A Distributed Message Board

Irtiza Ahmed Akhter
irtiza@cs.wisc.edu

Zainab Ghadiyali
zainab@cs.wisc.edu

November 26, 2012

1 Abstract

Distributed Systems are widely popular and increasingly important given the yearly increase in number of Internet users and scalable systems. In this paper, we describe the design, implementation and analysis of connect, a distributed system built on top of EC2 service.

2 Introduction

The increasing ubiquity of distributed and chat systems has paved way for an increasing number of advances in both. Deploying a chat system over a distributed system can be done in several ways. Internet Relay Chat (IRC) is one of the oldest chat systems found on the Internet. The backbone structure is a spanning tree of servers and users connect to one of these servers. The message then trickles down to all servers across the spanning tree. The protocol is simple and widely studied. Another type of chat dsystem is web based with each chat application differing in interface and functionality. AOL Instant Messenger, Facebook Messenger and Google Chat are some examples dominating this space.

3 Background

3.1 Distributed System

3.2 Chat System

As mentioned in 2, chat systems are of two main types.

3.2.1 IRC

Internet Relay Chat(IRC[2]) was a rather widely used chat system. Each IRC network comprises of a spanning tree structure of servers who typically listen on port 6667. A user idetnfies himself within the IRC-network through a designation username. The username is hence unique within a particular network. IRC networks comprise of several channels which form meeting and discussion avenues. Since channels are so critical to IRC functioning, it is imperative that these channels be unique and consistent throughout the network. An IRC operator works as an admin to override actions of remove users from network.

3.2.2 Web-chat

A web chat system uses a browser as a user interface and HTTP as underlying application protocol. Typically, a web chat service has a login system to authenticate users and a session id to maintain consistency and ensure that each user gets the chat message broadcasted.

3.3 Distributed Chat System

3.4 Design

In our chat system we use the TCP protocol for communication between the chat server and client while communication between the chat client and location server is via HTTP and TCP. TCP is a connection oriented reliable protocol and states related to each TCP connection need to be maintained at both client and server side. UDP could be used for transfer of chat messages, however we choose TCP over UDP due to TCP's reliability and the fact that packet loss may occur. The chat application is to be implemented using a clientserver architecture:

1. Clients connect to a chat server who in turn is responsible for broadcasting these messages.
2. Since the chat server has the key role of transmitting and storing messages, it is important to automate the chat server as much as possible in order to improve operational performance.
3. The network must be secure to protect messages irrespective of whether the clients connect through an insecure/public network.
4. The client must be able to use the chat service irrespective of changes to the chat server and internal application network.
5. The clients must be able to connect to the chat server irrespective of firewalls.
6. The client must have easy access to the chat application through all major browsers

4 Introduction To Connect

May be we can add a high level description of the front end web application, what it does, what are the expectations, some features, facebook integration, etc

5 Design & Implementation Of Distributed Back-End

As mentioned in the previous section that we intend to develop a web based message board system that relies on a distributed back-end. We aim to design the back-end to provide the following features - load balancing, replication, high availability, eventually consistent replicas, scalability/replication of service, fault tolerance and automated recovery. In this section we will cover each of these features and corresponding implementation one by one. One general idea for all these goals is that, we wanted to automate as many features as possible in order to minimize human intervention and to provide higher degree of transparency to the end-users. Before going into the implementation details, we will first present a brief overview of the tools and techniques we have used for that will be helpful for the reader to understand the limitations of the services provided by these tools and our implementation specific corresponding solutions.

5.1 Tools

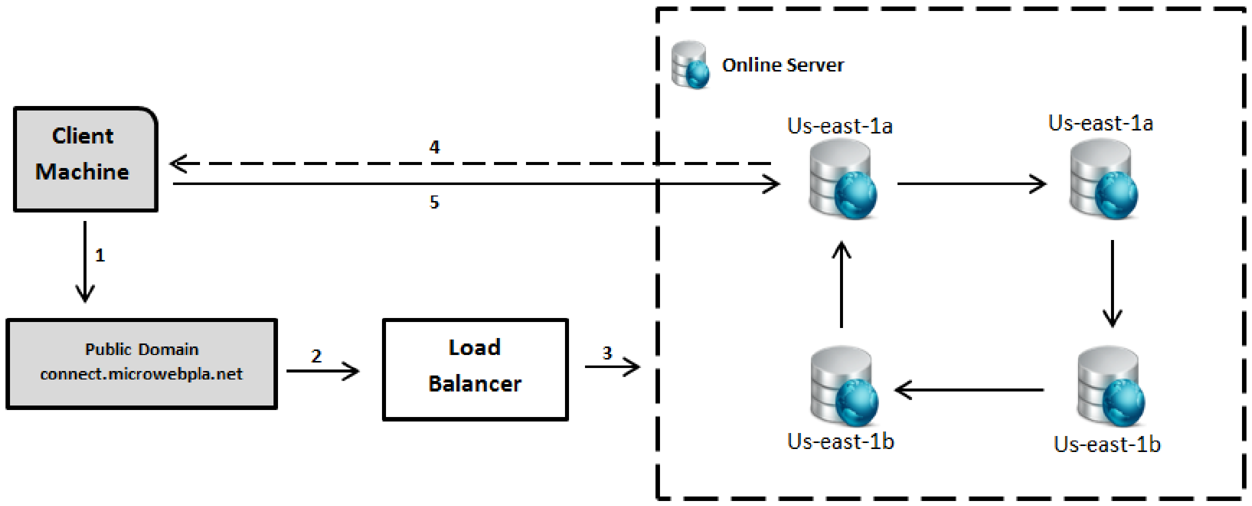

Figure 1: **Work-flow:** We have set up a public domain to make the service available to the Internet users. A client request first goes to connect.microwebpla.net (1), microwebpla.net points to the load balancer (2), the load balancer picks a suitable target server from the cluster and forwards the original request (3), the selected server gets back to the client with a reply and establishes a virtual communication channel (4), until the channel is closed, from this point client directly communicates with the selected server in step (3).

Our distributed message board application was built using several tools. Firstly we have used *Amazon's Elastic Cloud Computing (EC2)* [4] instances to accommodate the *web servers* and the *database* servers. We have spawned four *t1-micro EC2* [5] instances in two separate geographical zones (Us-east-1a , Us-east-1b) for this purpose. All the *EC2* instances are of type t1-micro and have identical configuration. We opted for *Ubuntu 11.14* 32 bit operating environment. Next we installed *Apache Web Server* [3] to host the web pages and *MySQL* [7] as the database to accommodate our simple data structure. We used a fifth *EC2* instance as a the *Load Balancer* and *Traffic Forwarder*. In order to make the service accessible from a domain name we have setup a public domain (*connect.microwebpla.net*) and configured its DNS records to point it to the load balancer. Namely, we have added a *CNAME* [9] record for the sub-domain *connect.microwebpla.net* to make it an alias of the load balancer. The domain and its DNS records are managed by *Name.com*. We used *PHP*, basic *HTML* and *JQuery* [8] to write the front end application, *MySQL* and *Shell Script* on the back end to implement different features of distributed system such as - load balancing, replication, fault tolerance, etc. And finally, the entire setup involved writing a number of custom configuration scripts for *Apache*, *MySQL* and *EC2* instances. In **Figure** 1 we have laid out the different components of the entire system and presented the work-flow of serving a client request from a high level. In the following sections we will present several distributed aspects of our design.

5.2 Replication & Consistent Update Propagation

The goal of Replication is to some extent twofold. First we want the service to be replicated in all the back-end nodes, in other words, in any case, any of the back-end nodes should be able to serve any client request. And the second one is to replicate the stored data in as many instances as possible for higher availability. The first one in our case can be interpreted as serving the web pages and accepting *MySQL* queries, which turned out be very straight forward. We simply replicated the web server with the web pages and the *MySQL* tables in all the instances. This way each *EC2* instance can serve the same web application to the end-users transparently. The latter is however turned out to be tricky. The blame to some extent goes to our specific selection of tool, namely *MySQL*. Every web server is coupled with a *MySQL* instance, however it is not strictly necessary to forward a specific database operation from a web server to the *MySQL* instance it is coupled with, we will elaborate more on this in **section** *5.5*. The second goal of replication is strictly tied to data availability and consistency. We wanted to design such a system, where data will be replicated almost instantly in all the database instances regardless of the location of the server that first accepts and processes the request, and thus providing eventual consistency. For that, at one point we decided to write code from the scratch to propagate the updates from one instance to the other. Then again, keeping in mind the fact that we did not want to re-invent the wheel, we researched a little bit more and found out that the latest *MySQL* installation offers a model for replication [10]. We ask the reader to see **Figure** 2 for a visual representation.

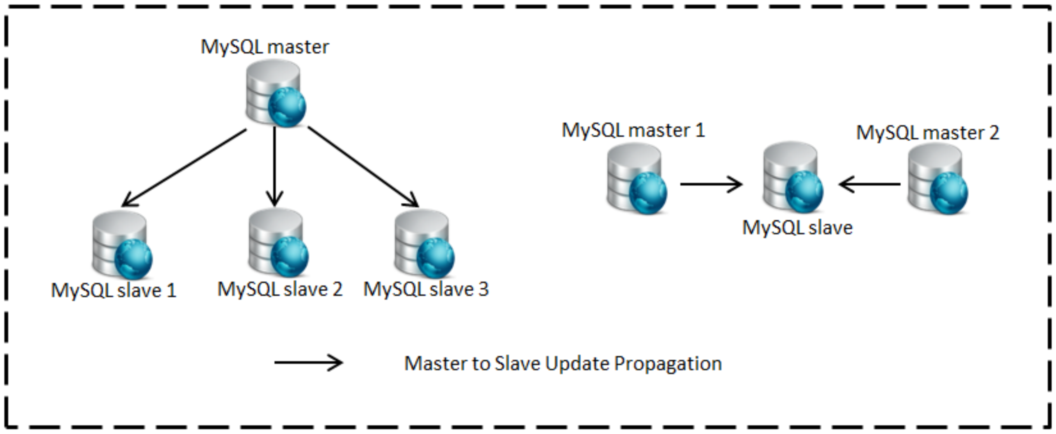

Figure 2: **MySQl Replication and Limitation:** *Left:* Native MySQl Replication. Single master, multiple slaves, unidirectional write propagation model. *Right:* Our requirement is multi master, multi slave and bi-directional model to support absolute replication, higher availability of service and strong consistency. To this date, MySQL does not have any support for this model.

MySQL came up with this model of replication with design goals and use cases different from ours. The use case of this model is to backup existing database(s) to one or more shadow servers from a single master database. For this reason, the *MySQL's* replication model allows one *master* replica to propagate updates to one (or more) *slave* replicas, so that in case of a *master* failure the *slave* can take over without loosing any data **Figure** 2 (left). At this point we would like to point the reader to the limitations imposed by this model. Note that, (1) this model only allows a *master* $\rightarrow$ *slave* update propagation, in other words propagation is unidirectional, (2) a single *master* can have multiple *slaves*, but a single *slave* cannot receive updates from multiple *masters*, (3) this model imposes the concept of *master* and *slave* and thus making the master a single point of failure and isolation of service and (4) the process of promoting the *slave* replica to become a *master* during a failure is not automated. Because of these limitations, this model is not adequate for our design goals. The first three limitations directly affect our design goals for replication, availability and consistency and the last one affects fault tolerance and recovery (we discuss this in **Section** 5.3)
Next we focused on how to leverage this existing replication model to ease our implementation at the same time not compromising any of the design goals. The intuitive solution we came up with can be deemed as a *ring topology*, where each node in the ring is a *MySQL* instance acting both as *master* and *slave* (thus eliminating the distinction between these two roles). As shown in **Figure** 3, update always propagates from *master* to *slave* in one direction. This update propagation can be seen as influenced by *Bayou's Anti Entropy* protocol, where each server talks to some other server and transfer the latest updates received by the first and unknown to the second (maintaining prefix property to have incremental write propagation). However to achieve this we had to experiment with configurations, as in the ring, each instance is a *master* of its successor and *slave* of its predecessor at the same time which is not supported by *MySQL* natively. Again this concept is highly related to our availability goal we stated before. Each instance should be able receive update as a *master*, log the update and propagate to its appropriate successor. All instances will have to be synchronized with their corresponding *master* and *slave* neighbors in terms of log position.

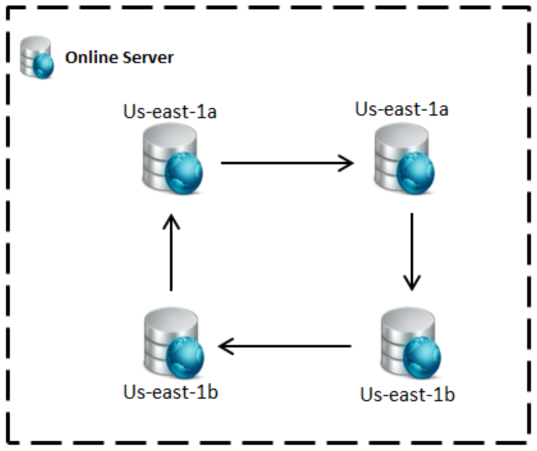

Figure 3: **Data Replication to Provide Eventual Consistency:** All the servers are connected to each other forming a daisy chain model. Data flows unidirectionally from one *master* to its single *slave*.

To keep the description succinct we are skipping the low level technical details. But the above design indeed gave us a very robust replication model. However this introduced new challenges to handle failure situation and inconsistent log position. We will talk about that in the next section. It is worth pointing out that we leveraged the original *MySQL* replication model without any violation. To be precise, in our model updates still flow unidirectionally from a single *master* to *slave*.

5.3 Automated Failure Detection & Self Healing

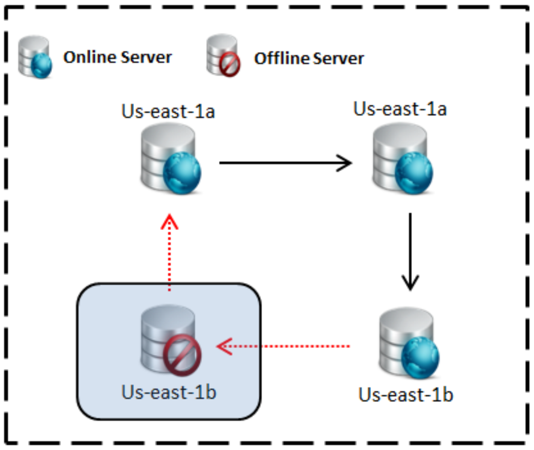

Figure 4: **Automated Fail Stop Fault Detection:** In case of a server failure the ring topology is broken and two neighbor servers (master and slave) are affected. We detect this failure dynamically.

It can be easily seen from **Figure** 3 that, if any one of the node fails, the ring will be broken (**Figure** 4) and a partition between instances will arise, which in turn will affect two of our goals, namely consistency and availability. To address this, first we define the granularity of failure we adopted. Failure can happen in one of three ways, those are (1) the *EC2* instance itself can go down, (2) the *Apache* web server instance can fail or (3) the *MySQL* can fail. Our policy is to take out the failed instance from the cluster entirely if any of the above situations occurs and thus eliminating the possibility of the failed instance being used by any client. We do this in a two step process. First a robust failure detection, followed by recovery. Both of these steps are automated and require zero human intervention.
For failure detection we implemented a *Gossip* type protocol where each instance in the chain keeps track of its *master* instance. It collects two types of heartbeats, it periodically probes the *MySQL* service that covers the third failure scenario and secondly it also probes the web server installed on the *EC2* instance which detects the first two failure scenarios. As mentioned earlier in this report, failure detection and recovery are done by custom *shell scripts*. Every instance runs a failure detection script that monitors these heartbeats. In case of any missing heartbeat, each instance dynamically repositions (recovery) itself to the proper location and completing the chain (**Figure** 5). It is worth mentioning that, this recovery takes less than a couple of seconds without interrupting any availability and raising any inconsistent situation. This failure recovery module is also written in shell script, which knows the address of all other instances and their positions.

Namely, each instance can go back ward and select a different master when the original master has failed. On the failed server, the same script detaches the entire instance from the cluster in case of a missing *MySQL* heartbeat. The other two failure cases (web server and the *EC2* instance itself) are handled by the load balancer (see **Section** 5.5).

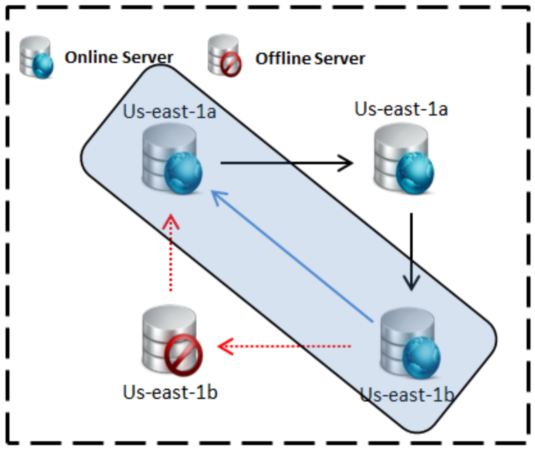

Figure 5: **Automated Fail Stop Fault Recovery:** In case of a server failure the ring topology is broken and two neighbor servers (master and slave) are affected. The recovery script simply bridges the newly created gap between these two endpoints.

At this point we would like highlight few significant gains achieved from this simple failure detection and recover model -

- Failure detection and recovery does not depend on the failed node(s)
- Any node can fail without affecting availability and consistency
- Failure detection and recovery processes are automated and fast
- As long as there is one functional instance alive, end-users will be able to access this service
- The entire process is absolutely transparent to the end-users

Once the original server is active and functional again, it will re-attach it self to the appropriate *master* (more on this in **Section** 5.4), and its heartbeat will be detected by the original *slave* and that will trigger the *slave* to re-attach itself to the *master* restoring the topology to its ideal state. Once the restoration completes, the recently recovered server starts getting all the missing updates from its *master* (assuming *master* has not failed by then) and synchronizes the log position.

5.4 Scalability

Next we talk about scalability of the design. Once we designed the dynamic fault detection and recovery, we saw that it inherently supports scalability with minimum effort. The scenario is depicted in **Figure** 6. When a new server is brought up on line, the gliding in process showed in **Figure** 6 is exactly same when a failed server comes back on line and relocate it self to the appropriate position in the ring. Therefore the system can support addition of arbitrary number of servers with minimum intervention except the following two. The new server will need a complete list of available server's IP addresses and location information. As of now this list will have to be manually embedded inside the new server's recovery script and we consider this as a bootstrap requirement. The second important requirement is that, the new server must have the right permission to access the existing instances.

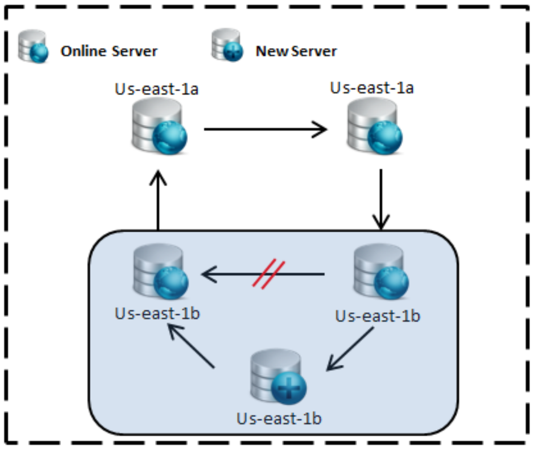

Figure 6: **Scalability:** When a new server is added to the cluster, the bootstrap script (a.k.a recovery script) automatically finds a suitable location for the new server to glide in. We try to keep the servers from the same zone in close proximity. The bootstrap script then automatically reconfigures the cluster to complete the ring topology

5.5 Load balancing, Availability and Transparency

We have used amazon's default load balancing service provided as part of the *EC2* services, which is also an *EC2* instance with minimum re-configuration to implement our design. We refer to **Figure** 1 to point out the position of the load balancer in the overall layout. The load balancer considers the system load in individual instance, its geographic location (in *EC2* terms) and the client's geographical location to forward a specific request to a specific instance in the cluster. Besides, it abstracts out the entire back-end and makes it absolutely transparent to the end-user. It also maintains the membership of each instance in the cluster. Every transaction aimed for the cluster internally or from client goes through the Load Balancer. For example, as we mentioned in **Section** 5.3 that we isolate an instance from the end-user in the face of any type of failure. We achieve this by manipulating the load balancer's configuration. Apart from our own monitoring module, the load balancer constantly monitors the concerned ports (3306, 80 and 22) and excludes an instance if it fails to receive a heartbeat from any of these ports, this makes the faulty instance unreachable from any client or even from within the cluster (unless accessed using explicit IP address). In addition to the load balancer we have an external domain name service that adds another layer of abstraction to the design and can be configured to point to a different cluster during system maintenance or catastrophic failure. However, updates in DNS records propagates asynchronously through the Internet, and hence it may take up to 6 hours (based on our experience) for the new information to be propagated all the root DNS servers.

6 Evaluation Of the Design & Results

We have conducted a number of tests to evaluate our system's performance and its features like consistency, replication strength, scalability, load distribution and capacity in terms of concurrent connections and queries. We only included three replicas in the chain purposely due to the limitation imposed by *EC2* [6] on I/O and data transfer. During this course we have written several *shell* and *MySQL* scripts to automate load generation, simulation and data collection. The load configuration was different for the different tests. We wrote a *shell* script that uses *mysqlslap* [13], a tool that comes with the *MySQL* release to generate concurrent queries on specified hosts with custom specification and schema. All the data presented in the following subsections are real time and collected from the live system under controlled environment.

6.1 Performance Improvement & Scalability

In this section we first present the limitation of a single *MySQL* instance in terms of concurrent writes it can accept and server under a specific hardware and system configuration. We then present the improvement achieved by replacing a single instance with a cluster of three instances. For this part of the simulation we varied the number of concurrent clients from 50 to 350, each submitting a single query 100 times. And then we plot average, minimum and maximum query completion time for each test set, however to evaluate performance we consider the average one as it reflects a reasonable negotiation of the both measures. In the first part of the test we submitted the aforementioned workload to a single *MySQL* instance and collect these metrics. We spawned a fresh instance for this purpose and directed the simulated load to that. In **Figure** 7 we can see the performance of a single replica under varied workload. For exactly *15000* queries the average response time was about *23.43* seconds.

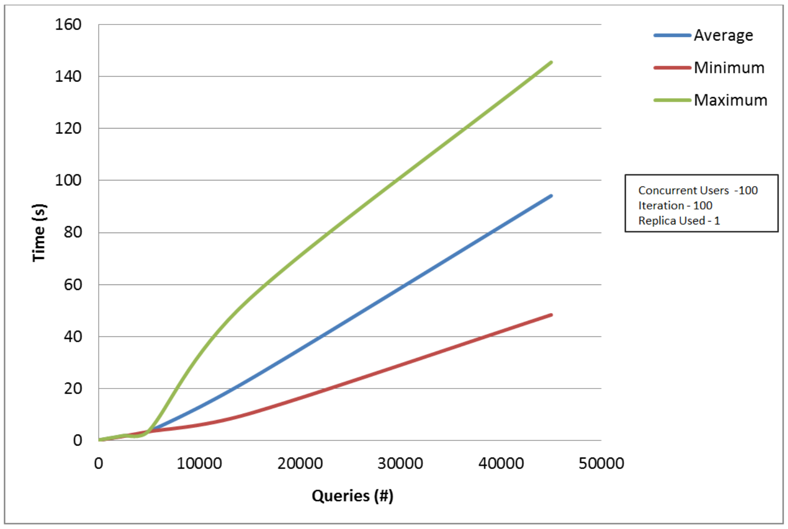

Figure 7: **Performance of a single MySQL instance:** This graph plots the average, minimum and maximum query completion time against different test sets involving varied number of concurrent queries.

Next, we submit a similar workload to a cluster of three replicas and report the results in

Figure 8. We notice the significant improvement in performance in terms of latency to serve each query. For exactly *35000* queries the average response time was about *2.6* seconds. This indeed is a non-trivial improvement.

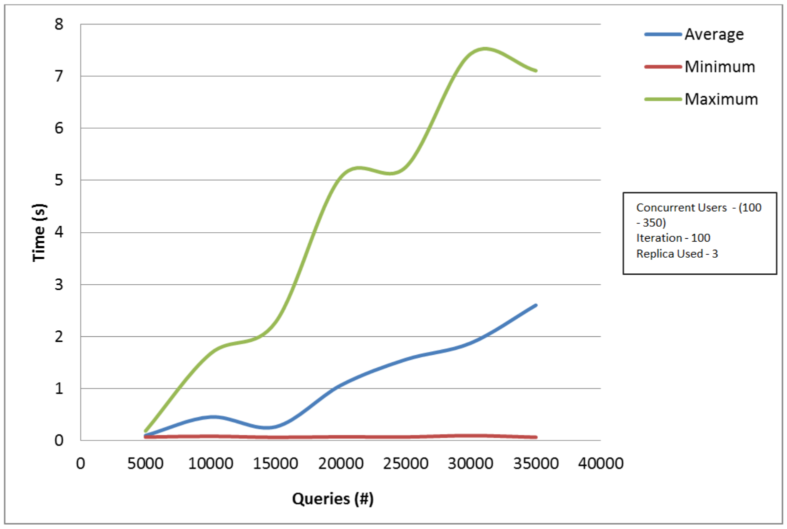

Figure 8: **Performance of a cluster of MySQL instances:** This graph plots the same metrics for a similar workload against total number of queries submitted to a cluster of three *MySQL* instances

6.2 Query Distribution & Replica Performance

Next we consider the performance of the load balancer. In other words we wanted to see how well the system distributes requests under extreme work loads. For this we simulated a *350* concurrent clients from three *EC2* instances, each submitting *100* queries to the cluster. **Figure** 9, 10, 11 and 12 depict the live scenarios in the three *MySQL* instances under this load. The first two show the number of concurrent queries and connections while the other two present a measurement of incoming and outgoing traffic. In all the four cases, the queries were distributed almost evenly across the three servers, however server 3 served more number of queries as we tweaked the number of incoming connections in server 3 to allow more. In addition to that, we wanted to see the reaction in the cluster in the face of a failure. For that purpose, at approximately 130th second during simulation we took down server1 and we can see that the remaining load was shared across the remaining two servers almost instantly.

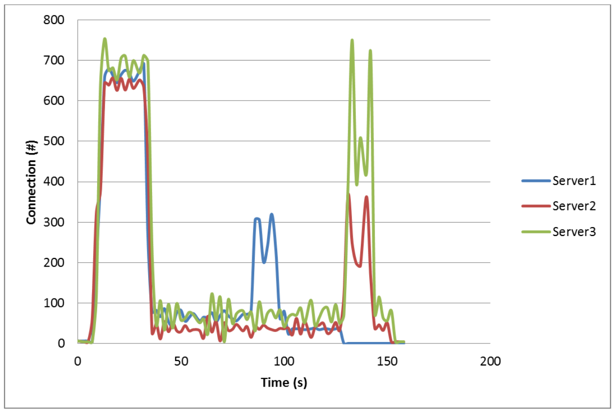

Figure 9: **Connection Distribution**

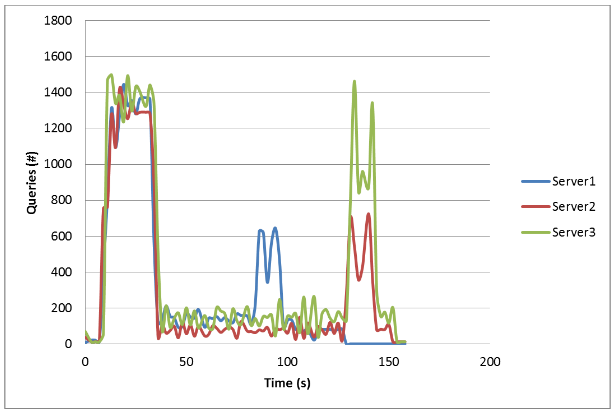

Figure 10: **Number of Queries Served by Each Instance**

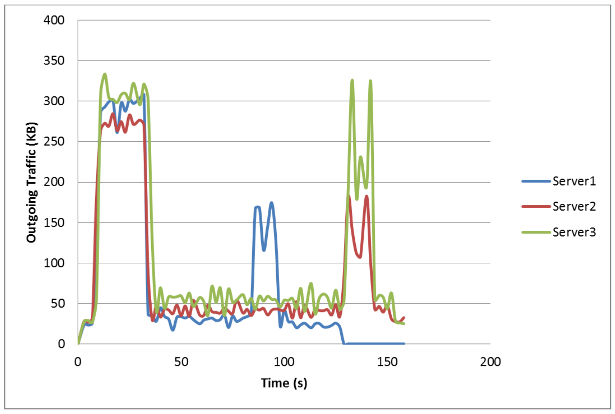

Figure 11: **Outgoing Traffic**

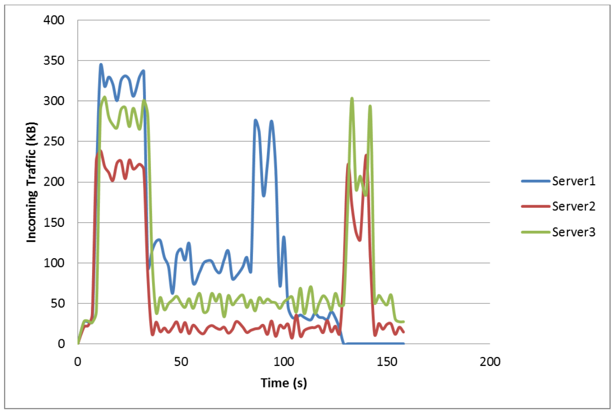

Figure 12: **Incoming Traffic**

6.3 Eventually Consistent Replication

To evaluate the consistency strength of the system, we simply calculated the hash of each **MySQL** instance and compared with others in real time under extreme load, distributed among the members in the cluster. This can be seen as taking a snapshot of all the tables and performing a comparison of those snapshots. To do this we used a third party tool *mk-table-checksum* [11], a part of *maatkit* [12] package. This program simply helps one to calculate checksum of specific database in a specified host. The load configuration for this test was, *250* simulated concurrent clients from each *EC2* instance submitting a single query *100* times. And we simultaneously generated similar load from all the the *EC2* instances. So the total number of concurrent queries submitted to three *MySQL* cluster was *(250 x 100 x 1) x 3* = 75000. A separate script on a remote machine calculated the checksum of these three replicas every *2* seconds while the queries were being served by the cluster and reported the checksums. In **Figure** 13, we report the outcome of this test. We can see from this graph that, the three checksums are not far away from each other and most of the time they are equal. And eventually after a certain period they reach to a consistent state. This lag period varies and depends on the transmission delay between any two instances, and in our case it was less 10 seconds under significant work load. This proves our claim of eventual consistency provided by the cluster.

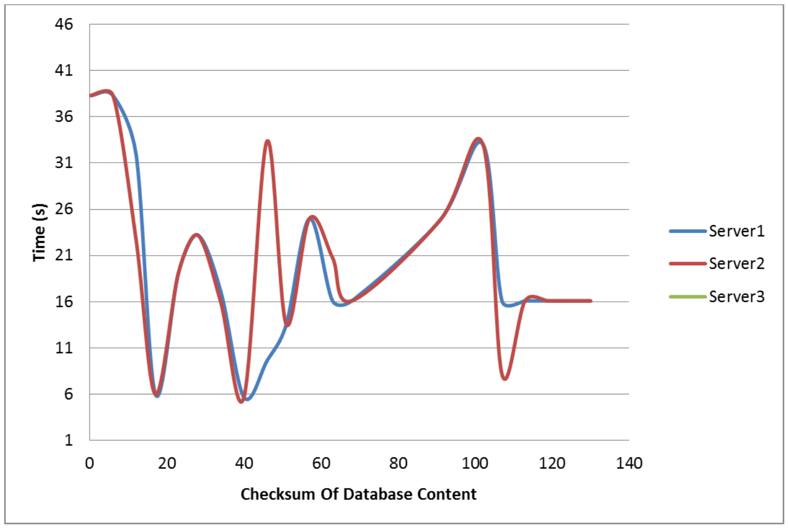

Figure 13: **Checksum Of Each MySQL Instance Against Time During Simulation**

6.4 Overloading the Web Server and Evaluation of Stress Test

7 Discussion

Note - We should include some more discussion, but the following part should be there Apart from the scripts that we wrote for implementing the distributed environment, we wrote a couple of simple scripts for source control and updating source files in all the three instances. We used *Github* [14] to store our source codes and the scripts inside each *EC2* instance periodically checks the Github repository and updates the necessary source files in the web server. Of course this can also be manually triggered if necessary.

8 Limitation & Future Work

In the previous sections we claimed and showed that the all the replicas in the *MySQL* cluster reach to a consistent state eventually. The update propagation happens eventually because each query submitted by a client is only written to one replica before returning to the client. The propagation is asynchronous and depends on latency between two replicas. However, a more complex algorithm can be implemented and used on this current configuration which will propagate the update in a semi-synchronous or even synchronous manner. The idea is to *try* to perform the submitted query on n ($n > 1$) replicas before getting back to the client. For this project we did not explore this option as the notion of eventual consistent replicas is strong enough for our client application (distributed message system) and the complexity of the algorithm required for this semi-synchronous update mechanism is dwarfed by the replication performance of the existing simple asynchronous design. Secondly,We have written bash scripts to automate failure detection, recovery and to make the system scalable. However, the topology information (e.g. location and IP addresses of the instances) are hard coded as of now. We originally planned to come up with a dynamic configuration file that these scripts can read from and write to and a tool to manage these configuration files from a single instance, but due to time constraint we could not reach that point. Nevertheless, the scripts work just fine. We introduced the functionality and position of the system's *load balancer* in **Section** 5.5. This *load balancer* is indeed a single point of failure and currently we do not have any shadow instance to substitute the failed *load balancer*. But as the *load balancer* in our design does very specific job it is very unlikely to be failed due to higher number of requests. In any case, coming up with a secondary *load balancer* should be pretty straight forward.

References

[1] Lamport, L., *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*, Addison-Wesley Pub Co., 2nd edition, August 1994.

[2] J. Oikarinen and D. Reed, Internet Relay Chat Protocol RFC 1495, 1993.

[3] `http://www.apache.org`

[4] `http://aws.amazon.com/ec2`

[5] `http://aws.amazon.com/ec2/instance-types`

[6] `http://aws.amazon.com/ec2/pricing`

[7] `http://www.mysql.com`

[8] `http://www.mysql.com`

[9] `http://en.wikipedia.org/wiki/CNAME_record`

[10] `http://dev.mysql.com/doc/refman/5.0/en/replication.html`

[11] `http://www.maatkit.org/doc/mk-table-checksum.html`

[12] `http://www.maatkit.org`

[13] `http://dev.mysql.com/doc/refman/5.1/en/mysqlslap.html`

[14] `http://github.com`