Practical Distributed Systems Notes

Research Notes

theroyakash*
Microsoft Corporation & Indian Institute of Technology Madras
hey@theroyakash.com, cs22m007@smail.iitm.ac.in

Last Compiled: December 8, 2023

Abstract

This note contains my understanding of the practical aspects of distributed systems. We cover various concepts starting from designing most of the basic systems, then we move on to cover storage systems (relation, non-relational), storage engines, designing high throughput systems, and information retrieval systems. At the end we disect some unqiue research topics in the distributed systems space.

1 Introduction to basics

We'll quickly go through the basics so that we can deep dive into the most interesting parts.

1.1 Scaling

Every other book on distributed systems starts with what it means to have a distributed systems, and when we should not implement / use a distributed system. This note will not start with that. We start with scaling. There are generally two types of scaling

- Horizontal scaling
- Vertical scaling

As we are discussing distributed systems by scaling for now on we'll only mean horizontal scaling. A horizontally scaled service can be scaled up or scaled down depending upon the traffic hence the cost can be adjusted as the amount of requests goes up or down. These are probably the best reasons as to why horizontal scaling is much more preferred than doing vertical scaling. If the user base is located in one single region (for example online GATE coaching) then a powerful server in the middle of the country is good enough for all India customers. But here we'll discuss internet scale. Hence there is no point in discussing advantages / disadvantages of vertical scaling.

1.2 Load Balancers

Think of load balancers as the request mapper to a server. Suppose we are Horizontally scaled and we have ten to fifteen Azure compute engine, where the applications are running. Now how can you distribute the requests into those servers such that each of the server works roughly the same?

We introduce a concept called **load balancing policies**,

*This note has been prepared during my masters in computer science at IIT Madras, as well as when I was a software engineer at Microsoft. This is affiliated with none. This work has been heavily inspired by the Arpit Bhayani's Course on Distributed Systems.

- Balancing according to number of requests: With predefined algorithms like round robin, least connection etc, we can distribute the incoming requests into our server evenly so that no computer is overworked or underserved seating idle,
- Balancing according to the network IO: Suppose our servers creates a long-lived TCP connection with the client and streams some (song / video). Then instead of the amount of computation used by the server, we can distribute the incoming requests by amount of network IO used. Then we'll have even amount of work set out for every node in the system.

By looking at the above policies, we can see that load balancers are a type of layer positioned just before the actual server. They route incoming requests to the appropriate server.

Hey reader, here is a **brilliant** use case. Suppose our API server generates some kind of authentication token that are short-lived and our API requests are stateless. Hence every request must store some credential in the header of the request for validation. We can store the recently validated / generated token into the memory of the load balancer. Now we can verify every request and route only the requests with proper authentication. We can also store a counter for each unqiue token in the in memory database in the load balancer. This way we can count the number of requests coming in from a particular token. This will be helpful during rate-limiting and mitigating DDoS attacks. As you can probably see it, we can add or remove API servers easily from the list of servers as we see fit if we have a load balancer. The load balancer will automatically adjust itself to distribute requests to fewer / more nodes.

Here is one small **question** for you to brainstrom.

Question 1.1: Update the design of the current system

Load balance a system (say this is a Python & Flask application) that has the following few APIs for customers to use. Some initial configurations and some observed metrics are given to you to decide few design decisions.

- *We currently have 3 servers, India, US, and EU one each,*
- *This service in US and EU is not that popular, but India business is booming*
- *Our India server configuration is 16 GB RAM + GPU and 128 GB SSD,*
- *Some usage metrics: India server is using 80% GPU 90% of the time to calculate some predictions, 90% CPU 95% of the time and 95% of RAM 99% of the time.*
- *`uploadImage()` uses significant network, `inferTopicFromImage()` uses GPUs and `calculatePredictionQualityForMetric(Metric M)` judges the quality of the predictions for some metric `M`, this uses lots of CPU,*
- *`calculatePredictionQualityForMetric(Metric M)` API is invoked multiple times for multiple metric calculations. Say for every prediction output from `inferTopicFromImage()` there will be 20-40 average `calculatePredictionQualityForMetric(Metric M)` call.*

Our users are facing considerable amount of lag. Fix this.

Solution: Let's look at the India server. As we are seeing 95% of RAM usage 99% of the time, that means we need more memory. Let's upgrade that to 32 GB of RAM. We are also told that our users are facing considerable amount of lag. That means our users are not served with the computation as soon as possible. Means they are waiting for the query to start executing (scheduled on the maching). This is a good sign that we should start horizontal scaling. We'll triple the number of India servers. Now to evenly distribute the computation among the three India servers we need good load balancing policy. Take a second to come-up with a good load balancing policy for this specific problem.

As you can see from the metrics and the API definitions some API uses CPU a lots, some API uses GPU a lots and some API uses network IO, memory a lots. It'll be a fools implementation to have very high CPU / GPU and RAM for all the API servers that we have.

- Let's have 2 low CPU but high GPU server
- Let's have 4 extremly fast CPU server with no GPU
- Let's have 32 GB RAM in all of them.

Now we write the policy for the load balancer. All the `inferTopicFromImage()` calls will be routed to the GPU servers. They'll be load balanced between those two only.

As for every predictions there will be 20 - 40 many call to `calculatePredictionQualityForMetric()` that means we are needed much more CPU intensive servers than GPU intensive servers. Through trial and error we can reach a good balance. For example here I say 4 is enough.

We add 32 GB RAM in all the machines. We reduce them if necessary after carefully monitoring the RAM usage. Depending on the URL (to identify what API is called) the load balancer will redirect the request to corresponding target group. □