**Topic 31: Distributed Systems (CDN + Storage) Design**

**Team Members**

Yani Jin ([yanijin@usc.edu](mailto:yanijin@usc.edu))

Xinyi Cai ([caixinyi@usc.edu](mailto:caixinyi@usc.edu))

Ryan Afranji ([afranji@usc.edu](mailto:afranji@usc.edu))

**Problem Description**, challenges especially in terms of software and algorithms development, motivation, any novelty (in terms of algorithms, math, etc.) compared to any prior work (3 points)

Challenge 1: We thought about establishing the web server with Cassandra database on our local virtual machine at the beginning. The other part of our system will need to access the public ip address of the web server. We had tried to use Ngrok (a tunnelling tool) to acquire a public ip address for the web server. However, we found that the public ip address generated by Ngork is dynamic. Then we cannot maintain it for further use.

Challenge 2: We reestablished our web servers as well as Cassandra database on AWS due to lacking of hardware devices. But we might need to deal with the limited RAM problem in the future development.

Challenge 3: Understanding the configuration of Nginx is also a challenge for us. Nginx provides not only a way to build web servers, but also a way to configure computers as load balancers and cache proxies.

Challenge 4: After having an appointment with Akshata, we might need further help and discussion about rest API. Due to lacking materials for now, we will complete this part in our next phase.

**Project Timeline** (what you have done so far, what you expect to do by Phase II and Phase III. (2 points).

Adding progress report for phase I, adding any information the course staff and supervisors should be aware of (any change of plan since the proposal and the reason (10 points).

What you have done so far:

1. We have successfully made connections among web servers, load balancer and cache proxies.
2. We have built a web server connecting to Cassandra database.

What you expect to do by Phase III:

1. By that time, we could implement more complex queries on the Cassandra database.
2. Adding self-built GeoDNS to the edge of our CDN.
3. Holding a hierarchy of web pages which provide REST APIs for our clients.

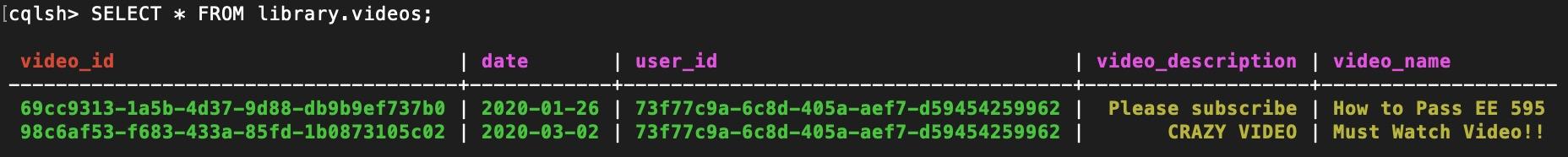
**Progress report**, (any change of plan, etc.)

After research on the difference among reverse proxy, load balancer and proxy cache, as well as popular tools providing these functionalities, we decided to build load balancer and proxy cache with Nginx instead of varnish.

**Analysis** of your solution, framework, algorithm set, mathematical models. Analysis of time and memory complexity. For the power try your best to model the power consumption. Writing the corresponding pseudo-codes, flow charts, etc. (20 points) \*\*

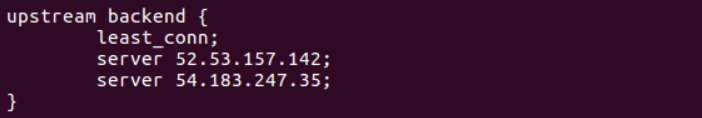
1. In the load balancer, the default algorithm is round robin. It sends requests to the backend servers in a rotational fashion. It is one of the simplest methods for delivering requests. Round robin load balancers would assume that all servers are basically the same from every aspect. This algorithm might not be very sensible when the server in order still has a lot of connections waiting to be answered. So here we change the algorithm to least connections. With the least connection method, the load balancer will compare the current number of active connections among those available servers. After that, the load balancer will send the request to the server which has the least active connections.
2. As for proxy caches, two key issues are considered; one is where and in what architecture to store the cache content in proxies, the other is specifying the cache policies. For the first one, we need to create a directory for cached responses, allocate a specific amount of memory for Shared Memory Space, specify directory structure levels, define a period of time after which we should release the cached response, limit the space for cached response. For the other, setting cache validity, content refreshing and when to deliver stale content is needed.

**Implementation** (submission requirements: code, short summary of any software packages you have used, and any other information related to the implementation work) (35 points). \*\*

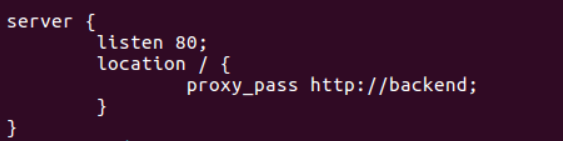
1. Database
   1. For our database platform, we chose Cassandra. We have a Cassandra process running on the EC2 instances that are running our web servers. To interface between our web server and this database, we downloaded a driver for node.js that allows our web server to make queries to the database we have created. We built the keyspace that was designed in our phase I report. The lines of code entered to populate our database via the shell cqlsh is provided. Below are a couple screenshots of some of the tables in our database.
   2. Screenshot of database table containing users:  
      Screenshot of database table containing list of videos:  
      
2. Load Balancer:

In this project, we change the algorithm of load balancer from the default round robin algorithm to the least connection algorithm. Round robin algorithm is one of the simplest algorithms to deal with the load balancing problem. However, it is not very reasonable when the serving situation is different among the servers. That is to say, if one of the servers could respond to the request slower than others, it will still receive the requests continuously. If we change the algorithm to the least connection algorithm, the load balancer could select the server that has the least active connection. Least connection algorithm could effectively distribute the requests across servers according to their capacity.

In the upstream backend block, we also need to define the public ip address of web servers. Load balancers could pick the ip address based on the defined algorithm. The selected ip address will be further delivered to the proxy cache.



In the server block, the listen port is defined as 80. The proxy\_pass is defined as the place where the load balancer will access the target web content.



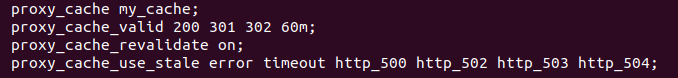
1. Proxy Cache

Use proxy\_cache\_path directive sets the path and configuration of the cache. Here we set the Shared Memory Space(key\_zone) to 10 megabytes, specify a two‑level directory hierarchy, and set the upper bound of the space of cache response by max\_size. Inactive sets the release of cached responses to 5 seconds for display purpose, i.e., cached responses disappear quickly and we could find out the change in original servers quickly. Set use\_temp\_path to off so that the cache will go to the directory specified directly without unnecessary copying.



In the server block, which refers to proxy, we need to activate above configuration using proxy\_cache directive.

Proxy\_cache\_valid enforces the expiration for the cached response and proxy\_cache\_revalidate is turned on. The former is quite different from inactive aforementioned. Think of inactive as erase cached data, while proxy\_cache\_valid only tags the cached response as expired and has nothing to do with the existence of that data. Turn on revalidate so that proxy will go fetch the expired content from the original server automatically.



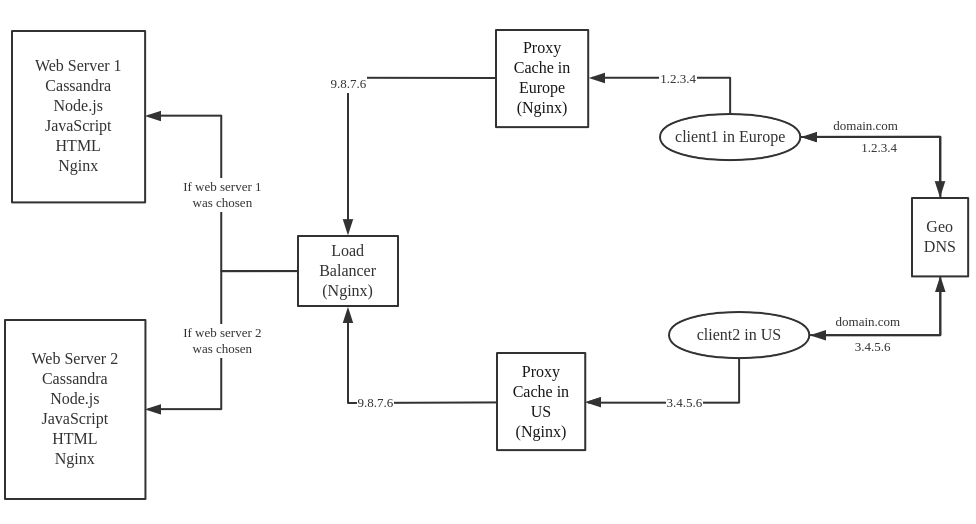
Proxy\_pass specifies the upstream node for this proxy. In our case, it will be the domain name or public IP address of the load balancer.



**Writing** section 3 (your model description) as well as part of Section 4 (experimental result section with some preliminary results)

*Model Description*

The model we proposed consists of EC2, Cassandra database, Web server, REST API, ngrok, load balancer, cache proxy and GeoDNS. After we making some updates on the system, our model looks like the following configuration:



We currently have our Cassandra database and web servers established on EC2 to ensure the static public ip address. In the Cassandra database, we will make it act as a video website. Here each user, video, and comment under the videos will have their specific ids. When the client needs to have access to the contents in the Cassandra database, it could make the request through the REST API in the web server. As the response of the REST API, it will send back the required web page to the client. Basically, each table in our Cassandra database will exist as a html web page. The load balancer sits in the local PC. To acquire the public ip address of the load balancer, we use ngrok here. The public ip address is needed for proxy cache. Both the load balancer and the proxy cache are currently set up on Nginx. When the client wants to access the target content, GeoDNS will assign the client to the DNS server which is the geographically closest to it.

*Experimental Result*

Currently, we couldn’t test our CDN’s performance as the whole system hasn’t been completed; Without the GeoDNS to dispatch the requests coming from clients, we couldn’t take a test on any end-to-end characteristics. However we could determine on which aspects we are going to perform tests. To reveal a CDN’s performance, there are so many features we could test on, and they are all related to time consumption, except for HTTP status code. Here is the list of features we are considering to take a look by the end of phase III: HTTP status code, DNS lookup time, TCP connection time, resolve time. There are a lot of free online tools providing CDN performance tests which cover the above mentioned features, such as Uptrends and CDNPlanet.

**List of papers**:

1. *An Improved Weighted Least Connection Scheduling Algorithm for Load Balancing in Web Cluster Systems*

In this paper, Gurasis Singh and Kamalpreet Kaur say that the load balancer will act like a “traffic cop” that presents between real web servers and end users to transmit the requests. The so-called improved weighted least connection is still maintaining the load among existing servers. This is achieved by preventing transmitting requests to the new real web server. The load balancer will “protect” the web server from working overload by excluding this specific server from the activated server list. This protection will be made only when the number of web requests assigned only to this web server has exceeded the maximum continuous allocation number C. Then after C - 1 allocation, this web server will be activated again and be arranged into the server scheduling list. So in our project, we are considering how to select the load balancing algorithm among so many existing algorithms. The details could be decided in response to the scale of our network.

1. *An Integrated Load-balancing Scheduling Algorithm for Nginx-Based Web Application Clusters*

In this paper, Ruoyu Li et al basically discuss the web application deployment architecture and its relationship with its container method. The analysis is concerned about the host;s load metrics, the host’s performance and the proportion of those free memory which has already been assigned to the container. They come up with the algorithm helping the web application response faster than the ordinary algorithms, like round-robin and least-connection algorithm. This proposed algorithm is called dynamic weighted least connection algorithm. Using this algorithm, we might arrange how the load balancer works dynamically. This will be useful when we deal with the real time problem.

1. *An Optimized Round Robin Scheduling Algorithm for CPU Scheduling*

In this paper, Ajit Singh, Priyanka Goyal, Sahil Batra talk about how to improve the ordinary round robin algorithm. For the existing algorithm, people usually want to maximize the throughput, avoiding indefinite postponement and starvation, minimizing overhead, achieving balance between response and utilisation etc.. To improve the round robin algorithm, the authors reduce waiting response time, and it is usually achieved by having less preemption and context switching.

1. *Bringing Innovative Load Balancer to NGINX*

In this paper, AdamSchwartz and EarlhamCollege discussed their ideas regarding the load balancer in NGINX. They make comparisons among the random algorithm, the round robin algorithm and the newly proposed “two-choices” algorithm. Although the random algorithm and round robin algorithm are enough for many cases, they do have some drawbacks when being implemented. The “two-choices” algorithm makes the client only compare two queues rather than pick the best among all choices. This could prevent the current best choice getting collapsed if many clients are picking the same best one at the same time. In our project, we would also make some improvements to our load balancer algorithm. This paper could help a lot as expected.

1. *Understanding Open Proxies in the Wild*

This paper introduced a methodology to measure characteristics on open proxies for the purpose of learning and analyzing their behavior. To understand the usage pattern of open proxies, authors firstly discovered proxies by crawling aggregator sites, then collected traffic-related data via probing, finally revealed why and how open proxies are utilized. Analyzing how proxy clients distributed also helped us understand the workload experienced by open proxies in the wild. Based on these usage characterizations, developers could come up with more secure forms of anonymous systems.

1. *Request Redirection Algorithms for Distributed Web Systems*

This paper describes a combination of DNS dispatching and request redirection mechanism to help direct traffic for distributed web systems, which could either be centralized control or distributed control. On the basis of our system design, the methodology described for centralized schema would be useful. Two most important kinds of algorithms this paper designed and introduced are redirection activation algorithm and node localization algorithm. This paper not only provided implementation but also offered models to compare the performance of those algorithms.

1. *Prefix based Chaining Scheme for Streaming Popular Videos using Proxy servers in VoD*

This paper proposed a combination of prefix caching technique and peer-to-peer client and proxy chaining to achieve efficient video-on-demand service. I would say the key word for this paper is chaining. Although chaining is one of the solutions to address the problem of dealing with too many asynchronous requests, in lots of existing work, chaining is not considered. This paper added chaining in prefix caching, which will further reduce aggregation transmission cost. The proposed architecture also makes proxy servers interconnected (another way of chaining) so that load sharing algorithms could be deployed among proxy servers and cost drops by decreasing the usage of expensive server-client path and taking advantage of relatively cheap proxy-client path.

1. *Web Application Hosting in the AWS Cloud Best Practices*

Web servers do not have constant usage. Their traffic is constantly changing. So AWS hosting websites need a mechanism to scale up or down depending on the traffic of the website at a given time. AWS offers something called Cloudfront which handles setting up cache proxies in an effective way. AWS’s Elastic Load Balancing handles the changes in usage of web servers by monitoring traffic and individual server’s usage to dynamically scale the number of needed instances up or down. For example, when CPU utilization gets to a certain threshold, a new instance will be added to reduce the workload.

1. *Performance Evaluation of Cassandra Scalability on Amazon EC2*

This paper analyzes the effects of scaling a cassandra database on an EC2 cluster on AWS. Since simply adding a node should not have much latency effect, this paper analyzes what happens when replication factors and consistency are still considered. When comparing results against MongoDB and Hbase, Cassandra outperformed both of them. Another observation is that increasing the replication factor increases latency when you scale up which makes sense. The auto scaling technique used in their experiment reduced latency of scaling up.

1. *Performance evaluation of Cassandra in AWS environment*

Cassandra is a good match for AWS because it supports automatically handling replication as the cluster scales and as nodes begin to fail. It was designed by Facebook to efficiently scale up for their huge amounts of data and for it to be able to handle failures at any point in the system. This paper goes into great depths about the features of both Cassandra and AWS. then goes to analyze different experiments that used different instance types on AWS to see how performance played out as the cluster scaled up in size. Surprisingly though, sometimes Cassandra did not offer better performance as the system scaled up. Yes it was able to reduce CPU utilization and the throughput, but these benefits were outweighed by the overhead associated adding a new node causing Cassandra to have to handle replication and consistency factors.