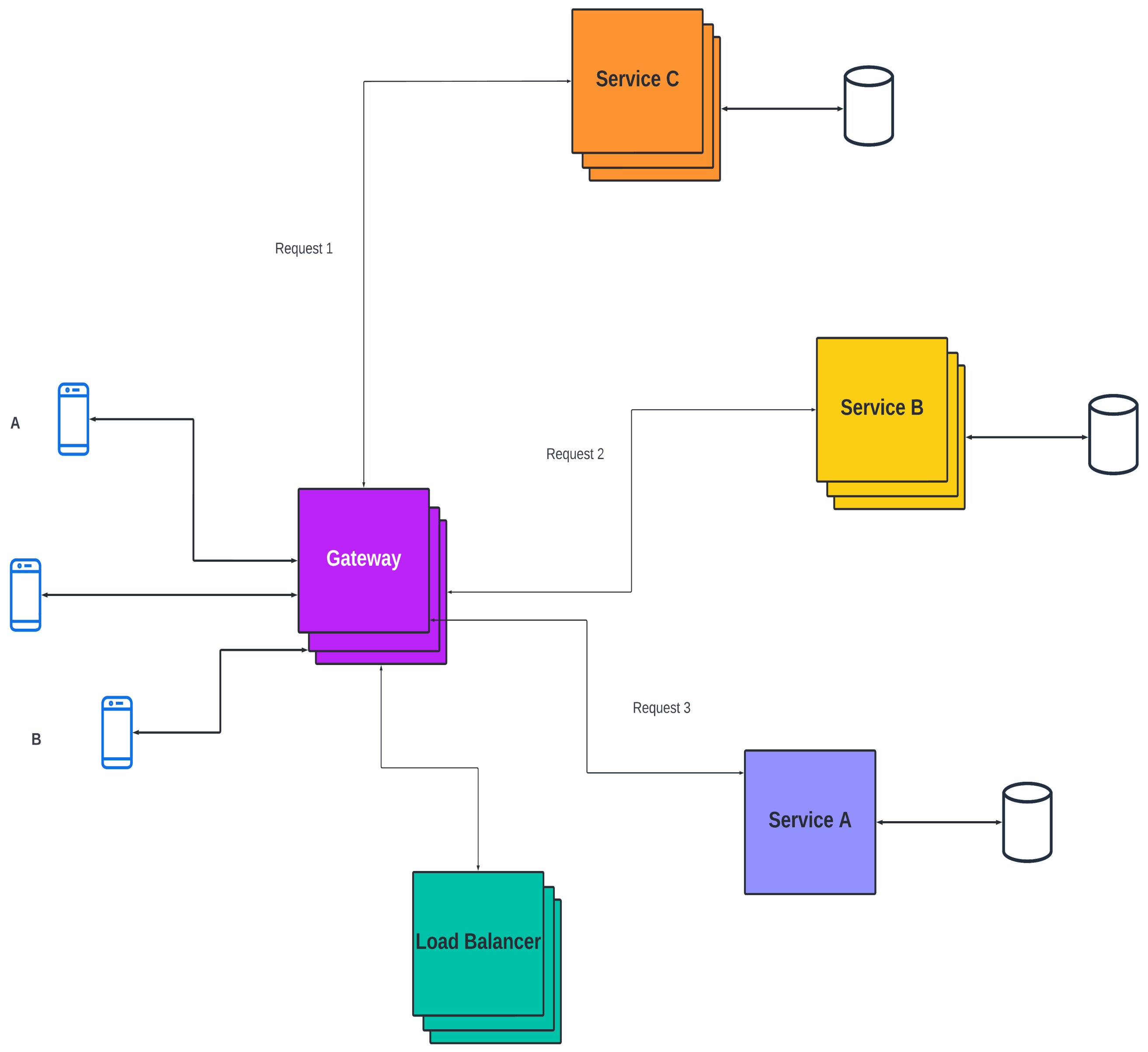
INTERVIEW READY

Load Balancers

What is load balancing ?

In a distributed system, load balancing refers to the distribution of incoming traffic across multiple servers.



We need to use load balancer in our system because:

1. It distributes client requests across multiple servers ensuring that no server is overworked. This ensures **high availability** of our system. It also prevents **traffic spikes** on a single server.
2. It provides the flexibility to remove servers which makes our system **fault tolerant** and **scalable**

3. It reduces the **response time of our system**. (Since our servers are not overworked our performance is not degraded)

Classification of Load balancing algorithms

There are 2 types of load balancing algorithm.

- **Dynamic Load Balancing algorithms :** These algorithms **consider the current state for each server** for distributing the incoming traffic.
  The algorithms are considered complex, but have better fault tolerance and overall performance. Also, these algorithms are extremely efficient when execution time varies greatly from one task to another.
  Dynamic Round Robin, Least Connection, Weighted Response Time are few examples.

- **Static Load Balancing Algorithms :** These algorithms do not consider the current state fo the servers. These algorithms are very simple and are very efficient in the case of fairly regular tasks.
  Round Robin, Consistent Hashing, Weighted Round Robin etc are few static algorithms.

Simple Hashing

How it works?

- There are **N** servers.
- Each server is labelled from **0** to **N-1**
- Output of hash function **H(x)** is uniformly random , where **x = Client Request ID**.

Every time we get a request we find the value of **H(x)%N**. This gives us a value **y** (which is between 0 to N-1). We then route the request to server y.

Considering the output of hash function to be uniformly random, if we have **M** requests, then each server will have at get **M/N** requests.

So the load factor in this case is **1/N**.

Limitations of this approach

For the same input, hash function always gives the same output. So the same client request is directed to the same server. To improve the performance each server uses cache to store relevant information. For e.g., if **client x** is directed to **server y** then **y** will cache some data for x.

But what happens when we add a new server. Now we need to find the value of **H(x)%(N+1)**. This will cause a lot of client requests to be routed to different servers. Due to this almost all servers have to recompute the cache data.

It is very inefficient to recompute the cache every time we add or remove servers. Therefore this method **does not provides the flexibility to add or remove servers**.

On average, Number of remappings = Number of requests.

Consistent Hashing

Consistent Hashing improves upon simple hashing by reducing the reallocation of requests to servers when a new server is added or removed. Therefore it provides the **flexibility to add and remove servers**

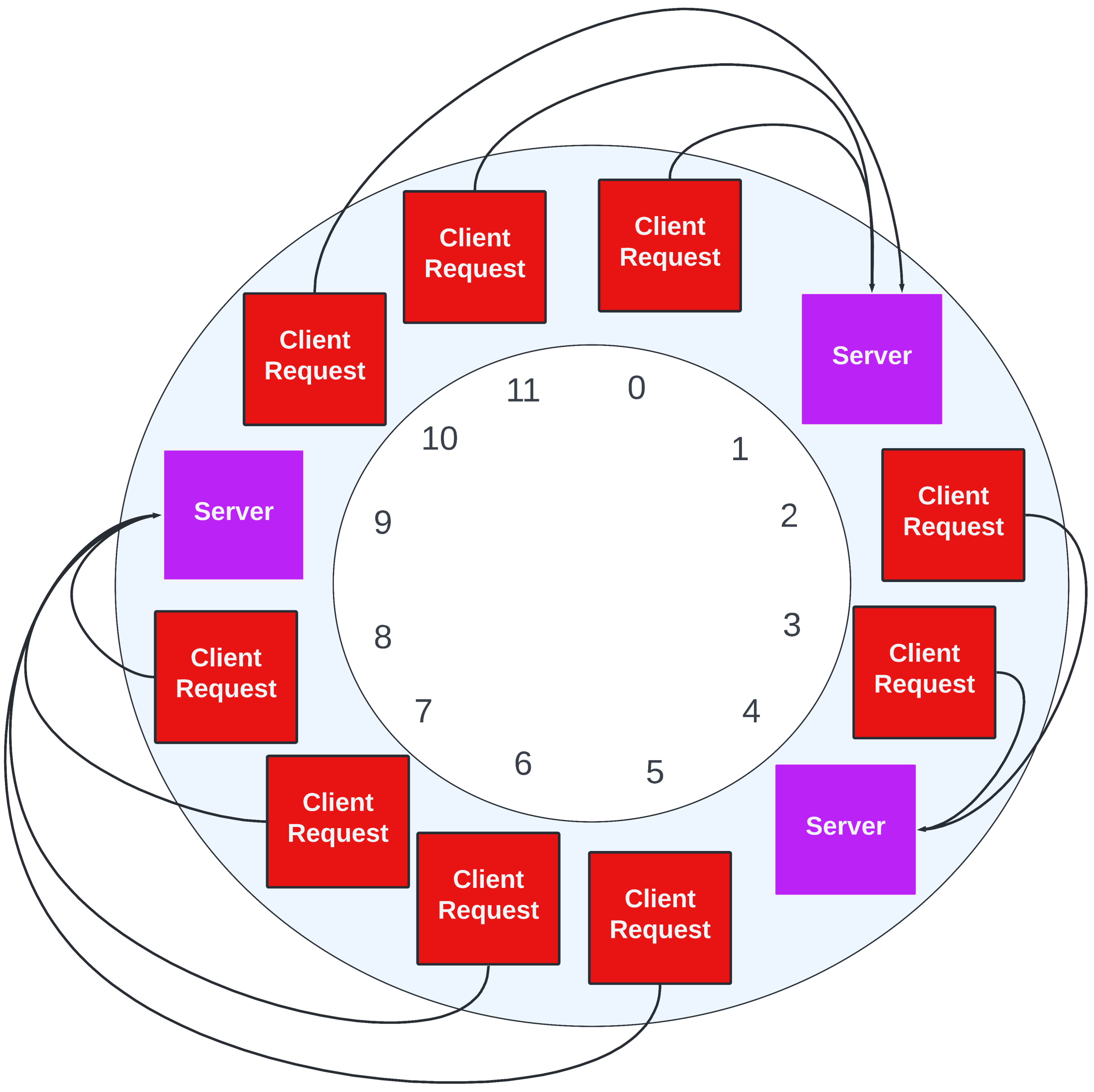
How it works?

In simple hashing we map each client request to an array ranging from **0** to **N-1**. In consistent hashing we have a **circular array** ranging from 0 to N-1.

Each client and server have IDs. So we hash these IDs and place the client and servers on the circular array.

For each client request **we go clockwise and find the nearest server. And route the request to that server**

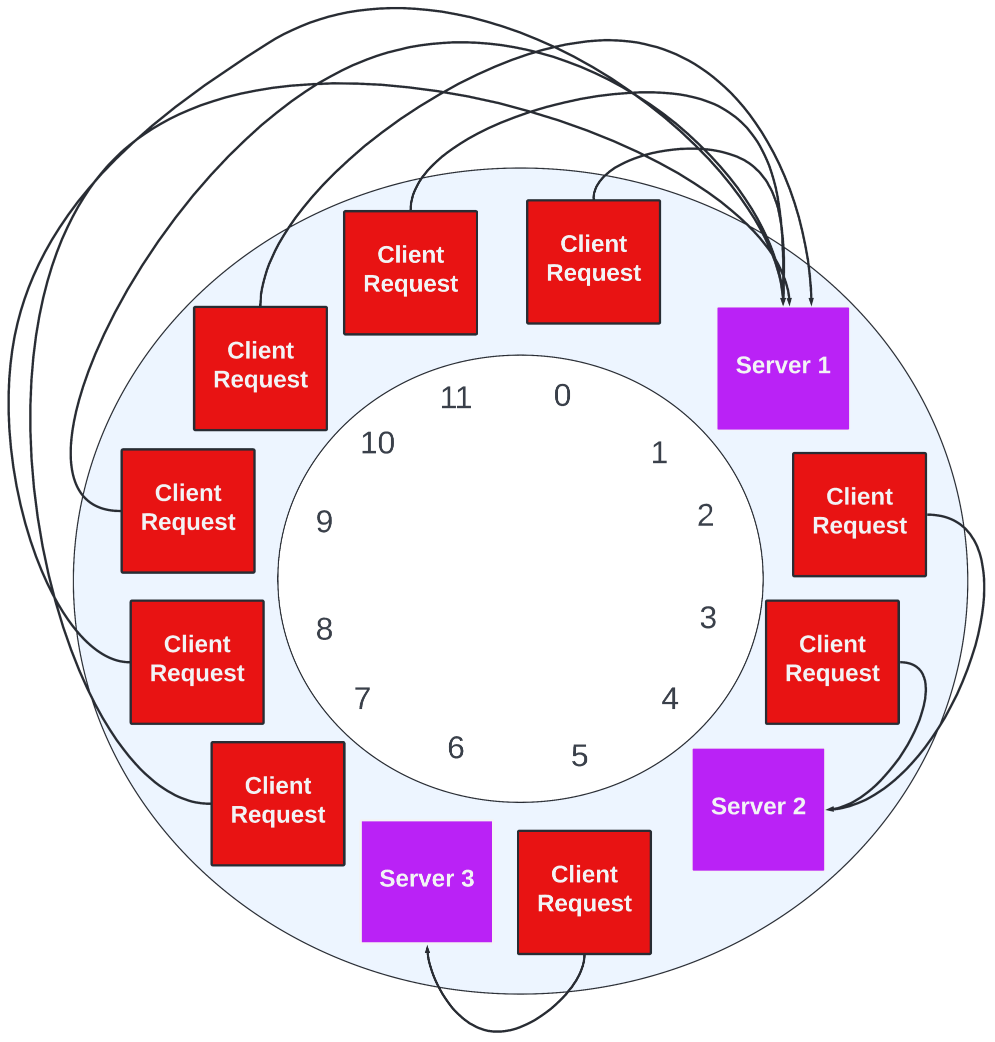
So the load factor (on average) in this case is **1/N**.



Circular array of size 12

Limitations of this approach

- Although load factor is 1/N in average case, **practically we may have skewed (uneven) distribution.** This might cause a few servers to be overloaded with a lot of requests while other servers are idle.
  Consider the example below



Skewed distribution

Server 1 receives 6 client requests whereas server 3 receives only 1.

- This approach won't work if servers have different capacities since it distributes the load evenly.

Solution to prevent skewed distribution

To solve this we can use the concept of **virtual servers.**

We want to increase the number of servers but getting actual servers is expensive, so for each server, instead of 1 we can generate K server IDs. We then hash these K IDs and place them on the circular array. Everything else remains the same. This **reduces the chance of load being skewed**.

Round Robin

How it works?

In this method, client requests are routed to servers in a cyclic manner. When it reaches the last server loops back to the first server and continues the process.

E.g.,

Request 1 is routed to Server 1. Request 2 is routed to Server 2. Request 3 is routed to Server 3. Request 4 is routed to Server 1 and so on.

Limitations

- This algorithm assumes that servers are similar enough to handle equivalent loads. This might cause the servers with less capacity to overload and fail more quickly while other servers remain idle.

Weighted Round Robin

Weighted Round Robin is used when we need to route requests to servers having **different capacities**.

How it works?

In this algorithm each server is **assigned some weight.** Assigned weight is **directly proportional to the capacity of the server.**

It then allocates requests in a cyclic manner similar to Round Robin.

e.g.,

Server **X** is assigned weight 4. Server **Y** is assigned weight 1. Server **Z** is assigned weight 2.

Suppose we receive 15 requests.

Server **X** is routed 8 requests. Server **Y** is routed 2 requests. Server **Z** is routed 4 requests. Server **X** is routed 1 request.

Least Connection Algorithm

How it works?

It is a **dynamic load balancing algorithm**. It finds out the number of active connections on each server. It then routes the request to the server with the lowest connections.

If there are multiple servers with lowest connections then it follows the round robin approach.

E.g.,

Server A has 3 active connections, B has 2 active connections and C also has 1 active connections.

Suppose we receive 3 client requests.

Request 1 is routed to server C. (Now C has 2 connections) Request 2 is routed to server C. (Now C has 3 connections) Request 3 is routed to server B. (Now B has 3 connections)

Limitations

- It has a complex algorithm compared to static load balancers.
- It needs more processing power.
- It only considers the number of connections not the capacities of the servers.

Weighted Least Connection Algorithm

How it works?

It is similar to Least Connection Algorithm except each server is assigned a weight. Weight of the server is directly proportional to its capacity.

If there are multiple servers with lowest connections then it routes the request to the server with the largest weight.

e.g.,

Server **A** is assigned weight 7. Server **B** is assigned weight 4. Server **C** is assigned weight 3.

Initially,

Server **A** has 3 active connections. Server **B** has 2 active connections. Server **C** has 2 active connections.

Suppose we receive 3 client requests.

Request 1 is router to server **B**. (It has least connection and weight(B) > weight(C)).

Request 2 is routed to server **C**. (It has least connection)

Request 3 is routed to server **A**. (It has least connection and weight(A) > weight(B) > weight(C))

Limitations

- Needs more processing power and the algorithm complex.

That's it for now!

You can check out more designs on our video course at InterviewReady.