

O'REILLY

Designing Data-Intensive Applications

THE BIG IDEAS BEHIND RELIABLE, SCALED,
AND MAINTAINABLE SYSTEMS



SECOND EDITION

SYSTEM DESIGN INTERVIEW



AN INSIDER'S GUIDE

Alex Xu

TIPS for System Design Interview

Friday, 31 December 2021 3:09 PM

Start with the Expectations, i.e. functional, non-functional requirements

① Don't go into details prematurely

↳ But, when it comes to DB design, ask if they want you to draw ER diagram right then (ideally, we also want to do that)

② Don't have a said Architecture 100%
i.e. not caring about requirements and designing as you've already read the question.

③ Keep it simple, stupid (KISS)

④ Don't make statements without justification.

↳ ex: saying you'll use a NoSQL DB without any relevant statements to back the design.

⑤ Be aware of current technologies

↳ ex: say you'll use a CDN, it might be nice to mention a real CDN tech. like Akamai
(or) cloud front.

(or)

For a message queue, say you'll use maybe Rabbit MQ / Kafka etc..

⑥ Don't do Capacity Estimation without any reason
↳ Maybe speak out with interviewer when you notice a particular service needs to know the capacity when you're 100% sure.

⑦ mention n/w protocols b/w communication of the systems in architecture !!! And mention pros & cons of using one over other (ex: HTTP vs FTP etc...)

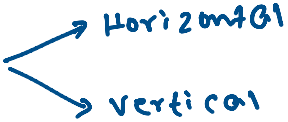
⑧ Speak out internals of system (ex: why a NoSQL DB like Cassandra over MySQL. What does Cassandra bring to the architecture other DB's cannot)

⑨ TRADE OFFS !!!

Terms & Analogy

Saturday, 25 December 2021

3:07 PM

① Scaling 

② Load Balancing

③ Backups (Avoid single point of failure)

④ Microservices Architecture (Isolate responsibilities)

⑤ Distributed System (Partitioning)

⑥ Decoupling

⑦ Logging & Metrics

⑧ Caching

⑨ API (Application Programmable Interface)

⑩ CAP Theorem (Consistency, Availability, Partition Tolerance)

⑪ CDN (Content Delivery Network)

⑫ DNS (Domain Name System)

⑬ Ports (ex. HTTP - 80, HTTPS - 443)

⑭ Network Protocols (IP, HTTP, TCP)

Snippets

Friday, 14 January 2022

5:23 PM

Read heavy?

↳ Throw a cache

write heavy?

↳ Push write reqs to a Message Queue.

Search Data?

↳ put a search index

More users? need to scale Database?

↳ need ACID properties?

↳ shard the RDBMS Database

↳ ACID Properties not mandatory?

↳ use NoSQL Database

High throughput?

↳ scale application servers. & put load balancer with
consistent hashing

Network Protocols

Wednesday, 12 January 2022

10:39 PM

↳ IP

↳ TCP

↳ HTTP

↳ Just a set of rules to interact b/w 2 things.

IP (Internet Protocol)

↳ 2^{16} bytes ≈ 0.06 MB

2 main things in IP packet are

① Header (src, target, total size of packet, IP version etc.)
↓
very small in general

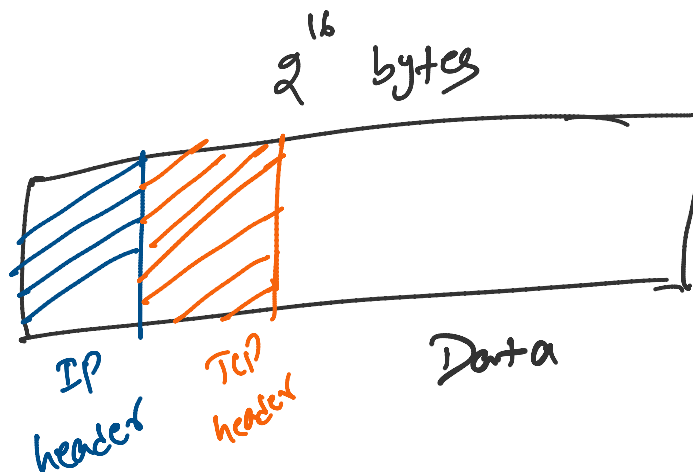
IPv4/IPv6

② Data

TCP (Transmission Control Protocol)

↳ Built on top of Internet Protocol

→ sends IP packets in order, reliable, error free way



... and User Text Transfer Protocol

HTTP (Hyper Text Transfer Protocol)

↳ Built on top of TCP

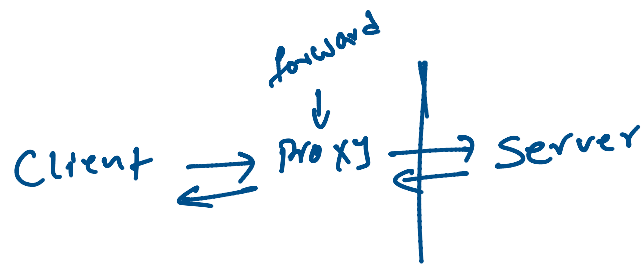
↳ This abstraction is called request-response paradigm.

Proxy (on Behalf)

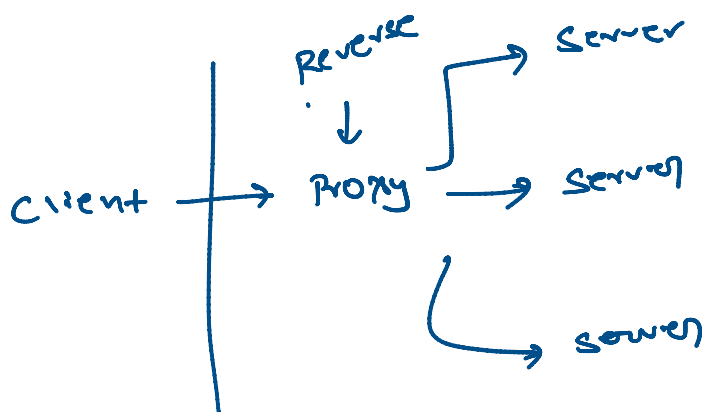
Friday, 31 December 2021 8:07 PM

Forward proxy → Server doesn't know which client it is connecting from (VPN)

Reverse proxy → Client doesn't know which server it is connecting to (balancer)



Forward proxy :- Proxy sits b/w client & server and disguises client IP address, block malicious traffic from clients etc.



Reverse proxy :- It's basically a server side proxy.

↳ Can be used for traffic control, Load Balancing, caching response from servers etc...

CAP Theorem

Sunday, 2 January 2022 12:04 AM

- ↳ consistency
- ↳ Availability
- ↳ Partition Tolerance

Any n/w shared system (distributed system) cannot have all 3 properties (nearly impossible). You need to forfeit one of them

Consistency :- A read happening after a write should always return the latest value of the recently done write operation.

Availability :- Every available node in system should respond in a non error format to any read request, without the guarantee of returning the latest write.

Partition Tolerance :- System will be responding to all read & write requests even if the communication channel (or middleware) b/w nodes is broken (or partitioned)

→ Partition Tolerance mostly happens through n/w failures. In a distributed system, this is almost unavoidable. (so, we cannot give up on this property)

→ we can design a system with either

① Partition Tolerance + Availability (AP)

(or)

② Partition Tolerance + consistency (CP)

Degrees of consistency & Availability

→ So, we know that we cannot provide 100% consistency and 100% availability with partition tolerance in picture.

→ But we can provide some extent of consistency, and availability!
or
degree

↳ This absolutely depends on usecase if we can achieve it or not!!

(eg: In ATM machine, no way we can be inconsistent, it always 100% consistent right. (or) Booking Tickets System.

Ex: 1) In ATM machine, no way we can have to be 100% consistent right. (or) Booking Tickets System.

2) Let's say, this time we're talking about no. of views for a youtube video. Here, you can forfeit the 100% consistency (The nodes will eventually reconcile & update anyways).

Tuesday, 28 December 2021 7:44 PM

Tuesday, 28 December 2021 7:44 PM

Tuesday, 28 December 2021 7:44 PM

- Tuesday, 28 December 2021 7:44 PM

Tuesday, 28 December 2021 7:44 PM

Content Delivery n/w

Wednesday, 29 December 2021

8:47 PM

↳ example Akamai, S3

CDN does: (Primarily focuses insanely fast data delivery by having multiple data centers geographically distributed!!!)

- ① Host the data close to users
- ② Follow Regulations
- ③ Provides some interface to update/remove? data from the data centers?
↳ The CDN basically?
- ④ can configure how much time to store some data in this CDN.

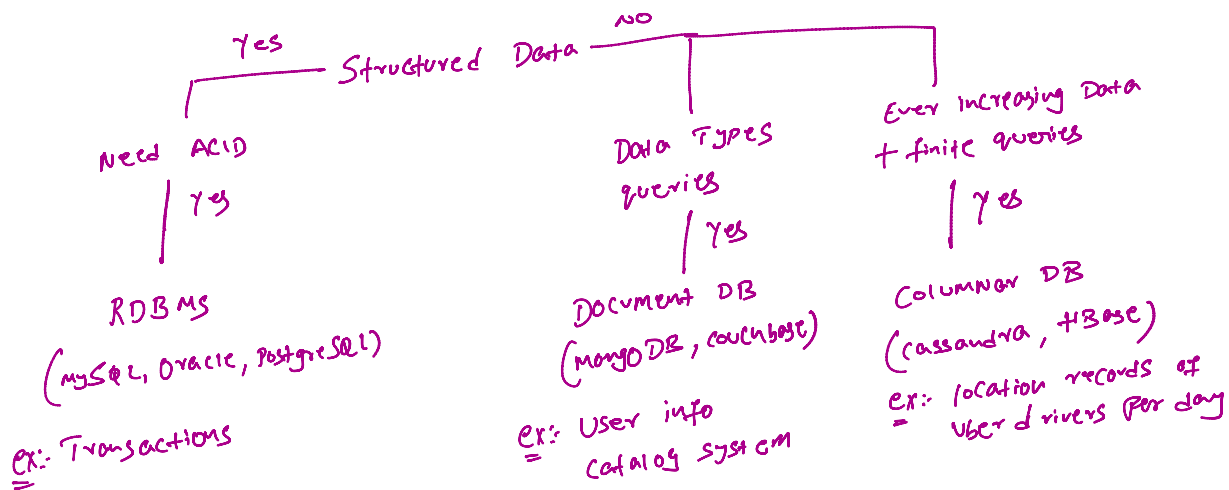
Cloudfront

- ↳ super cheap
- ↳ Reliable & easy to use

SQL vs NoSQL

Monday, 24 January 2022 1:35 AM

- ① Always think of caching to reduce latency.
 - ↳ Server API can cache
 - ↳ Redis, Memcached (key value stores) → non relational DBs
- ② Storing Images, Videos? These are blob storages & not queried upon.
 - ↳ So use Amazon S3 to store
 - ↳ use a CDN in front of S3 to distribute same data geographically! (faster way)
- ③ Text searching Feature (ex: Amazon product etc search / Netflix show title search etc..)
 - ↳ Elasticsearch / Solr



→ Feel free to use a combination of database types based on the scenario !!!

① Relational Databases

- ↳ Tables & rows with a fixed schema
- ↳ ACID Properties
- ↳ usecase → Entirely structured data / need acid properties
- ex: Employee Table, Transactions etc..
- ↳ Vertical scaling is easy (add more RAM, CPU etc)
- ↳ horizontal scaling is tough here (i.e. sharding)

... Databases (Horizontal Partitioning inbuilt mostly)
... (i.e.)

↳ Built for scale

② non relational Databases (Horizontal Partitioning inbuilt mostly)

↳ Key-value store (Redis, Memcache)

↳ fast & provide quick access & they are in memory

↳ used in caching solutions

↳ Built for scale

↳ Built for aggregations

↳ Document Based (MongoDB)

↳ used when not sure about structure of data & would evolve over time (i.e. no fixed schema, and can be changed easily)

↳ supports heavy reads & writes, provides sharding capabilities

↳ They have collections (like Tables) and documents (like rows)

↳ usecase → Product details of Amazon (since we want all data of one product at one place, no joins etc.)

↳ Cons

↳ NO ACID properties though we can handle it through application code, main loss is strong consistency.

↳ Joins are tough to achieve.

↳ Column DB (Cassandra, HBase, Sybase)

↳ Sort of mid way of relational & Document DBs

↳ Doesn't support ACID

↳ usecase → streaming data / event data for analytics

↳ Distributed databases in general

↳ Search Databases (Elasticsearch, Solr)

↳ Full text Search

↳ Fuzzy Search

↳ Keep in mind, this is NOT PRIMARY DATABASE!

For Reference

Tuesday, 28 December 2021

9:46 PM

a day.

#doubt →

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zip	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA→Netherlands→CA	150,000,000 ns

Google

10 processors

⇒ 400 MB/sec

How to avoid cascading failures

Friday, 31 December 2021 1:47 PM

Problems

- ① Cascading failures → Rate limiting
- ② Going viral → Pre scale
- ③ Predictable load increase → Auto scale
- ④ Bulk job scheduling → Batch processing
- ⑤ Popular posts → Approximate statistics.

Database Sharding

Sunday, 26 December 2021 3:59 PM

Problem :- How to query huge data in DB?

① Optimise queries (but lot of data, even with optimised queries, it's not good enough)

② Index on table (fine, but got lots of data!!!)

③ Sharding

→ Horizontal Partitioning

Takes a key and partitions the data to different servers.
(This key is generally an attribute of data we're storing)

Database servers!!

→ Practice of separating one table's rows into multiple different tables, known as partitions → sharding

Problems

① Joins across multiple shards.

② Fixed no. of shards

↳ use Hierarchical Sharding

i.e. Let's say even after sharding the data in one shard is huge.

... in this shard we can again ... in multiple

one shard - -
→ Now, in this shard we can again
shard/Partition the big shard into multiple
shards.

② what happens if a shard goes down

↳ use master/slave architecture

↳ master is the most updated one &
slaves keep reading from master for
updates

↳ All writes go to master, Reads are
distributed to slaves

↳ If master fails, one slave becomes
master.

- Querying DB for commonly used data?
 - ↳ use cache! (in memory cache)

Helps in saving n/w calls

- Avoid computations again & again
 - ↳ Instead of computing it every time from querying DB, query once & store in cache.

- This also avoids load on DB.

Caches generally run on SSD (Solid State Drive)
which are expensive!!!

- ① when to load data into cache
 - ② when to evict/remove data from cache.
- } cache policy!

- Cache policy determines how cache performance is.
 - ↳ ① LRU (least Recently used) → mostly used!!!
 - ② LFU (least frequently used) → not much used

Thrashing

- ↳ constantly loading & removing data from cache without ever using the results!

Problem

→ How can data be consistent in caches?
i.e. server 2 updates info in DB, which is
cached in server 1 cache. now what???

① Write Back Cache

↳ when updating data, hit cache (if this data exists,
then update it here in cache, then update DB)

↓
This method will have
some data inconsistency
if multiple caches are
there

↓
→ This update can be delayed if
data updated in cache is not of high
importance.

→ Then all updates on cache can
at once be sent as a batch to
DB for update.

② write Through Cache

↳ First hit the DB & then update in cache. (Synchronous)
↳ Reliable & better (But slow as update needs to
happen in both places)

Load Balancing

↳ let's say we have "N" servers.
and we need to balance load properly across
those "N" servers. i.e. load balancing.

→ consistent Hashing let's us do load balancing efficiently.

Example: $h(\text{req id}) / \text{no. of servers}$ (General old fashioned hashing) doesn't work.

↓
hash function is uniformly Random
i.e. takes request id & gives out some
random no./string/something
AND every server will have uniform load.

X Requests → Load = X/N → load factor = $1/N$
↓
on each server

Problem?

After adding new servers (scaled horizontally),
the requests are routed again

→ i.e. In real world, a request id is generally
uniform. And, when we hash this, we always
get same output.

→ if this req. is previously sent to server "A"
because of addition of new servers, same
req. would go to some other server "M".

(And this is a problem with simple hashing)

Why? → Bcz, there could be cached data
we want to reuse.
And, we want to send a same req.
with same id to same server.

Solution (use consistent Hashing)

... in $[0, M-1]$ range
Total M

Solution (use consistent Hashing)

- ① Let's say our hash function outputs in $[0, M-1]$ range ^{Total}
- ② Servers themselves have id's also, we hash these id's also and take remainder with M .

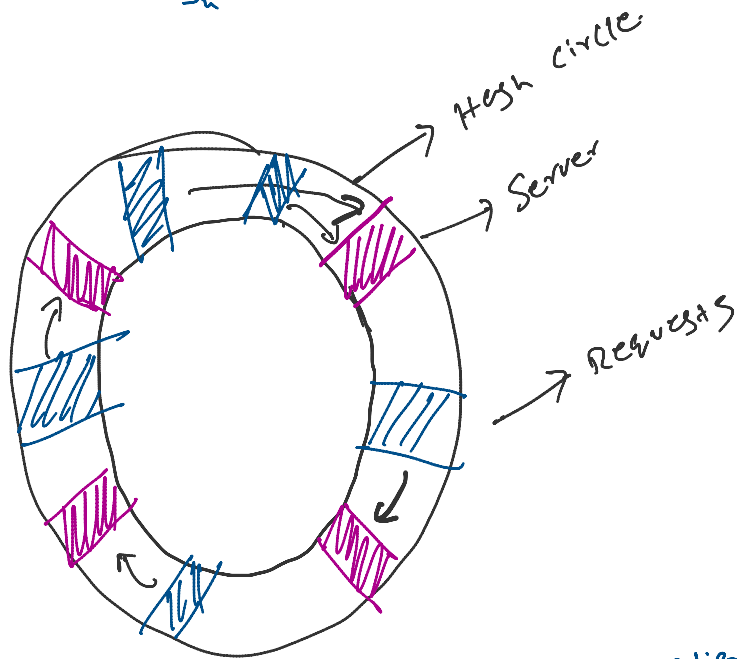
$$\text{e.g. } S_0 \rightarrow h(0) \% M$$

$$S_1 \rightarrow h(1) \% M$$

$$S_2 \rightarrow h(2) \% M$$

$$S_3 \rightarrow h(3) \% M$$

$$S_n \rightarrow h(n) \% M$$



→ The request is served by the immediate server on the clockwise direction.

→ If a new server is added, it goes into the hash ring.

→ If a server goes down, all requests that would go to this server will now be served by the next server on clockwise direction.

Problem

→ If there are 5 requests that have server "A" on clockwise direction. If this server "A" goes down, all these requests will now be served by the next server on clockwise direction.

- If this -
on clockwise direction. If this server "A" goes down,
then all these 5 requests go to next server on
clockwise direction, let's say server "B".
- Due to this server "B" load increased a lot!!!

Solution:-

Use multiple hash functions for servers

$$\begin{array}{ll} \text{i.e. } S_0 \rightarrow h_1(0) & h_2(0) \rightarrow \dots \\ S_1 \rightarrow h_1(1) & h_2(1) \rightarrow \dots \end{array}$$

- If we have "K" hash functions, we can have K points
for each server.

- Likelihood of one server getting lot of load is now very less
- If a server is removed, then remove the K points of that server.

Message Queue

Sunday, 26 December 2021

3:34 PM

- Takes tasks, persists the tasks, assign tasks to right server,
waits for task completion, if taking long time/server dead
assigns to another server

All of these are done by a
" Message Queue "

Ex:- RabbitMQ, JMS (Java Messaging Service)

Horizontal vs Vertical Scaling

Saturday, 25 December 2021

2:46 PM

→ Host services on cloud (AWS/GCP/Azure)

↓

Can be your product/service/anything.

Problem → How to handle more requests
i.e. Scalability

- ① Buy bigger machine → vertical scaling
- ② Buy more machines → Horizontal scaling

Trade offs

Horizontal	vertical
① need load balancing	① N/A
② Resilient	② Single Point failure
③ n/w calls (RPC) between servers	③ Inter Process communication
④ Data consistency issue	④ consistent
⑤ Scales well	⑤ Hardware limit

Solution

Horizontal scaling but, each server is powerful enough