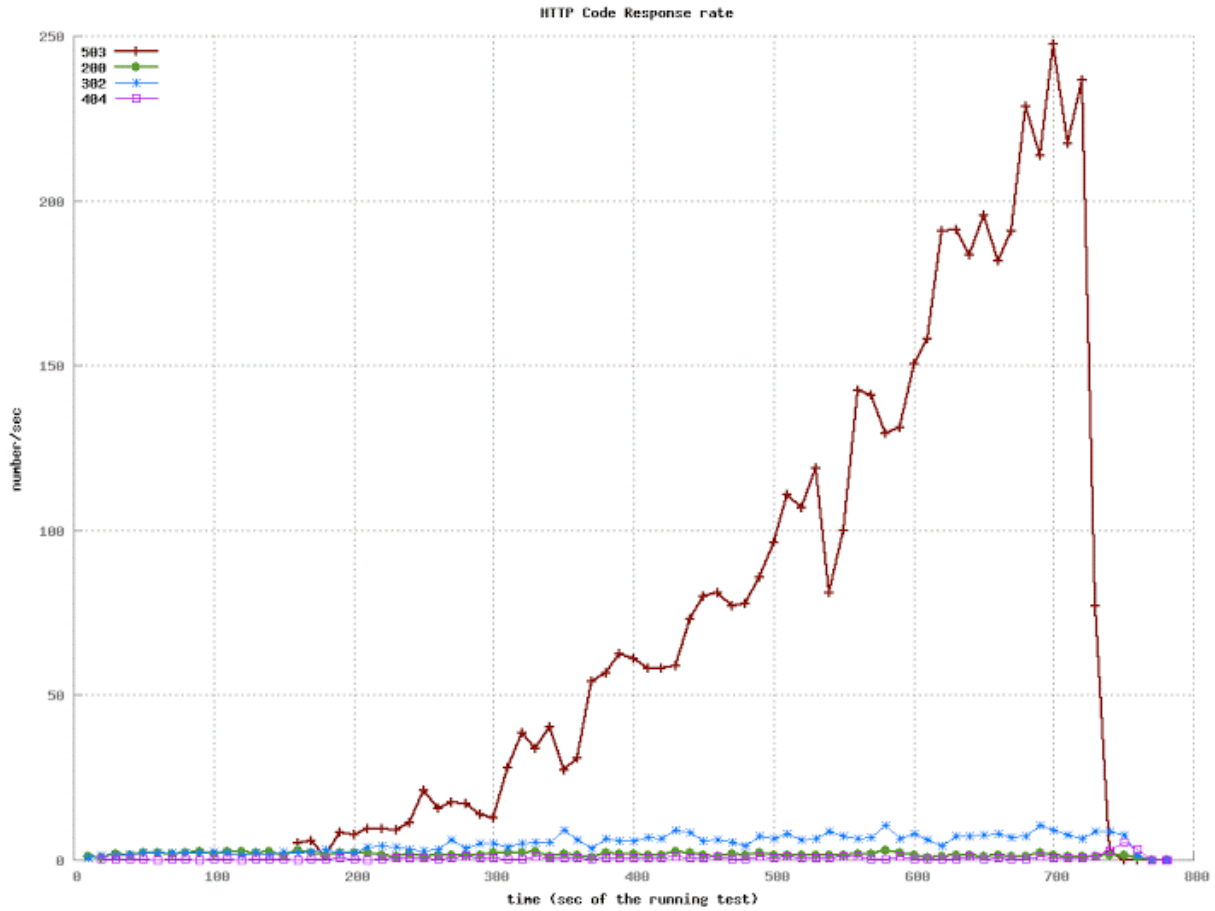


Figure: HTTP Return Code Status (Rate)

m3.xlarge : This again, belongs to M3 series which provides a balance of computation, memory, and network resources, and it is a good choice for many applications. This particular instance has double the number of resources compared to m3.large as can be seen from the table. Hence as expected, it performed a little better than it though not by much. The mean page access time was a high 1.05s and the mean request time was 0.97s.

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	7.2/sec	2639
302	17.9/sec	6886
404	8.1/sec	904
503	221.2/sec	45600

- The number of 503 HTTP responses reduced by around 5000 and the Highest rate also decreased by a small amount. The server is able to handle 1.5 users per second easily and starts deteriorating when 2 users are arriving per second. Around 250 seconds when there are 2 users per second arriving, the 503 starts increasing.

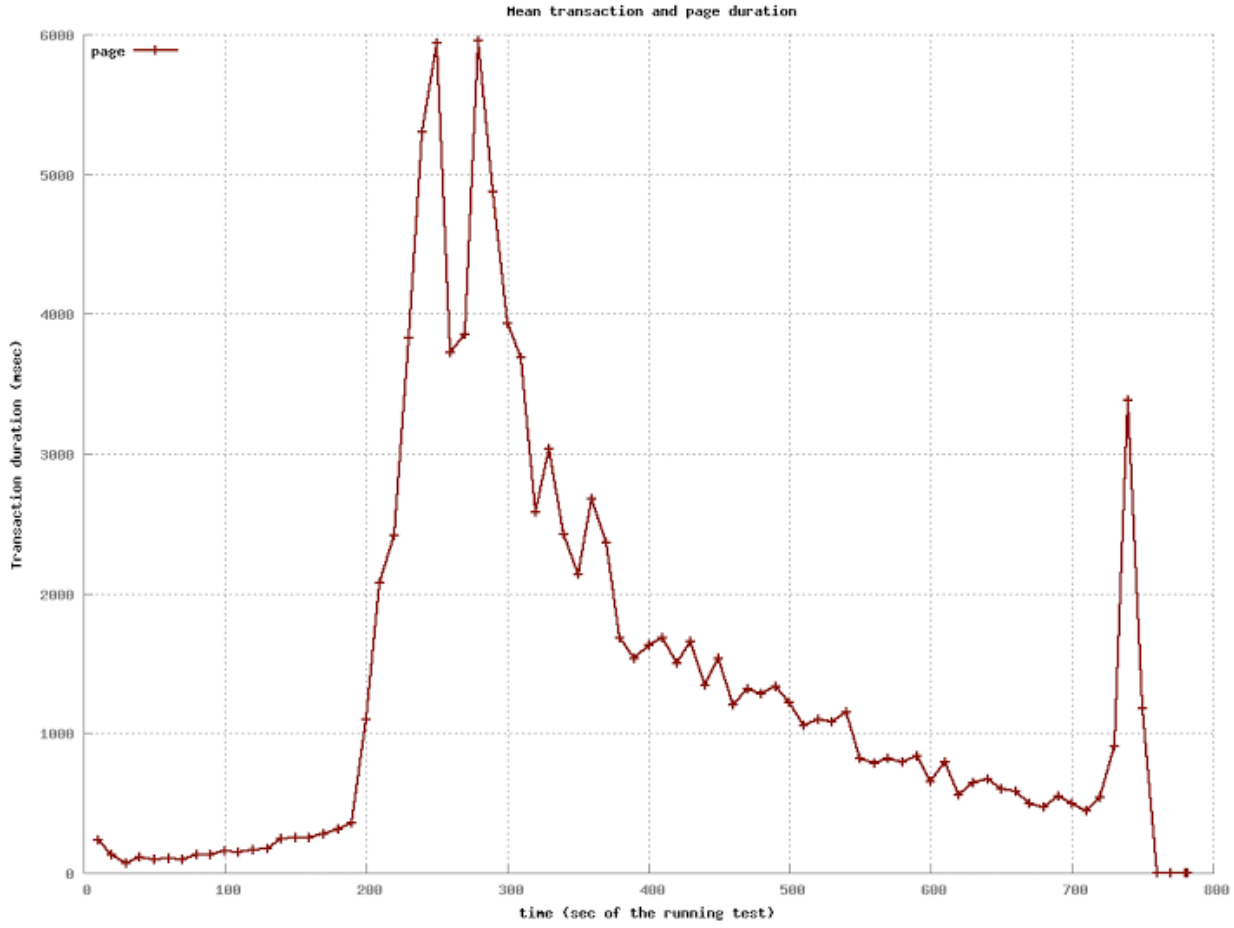


Figure: Transaction/Pages Response Time

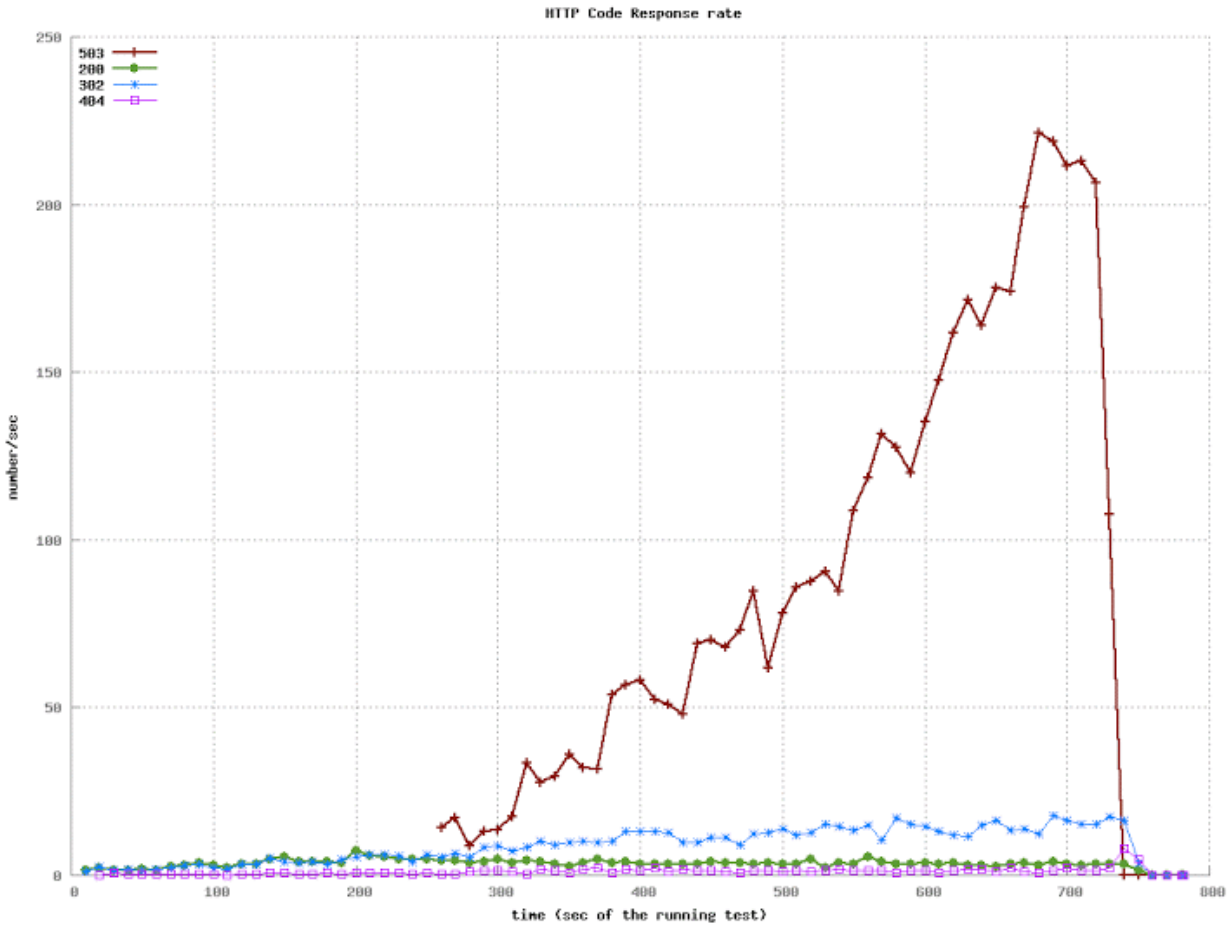


Figure: HTTP Return Code Status (Rate)

m3.2xlarge : This again, belongs to M3 series which provides a balance of compute, memory, and network resources, and it is a good choice for many applications. This particular instance has double the number of resources compared to m3.large as can be seen from the table. But we can hardly notice any improvement in the performance of the site from m3.xlarge. The number of 503 errors reduced by a mere 2000 and it still couldn't handle 2 users arriving per second. Hence the graphs are not included in this section. So there was technically very little improvement from m3.xlarge. The mean page access time was a high 1.03s and the mean request time was 0.95s.

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	9/sec	3255
302	24.9/sec	7770

404	9.1/sec	1041
503	217.6/sec	43693

As can be seen from the errors, there are 503s as expected. But there are also 404s. This was due to the way we wrote the test.xml file for tsung. We pick a random post to bid on in one of the sessions and during multiple phases. If the same user picks the same post to bid on, the request to bid would be redirected to editing the bid URL. But while creating a new bid, we keep the post id in the session and while editing it, we don't have that variable in the session. Hence, due to this redirection, the 404 errors were seen. From this point on, we changed the logic at backend with respect to the session and made sure the 404s did not exist anymore.

Also another problem with our backend was that whenever a new bid was placed, a notification was added to the notification table and it is removed only after the user checks his notifications. Since tsung tests create the bids and the notifications but it doesn't emulate the user viewing those posts, the notifications table was growing very fast. A side effect of this was that, we were checking the notifications of a particular user in application controller before any action was performed and hence the processing time for every request for huge. This was one of the main reasons we see so many 503 HTTP status returned. Thus, for testing purposes, we removed the notification checking logic in the backend and performed the tests and as expected the results started getting better.

III: Horizontal Scaling

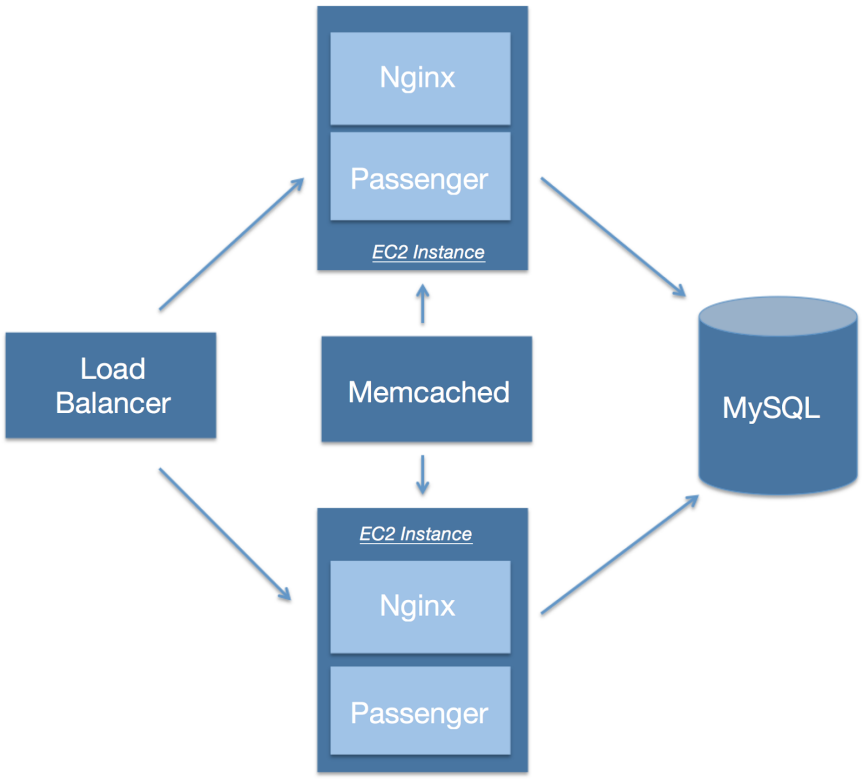


Figure: Amazon EC2 Setup

Since one single server could not handle 1000s of users, we now try to scale the system horizontally. Horizontal scaling is the use of multiple app servers with a load balancer which divides the load across these app servers. We further use a remote database server to which all the app servers point to, and all the database actions happen remotely. One major change in the configuration file is the removal of the search feature. Since we used Solr server for building a search index and used it for searching, this created a issue during horizontal scaling. We could not host the solr server in a remote server like the remote database server, and hence ignored the search feature in the testing.

We tried different types of configurations for this. There are different types of improvements we can do here. First is to try increasing the number of HTTP server instances(Nginx). Second is to try different types of machines like t1 micro, m3 large and m32x large and finally to see if the load on the database server is high and see if vertical scaling of the database server improves the system.

Since we already know which is the best instance type from vertical scaling (i.e. m3.2xlarge), we will be having multiple instances of m3.2xlarge type and performing horizontal scaling. We used m3.2xlarge instances for the HTTP servers and m3.large instance for the database server, unless specified otherwise.

Also we use a bigger seed file of 2,500 users which increases the overall database size from around 5000 records to 15000 records. And as mentioned above we also removed the notifications for load testing, as for every action the entire notifications table was being read which caused a huge amount of 503 errors.

4 Instances of HTTP server and 1 instance of database server :

The mean page and request time for this configuration was seen to be around 2.36 and 2.01 seconds. This is high compared to a single vertical m3.2x large configuration, but this is because more requests are served overall in the load balancing configuration.

We also see that the errors have reduced by a huge extent compared to vertical scaling. We further try to identify the best possible configuration for load balancing and hence increase the number of instances. The server handles up to 16 users per second and then starts performing badly.

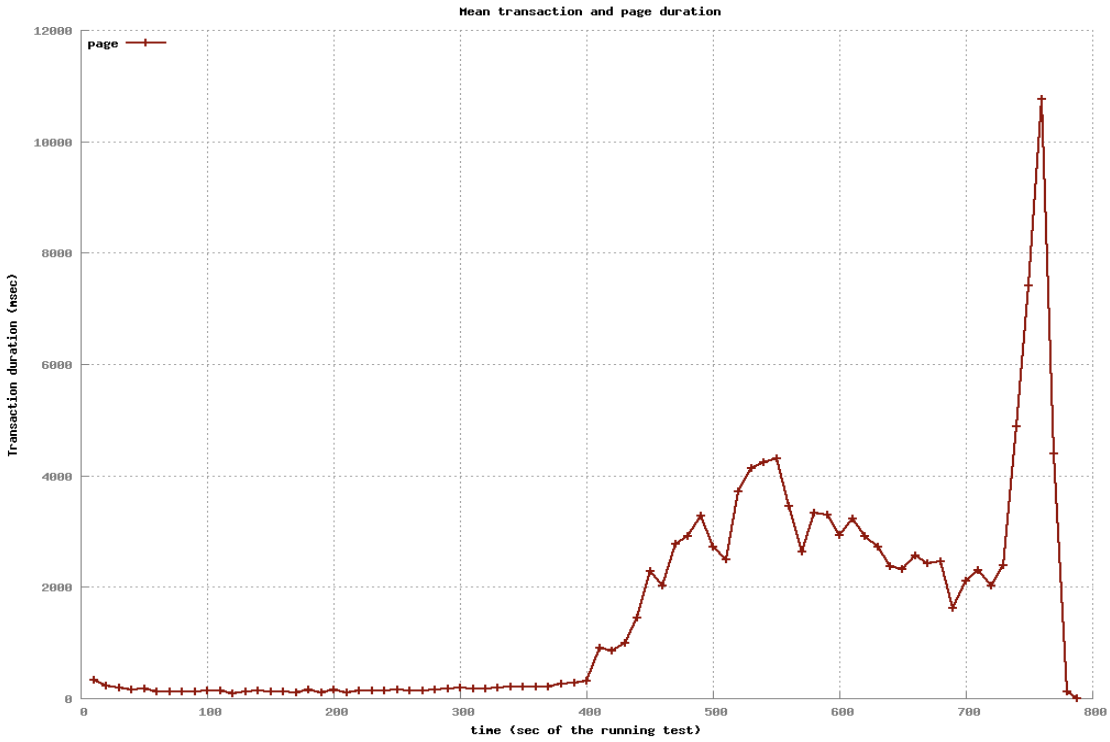


Figure: Mean Transaction Time

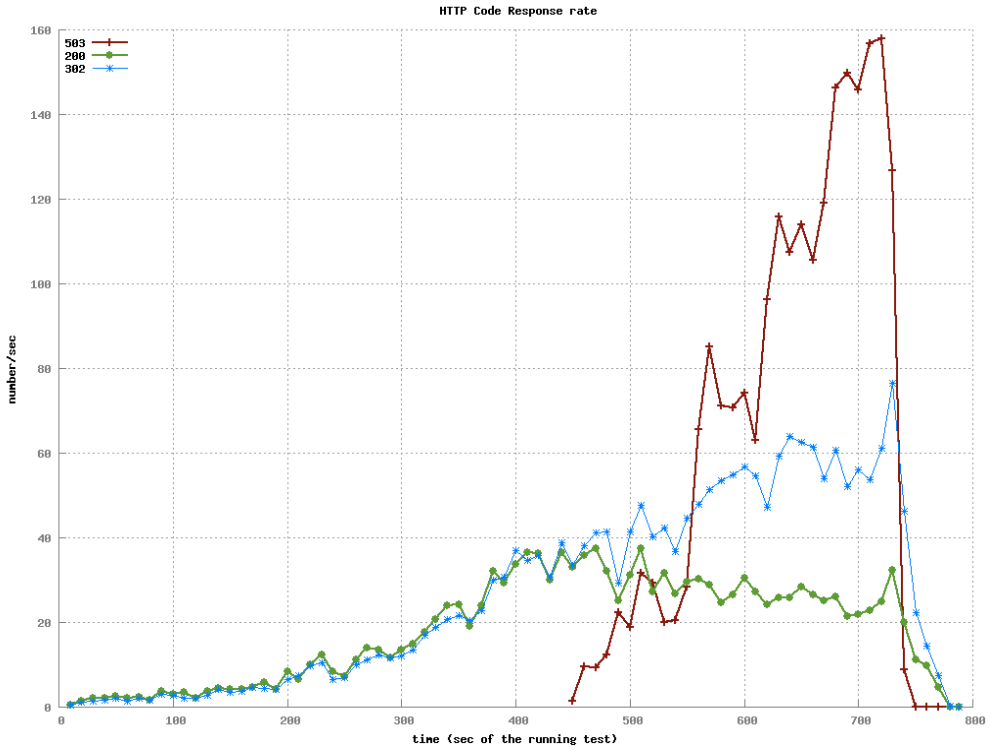


Figure: HTTP Status Codes (Errors start around 450th second)

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	37.5/sec	14196
302	76.6/sec	20832
503	157.9/sec	21831

6 Instances of HTTP server and 1 instance of database server :

The mean page and request time for this configuration was seen to be around 3.24 and 2.75 seconds. There is a increase in the average time, this may be because there is more redirection because of load balancer and with more instances the average time is increasing. The amount of 503 errors reduced by around 2000 from a 4 instance of HTTP server and 1 database server. We see that the server starts deteriorating after around 450 seconds. The server handles approximately up to 20 users per second.

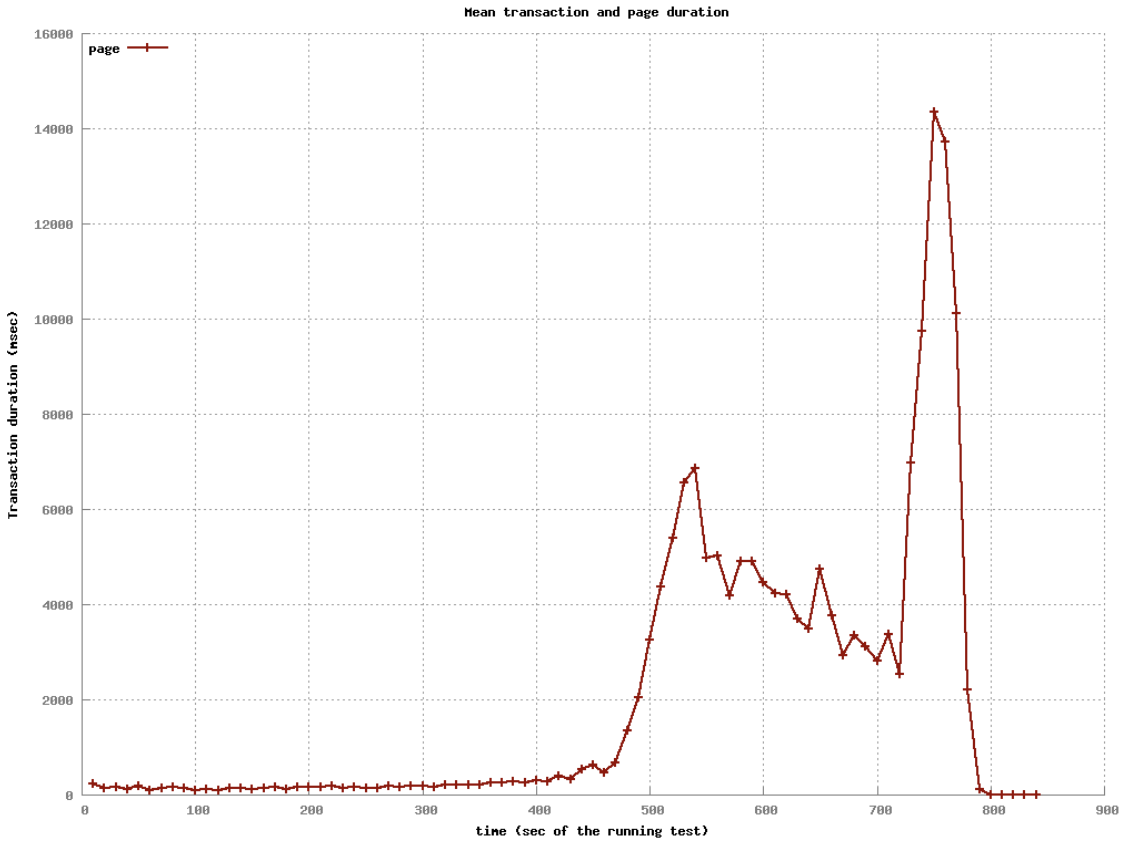


Figure: Mean Transaction Time and Page Duration

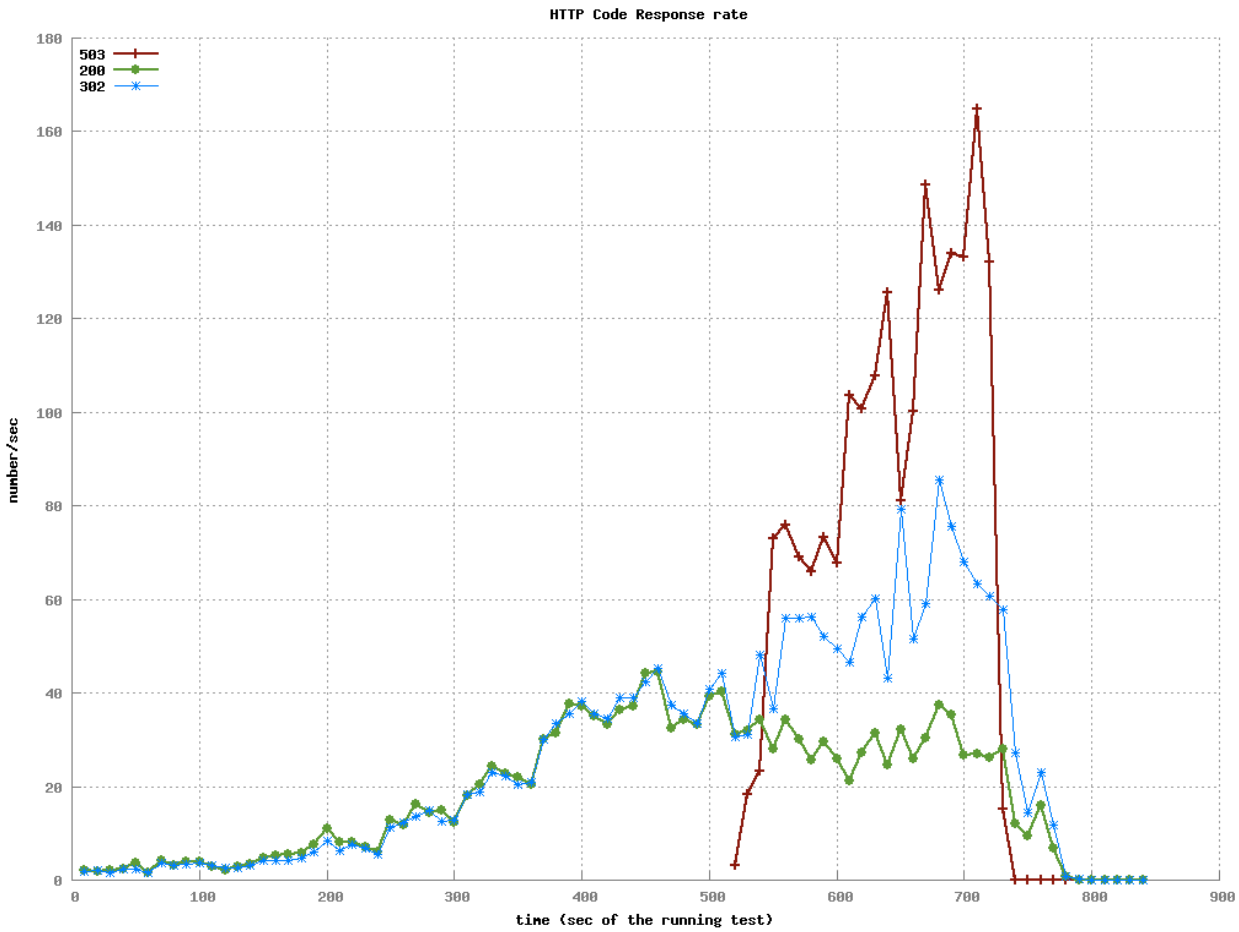


Figure: HTTP Status Codes

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	44.4/sec	15547
302	85.6/sec	21625
503	164.9/sec	19429

8 Instances of HTTP server and 1 instance of database server :

The mean page and request time for this configuration was seen to be around 4.31 and 3.68 seconds. Here we see that the mean transaction time and the errors in 6 instances

and 8 instances are almost the same. This again handles up to 20 users per second. This indicates there is not much improvement in the increase in configuration.

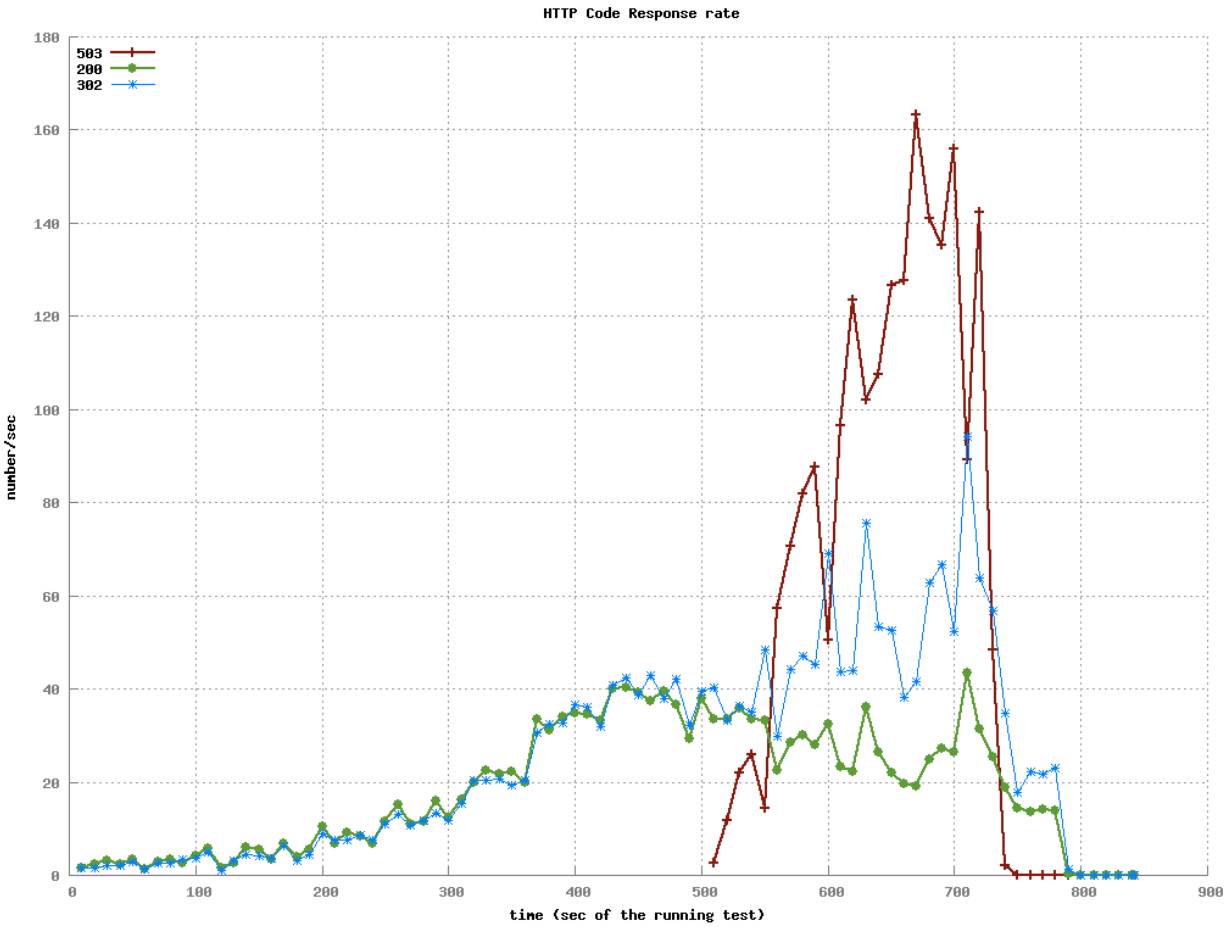


Figure: HTTP Status Codes

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	43.4/sec	15456
302	94.1/sec	21000
503	163.2/sec	19860

At this point, we know that with a single instance of database server, the best option we have is 6 instances because the number of 503 responses were almost the same with 6 and 8 instances. So with 2 more extra instances, we did not gain anything. Further in some

runs we saw that 500 errors were there with 8 instances, which is basically because of the increased load in the database server. Hence, we decided to take the 6 instance cases and increase the size of the database server to the best possible available instance type, which is m3.2xlarge and see if the performance increases.

Database Vertical Scaling

Since we see that increasing the number of app instances did not help with performance. We increase the database configuration and try to see if there is increase in performance. We saw that transaction time of both m3.large and m3.2xlarge database were almost the same. However, there was a decrease in the errors in m3.2x large from 20 thousand errors to around 15 thousand errors, which is a good increase in performance. Further we see that the errors start later at around 530th second compared to a m3.large database for which it starts at around 500th second.

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	45.7/sec	17343
302	84.8/sec	23165
503	140.9/sec	15648

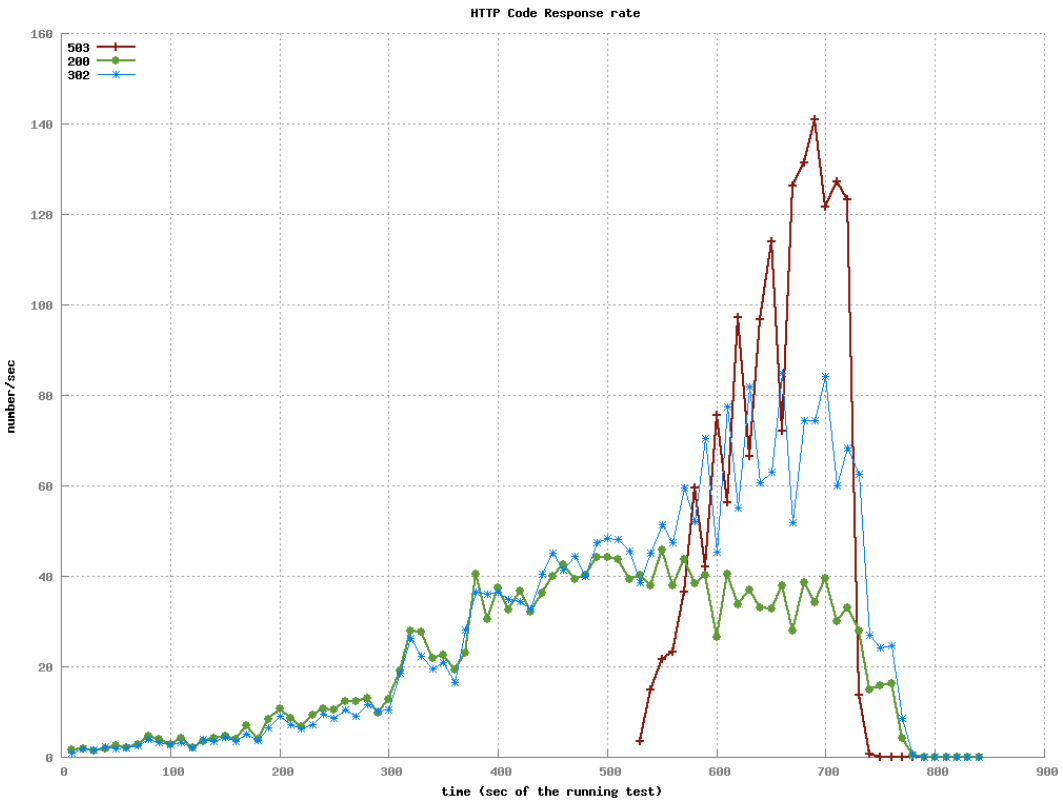


Figure: HTTP Code Response Rate

Since m3.2xlarge database performs way better, we used this database configuration with 6 m3.2xlarge HTTP servers for further load testing with optimization and caching.

Optimizations and Caching

We used a profiler named rack profiler to see how much time every request is taking within the application. The profiler tells every information about the request. It mentions the amount of time taken per query, number of queries, rendering time and the processing time. This helped us to identify blocks in our system which were not performing well.

I: Pagination

One thing we noticed is that the number of posts increased our login page and the user posts page took a huge amount of time for loading. Since a user would not want to see all the posts at a time we implemented paging, where a user would only be able to see 30 posts at a time. We achieved pagination using a library called will_paginate, which is highly optimized for database queries that return a lot of results. This improved the performance to a huge extent. From the profiler we saw improvements of over 2 seconds in the initial response time. This was also noticed in load testing where a server with pagination performed very well compared to no pagination.

II: Javascript Optimizations

We saw that the initial load time of the application was very high. So we minified the javascript and the css files. Also we removed any unwanted javascript files as they were going into response as they were in assets folder.

III: SQL Optimizations

In this phase of optimizations, we revisited some of the intensive queries we had written in the app. In the first step of optimization,we realized that while displaying the pins of the posts, we had a complex sql query in the view. We modified the view then and moved the logic into the controller. In the second step, we noticed another complex query when notifying the user of new bids on his post and when a user wins a bid on the post. Instead of searching for the bids belonging a post and then searching if it's new or instead of searching for the posts the user has bid on, we were going through all the new bids in the notification table and searching the ones belonging to the post posted by this user. We changed the logic related to this we saw huge improvements because of these 2 optimizations combined with javascript optimizations and pagination library, led to almost unbelievable results as can be seen :

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	109.5/sec	26975
302	128.6/sec	29480

- We see that there is no errors in the response and average page and request time were in order of milliseconds. Further the average transaction time was around 200 ms. This is way better than any of the results we had till now, and is mainly because of all the optimizations we performed.

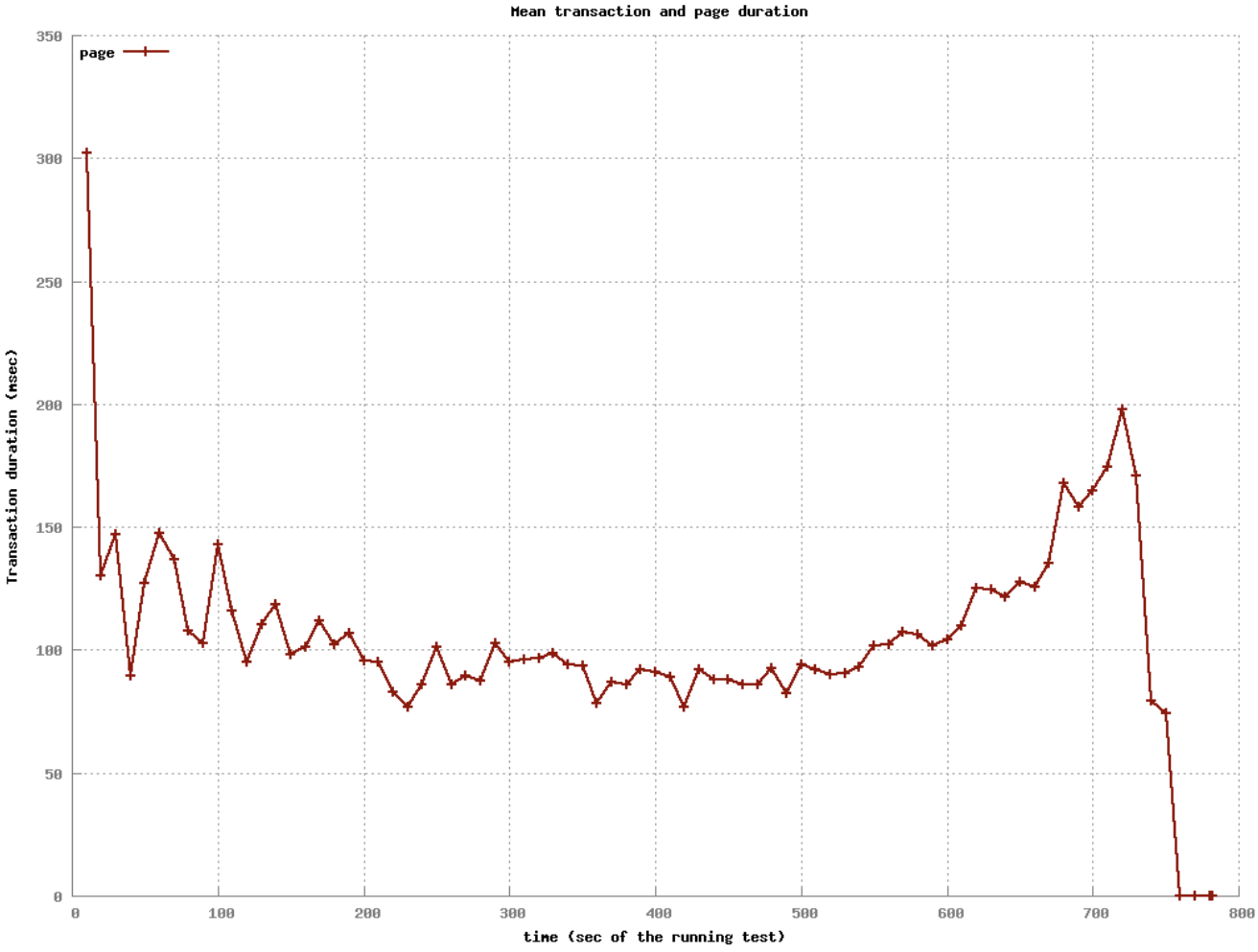


Figure: Mean Transaction Time (never goes beyond 310 ms)

Phase Increase:

Since we did not get any errors we further increased the phase rate with 100 and 200 users per second for 60 seconds. Once we added that we see that in the phase of 100 users per second it starts giving 503 errors and the total 503 errors are around 40000. This is logical as the system breaks as soon as it starts getting 100 users per second and the entire time it gives 503 mostly.

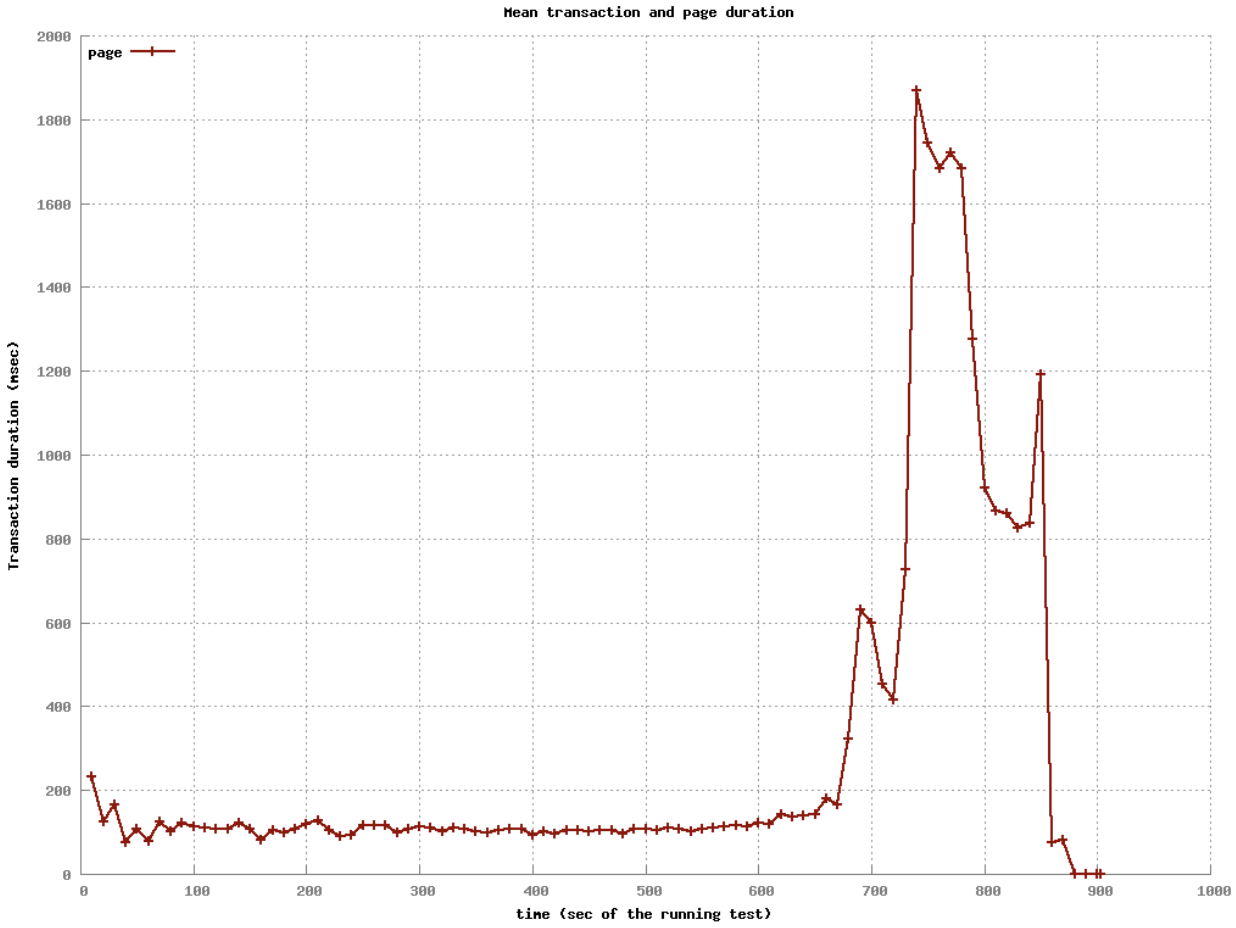


Figure: Mean Transaction Time (breaks with 100 users/sec)

.

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	118.2/sec	38980
302	253.1/sec	55057
500	0.2/sec	4

503	573.6/sec	39737
504	0.1/sec	4

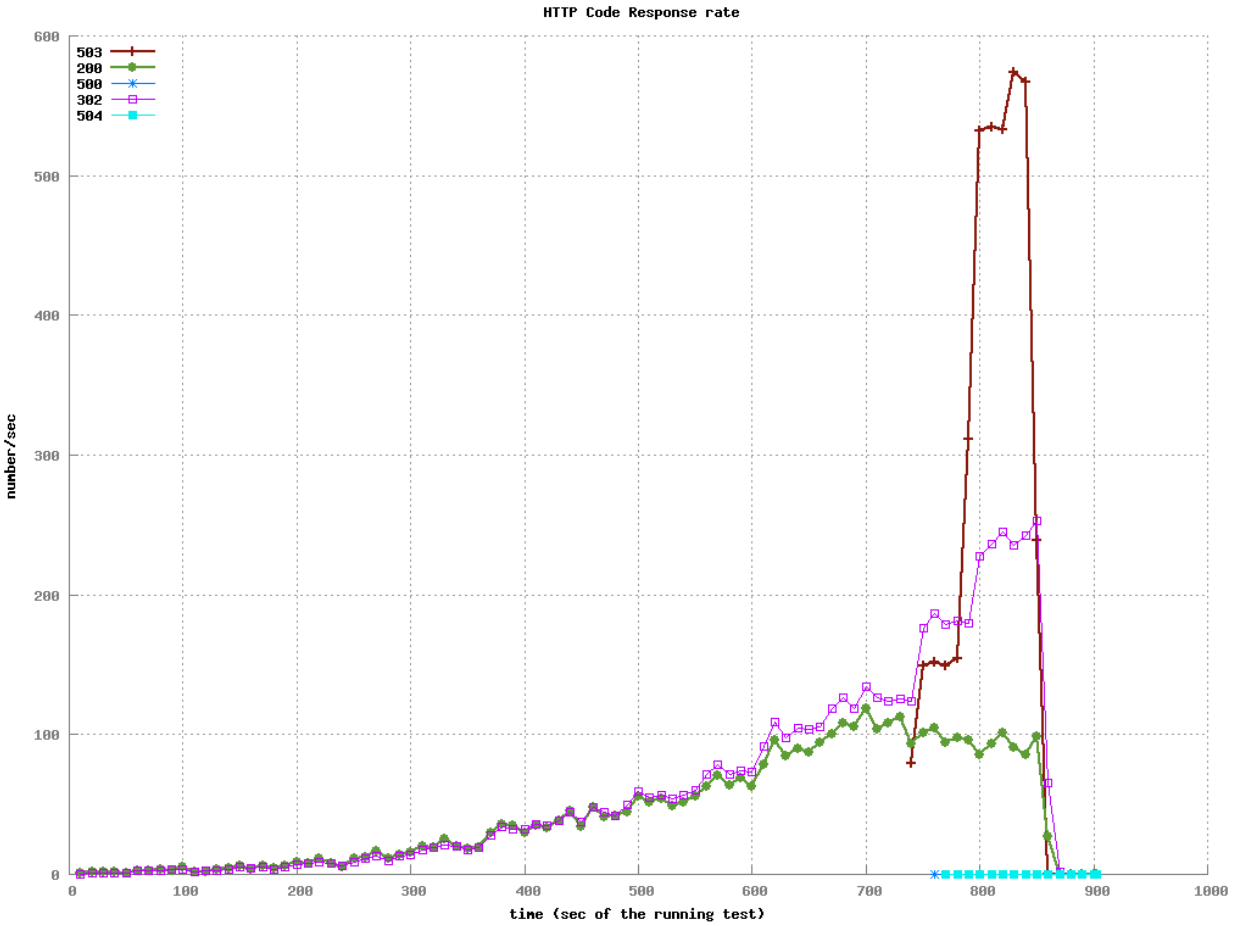


Figure: HTTP Code Response (Errors occur at 100 users/sec)

IV: Server Side Caching

Web applications receive many duplicate requests by different users. The responses to these users are almost the same, and hence reprocessing the request to get the response is a fallback in the application. This can be stopped by the use of caching, but the downside of this is the heavy use of memory in the system.

We noticed that the initial page of seeing all the posts in the system is essentially the same response for every user. Hence, we used fragment caching for this. We cached every post pin based on post id and its timestamp. This helped in fetching the posts from cache rather than performing a database query and reprocessing the result. The cached response

was much faster compared to a non cached response. Similar to this we also cached profile page of users as they remain stagnant over time and do not change much. Since we did not want to have a huge load on memory we cached the pages with an expiration time of a day and a limit of 1000 users cached at a time.

For the posts page, since the posts seen by users are also the same until new posts arrive, we further performed Russian doll caching by caching a fragment within a cached fragment. We cached the entire welcome page, and within the welcome page we cached each post. If a new post arrives the first cache does not get used, but the posts pin cache still remains valid and is used. We also did low level caching to remove multiple sql queries of posts table in bids controller. We tested the caching in two following ways: cache in memory, and memcache where a remote server is used for caching.

Our memory cache and memcache both performed the same way as sql optimization instance, but with around 38000 errors and improvement in mean transaction time. The system handles 100 users per second but then breaks in the next phase with 200 users per second. We further see that the load on database is huge and we get some '500' errors. We were expecting major changes with cache but our results were not as good as we expected. We think this is mainly because of cached pagination already happening and the huge creation of new posts and updation of posts in our load testing scripts.

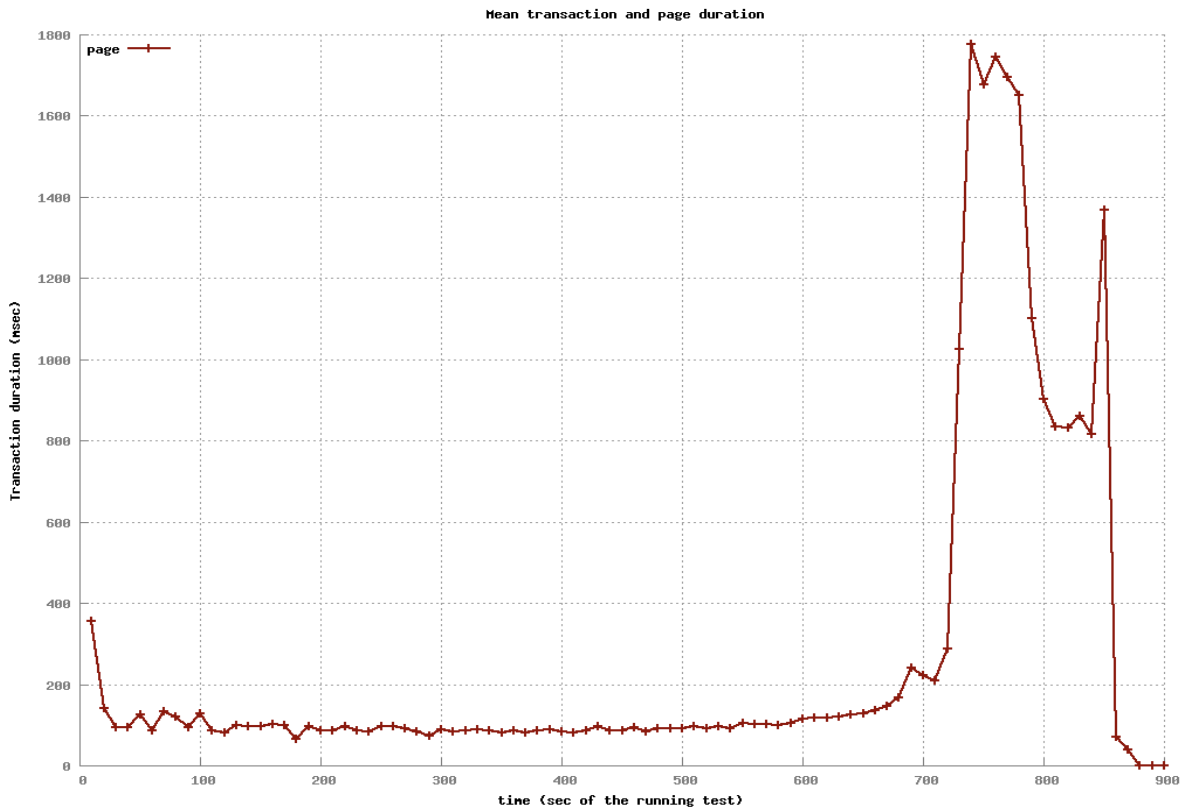


Figure: Mean Transaction Time (Breaks at 200 users/sec)

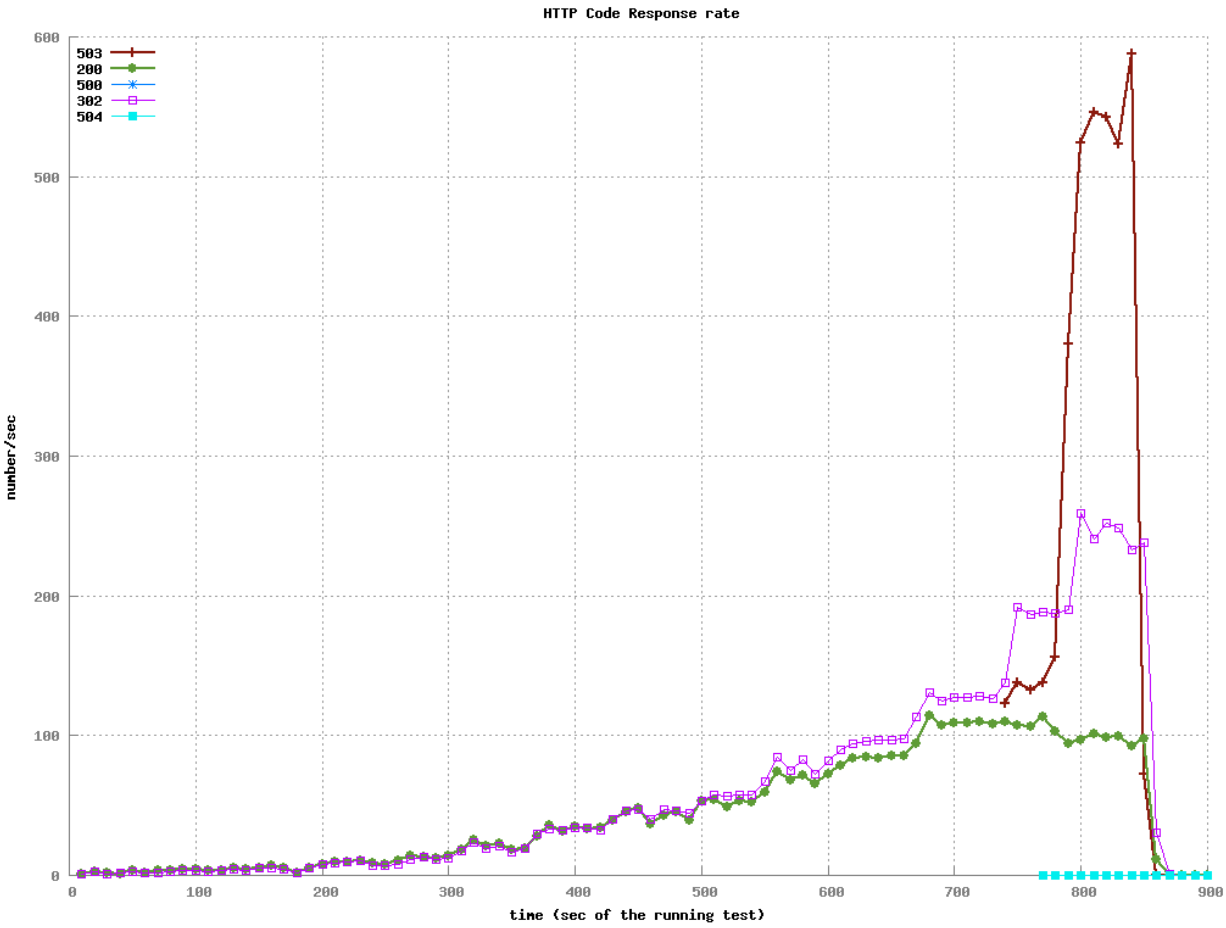


Figure: HTTP Code Rate (Breaks at 200 users/sec)

Table: HTTP Responses:

Code	Highest Rate	Total Number
200	114.6/sec	38597
302	259.2/sec	55701
500	0.2/sec	3
503	587.4/sec	38309
504	0.1/sec	3

Future Work

While doing vertical scaling, the best instance we could try was the m3.2xlarge instance which is balanced for CPU utilization and is memory optimized. We couldn't try a CPU optimized instance such in C3 series, as it was never available for us to use. Since our application isn't a memory intensive application, we expect it to perform better on any of the CPU optimized instances. Once we reached the peak with available resources during horizontal scaling, we wanted to test our database with I2 configuration as it is a high IO optimized instance and would be perfect for database. But since it was not available we could not check that. That would be a good vertical scaling option test for the database server. Another major test which we could not perform was sharding. We wanted to try multiple database instances with a load balancer with multiple app based instances. This would further reduce the bottleneck of database and improve the overall application. In one of the load tests we ran for 8 instances of the app server and single database server, we reached maximum connections on mysql2 database server and got many 500 http responses. If we perform either sharding or database replication where appropriate, we expect our application to perform even better. Also we noticed that some changes to our relation schema design could have benefitted us significantly in scaling our application. We would like to explore this option further because we believe the gains from this would be significant compared to other optimizations we may have missed.

Conclusion

We built a simple application which was basically an adaptation of the idea of UBER. We wanted to create an application where a user who needs some service can post an ad and people who want to provide the service for that ad can bid and compete against each other. Once we had the basic application in place, we focussed making the application scalable by trying various configuration, subject to the resources we had. First we started with vertical scaling of a single application server and reached a point where we couldn't get any benefits by scaling vertically. At this point, we started horizontal scaling by having multiple instances of the app servers, with a shared database server. We again noticed after some point, investing more money in horizontal scaling wouldn't provide any more benefits. This is when we started some optimization in Javascript and database access/SQL. This was promising and helped us scale to a very large extent, we didn't believe would be possible. We increase the load during testing phase here and tried some server side caching using fragment caching and low level controller caching. We couldn't see any noticeable gains from server side caching other than improvement in mean transaction time because the critical path in our load tests couldn't exploit the gains from caching. Though we had caching in other controllers and views which were not accessed in the critical path, there was no way for us to see how it helped us scale in a large scale. Finally, we not only learnt the different paths that can be taken while scaling an internet application, but also to choose it wisely as we see fit for our particular application.