What Distributed Systems Achieve for us?

M T W T F S S
Page No.:
Date:
YOUVA

* Characteristics of a Distributed System *

1) Fault Tolerance:-

Anything that can go wrong, will go wrong
— Edward A. Murphy Jr.

To build a robust system, your system needs to be able to handle adverse scenarios. A developer has to put extensive effort into achieving fault tolerance.

From a distributed system's perspective, when things go wrong in any part of the system, we say that there are faults.

A few faults :-

* One of the nodes running server program is suddenly handling too many connections. As a result, the node runs out of RAM and crashes.
* Network communication between two database nodes dropped which means the data sync among your database nodes has stopped.
* Hardware components like a disk drive or logic board chips got damaged in any of the nodes of your system.
* Your server nodes are unable to handle a sudden spike in traffic, resulting in crashes.
* There was a new feature added to your app. The code that was deployed contained a nullPointerException bug which caused the crash of server programs running on the nodes.

The idea of being able to tolerate faults in a distributed system is called Fault tolerance.

Why we need fault tolerance?

Things can go wrong in distributed systems such as Hardware failures, software bugs, network drop or maybe the system is fine but the third party dependency faces an issue (such as API contract changed)

Faults usually happen in system and a small fault can bring down whole part of the system where it occurred.

So, Distributed systems have to be able to tolerate faults so that they can continue to function even if faults appear Distributed systems have to be fault tolerant by nature

2) Reliability :-

Reliability is the first pillar of fault-tolerant systems.

What is Reliability?-

A reliable system is capable of "continuing to work correctly, even when things go wrong."

System should be capable of handling the following:

→ The system can serve users' expectations. For example, if user uploads a photo and that photo is not shown on their profile, this leads to bad user experience.

→ If user makes mistakes, system should tolerate them. For example, if user is allowed to upload only photos but uploaded video, the app should show some error message instead of breaking the flow.

→ The system should be performant. For example, If user opens feed and waits for 2 minutes for it to load, users wont be very happy.

→ If some malicious user tries to abuse your system, the system should endure it.

A reliable system should be built that should handle failure scenarios such as hardware issues, network drop, power outages etc.

Note: When faults occur, a reliable system is capable of identifying, isolating and potentially correcting it by itself. If correction is not possible, it should trigger a higher-level recovery mechanism, for example, triggering fallbacks to replacement hardware or software components. If that does not work, it should halt the affected program, and stop the entire system if required. And notify the system owner or developers accordingly.

A reliable system never continues to operate on corrupted data without any recovery effort. If this happens, the system is unreliable.

* What hinders reliability?
  - Hardware faults
  - Software faults.

Hardware faults:-

Hard disks are reported as having a mean time to failure (MTTF) of about 10 to 50 years. Thus, on a storage cluster with 10,000 disks, we should expect on average one disk to die per day.

There are many hardware components on a single machine. The chance of failures just increases when you have tens of thousands of machines. Also networks that bind the nodes are unpredictable too.

Software faults:-

Machines run server programs to serve user requests. And programs tend to have bugs.

Due to software bugs, the server program may crash and restart. Due to misconfiguration of server startup parameters, program crashes every time it tries to restart. Sometimes bugs are not generic. You won't encounter them until a very specific set of parameters are passed to a particular function. If you don't know the specific set of inputs, it may be even impossible to reproduce the bug. Sometimes, bug can also be due to third-party library.

3) Handling Hardware and Software Faults :-

* Handling Hardware Faults :-

For disk failures, it's a very common practice to add redundancy by having more than one disk drive to store the same data.

Arranging multiple disks to store the same copy of data is called RAID (Redundant Array of Independent Disks)

RAID scheme works when downtime is tolerable. If things go wrong, the system will be down and won't be available for the time being. But the redundancies will ensure correctness as the data is not lost - and the system will not serve users with any corrupted data.

On the other hand, in a distributed environment, we don't want to let our whole system stop working every time there is a random node crash. We want it to be "available" all the time and at the same time "reliable".

For availability: multi-machine redundancy
& reliability

With multi-machine redundancy, we achieve two things here:
① When a node crashes, other nodes can serve ⇒ Availability.
② There is no data loss. Other nodes can serve the users with correct data ⇒ Reliability.

Conclusion: Create redundancy - to handle hardware faults.

* Handling software faults:-

① Write tests
- Unit tests
- Integration tests
- End-to-end tests

② Handle errors

③ Monitor critical parts.
↳ add logs.
↳ you get notified of any abnormality to react immediately & ensure fault tolerance.
↳ can use third-party GUI which you can see all kinds of statuses of your system.

4) Availability :-

While reliability ensures correctness of the system, availability ensures that the system is functional.

Availability is a property of a distributed system that ensures that the system is ready to serve users whenever users need it.

U - uptime D - downtime A - Availability

$$A = \frac{U}{U+D}$$

If A% is 99% ⇒ system is down for 1% ⇒ 3.65 days. For many systems, 3.65 ≈ 4 days of downtime over a year is bad.

Note:-

When we define availability, correctness is implicitly assumed. If the system was up all the time but the data was lost or corrupted, then that does not really mean the system was serving the users as expected.

* Motivation behind available systems

Applications like Amazon want to be highly available. Because e-commerce websites can sell more products when available all the time.

Similarly, any other websites want to be generally up, so that users can use them.

* The availability chart

Availability Level	Allowed Downtime
90%	36.5 days.
95%	18.25 days.
99%	3.65 days.
(2-and-half nines) 99.5%	1.83 days.
99.9%	8.76 hours.
99.95%	4.38 hours.
(4-nines) 99.99%	52.6 minutes.
~~(2-and-half nines)~~ 99.999%	5.26 minutes.

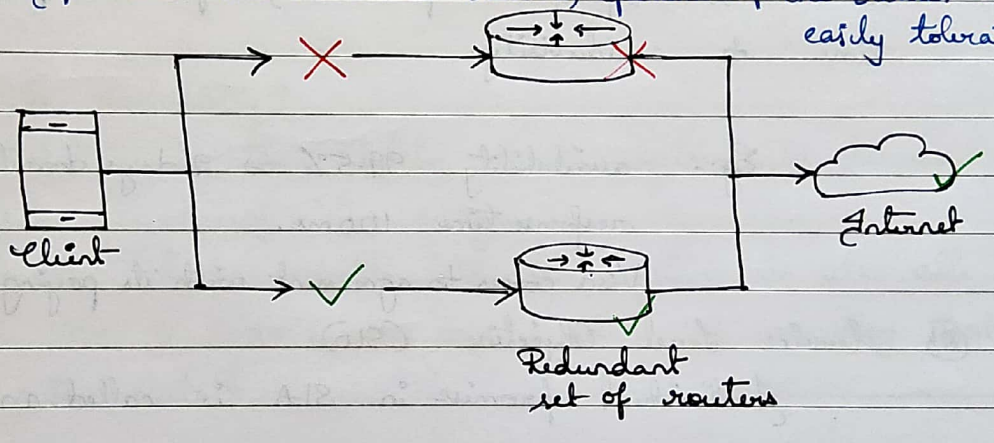
5) Achieving Availability :-

* How to achieve availability in distributed systems?

① Add redundant resources – already discussed.
② Avoid SPOF.

SPOF :- Single Point of Failure (SPOF) in a distributed system means a component that can bring the entire system down if there is any failure in the node itself.

Fig:- If there are redundant routers, failure of one router is easily tolerated.



* Setting up expectations :-

⇒ How far should you go as a distributed system owner?

Assume your app has 5 nodes running the server program. Due to well-designed architecture, the system is capable of losing 1 node or 2 nodes. What happens if 3 nodes crash? What if all the nodes crash at the same time?

⇒ Somewhere down the line, you need to set expectations

Any hardware dies at a point. There can be bug that can cause cascading failures. Even systems owned by Google and Amazon face outages.

So in practice, distributed systems define some set of expectations and communicate them to their clients. That's where the availability percentage measurements are used.

* How to define expectations?

① Service Level Agreement (SLA):
SLA is an agreement between the system and its clients on specific requirements, for example, response time or availability.

Eg:- availability 99.5% ⇒ 2 days downtime over a year.
response time 100ms.
Visa comes to agreement with its paying customer as 99.99

② Service Level Objective (SLO):
Individual promise in SLA is called an SLO.

Eg:- Visa could have an SLO of procesing each transaction within 2s.

③ Service Level Indicators (SLI).
SLI is the actual measurement in reality.

Eg:- Visa SLA: 99.99 SLI: 99.95

6) Scalability:-

So far we discussed about achieving Availability and Reliability.

But a system that is available and reliable today won't necessarily be the same tomorrow.

If we have an app that currently has 100 concurrent users (users connected to your system at the same time), but what if it has 10K or 100K concurrent users tomorrow? The system is mostly likely does not not remain reliable and available.

Hence, we learn about scalability.

What is Scalability?

Scalability is the ability of a system to handle the increased load of its usage.

In the above example, if the system is also able to handle 10K or 100K user count, then we can say that the system is scalable.

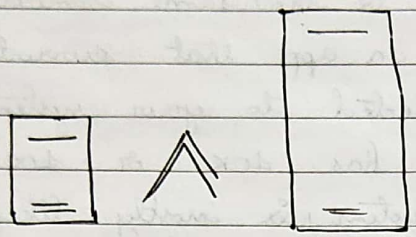
Scalable system is an answer to certain questions:

1. If the user base increases, how does that affect our system?
2. Can the system cope with the increased count of users, i.e., the increased load?
3. How can we add computing resources to handle the load?
4. Can we add resources seamlessly so that there is no visible impact from the users' perspective?

* How to achieve scalability?

① Vertical scaling aka scaling up.
② Horizontal scaling aka scaling out.

① Vertical scaling aka scaling up :-



Fig, In vertical scaling, a node is upgraded with more resources.

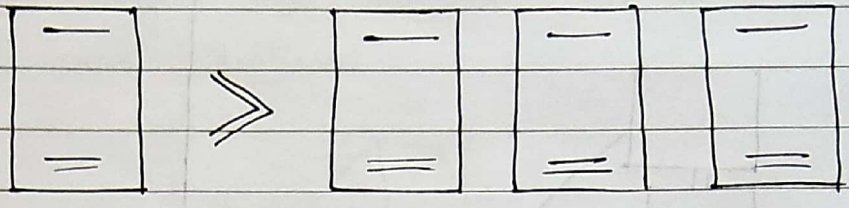
Vertical scaling can mean different things in the context of different systems. For example:

→ A node unable to handle many connections can be scaled up by adding more RAM and CPU.
→ A database node unable to store more data can be scaled up by adding more storage.
→ A router incapable of supporting an increased count of devices could be scaled up by replacing it with a more powerful router.

* Vertical scaling may work for small systems with not-so-heavy increased load.
* Eventually, it becomes infeasible with the system's growth:
  → Continuously buying or renting a bigger machine is expensive. Hardware also has its own limitations.
  → One machine is a single point of failure (SPOF)

"Scope of Vertical Scaling is limited".

② Horizontal scaling aka scaling out:



fig, In horizontal scaling, load is distributed among multiple nodes.

* Add more nodes to handle increased load - horizontal scaling.
* For different systems, this has different implications:
→ A node running a server program can be scaled out by running the same program in multiple nodes and distributing the load among them evenly.
→ A database is scaled out by adding replica nodes, commonly for reads.
→ A network is scaled out by setting up multiple routers to work in sync.

Horizontal scaling is more or less the industry standard and large systems depend on Horizontal scaling.

* Cost effective → cheaper hardware with added redundancy.
* Seemingly infinite capacity.
* can easily avoid SPOF.

We evenly distribute the load among horizontally scaled nodes. This is achieved with "Load Balancing".

7) Load Balancing in Distributed Systems :-

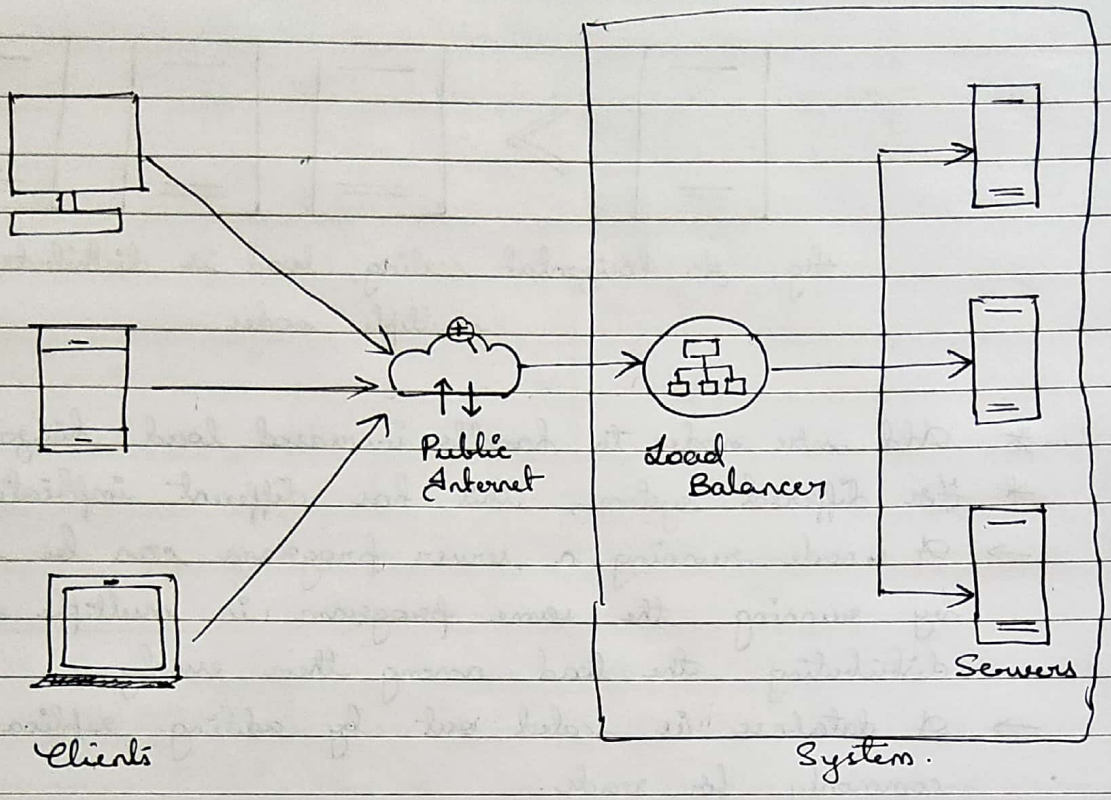


Fig., A load balancer balances the load among multiple servers so that no one server is overloaded.

Definition :

A load balancer (LB) is basically a node in a distributed system that tries to evenly distribute traffic among the actual server nodes.

The diagram shows a common scenario of load balancing in distributed systems :

↳ Requests from clients come to your system via the public internet and hit the LB. Basically, the LB is sitting in between the clients and the servers.

↳ The LB routes requests to one of the server nodes based on some predefined algorithm or a combination of algorithms.

↳ Because of the LB, it is now easier to control traffic and distribute the workload.

* Health checks and high availability using LB:

With proper usage, load balancers will make a system robust and resilient.

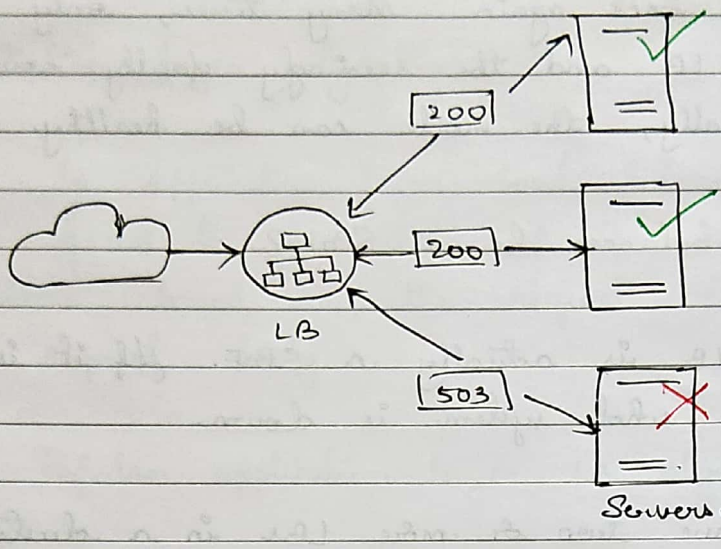


Fig., Unhealthy nodes can be detected by load balancer

LB needs to know which nodes are available to forward a request to, otherwise a request might just be dropped & take more time and resource than required.

In above diagram, LB sends request to all servers every 30 seconds to check its health of each server, like by using a /status endpoint and take further actions to make sure the system remains highly available.
Some of these actions are:

↳ Stopping the forwarding of requests to unhealthy node. Users won't see a longer response time due to retry and node failures.

↳ Triggering alerts to system owners.

↳ If required, initiating the process of spinning up a new node automatically. This helps to achieve high availability in the system when node failures occur.

↳ At the same time, LBs can help to scale out the

system based on load. If the LB sees more requests than a particular threshold, it can bigger the initiation of more nodes.

↳ Periodically checking the status of the unhealthy node whether it's back again. Many times, only the network between the LB and the seemingly faulty node is the issue. Originally, the node can be healthy.

* Can a load balancer be a SPOF?

A single LB is actually a SPOF. If it is down, then essentially whole system is down.

Solution: Have two or more LBs in a cluster node. Then let IP addresses of these balancer nodes propagate to DNS servers.

When clients try to connect to your backend, they get all the three IPs and randomly connect to one of them. If one is down, clients can try to connect to the other one.

8) Load Balancing Algorithms :-

* application-layer algorithms
* network-layer algorithms

* Application-layer algorithms :-

In application-layer load balancing, the load balancer has access to the data of the request. It can take decisions based on the request headers as well as the request body.

Popular application-layer LB algorithms:

* Hashing
* Endpoint evaluation.

** ① Hashing :-

The LB can hash a set of pre-defined attributes and generate hash value. The hash value is then mapped to one of the server nodes.

Example:
Request body has attribute user_id. LB does following steps:

1. Hash user_id using any pre-configured hash function.
   user_id = abc123 ⇒ hash value = 981723123.
2. Map the hash value to a server node, using mod. If there are 3 servers, hash(user_id) % 3 = 2 (let's say).
3. Route request to node 2.

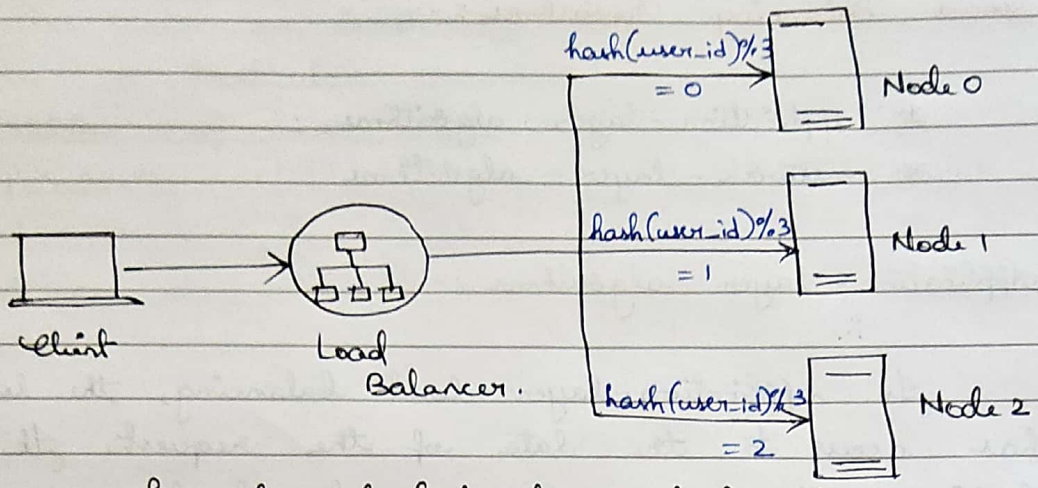


fig., how hash-based LB looks like.

② Endpoint Evaluation:-

In this mechanism, the endpoint of the request is considered and routed accordingly.

For instance, for two endpoints /v1/app/photos and /v1/app/videos, you might have two set of servers serving photos and videos respectively.

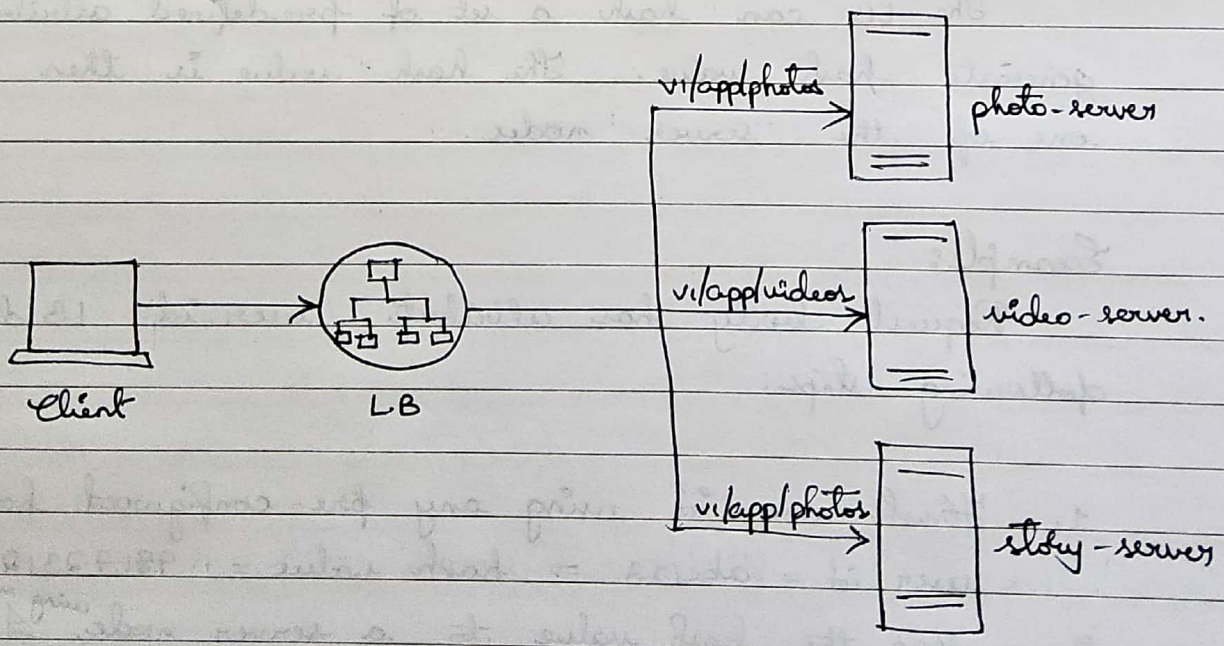


fig., In endpoint evaluation, different endpoints are served by different server nodes.

M T W T F S S

Page No.: YOUVA

Date:

* Network - layer algorithms :-

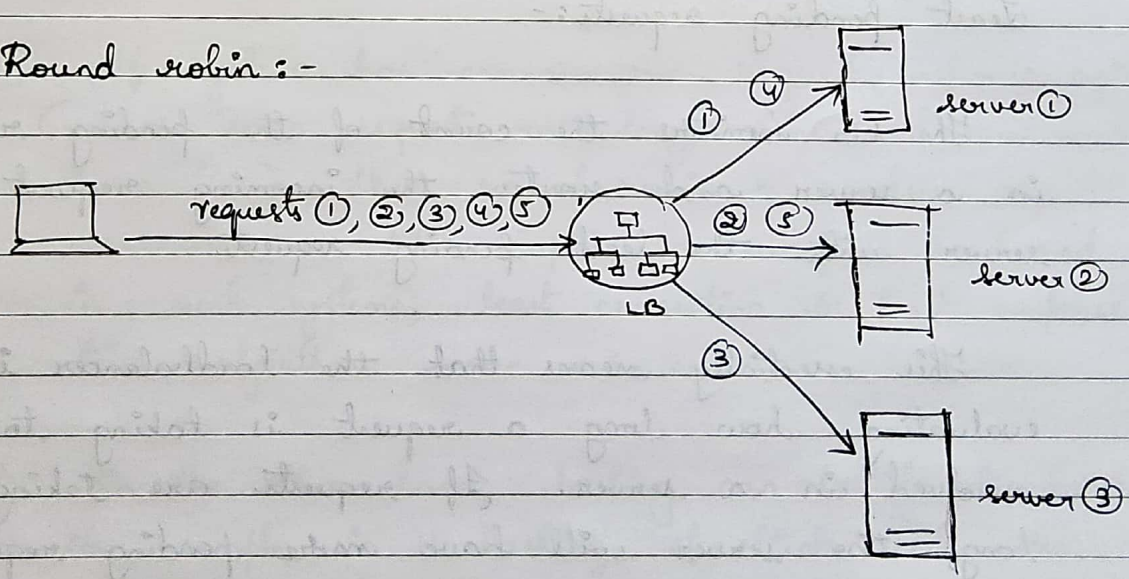
Load Balancer only sees source and destination of request but not what is there in the request.

* Random selection
* Round robin
* Least connection
* IP hashing
* Least pending.

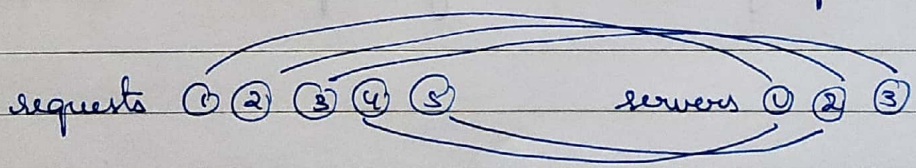
① Random selection :-

→ sends request to any of the server nodes.
→ calculates probability for choosing server nodes.
→ sometimes a sophisticated random number generation is used to choose server node.

② Round robin :-



Requests are distributed in Round robin fashion.



The LB will keep a state of the server and route request accordingly.

③ Least Connection :-

LBs keep track of how many persistent connections there are in the server nodes and choose the one with the least number of connections.

** In systems where persistent connections are common (think of chat application backends), least connection can be a beneficial choice for load balancing.

④ IP Hashing :-

source IP is hashed. Base on the hash, request is routed to specific server.

⑤ Least pending requests :-

The LB monitors the count of the pending requests in a server and routes the incoming request to the server with the least pending requests.

This essentially means that the loadbalancer is evaluating how long a request is taking to be resolved in a server. If requests are taking too long, the server will have more pending requests. As a result, the LB will try to optimize requests by sending new incoming requests to the less busy servers.

* Which algorithm should you choose?

This is very system-specific. One algorithm working for a system does not necessarily mean it will be similarly suitable for other systems. For instance:

① You first need to decide which type of load balancing is needed for your system: network-layer or application-layer. Based on that, you can choose a particular algorithm.

② If a system has very short request-response patterns, then round-robin or random both can be suitable algorithms.

short request-response pattern ⇒ short time frame to complete a request & response

③ If system has non-uniform request-response patterns (means some requests need longer time and some need lesser time), then round-robin & random are unnecessary. This implies non-uniform load and in such systems, least connection or least response time algorithms make more sense.

④ On the other hand, hashing can also be disadvantageous. For example, in IP hashing, if a client sends more requests, means many requests route to same server for same client, then it puts more load on the server.

9) Measuring Load and Performance:-

To scale a system, you need to define the load and performance of your system. Based on the numbers you see, you decide whether to scale your system, and how far you should go with scaling.

* Measuring Load:-

1. Queries per second (QPS)
2. Read-to-write ratio (r/w)
   - Read-heavy systems
   - Write-heavy systems
3. Measuring performance.
   - Percentiles.

1. Queries per second (QPS):

also known as requests per second (RPS).

Eg:- A server receives 10M requests per day.

day to seconds conversion ⇒ 24 hours × 60 minutes × 60 seconds.

$$\therefore \frac{10M \text{ requests}}{\text{day}} = \frac{10 \times 10^6 \text{ requests}}{24 \times 60 \times 60 \text{ second}}$$

$$= 11.57 \text{ requests/sec. (RPS) (QPS)}$$

$$\approx 11.6 \text{ QPS.}$$

But these 10 M requests in a day are not uniformly distributed throughout the day.
Eg:- payment server is more likely to receive more requests during the day rather than at night.

Using QPS and user behavior, we can measure the load of the system.

User behavior implies that =>

For example, users shop more during Christmas or Black Friday etc festivals on Amazon. So Amazon will likely have more QPS these days.

② Read-to-write ratio (r/w):-

r/w ratio is another common way, generally used in DBs.

* read-heavy
① r/w is higher
② adding more read-replicas to the DB cluster
③ read-replicas contain copy of original data.
④ read requests are generally forward to read replicas in order to not overload the write servers.

* write-heavy.
① r/w is lower
② similar or more amount of writes than reads.
③ scaling write-heavy system is more complicated compared to read-heavy.
④ add more nodes to handle writes & carefully synchronize the write nodes strictly.

Note:- Apart from QPS, r/w ratio, there are some other ways to measure load in a system.
For example, data size, request-response size, number of concurrent users connected to a node, etc. Sometimes, this is system-specific.
As an owner, you need to build intuition for business use cases, and consider additional metrics to measure load.

(5) Measuring performance:-
This is also system-specific.
For example, for a system which is having request-response pattern, response time measures performance.

Response time is the time witnessed by the client to receive the response for a request.

How do we use response time as a metric for performance measurement?

using "average response time"

$$\text{average response time} = \frac{\sum_{i=1}^{n} t_i}{n} = \frac{(\text{sum of time taken by } n \text{ requests})}{\text{no. of requests } n}$$

n = total no. of requests
t_i = time for each requests

This doesn't show full picture of response time ⇒ does not denote how many users experienced some amount of delay. The better approach is "percentiles".

Percentiles :-

Median response time :

Example: If we have 100 requests with response times, sort those requests by response time. The response time of middle one determines 50th percentile. Let's say, the response time of middle request is 2 ms which says that half of the requests are served in less than 2s. In other words, 50th percentile (p50) of our system is 2s.

p50 = 2 seconds.

Distributed systems commonly evaluate high percentiles such as 95th, 99th, 99.9th.

[99th percentile p99 = 500 ms.
means out of 100 requests, 99 requests served under 500 ms.]

** lower response time for a higher percentile means good performance.

Key takeaways :-

1. Measuring load and performance will show you the correct direction for scaling out your system.
2. As an owner, you have to proactively measure these metrics to ensure that the system has resources according to its needs.

10) Maintainability :-

A maintainable distributed system should be :-

① Easy to operate
② Easy to understand
③ Easy to extend.

① Easy to operate :- As a developer, it is crucial to put effort into ensuring the operability of your system.

↳ Adding visible monitoring and logging: Add monitoring and logging mechanisms to your system and make them visible via specific platforms.
Eg: Datadog monitors system.

↳ Building automation support: It should be simple to quickly bootstrap the critical parts of your system so that in times of disaster, you can quickly respond.

↳ Creating proper documentation: Good documentation will help build and scale a skilled developer team.
Also, when things fail, developers should have proper docs specifying recovery procedure.

↳ Implementing a self-healing mechanism: Self-healing mechanism is difficult but helps to build a robust s/m.
It should allow developers to take control if required.

↳ Minimizing surprises: Document what could go wrong and recovery mechanisms for same. If you encounter new surprises, learn from them and document what you learned & enhance recovery mechanism.

② Easy to understand:-

① Avoid complicated dependencies → dependency on other systems or 3rd party services should be kept simple and easy to understand.

② Avoid making assumptions → systems get overly complicated just because someone down the line makes an assumption with no convincing logic.

③ Easy to extend:-

* Develop good coding practices: If you can ensure good practices, it will be easier to extend the codebase when adding new features.

* Create the habit of refactoring when required: If developers avoid refactoring, eventually the things that need to be refactored add up and the task becomes so massive that it never gets done.

* Build a well-defined deployment procedure: Developers should be able to go from one development to another with ease. To achieve this, build automation and create documentation where required

Key takeaways:-

① Maintainable system will be alive in the long run.
② A lot of effort will be put into maintaining once it is built.
③ As an owner, ensure maintainability from very beginning of development process.