Distrivia: A Distributed Trivia Game

Brian Gianforcaro
Rochester Institute of Technology
bjg1955@rit.edu

Steven Glazer
Rochester Institute of Technology
sfg6126@rit.edu

Samuel Milton
Rochester Institute of Technology
srm2997@rit.edu

1. WHAT WE'VE DONE

Much of our system has been implemented and is in the debugging phase. The server application, web client, and Android client are nearly complete with some revisions happening as necessary. We have also started development on an application for the iPhone platform. The extent of what has been finished for each part of our system is explained below.

1.1 Architecture Overview

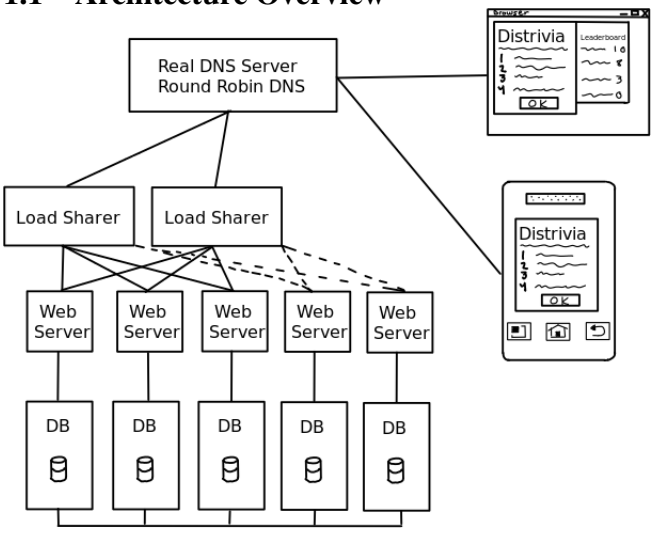

Figure 1: High level overview of the system

1.2 Servers

Currently, we have 6 servers running on Amazon's Elastic Compute Cloud [1]. One server is being used as a web-monitor host running Monit[5]. We are able to login to the web-monitor to see the status of all our servers, databases, and load sharers. This will help us manage whether or not the services are running and allow us to track down any problems, if they arise. Three servers are all running our server application and hosting the databases. We also have two final servers that we have setup as hot spares in the event of partial or total failure. Each server contains the server application that responds to clients as well as the Riak [9] databases. Riak auto-synchronizes the databases between the 5 servers. Note that the hot spares are setup to be part of the database cluster so they have a full copy of the database on hand, however they don't accept any reads or writes directly. This allows for them to immediately be ready for read/writes in the event that the original servers are down. With this configuration if we were to add a server, we would just need to update Riak and it would sync the databases to any additional servers as well. These servers, therefore, all contain the same information. If any servers go down, the distributed system would still run fine. We could have all servers go down except one and still be reliably handling requests.

1.3 Server Applications

The server is currently almost feature complete, but still needs a lot of testing. Client registration, and login are all supported. A client can join a game, retrieve questions for that game, and answer questions. Both a global leader board and individual per game leader board can also be requested. We recently refactored our API to be more universally consistent. The servers and clients now work on a more consistent basis where the clients all expect the same information and use the same calls.

1.4 Traffic

We have two load sharers set up. Both sharers are currently configured to divide the requests among the three active servers and, if need be, the two hot spares. The duplicate sharers allow for a load distributer to go down and we would still be able to serve requests to the servers. These load sharers are selected by the clients using round robin DNS where distrivia.lame.ws will select either load distributer that is up at random. If a load distributer should go down or become unreachable the DNS service will no longer serve that hosts DNS record until it is again reachable. This provides us with stability for accessing the servers. The load sharers are set up to attempt to connect to a server twice in a one second period before deeming it dead and removing it from its list of servers to direct clients towards. Currently we are using the NGINX [6] in a reverse proxy configuration for our load sharing needs.

1.5 Clients

Our focus for clients has been on a web-based client and an Android application. Both clients have most of the workings completed. The web-based client will be run from desktop platforms and laptop platforms. The Android client will run on any Android phone running OS 2.1 or higher and does not use any unique services that require specific hardware. Both clients are nearly fully-functional. They can, currently, login/register players, view leaderboards, join public games,

and compete in public games. We are actively adding features to both clients, daily. We have also started development on an iPhone application for distrivia. This client does not yet communicate with the server, but it does implement nearly all the views seen in the other clients. We have been setting up the network communication for the iPhone platform as well.

1.6 Messages

We are currently using the https protocol for our messages. This provides a secure and usable messaging system that we have designed our client and server APIs to use. The actual messages are sent and received in two way's. Initially a user session key is generated using the UUID[7] algorithm and returned to the user on successful authentication. This session key is then passed back to the server with every in coming message. Messages are sent to the server using POST and GET request methods supported by the HTTP protocol. Responses from the server are in the form of JSON [8] objects. JSON is a small, easily parseable format, representing objects in a human readable format as they would be represented in JavaScript. With the web client the JSON object returned from the server is instantly usable, however for the Android client we have to parse the JSON object into a native Java object. The Android SDK provides libraries to do this, located in the org.json [2] package name space.

2. WHAT'S LEFT

2.1 Server

The server is nearly completed. The server application just needs to have some tweaks and further testing.

2.2 Web Client

A lot of the main features of the web client are completed. However, there are still a few remaining features that need to be implemented. The only major feature remaining is private games, which are largely unstarted, although it will reuse most of the public game code. In addition to this, buttons should be disabled when clicked to prevent users from spamming the server with repeat messages.

2.3 Android

Private game play is the largest thing left for the Android client. Besides this we've just been fixing bugs and cleaning up the user interface.

2.4 iPhone

The iPhone application just started development in the last couple days. We have built most of the user interface but need to connect all of the actions to the server. We will need to implement everything, including login/registration, joining public/private games, viewing leaderboards, and competing in rounds.

3. MEETING THE REQUIREMENTS

Our system has three unique clients. One of them is designed for the Android platform as an application. Another is designed for the iPhone platform as a native application. The third is a web client, which is specifically being tested on Desktop and Laptop browsers, however it may work on many hand held web browsers as well.

Our system supports having any number of servers and allowing any subset of them to fail. We are using Riak, which supports a 'shared nothing' architecture. We can add a new server by joining it to our database and starting the web server. This can be done as many times as needed, allowing us to scale up our system. These same principles also give us great fault tolerance, since any server is capable of running independently. As long as one server is still up, the service will be available.

We use round robin DNS to allow our clients to be completely ignorant of our servers. Load sharers can be added and removed from the DNS server and the clients will be completely ignorant of this and unaffected. The load sharers in turn know about our data servers and do their best to distribute the load across them. When a new server is added, it is just added to our load sharer configuration file and clients can start using it. When a server fails, the sharers will stop directing traffic to it.

We are currently using Riak as our database backend. We chose Riak for multiple reasons, but the biggest was it's focus on scalability, fault-tolerance, availability, and replication. Riak is a Amazon Dynamo [3] inspired database, it supports full masterless replication. In Riak every node can serve client requests and they are propagated to the rest of the cluster. In our system we have made sure that this replication occurs between nodes on Amazon EC2's internal Gigabit ethernet so it is as fast as possible. Riak also has the unique property that if all of the nodes but one fail, the single node can still accept both reads and writes to the database. When the other nodes return to operational state from failure, all changes will automatically be propagated to all of the nodes. This feature is the basis for our ability to support and recover from partial failures. In the event that two Riak nodes accept writes from two separate clients modifying the same data and Riak is unable to resolve the conflict the user is given both versions and allowed to resolve the conflict them selves. However we have configured our installation so this can not occur, we have specified that every available node must agree on a write before it is committed. Given the relatively small number of nodes (five) that will take part in this agreement, we decided that the possible overhead was acceptable for the increased level of consistency it gives us.

Our team is taking a very proactive stance towards security. All of our machines are properly firewalled so that outside clients can not authenticate to our database cluster. Also we have implemented SSL through our entire infrastructure. Our load sharer's automatically redirect http:// to https:// and once you are in https:// at the load sharer, SSL continues all the way back to our actual web application back end serving responses over a SSL wrapped socket. This way no passwords are ever sent in clear text. Our group even found a SSL certificate authority who was willing to sign our SSL certificate for free. This gives us actual fully functional, authenticatable SSL infrastructure. In order to make sure that our system properly secures user information, we stuck with the industry standard of storing a hashed version of the users password instead of the password itself. We decided on using the cryptographically secure hashing function designed for passwords known as bcrypt [4].

4. REFERENCES

[1] Amazon elastic compute cload server hoasting. http://aws.amazon.com/ec2/.

[2] Android sdk: org.json package. http://developer.android.com/reference/org/json/package-summary.html.

[3] Dynamo: Amazon's highly available key-value store. http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.

[4] A future-adaptable password scheme. http://www.openbsd.org/papers/bcrypt-paper.pdf.

[5] Monit: Process supervision tool. http://mmonit.com/monit/.

[6] Nginx: A lightweight, high-performance web server/reverse proxy. http://nginx.org/en/.

[7] Rfc 4122: A universally unique identifier (uuid) urn namespace. http://tools.ietf.org/pdf/rfc4122.pdf.

[8] Rfc 4627: The application/json media type for javascript object notation (json). http://tools.ietf.org/pdf/rfc4627.pdf.

[9] Riak: Scalable data store. http://www.basho.com/products_riak_overview.php.