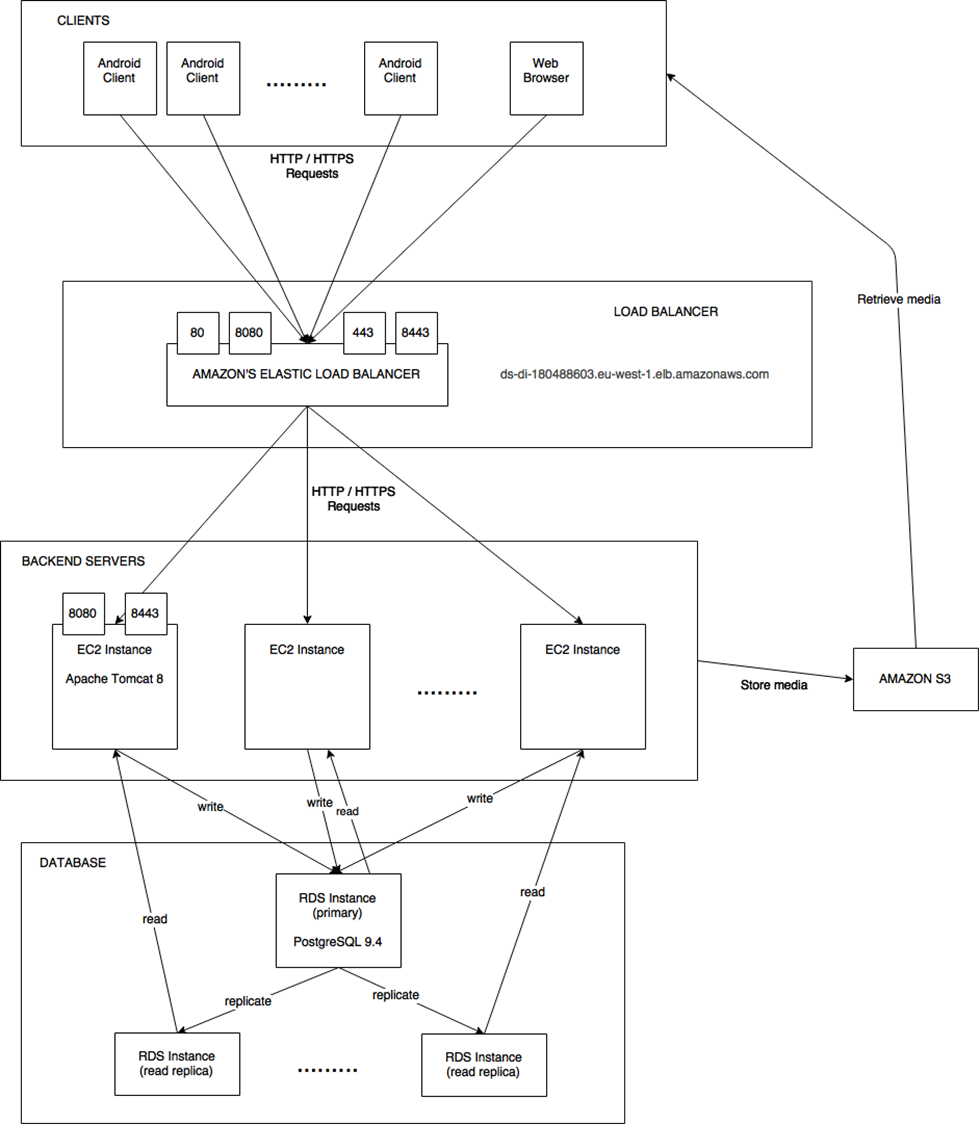
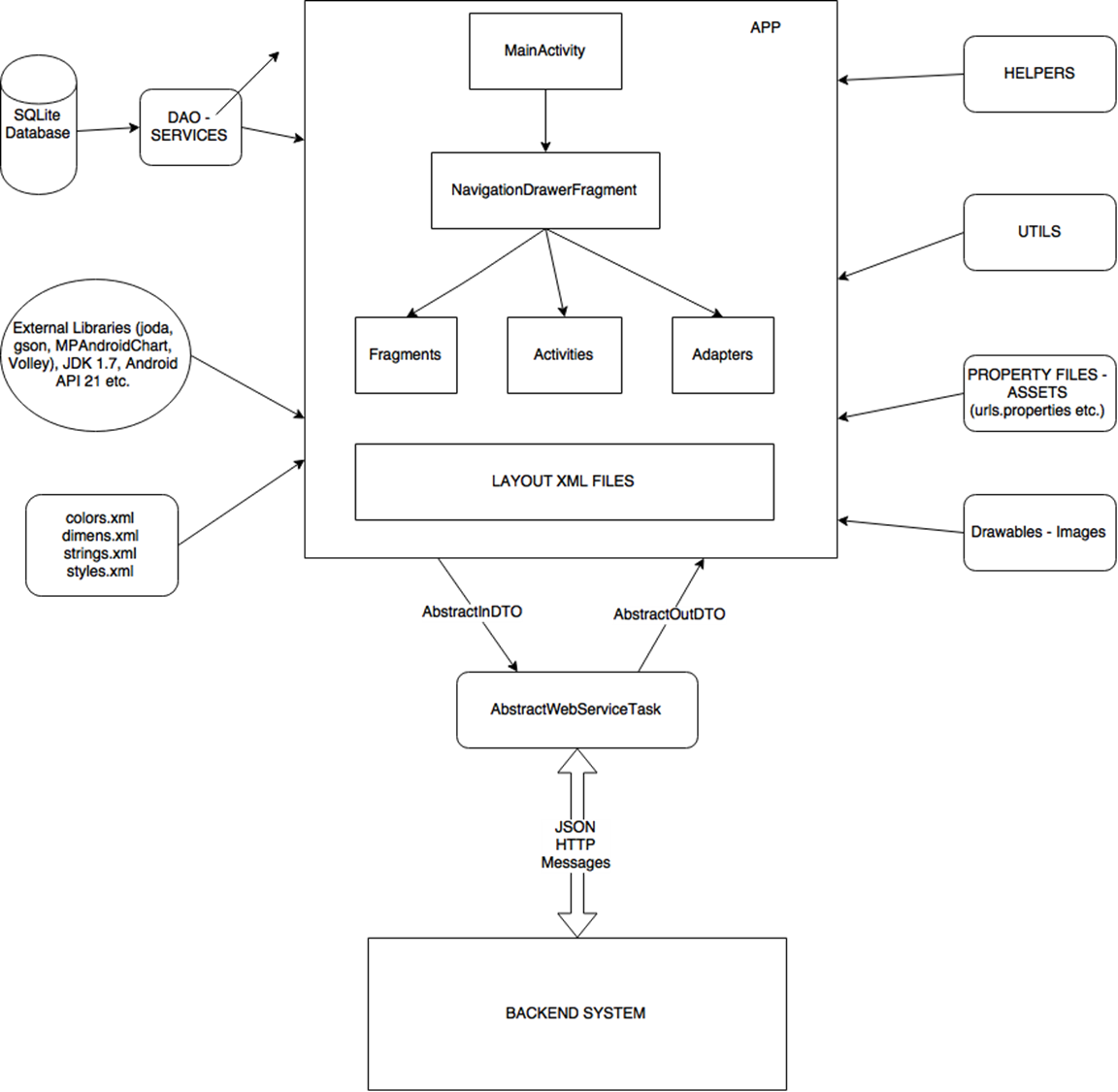
**Team members:** ALTIN CIPI (M1309), VASILIS SKOURTIS (M1341), EYAGGELOS KARAGEORGOS (M1364)

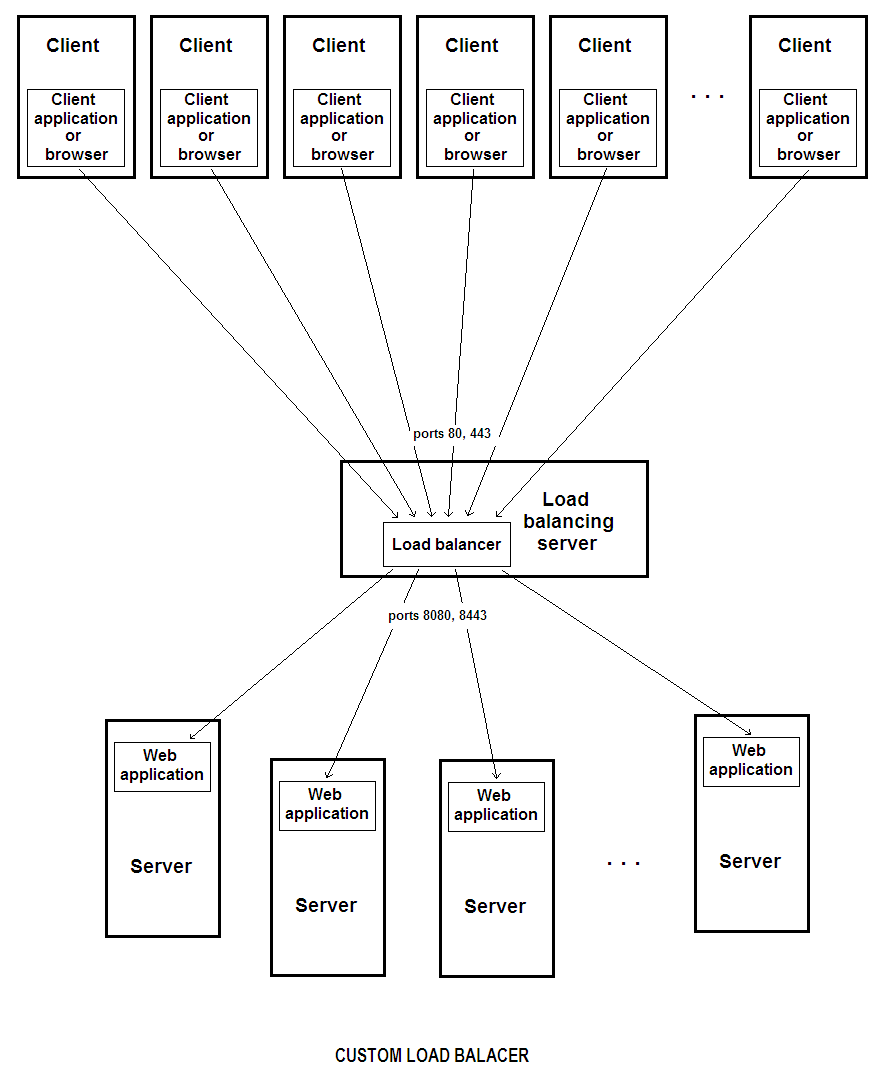
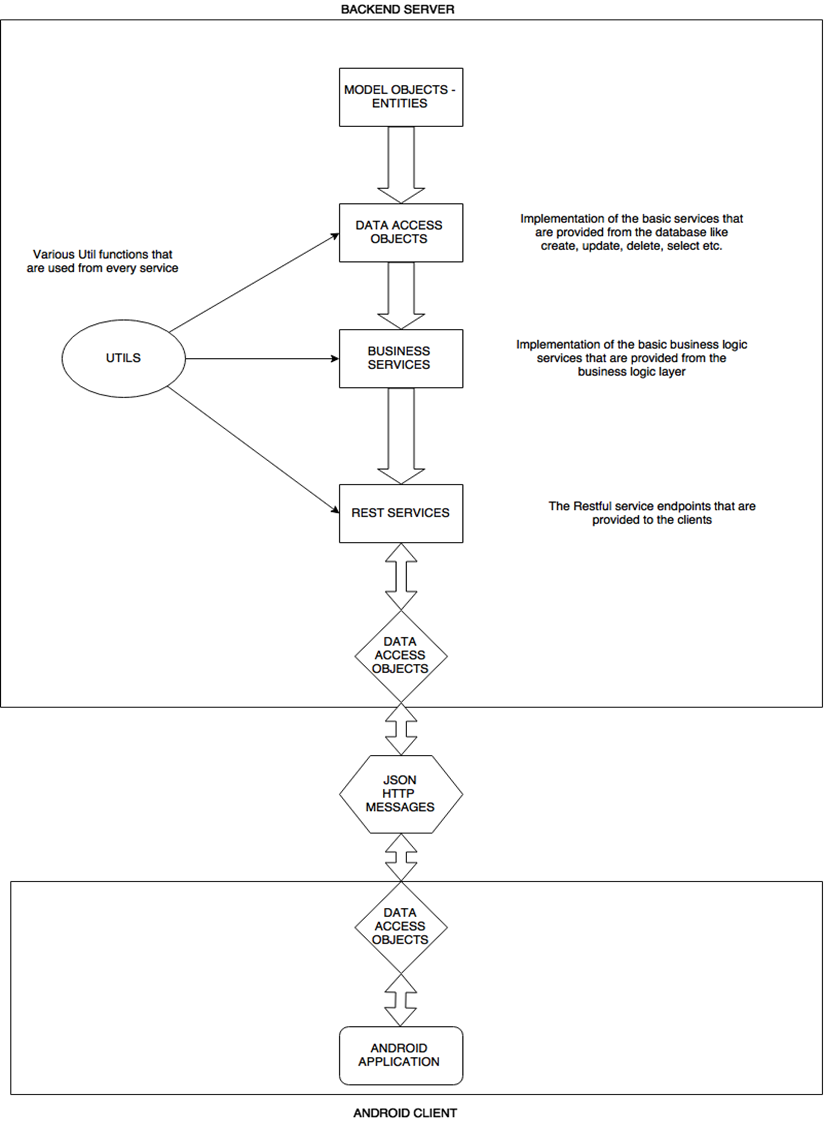
**Project title:** E-voting android mobile application + system backend (VoteForIt)

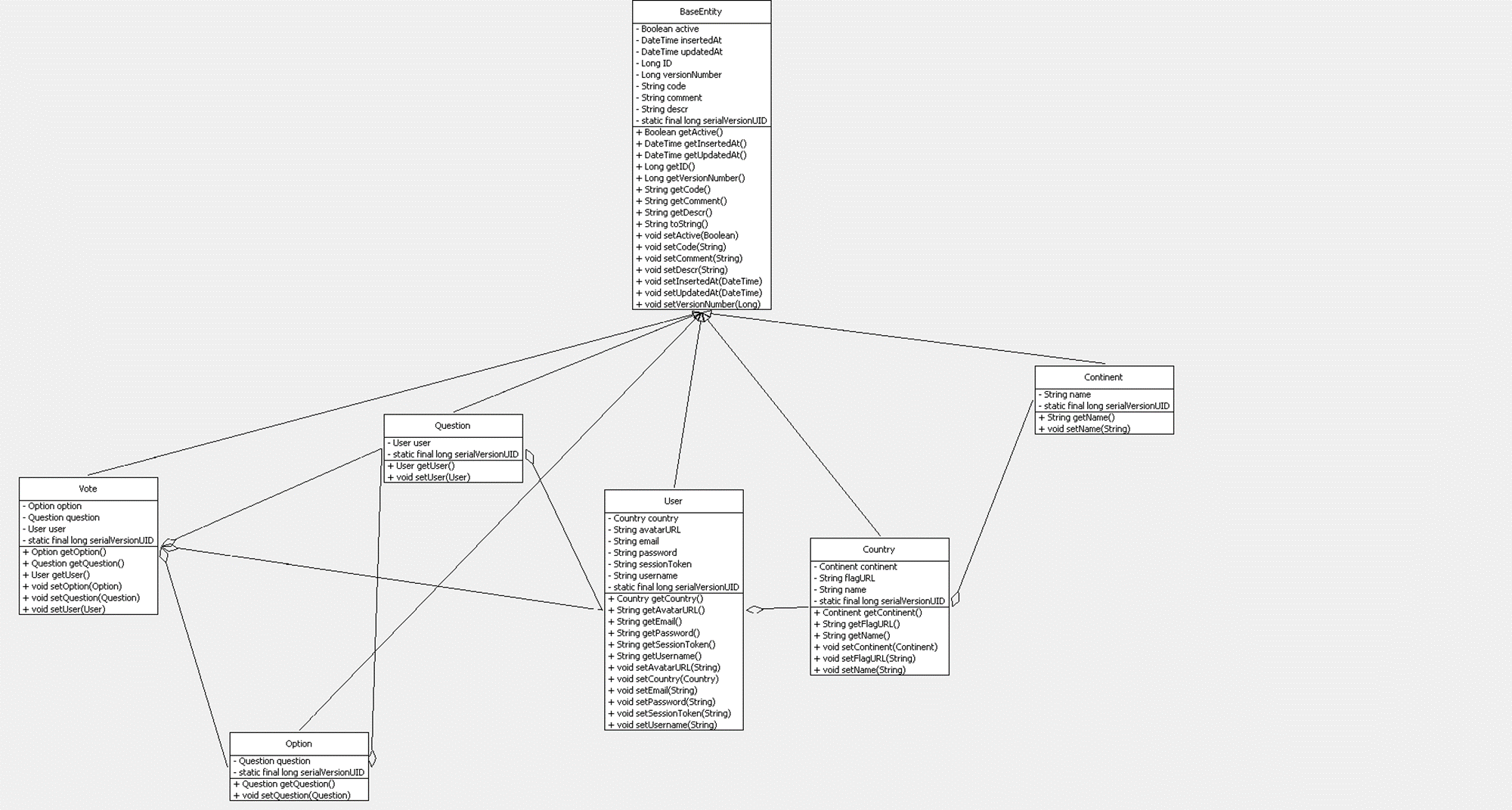
**What activities we set out for this project:**

* Create an email account at Gmail for our application ([ds2015.dit@gmail.com](mailto:ds2015.dit@gmail.com)) (achieved)
* Create an account at Amazon Web Services with the above email. (achieved)
* Create 3 EC2 instances (t2.micro, 1CPU, 1GB RAM) in Ireland Availability Zone. (achieved)
* Install in them Java 1.7.75 and Apache Tomcat 8.0.15. (achieved)
* Create an Amazon Elastic Load Balancer. (achieved)
* Configure it to listen to ports 80, 443, 8080, 8443. (achieved)
* Attach to it the 3 EC2 instances. (achieved)
* Create from this site <http://www.selfsignedcertificate.com/> a self-signed SSL certificate for the domain name [www.voteforit.com](http://www.voteforit.com). (achieved)
* Attach this certificate to the load balancer mentioned above on his ports 443 and 8443. (achieved)
* Purchase a valid certificate from a trusted CA for the above domain name. (unfinished)
* Use SSL for the communication between the EC2 instances and the RDS instances. (unfinished)
* Create an AWS S3 bucket. (achieved)
* Make the bucket read-only for everyone except for authenticated users (EC2 instances). (achieved)
* Create a primary database RDS instance (db.t2.micro, 1CPU, 1GB RAM, 20GB SSD). (achieved)
* Enable automated backups on this database instance. (achieved)
* Create a read replica of this database instance. (achieved)
* Create a Virtual Private Cloud. Apply security restrictions to it. (achieved)
* Put the EC2 instances and the RDS instances in that VPC. (achieved)
* Create an authenticated user named “ds” and give him full access privileges to the S3 system. (achieved)
* Create access keys for that user. (achieved)
* Create the DSBackendServer webapp in Eclipse Luna. (achieved)
* Implement all the functionalities and the REST Services needed.
  + Create the model objects (achieved)
  + Create DAOs and Services for each Model object (achieved)
  + Create the DTOs (achieved)
  + Create the REST Services (achieved)
* Upload the webapp on the 3 EC2 instances via FileZilla and start the tomcat servers. (achieved)
* Create the DSAndroidApp project in Android Studio. (achieved)
* Implement all the screens that will be needed.
  + View my voting subjects (achieved)
  + User Profile (achieved)
  + Create a Poll (achieved)
  + Browse through all voting subjects (achieved)
  + Vote for a poll (achieved)
  + View a poll’s results (achieved)
  + Login (achieved)
  + Register (achieved)
  + Logout (achieved)
  + Settings (unfinished)
  + View user’s photo (achieved)
  + Subscribe to another user (unfinished)
  + Get a notification at the end of a poll I have voted (unfinished)
  + Users have points that will increase each time they participate in a voting (unfinished)
  + Categorize polls (unfinished)
  + Create poll only if have enough points (unfinished)
  + Upload the final android app to PlayStore. (unfinished)
* Upload the android app to a mobile phone for testing. (achieved)
* Implement a session token technique on every REST service request. (achieved)
* Implement password advanced encryption. (achieved)
* Implement salt hashing. (achieved)
* Develop a tool that will trigger a very large amount of client requests to test the Amazon ELB. (achieved)
* Develop functionality in the backend web app to support multiple databases and read replicas. (achieved)
* Develop a custom load balancer. (achieved)
* Test the custom load balancer. (achieved)
* Develop a custom database replication system. (unfinished)
* Develop a custom database load balancer. (unfinished)
* Purchase a domain name for [www.voteforit.com](http://www.voteforit.com) (unfinished)
* Configure the Amazon’s DNS servers to route all requests made on this domain to our elastic load balancer. (unfinished)
* Develop various test scenarios. (achieved)
* Compose the final presentation and demo. (achieved)
* Create a repository in GitHub and upload all the code there. The repository is this <https://github.com/std08010/ds2015> (achieved)









**Load Balancer Architecture:**

In our custom application load balancer design, our main objectives were two. First, to keep a high performance and very responsive load balancer, by adding as less functionality as possible and minimizing delays. On the other hand, a competitive objective, to make our balancer as smart as possible. The more information the balancer has, the better his decision making will be. Moreover, the fault tolerance perspective was always part of the design.

In order to improve our balancer's decision making, comparing to widely adopted simple techniques like round robin or just ping information, we had to provide him with high quality information, allowing him to make correct or less error-prone estimations and assumptions with negligible processing. To provide such information, we came up with a hybrid ping.

Periodically, load balancer sends a specific REST request to its joined servers. After receiving that request, each server performs a system call to get the CPU and memory usage while it also executes a database query, responding to balancer with the system values. Balancer measures response time and keeps the values that got from server. Based on these two measurements, the balancer calculates the rank of each server and sorts them in a list by rank ascending order. In that way, balancer has a clearer image about the whole system state. Ping functionality provides information only about the network. A network closer server could have larger request response times either because it is too busy or because it is away from other resources, such as database. Hybrid ping provides balancer information about network, congestion, general state, resulting in better request routing and smaller response times.

Moreover, if balancer gets a failure response or a timeout, then it adds a large penalty to that server. That server will be moved to the last place(s) of the servers list and actually it will not be assigned any requests. After that server starts to respond normally, its rank value will be improved gradually and after some iterations, server will take place normally in available servers list.

After having updated its system status, the balancer routes the incoming requests based on that. Each incoming request is forwarded to the first server in server list. While balancer forwards that request, it updates its local system state insight, by increasing the load of the corresponding server and re-sorting list if needed. An overall load balancer functionality is depicted in the figure below.



**Evaluation Section**

**Backend web application Evaluation:**

We have built some test classes in order to check some of our services and some of our REST Services. Those are AmazonS3ClientTest, Log4jConsoleTest, CountryClient and RegistrationClient.

**Android App Evaluation:**

We used the final mobile app in every way possible and we have tested as many case scenarios as we could think of in order to see how the application responds. When we were able to find a bug we made the appropriate corrections. The app seems to behave correctly in all normal test cases.

**Elastic Load Balancer Evaluation:**

In order to evaluate the Amazon’s Elastic Load Balancer we built a test class named StressTestClient in which we generate a large number of Threads. Each thread makes a HTTP Request to the REST Service StressTestRestService. This Rest Service all it does is to sleep for one minute. By this we can test how many concurrent connections our system can handle and how does our elastic load balancer behave in a large amount of requests. We have configured our tomcat servers to have a thread pool of 200 threads. This means that one server can handle 200 concurrent requests. Since we have 3 servers this means that we can handle 600 concurrent requests. What happens if we have more than 600 requests simultaneously? The answer is simple. All the other requests are pending for a thread in the thread pool to finish and become available again. If the client has set a timeout for the response then if our system has insufficient resources to handle the request then the client will get a request timeout error.

In the below article you can find a lot of information about the ELB and how can someone evaluate it. There exist various tools that can perform this evaluation. We have not proceeded in using any of those due to lack of time.

<http://aws.amazon.com/articles/1636185810492479>

**Fault Tolerance Evaluation:**

In order to test fault tolerance we were shutting down our ec2 instances one by one and we were running our test classes in order to observe how well the system was responding with the remaining instances, until there was no instance left and the system was unresponsive. We conclude that the more ec2 instances the system has the more scalable and fault tolerant it is.

**Custom Load Balancer Evaluation:**

In order to test and evaluate the load balancer, we deployed it on local machines and connected it to multiple instances of the backend web applications, deployed on the same and different machines. We developed a simple tester class that floods the balancer with requests on a rest service (that queries the database) continuously while configuring the load balancer to record all the requests along with additional metadata in order to analyze its behavior. During the testing, we disable application servers one-by-one and monitor the CPU usage and network load of the entire system. We observe that the system is resilient and is operational with down to only one server online. As we gradually enable all the servers, we observe that the load is being distributed equally again.

Our load balancer was tested in various scenarios. With all servers operating normally during the whole experiment, with some servers closer to load balancer having small network delays, with some servers accessing remote databases and with some servers failing during the experiment.

To setup the environment to materialize all those scenarios, we launched 4 tomcats simulating the web servers on cloud. Tomcat servers were running in 2 different PCs, 2 on each PC. The load balancer was running on one of these machines. A LoadBalancerTester, running on another or same machine, was creating a number of threads in order to hit the balancer with concurrent requests.

In one of the most representative scenarios, 100 threads were hitting the balancer in parallel with same requests. First, we stopped one server manually to monitor balancer's behavior with 3 servers. Later, we restarted the server that failed and the system was running with full resources. At some point, we had stopped 2 and 3 servers and one of them was not able to be relaunched again. At the end of the experiment, the system had 3 available servers.

The load balancer evaluation results are given in a separate file, with some more charts displaying servers' and balancer's behavior. Below, we have added some figures showing the anticipated balancer behavior during the described above experiment.

