

System Expert

Introduction

- Goal : Design Robust, Functional and Scalable Systems
- Where the coding interview serves primarily as an assessment of your problem-solving ability, the systems design interview is a test of your engineering knowledge veiled behind the facade of an open-ended design question.
- Welcome to the crucible of modern software.

What Are Design Fundamentals?

- Building scalable, production-ready applications is both art and science. Science, in that it requires knowledge of many topics in computer engineering; art, in that it demands an eye for making smart design choices and piecing together the right technologies.
- Master both disciplines and you, too, can become a Systems Expert.
SQL, Server, Cache, Polling, Load Balancer, Leader Election, Peer-to-peer, Availability, Proxies, Nginx, Mapreduce, Client, HTTP, Database, Hashing, Replication
- Things to take into consideration:
- Your job to take 1 line question and make it a 45 mins discussion
- Justify your solution. Why is it best? Why one of many? Clear doubts of the interviewer.
- 3 Components
 - a) **Foundational knowledge** : Client Server Model, Network Protocols
 - b) **Key Characteristics** : Availability, Latency, Throughput, Redundancy, Consistency
 - c) **Actual Components** : Load balancer, Caching, Leader Election, Proxies, Rate Limiting
 - d) **Actual Tech** : Products and Services : Zookeeper, AWS, Nginx etc

Client—Server Model

A client is a thing that talks to servers. A server is a thing that talks to clients. The client—server model is a thing made up of a bunch of clients and servers talking to one another.

And that, kids, is how the Internet works!

Browser > DNS query for IP address > IP > HTTP Request > Server

Important points:

- Dig + domain name for IP address details
- Source IP is in the HTTP Request
- Any machine has 16000 ports. HTTP is on port 80, HTTPS is on port 443
- Exercise => nc -l 8081, nc 127.0.0.1 8081

Client

A machine or process that requests data or service from a server.

Note that a single machine or piece of software can be both a client and a server at the same time. For instance, a single machine could act as a server for end users and as a client for a database.

Server

A machine or process that provides data or service for a client, usually by listening for incoming network calls.

Note that a single machine or piece of software can be both a client and a server at the same time. For instance, a single machine could act as a server for end users and as a client for a database.

Client—Server Model

The paradigm by which modern systems are designed, which consists of clients requesting data or service from servers and servers providing data or service to clients.

IP Address

An address given to each machine connected to the public internet. IPv4 addresses consist of four numbers separated by dots: **a.b.c.d** where all four numbers are between 0 and 255. Special values include:

- **127.0.0.1**: Your own local machine. Also referred to as **localhost**.
- **192.168.x.y**: Your private network. For instance, your machine and all machines on your private wifi network will usually have the **192.168** prefix.

Port

In order for multiple programs to listen for new network connections on the same machine without colliding, they pick a **port** to listen on. A port is an integer between 0 and 65,535 (2^{16} ports total).

Typically, ports 0-1023 are reserved for *system ports* (also called *well-known ports*) and shouldn't be used by user-level processes. Certain ports have pre-defined uses, and although you usually won't be required to have them memorized, they can sometimes come in handy. Below are some examples:

- 22: Secure Shell
- 53: DNS lookup
- 80: HTTP
- 443: HTTPS

DNS

Short for Domain Name System, it describes the entities and protocols involved in the translation from domain names to IP Addresses.

Typically, machines make a DNS query to a well known entity which is responsible for returning the IP address (or multiple ones) of the requested domain name in the response.

Network Protocols

- IP packets. TCP headers. HTTP requests.
- As daunting as they may seem, these low-level networking concepts are essential to understanding how machines in a system communicate with one another. And as we all know, proper communication is key for thriving relationships!
- IP - Header and Data : Packets of bytes : 2^{16} bytes = 0.065Mb
 - Header : Source and Destination IP, Total size, Version of Internet Protocol (IPv4, IPv6)
 - TCP Header : Info of ordering of packets
 - Data : What we actually want to send
- TCP - Transmission Control Protocol :
 - Send IP packets in ordered way, reliable way, error free way
 - Send long pieces of data
 - Connection handshake, Timeout, Ending connection via special message
- HTTP - Built over TCP : Higher level abstraction : Requests and Responses
 - Details below : “path” also comes after method and before headers
 - Get = retrieve data from server, POST = providing data to server, DELETE = to delete some data

```
JS http_request_example.js > [o] httpResponse > ↴ statusCode
1  const httpRequest = {
2    host: 'localhost',
3    port: 8080,
4    method: 'POST', // GET, PUT, DELETE, etc.
5    path: '/payments',
6    headers: {
7      'content-type': 'application/json',
8      'content-length': 51,
9    },
10   body: '{"data": "This is a piece of data in JSON format."}'
11 }
12
13 const httpResponse = {
14   statusCode: 200,
15   headers: {
16     'access-control-allow-origin': 'https://www.algoexpert.io',
17     'content-type': 'application/json',
18   },
19   body: '{}'
20 }
```

IP

Stands for **Internet Protocol**. This network protocol outlines how almost all machine-to-machine communications should happen in the world. Other protocols like **TCP**, **UDP** and **HTTP** are built on top of IP.

TCP

Network protocol built on top of the Internet Protocol (IP). Allows for ordered, reliable data delivery between machines over the public internet by creating a **connection**.

TCP is usually implemented in the kernel, which exposes **sockets** to applications that they can use to stream data through an open connection.

HTTP

The **HyperText Transfer Protocol** is a very common network protocol implemented on top of TCP. Clients make HTTP requests, and servers respond with a response.

Requests typically have the following schema:

```
host: string (example: algoexpert.io)
port: integer (example: 80 or 443)
method: string (example: GET, PUT, POST, DELETE, OPTIONS or PATCH)
headers: pair list (example: "Content-Type" => "application/json")
body: opaque sequence of bytes
```

Responses typically have the following schema:

```
status code: integer (example: 200, 401)
headers: pair list (example: "Content-Length" => 1238)
body: opaque sequence of bytes
```

IP Packet

Sometimes more broadly referred to as just a (network) **packet**, an IP packet is effectively the smallest unit used to describe data being sent over **IP**, aside from bytes. An IP packet consists of:

- an **IP header**, which contains the source and destination **IP addresses** as well as other information related to the network
- a **payload**, which is just the data being sent over the network

Storage

- An entire video dedicated just to the storage of data?
- Yes! Multiple videos, in fact, as you'll see later on when we discuss databases. As it turns out, information storage is an incredibly complex topic that is of vital importance to systems design.
- Don't even think of skipping this lesson!
- Data in disk is persistent but Data in memory is not (like hash tables etc)
- Distributed storage : Copy or divide?
- Consistency issues!

4 Key Terms

Databases

Databases are programs that either use disk or memory to do 2 core things: **record** data and **query** data. In general, they are themselves servers that are long lived and interact with the rest of your application through network calls, with protocols on top of TCP or even HTTP.

Some databases only keep records in memory, and the users of such databases are aware of the fact that those records may be lost forever if the machine or process dies.

For the most part though, databases need persistence of those records, and thus cannot use memory. This means that you have to write your data to disk. Anything written to disk will remain through power loss or network partitions, so that's what is used to keep permanent records.

Since machines die often in a large scale system, special disk partitions or volumes are used by the database processes, and those volumes can get recovered even if the machine were to go down permanently.

Disk

Usually refers to either **HDD (hard-disk drive)** or **SSD (solid-state drive)**. Data written to disk will persist through power failures and general machine crashes. Disk is also referred to as **non-volatile storage**.

SSD is far faster than HDD (see latencies of accessing data from SSD and HDD) but also far more expensive from a financial point of view. Because of that, HDD will typically be used for data that's rarely accessed or updated, but that's stored for a long time, and SSD will be used for data that's frequently accessed and updated.

Memory

Short for **Random Access Memory (RAM)**. Data stored in memory will be lost when the process that has written that data dies.

Persistent Storage

Usually refers to disk, but in general it is any form of storage that persists if the process in charge of managing it dies.

Latency And Throughput

If you've ever experienced lag in a video game, it was most likely due to a combination of high latency and low throughput. And lag sucks.

It is therefore your Call of Duty to master these two concepts and to join the crusade against high ping.

Latency

The time it takes for a certain operation to complete in a system. Most often this measure is a time duration, like milliseconds or seconds. You should know these orders of magnitude:

- **Reading 1 MB from RAM:** 250 μ s (0.25 ms)
- **Reading 1 MB from SSD:** 1,000 μ s (1 ms)
- **Transfer 1 MB over Network:** 10,000 μ s (10 ms)
- **Reading 1MB from HDD:** 20,000 μ s (20 ms)
- **Inter-Continental Round Trip:** 150,000 μ s (150 ms)

Throughput

The number of operations that a system can handle properly per time unit. For instance the throughput of a server can often be measured in requests per second (RPS or QPS).

Availability

Some things would be critical to the system hence need to be highly available like the services. However, some would not be critical for example a dashboard, hence need not be highly available.

Redundancy helps in High Availability.

- Passive : Multiple components. If any component fails, others continue smoothly. Example of airplane twin engines
- Active : Only one or few take care of the work, if that fails, other one takes over.

Availability

The odds of a particular server or service being up and running at any point in time, usually measured in percentages. A server that has 99% availability will be operational 99% of the time (this would be described as having two **nines** of availability).

High Availability

Used to describe systems that have particularly high levels of availability, typically 5 nines or more; sometimes abbreviated "HA".

Nines

Typically refers to percentages of uptime. For example, 5 nines of availability means an uptime of 99.999% of the time. Below are the downtimes expected per year depending on those 9s:

- 99% (two 9s): 87.7 hours
- 99.9% (three 9s): 8.8 hours
- 99.99%: 52.6 minutes
- 99.999%: 5.3 minutes

Redundancy

The process of replicating parts of a system in an effort to make it more reliable.

SLA

Short for "service-level agreement", an SLA is a collection of guarantees given to a customer by a service provider. SLAs typically make guarantees on a system's availability, amongst other things. SLAs are made up of one or multiple SLOs.

SLO

Short for "service-level objective", an SLO is a guarantee given to a customer by a service provider. SLOs typically make guarantees on a system's availability, amongst other things. SLOs constitute an SLA.

Caching

At client level or At server level as per usecase.

Redis : Key Value store detached cache

Cache

A piece of hardware or software that stores data, typically meant to retrieve that data faster than otherwise.

Caches are often used to store responses to network requests as well as results of computationally-long operations.

Note that data in a cache can become **stale** if the main source of truth for that data (i.e., the main database behind the cache) gets updated and the cache doesn't.

Cache Hit

When requested data is found in a cache.

Cache Miss

When requested data could have been found in a cache but isn't. This is typically used to refer to a negative consequence of a system failure or of a poor design choice. For example:

If a server goes down, our load balancer will have to forward requests to a new server, which will result in cache misses.

Cache Eviction Policy

The policy by which values get evicted or removed from a cache. Popular cache eviction policies include **LRU** (least-recently used), **FIFO** (first in first out), and **LFU** (least-frequently used).

Content Delivery Network

A **CDN** is a third-party service that acts like a cache for your servers. Sometimes, web applications can be slow for users in a particular region if your servers are located only in another region. A CDN has servers all around the world, meaning that the latency to a CDN's servers will almost always be far better than the latency to your servers. A CDN's servers are often referred to as **PoPs** (Points of Presence). Two of the most popular CDNs are **Cloudflare** and **Google Cloud CDN**.

```
JS server.js × JS database.js
JS server.js > ...
1 const database = require('../database');
2 const express = require('express');
3
4 const app = express();
5 const cache = {};
6
7 app.get('/nocache/index.html', (req, res) => {
8   database.get('index.html', page => {
9     res.send(page);
10   });
11 });
12
13 app.get('/withcache/index.html', (req, res) => {
14   if ('index.html' in cache) {
15     res.send(cache['index.html']);
16     return;
17   }
18
19   database.get('index.html', page => {
20     cache['index.html'] = page;
21     res.send(page);
22   });
23 });
24
25 app.listen(3001, function() {
26   console.log('Listening on port 3001!');
27 });

JS server.js JS database.js
JS database.js > ⚡ get > ⚡ module.exports.get
1 const database = {
2   ['index.html']: '<html>Hello World!</html>',
3 };
4
5 module.exports.get = (key, callback) => {
6   setTimeout(() => {
7     callback(database[key]);
8   }, 3000);
9 };
```

1. **Write through cache:** Writes both in cache and database both at same time. Every time going to database.
2. **Write back cache:** Only the cache gets update and system will asynchronously update the database later some time. If cache is lost due to any reason then data might get lost.

Caches can become stale if not designed properly. For instance we have on server cache and multiple servers, that will not make sure the same things are available to all customers across servers (in case of distributed servers). Therefore we move the Cache out of the servers, that is to Redis etc.

For immutable data caching is beautiful.

If things are mutable caching can get bit tricky. Eviction policy is to be decided on the bases of usecase.

Proxies

Often used by nefarious hackers to conceal their identity and obfuscate their location, these special intermediary servers boast many important real-life applications within the context of caching, access control, and censorship bypassing, amongst other things.

Forward Proxy

A server that sits between a client and servers and acts on behalf of the client, typically used to mask the client's identity (IP address). Note that forward proxies are often referred to as just proxies.

Reverse Proxy

A server that sits between clients and servers and acts on behalf of the servers, typically used for logging, load balancing, or caching.

Nginx ⚡

Pronounced "engine X"—not "N jinx", Nginx is a very popular webserver that's often used as a **reverse proxy** and **load balancer**.

Reverse proxies help a lot:

- Filter out certain kind of requests
- Cache stuff
- Want to log stuff or certain metrics
- Use as a load balancer

Load Balancers

Relentlessly distributing network requests across multiple servers, these digital traffic cops act as watchful guardians for your system, ensuring that it operates at peak performance day and night.

Load Balancer

A type of **reverse proxy** that distributes traffic across servers. Load balancers can be found in many parts of a system, from the DNS layer all the way to the database layer.

Server-Selection Strategy

How a **load balancer** chooses servers when distributing traffic amongst multiple servers. Commonly used strategies include round-robin, random selection, performance-based selection (choosing the server with the best performance metrics, like the fastest response time or the least amount of traffic), and IP-based routing.

Hot Spot

When distributing a workload across a set of servers, that workload might be spread unevenly. This can happen if your **sharding key** or your **hashing function** are suboptimal, or if your workload is naturally skewed: some servers will receive a lot more traffic than others, thus creating a "hot spot".

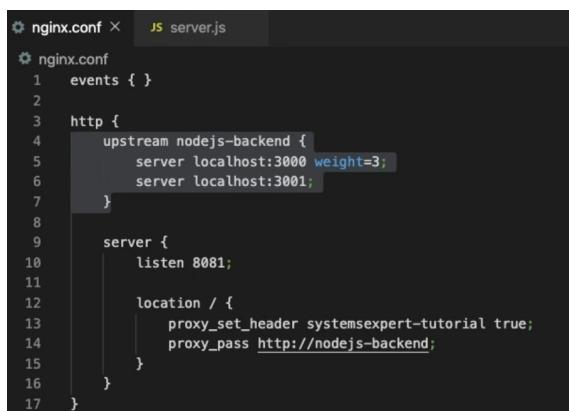
Nginx ⚡

Pronounced "engine X"—not "N jinx", Nginx is a very popular webserver that's often used as a **reverse proxy** and **load balancer**.

1. Random Load Balancing
2. Round Robin Load Balancing - 123123
3. Weighted Round Robin Load Balancing - 12331233
4. Performance Based Load Balancing - Distribution according to Performance and response time
5. IP Based Load Balancing - Redirecting a user to a set server to optimise cache to maximum

But what if the load balancer get loaded? We put multiple load balancers there.

One load balancer can forward requests to other load balancers too. For instance, One load balancer as IP based followed by 2 RR Load balancers.



```
nginx.conf × JS server.js
nginx.conf
1 events { }
2
3 http {
4     upstream nodejs-backend {
5         server localhost:3000 weight=3;
6         server localhost:3001;
7     }
8
9     server {
10        listen 8081;
11
12        location / {
13            proxy_set_header systemexpert-tutorial true;
14            proxy_pass http://nodejs-backend;
15        }
16    }
17 }
```

Hashing

Hashing? Like from hash tables? Should be simple enough, right?

The good news is that, yes, hashing like from hash tables.

The bad news is that, no, not simple enough. The video duration and thumbnail should be ominously indicative.

Consistent Hashing

A type of hashing that minimizes the number of keys that need to be remapped when a hash table gets resized. It's often used by load balancers to distribute traffic to servers; it minimizes the number of requests that get forwarded to different servers when new servers are added or when existing servers are brought down.

Rendezvous Hashing

A type of hashing also coined **highest random weight** hashing. Allows for minimal re-distribution of mappings when a server goes down.

SHA

Short for "Secure Hash Algorithms", the SHA is a collection of cryptographic hash functions used in the industry. These days, SHA-3 is a popular choice to use in a system.

We generally do hash client names and mod that by number of servers to actually do the mapping of client to servers to utilise caching on that server to the fullest.

MD5, SHA1, BCrypt

But what happens if server fails or we add. Another server? We can not keep on moving by different numbers, even if we do then there would be cache misses.

Therefore we use Consistent Hashing and Rendezvous Hashing.

- Consistent Hashing maintains some level of cache hits. As servers are added and removed from a circular setup which is based on Hashing.
- Rendezvous Hashing is based on a score which is calculated using industry standard score calculation algos.

Relational Databases

Tables and ACID.

No, we're not describing a drug lord's desk, but rather referring to key properties of relational databases. There's a lot of material to cover here, so hit the play button, kick back, and get ready to store tons of knowledge in the biggest database of them all: your brain.

The indexing in RDBMS comes real handy however makes writes a bit slower, but reads are much much faster.
Issues in NoSQL : No powerful querying, Updates not reflected giving stale data, Eventual consistency.

Relational Database

A type of structured database in which data is stored following a tabular format; often supports powerful querying using SQL.

Non-Relational Database

In contrast with relational database (SQL databases), a type of database that is free of imposed, tabular-like structure. Non-relational databases are often referred to as NoSQL databases.

SQL

Structured Query Language. Relational databases can be used using a derivative of SQL such as PostgreSQL in the case of Postgres.

SQL Database

Any database that supports SQL. This term is often used synonymously with "Relational Database", though in practice, not every relational database supports SQL.

NoSQL Database

Any database that is not SQL-compatible is called NoSQL.

ACID Transaction

A type of database transaction that has four important properties:

- **Atomicity:** The operations that constitute the transaction will either all succeed or all fail. There is no in-between state.
- **Consistency:** The transaction cannot bring the database to an invalid state. After the transaction is committed or rolled back, the rules for each record will still apply, and all future transactions will see the effect of the transaction. Also named **Strong Consistency**.
- **Isolation:** The execution of multiple transactions concurrently will have the same effect as if they had been executed sequentially.
- **Durability:** Any committed transaction is written to non-volatile storage. It will not be undone by a crash, power loss, or network partition.

Database Index

A special auxiliary data structure that allows your database to perform certain queries much faster. Indexes can typically only exist to reference structured data, like data stored in relational databases. In practice, you create an index on one or multiple columns in your database to greatly speed up **read** queries that you run very often, with the downside of slightly longer **writes** to your database, since writes have to also take place in the relevant index.

Strong Consistency

Strong Consistency usually refers to the consistency of ACID transactions, as opposed to **Eventual Consistency**.

Eventual Consistency

A consistency model which is unlike **Strong Consistency**. In this model, reads might return a view of the system that is stale. An eventually consistent datastore will give guarantees that the state of the database will eventually reflect writes within a time period (could be 10 seconds, or minutes).

Postgres ⚡

A relational database that uses a dialect of SQL called PostgreSQL. Provides ACID transactions.

```
/*
Transactions
*/

-- Transfer 100 from Clement to Antoine.
BEGIN TRANSACTION;
UPDATE balances SET balance = balance - 100 WHERE username = 'clement';
UPDATE balances SET balance = balance + 100 WHERE username = 'antoine';
COMMIT;

/*
Indexes
*/

-- Find the 10 largest ints.
SELECT * FROM large_table ORDER BY random_int DESC LIMIT 10; I

-- Create an index on the ints in the table.
CREATE INDEX large_table_random_int_idx ON large_table(random_int);
```

Key-Value Stores

One of the most commonly used NoSQL paradigms today, the key-value store bases its data model on the associative array data type.

The result? A fast, flexible storage machine that resembles a hash table. That's right folks, our favorite friendly neighborhood data structure strikes again!

Key-Value Store

A Key-Value Store is a flexible NoSQL database that's often used for caching and dynamic configuration. Popular options include DynamoDB, Etcd, Redis, and ZooKeeper.

Etcd ⚡

Etcd is a strongly consistent and highly available key-value store that's often used to implement leader election in a system.

Redis ⚡

An in-memory key-value store. Does offer some persistent storage options but is typically used as a really fast, best-effort caching solution. Redis is also often used to implement **rate limiting**.

ZooKeeper ⚡

ZooKeeper is a strongly consistent, highly available key-value store. It's often used to store important configuration or to perform leader election.

Redis : in memory key value store

Example show in image below : EX, 10 means that the keyword would be removed after 10 seconds from redis.

With the caching example as mentioned in caching section, we would loose the data if server crashed but the same would not happen in case of redis.



```
JS server.js × JS database.js
JS server.js > ⓘ app.get('/withcache/index.html') callback > ⓘ redis.get('index.h
1  const database = require('../database');
2  const express = require('express');
3  const redis = require('redis').createClient();
4
5  const app = express();
6
7  app.get('/nocache/index.html', (req, res) => {
8    database.get('index.html', page => {
9      res.send(page);
10   });
11 });
12
13 app.get('/withcache/index.html', (req, res) => {
14   redis.get('index.html', (err, redisRes) => {
15     if (redisRes) {
16       res.send(redisRes);
17       return;
18     }
19
20     database.get('index.html', page => [
21       redis.set('index.html', page, 'EX', 10),
22       res.send(page),
23     ]);
24   });
25 });
26
27 app.listen(3001, function() {
28   console.log('Listening on port 3001!');
29 });
```

Specialized Storage Paradigms

Blob Storage

Widely used kind of storage, in small and large scale systems. They don't really count as databases per se, partially because they only allow the user to store and retrieve data based on the name of the blob. This is sort of like a key-value store but usually blob stores have different guarantees. They might be slower than KV stores but values can be megabytes large (or sometimes gigabytes large). Usually people use this to store things like **large binaries, database snapshots, or images** and other static assets that a website might have.

Blob storage is rather complicated to have on premise, and only giant companies like Google and Amazon have infrastructure that supports it. So usually in the context of System Design interviews you can assume that you will be able to use **GCS** or **S3**. These are blob storage services hosted by Google and Amazon respectively, that cost money depending on how much storage you use and how often you store and retrieve blobs from that storage.

Time Series Database

A **TSDB** is a special kind of database optimized for storing and analyzing time-indexed data: data points that specifically occur at a given moment in time. Examples of TSDBs are InfluxDB, Prometheus, and Graphite.

Graph Database

A type of database that stores data following the graph data model. Data entries in a graph database can have explicitly defined relationships, much like nodes in a graph can have edges.

Graph databases take advantage of their underlying graph structure to perform complex queries on deeply connected data very fast.

Graph databases are thus often preferred to relational databases when dealing with systems where data points naturally form a graph and have multiple levels of relationships—for example, social networks.

Google Cloud Storage

GCS is a blob storage service provided by Google.

S3

S3 is a blob storage service provided by Amazon through **Amazon Web Services (AWS)**.

InfluxDB

A popular open-source time series database.

Prometheus

A popular open-source time series database, typically used for monitoring purposes.

Neo4j

A popular graph database that consists of **nodes, relationships, properties, and labels**.

Cypher

A **graph query language** that was originally developed for the Neo4j graph database, but that has since been standardized to be used with other graph databases in an effort to make it the "SQL for graphs."

Cypher queries are often much simpler than their SQL counterparts. Example Cypher query to find data in **Neo4j**, a popular graph database:

```
MATCH (some_node:SomeLabel)-[ :SOME_RELATIONSHIP ]->(some_other_node:SomeLabel {some_property:'value'})
```

Spatial Database

A type of database optimized for storing and querying spatial data like locations on a map. Spatial databases rely on spatial indexes like **quadtrees** to quickly perform spatial queries like finding all locations in the vicinity of a region.

Quadtree

A tree data structure most commonly used to index two-dimensional spatial data. Each node in a quadtree has either zero children nodes (and is therefore a leaf node) or exactly four children nodes.

Typically, quadtree nodes contain some form of spatial data—for example, locations on a map—with a maximum capacity of some specified number **n**. So long as nodes aren't at capacity, they remain leaf nodes; once they reach capacity, they're given four children nodes, and their data entries are split across the four children nodes.

A quadtree lends itself well to storing spatial data because it can be represented as a grid filled with rectangles that are recursively subdivided into four sub-rectangles, where each quadtree node is represented by a rectangle and each rectangle represents a spatial region. Assuming we're storing locations in the world, we can imagine a quadtree with a maximum node-capacity **n** as follows:

- The root node, which represents the entire world, is the outermost rectangle.
- If the entire world has more than **n** locations, the outermost rectangle is divided into four quadrants, each representing a region of the world.
- So long as a region has more than **n** locations, its corresponding rectangle is subdivided into four quadrants (the corresponding node in the quadtree is given four children nodes).
- Regions that have fewer than **n** locations are undivided rectangles (leaf nodes).
- The parts of the grid that have many subdivided rectangles represent densely populated areas (like cities), while the parts of the grid that have few subdivided rectangles represent sparsely populated areas (like rural areas).

Finding a given location in a perfect quadtree is an extremely fast operation that runs in $\log_4(x)$ time (where **x** is the total number of locations), since quadtree nodes have four children nodes.

Replication And Sharding

A system's performance is often only as good as its database's; optimize the latter, and watch as the former improves in tandem!

On that note, in this video we'll examine how data redundancy and data partitioning techniques can be used to enhance a system's fault tolerance, throughput, and overall reliability.

Replication

The act of duplicating the data from one database server to others. This is sometimes used to increase the redundancy of your system and tolerate regional failures for instance. Other times you can use replication to move data closer to your clients, thus decreasing the latency of accessing specific data.

Sharding

Sometimes called **data partitioning**, sharding is the act of splitting a database into two or more pieces called **shards** and is typically done to increase the throughput of your database. Popular sharding strategies include:

- Sharding based on a client's region
- Sharding based on the type of data being stored (e.g: user data gets stored in one shard, payments data gets stored in another shard)
- Sharding based on the hash of a column (only for structured data)

Hot Spot

When distributing a workload across a set of servers, that workload might be spread unevenly. This can happen if your **sharding key** or your **hashing function** are suboptimal, or if your workload is naturally skewed: some servers will receive a lot more traffic than others, thus creating a "hot spot".

Leader Election

Citizens in a society typically elect a leader by voting for their preferred candidate. But how do servers in a distributed system choose a master node? Via algorithms of course!

This form of algorithmic democracy is known as "leader election", though we personally think "algorithmocracy" sounds way cooler.

Leader Election

The process by which nodes in a cluster (for instance, servers in a set of servers) elect a so-called "leader" amongst them, responsible for the primary operations of the service that these nodes support. When correctly implemented, leader election guarantees that all nodes in the cluster know which one is the leader at any given time and can elect a new leader if the leader dies for whatever reason.

Consensus Algorithm

A type of complex algorithms used to have multiple entities agree on a single data value, like who the "leader" is amongst a group of machines. Two popular consensus algorithms are **Paxos** and **Raft**.

Paxos & Raft

Two consensus algorithms that, when implemented correctly, allow for the synchronization of certain operations, even in a distributed setting.

Etcdbolt

Etcdbolt is a strongly consistent and highly available key-value store that's often used to implement leader election in a system.

ZooKeeper

ZooKeeper is a strongly consistent, highly available key-value store. It's often used to store important configuration or to perform leader election.

Etcdbolt implements Raft algorithm.

We can use self made consensus algorithms but that would be too complicated, therefore we use Etcdbolt.

Peer-To-Peer Networks

Equality for all.

Sharing is caring.

Unity makes strength.

The more the merrier.

Teamwork makes the dream work.

Welcome to peer-to-peer networks!

Peer-To-Peer Network

A collection of machines referred to as peers that divide a workload between themselves to presumably complete the workload faster than would otherwise be possible. Peer-to-peer networks are often used in file-distribution systems.

Gossip Protocol

When a set of machines talk to each other in a uncoordinated manner in a cluster to spread information through a system without requiring a central source of data.

If we have a 5GB file, we split it into 1000 5 MB files and transfer that to 1000 machines and then let those machines communicate with each other to get the other parts to make up the whole file.

In a system of throughput of 5GBps, 5MB will be transferred in 0.001 seconds.

Tracker helps to figure out with which peer to talk next?

The same can be done via the Gossip protocol. (Or epidemic protocol)

Kraken - p2p powered docker registry.

Torrent works on same principle.

Polling And Streaming

You can think of polling and streaming kind of like a classroom; sometimes students ask the teacher lots of questions, and other times they quiet down and listen attentively to the teacher's lecture.

Now fire up the video and get ready to stream; you won't be able to poll here. Class is in session!

Polling

The act of fetching a resource or piece of data regularly at an interval to make sure your data is not too stale.

Streaming

In networking, it usually refers to the act of continuously getting a feed of information from a server by keeping an open connection between the two machines or processes.

Pooling = Non Instantaneous, Streaming = Instantaneous

Configuration

The config file is like the genome of a computer application; it stores parameters that define your system's critical settings, much like your DNA stores the genes that define your physical characteristics.

Unlike its biological counterpart though, the config file is easily editable. No gene therapy needed!

Configuration

A set of parameters or constants that are critical to a system. Configuration is typically written in **JSON** or **YAML** and can be either **static**, meaning that it's hard-coded in and shipped with your system's application code (like frontend code, for instance), or **dynamic**, meaning that it lives outside of your system's application code.

2 Types:

Static : It is bundled with whole package and every-time if changes are done, the whole package needs to be deployed again on the server. Hence server restart required.

Dynamic : It is decoupled from rest of the code base but a bit complex to implement

Rate Limiting

Rate Limiting

The act of limiting the number of requests sent to or from a system. Rate limiting is most often used to limit the number of incoming requests in order to prevent **DoS attacks** and can be enforced at the IP-address level, at the user-account level, or at the region level, for example. Rate limiting can also be implemented in tiers; for instance, a type of network request could be limited to 1 per second, 5 per 10 seconds, and 10 per minute.

DoS Attack

Short for "denial-of-service attack", a DoS attack is an attack in which a malicious user tries to bring down or damage a system in order to render it unavailable to users. Much of the time, it consists of flooding it with traffic. Some DoS attacks are easily preventable with rate limiting, while others can be far trickier to defend against.

DDoS Attack

Short for "distributed denial-of-service attack", a DDoS attack is a DoS attack in which the traffic flooding the target system comes from many different sources (like thousands of machines), making it much harder to defend against.

Redis

An in-memory key-value store. Does offer some persistent storage options but is typically used as a really fast, best-effort caching solution. Redis is also often used to implement **rate limiting**.

Logging And Monitoring

In order to properly understand and diagnose issues that crop up within a system, it's critical to have mechanisms in place that create audit trails of various events that occur within said system.

So go ahead, unleash your inner Orwell and go full Big Brother on your application.

Logging

The act of collecting and storing logs--useful information about events in your system. Typically your programs will output log messages to its STDOUT or STDERR pipes, which will automatically get aggregated into a **centralized logging solution**.

Monitoring

The process of having visibility into a system's key metrics, monitoring is typically implemented by collecting important events in a system and aggregating them in human-readable charts.

Alerting

The process through which system administrators get notified when critical system issues occur. Alerting can be set up by defining specific thresholds on monitoring charts, past which alerts are sent to a communication channel like Slack.

Using stack log by google

- Server sends logs periodically to some Time series DB : **Prometheus**, influx DB, graphite.
- For making graphs : **Grafana**
- For alerting and monitoring : For instance when error rate goes up. You can hookup monitoring system to Slack.

Publish/Subscribe Pattern

Publish/Subscribe. Press/Tug. Produce/Consume. Push/Pull. Send/Receive. Throw/Catch. Thrust/Retrieve.

Three of these can be used interchangeably in the context of systems design. The others cannot.

Idempotent Processes : That is outcome of multiple executions is one time same output only

Ordering of messages : Order is same as they are sent it.

Replayability to a previous snapshot is also there.

Example : Apache Kafka, Google Pub/Sub

Publish/Subscribe Pattern

Often shortened as **Pub/Sub**, the Publish/Subscribe pattern is a popular messaging model that consists of **publishers** and **subscribers**. Publishers publish messages to special **topics** (sometimes called **channels**) without caring about or even knowing who will read those messages, and subscribers subscribe to topics and read messages coming through those topics.

Pub/Sub systems often come with very powerful guarantees like **at-least-once delivery**, **persistent storage**, **ordering** of messages, and **replayability** of messages.

Idempotent Operation

An operation that has the same ultimate outcome regardless of how many times it's performed. If an operation can be performed multiple times without changing its overall effect, it's idempotent. Operations performed through a **Pub/Sub** messaging system typically have to be idempotent, since Pub/Sub systems tend to allow the same messages to be consumed multiple times.

For example, increasing an integer value in a database is *not* an idempotent operation, since repeating this operation will not have the same effect as if it had been performed only once. Conversely, setting a value to "COMPLETE" *is* an idempotent operation, since repeating this operation will always yield the same result: the value will be "COMPLETE".

Apache Kafka

A distributed messaging system created by LinkedIn. Very useful when using the **streaming** paradigm as opposed to **polling**.

Cloud Pub/Sub

A highly-scalable Pub/Sub messaging service created by Google. Guarantees **at-least-once delivery** of messages and supports "rewinding" in order to reprocess messages.

MapReduce

"MapReduce is a programming model for processing and generating big data sets with a parallel, distributed algorithm on a cluster."

Does Wikipedia's nebulous definition confuse you? Of course it does. In this video, we'll map out this complex topic and reduce it to clear, easily-understood concepts. See what we did there?

What Map function to use? What Reduce function to use?

MapReduce

A popular framework for processing very large datasets in a distributed setting efficiently, quickly, and in a fault-tolerant manner. A MapReduce job is comprised of 3 main steps:

- the **Map** step, which runs a **map function** on the various chunks of the dataset and transforms these chunks into intermediate **key-value pairs**.
- the **Shuffle** step, which reorganizes the intermediate **key-value pairs** such that pairs of the same key are routed to the same machine in the final step.
- the **Reduce** step, which runs a **reduce function** on the newly shuffled **key-value pairs** and transforms them into more meaningful data.

The canonical example of a MapReduce use case is counting the number of occurrences of words in a large text file.

When dealing with a MapReduce library, engineers and/or systems administrators only need to worry about the map and reduce functions, as well as their inputs and outputs. All other concerns, including the parallelization of tasks and the fault-tolerance of the MapReduce job, are abstracted away and taken care of by the MapReduce implementation.

Distributed File System

A Distributed File System is an abstraction over a (usually large) cluster of machines that allows them to act like one large file system. The two most popular implementations of a DFS are the **Google File System** (GFS) and the **Hadoop Distributed File System** (HDFS).

Typically, DFSs take care of the classic **availability** and **replication** guarantees that can be tricky to obtain in a distributed-system setting. The overarching idea is that files are split into chunks of a certain size (4MB or 64MB, for instance), and those chunks are sharded across a large cluster of machines. A central control plane is in charge of deciding where each chunk resides, routing reads to the right nodes, and handling communication between machines.

Different DFS implementations have slightly different APIs and semantics, but they achieve the same common goal: extremely large-scale persistent storage.

Hadoop

A popular, open-source framework that supports MapReduce jobs and many other kinds of data-processing pipelines. Its central component is **HDFS** (Hadoop Distributed File System), on top of which other technologies have been developed.

Security and HTTPS

Man-In-The-Middle Attack

An attack in which the attacker intercepts a line of communication that is thought to be private by its two communicating parties.

If a malicious actor intercepted and mutated an IP packet on its way from a client to a server, that would be a man-in-the-middle attack.

MITM attacks are the primary threat that encryption and **HTTPS** aim to defend against.

Symmetric Encryption

A type of encryption that relies on only a single key to both encrypt and decrypt data. The key must be known to all parties involved in communication and must therefore typically be shared between the parties at one point or another.

Symmetric-key algorithms tend to be faster than their asymmetric counterparts.

The most widely used symmetric-key algorithms are part of the Advanced Encryption Standard (**AES**).

Asymmetric Encryption

Also known as public-key encryption, asymmetric encryption relies on two keys—a public key and a private key—to encrypt and decrypt data. The keys are generated using cryptographic algorithms and are mathematically connected such that data encrypted with the public key can only be decrypted with the private key.

While the private key must be kept secure to maintain the fidelity of this encryption paradigm, the public key can be openly shared.

Asymmetric-key algorithms tend to be slower than their symmetric counterparts.

AES

Stands for **Advanced Encryption Standard**. AES is a widely used encryption standard that has three symmetric-key algorithms (AES-128, AES-192, and AES-256).

Of note, AES is considered to be the "gold standard" in encryption and is even used by the U.S. National Security Agency to encrypt top secret information.

HTTPS

The **HyperText Transfer Protocol Secure** is an extension of **HTTP** that's used for secure communication online. It requires servers to have trusted certificates (usually **SSL certificates**) and uses the Transport Layer Security (**TLS**), a security protocol built on top of **TCP**, to encrypt data communicated between a client and a server.

TLS

The **Transport Layer Security** is a security protocol over which **HTTP** runs in order to achieve secure communication online. "HTTP over TLS" is also known as **HTTPS**.

SSL Certificate

A digital certificate granted to a server by a **certificate authority**. Contains the server's public key, to be used as part of the **TLS handshake** process in an **HTTPS** connection.

An SSL certificate effectively confirms that a public key belongs to the server claiming it belongs to them. SSL certificates are a crucial defense against **man-in-the-middle attacks**.

Certificate Authority

A trusted entity that signs digital certificates—namely, SSL certificates that are relied on in **HTTPS** connections.

TLS Handshake

The process through which a client and a server communicating over **HTTPS** exchange encryption-related information and establish a secure communication. The typical steps in a TLS handshake are roughly as follows:

- The client sends a **client hello**—a string of random bytes—to the server.
- The server responds with a **server hello**—another string of random bytes—as well as its **SSL certificate**, which contains its **public key**.
- The client verifies that the certificate was issued by a **certificate authority** and sends a **premaster secret**—yet another string of random bytes, this time encrypted with the server's public key—to the server.
- The client and the server use the client hello, the server hello, and the premaster secret to then generate the same **symmetric-encryption** session keys, to be used to encrypt and decrypt all data communicated during the remainder of the connection.

TLS Handshake important : After pre-master secret is generated, session keys are generated. For instance the sessions keys can be generated via cHello, sHello and pre master secret

API Design

API Design to begin with API definition which will be just a text file. Most important is list of endpoint and which ones will need pagination.

Pagination

When a network request potentially warrants a really large response, the relevant API might be designed to return only a single **page** of that response (i.e., a limited portion of the response), accompanied by an identifier or token for the client to request the next page if desired.

Pagination is often used when designing **List** endpoints. For instance, an endpoint to list videos on the YouTube Trending page could return a huge list of videos. This wouldn't perform very well on mobile devices due to the lower network speeds and simply wouldn't be optimal, since most users will only ever scroll through the first ten or twenty videos. So, the API could be designed to respond with only the first few videos of that list; in this case, we would say that the API response is **paginated**.

CRUD Operations

Stands for **Create, Read, Update, Delete** Operations. These four operations often serve as the bedrock of a functioning system and therefore find themselves at the core of many APIs. The term **CRUD** is very likely to come up during an API-design interview.

```
api.txt
1 # API Definition
2
3 ## Entity Definitions
4 ### Charge:
5 - id: uuid
6 - customer_id: uuid
7 - amount: integer
8 - currency: string (or currency_code enum)
9 - status: enum ["succeeded", "pending", "failed"]
10
11 ### Customer:
12 - id: uuid
13 - name: string
14 - address: string
15 - email: string
16 - card: Card
17
18 ### Card
19
20 ## Endpoint Definitions
21 ### Charges
22 CreateCharge(charge: Charge)
23 | => Charge
24 GetCharge(id: uuid)
25 | => Charge
26 UpdateCharge(id: uuid, updatedCharge: Charge)
27 | => Charge
28 ListCharges(offset: integer, limit: integer)
29 | => Charge[]
30 CaptureCharge(id: uuid)
31 | => Charge
32
33 ### Customers
34 CreateCustomer(customer: Customer)
35 | => Customer
36 GetCustomer(id: uuid)
37 | => Customer
38 UpdateCustomer(id: uuid, updatedCustomer: Customer)
39 | => Customer
40 DeleteCustomer(id: uuid)
41 | => Customer
42 ListCustomers(offset: integer, limit: integer)
43 | => Customer[]
```

Important Links

- 1.<https://www.geeksforgeeks.org/search-an-element-in-a-sorted-and-pivoted-array/>
- 2.<https://www.geeksforgeeks.org/find-position-element-sorted-array-infinite-numbers/>
- 3.<https://www.geeksforgeeks.org/time-complexities-of-different-data-structures/>
- 4.<https://www.indeed.com/career-advice/interviewing/system-design-interview-questions>

Things to do:

- 1.Reddit and Uber API design questions
- 2.Pick your favourite service and go through their document. For instance stripe
- 3.3 Pick API, design API and match with what they actually have, Eg Youtube API