System Design

Table of Contents

Client – Server Model

A client is essentially a thing that talks to computers, a server is a thing that talks to clients (vice-versa). For example, imagine a browser(client) and a algo Expert(Server). HTTP request will request information and the server will send response via the return address stored in the original request.

Client - A machine or process that requests data or a service from a server (A single machine or software could be both a client and a server)

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

IP Address – An address given to each machine connected to the public internet

Port – In order for multiple programs to listen for new network connections on the same machine without colliding they pick a port to listen to. (2^16 in total)

DNS – Domain Name System, it describes the entities and protocols involved in the translation of a DN to IP address.

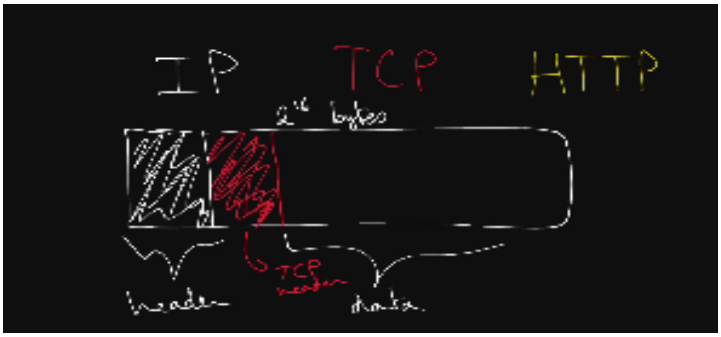
Network Protocols

For communication between machines, this allows us to understand how a system does this.

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. They are made up of bytes (memory allocation, 8 bits = 1 byte)

- **Header:** contains the source IP address (where its coming from) and the destination IP address (where its going), it also contains how much data the request contains the and the version of IP being used (IPv4 /IPv6 etc).
- **Data:** Can contain up to 2^16 bytes (isn't actually that much) – in this case you would have multiple IP packets (which makes things more complicated, that is you don't have a way of guaranteeing that all of these packets will be delivered or ordered correctly).

To overcome this, we then apply the TCP header which takes up more bytes in the packet. This will include information about the ordering of packets. The core idea about TCP is that when first accessing as server you will create a TCP connection (Handshake), a special TCP interaction where the two services send a few packets between each other to establish a connection. If one of the two machines don't send anything in a given period, the connection will time out (or its ended by a server/client). However, what It lacks is a robust framework that allows developers to define meaningful and easy to use communication channels for clients and servers in a system.



This is where http comes into play, which is a protocol that was built on top of TCP which introduces a higher-level abstraction above the two previous protocols known as the request-response paradigm (see screenshot in http section). Essential for implementing business logic!

TCP – Network protocol built on top of the 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 implement on top of TCP. Clients make HTTP requests and server 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, asides from bytes. An IP pack consist of:

- An IP Header, which contains the source and destination IP addresses as well as other information just related to the network
- A payload, which is just the data being sent over the network

Storage

Information storage is incredibly complex and is an extremely important concept to understand!

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 ling 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 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 must 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 large scale systems, 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 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 store for a long time and SSD will be used for data that's frequently accessed and updated.

Memory: Short from 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.

A Database is not a magical box, what it really is, is a server.

Latency & Throughput

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

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**: 0.25ms
- **Reading 1 MB from SSD:** 1ms
- **Transfer 1 MB over 1gbps network:** 10ms
- **Reading 1MB from HDD:** 20ms
- **Inter-continental round trip:** 150ms

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.

Depending on what context we're talking about, latency will still apply in the same dynamic. When you're design a system, you'll typically want to optimise them by lowering the latency. However certain systems might not care too much about the latency (Games really do!). Imagine we have 5 clients all requesting from the same server. The throughput would be how many requests the sever can handle at a given time (per second). 50mbs means the network can handle transferring 50megabytes per second. Blindly increasing throughput doesn't make sense, a better way to fix this system is to have multiple servers handle these requests (cost). You cannot make assumptions about latency or throughput based on the other.

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 (measured for a given year)).

High Availability: Used to describe systems that have particularly high levels of availability, for examples 5 nines of more – Abbreviated to "Ha" typically.

Nines: Typically refers to percentages of uptime. For example, 5 nines of availability means an uptime of 99.999% of the time. List of downtimes expected:

- 99 % (two nines) : 87.7hours
- 99.9% (three): 8.8hours
- 99.99%: 52.6 mins
- 99.999%: 5.3mins

Redundancy: The process of replicating parts of 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 systems availability, amongst other things SLA's are typically made up of one more SLO's

SLO: Short for "service level objective", an SLO is a guarantee given to a customer by a service provider. Typically made on a systems availability, amongst other things.

In reality, there is an implied availability guarantee (you expect it to operational all the time). Some systems will have an explicit availability guarantee. It is difficult to ensure high levels of availability (it could come at higher costs such as higher latency or lower throughput). For example, the area in which payments are handled would have a very high availability, whereas the dashboard in which a company can monitor sales would not be as critical to ensure uptime. You as the designer, need to understand what is core to your product and what is not (critical).

You want to make sure that your system doesn't have single points of failure (remove this by using **redundancy**). For example, if you had clients interacting with a server and a database. In this case you would introduce multiple servers. (add more machines (gets more expensive etc)). Passive redundancy is where you have multiple components at a given layer in a system and if it any point one of the dies, nothing is really going to happen. Active redundancy is where you have multiple machines where only a few handle traffic and if one of these fails the other machines will somehow know that it failed and takeover.

Caching

Cache: A price 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 the truth for that data (i.e. the main database behind cache) gets updated and the cache doesn't.

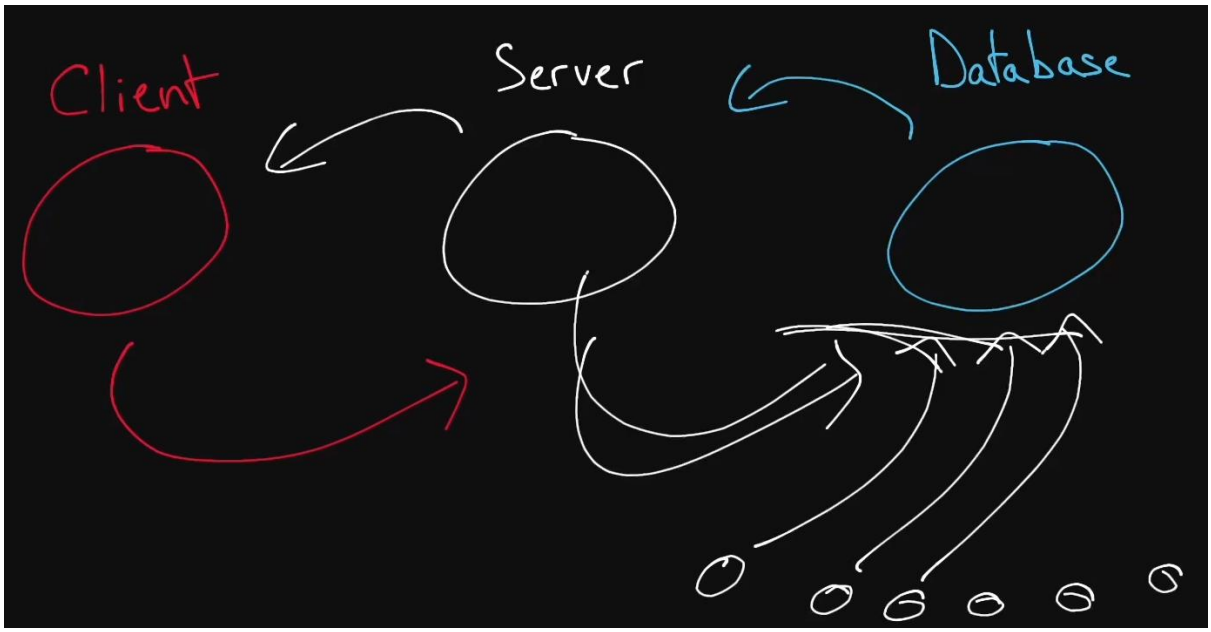
Cache Hit: When the requested data is found in the cache.

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 loud 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 be far better than the latency to your servers. A CDN's server are often referred to as PoPs (points of presence). Two of the most popular CDN's are **Cloudflare** and **Google Cloud CDN.**

Video Notes: There is actually a lot of caching that happens at the CPU and hardware level (not necessarily relevant here but its good to understand that it happens at many different levels in a system. An instance where caching is very helpful is where you perform a very computationally long request and you want to save the result of that request in order to save time, or you may have clients that make the same request multiple times. Imagine if we have a bunch of servers, all making the same request, Here instead of reading from the database a million times we could cache the database result and the servers would then point to that result.



Another concrete example of caching is running code on algo expert. When users run algoexperts solutions the results to the code have been cached

```
const database = require('./database');
const express = require('express');

const app = express();
const cache = {};

app.get('/nocache/index.html', (req, res) => {
  database.get('index.html', page => {
    res.send(page);
  });
});

app.get('/withcache/index.html', (req, res) => {
  if ('index.html' in cache) {
    res.send(cache['index.html']);
    return;
  }

  database.get('index.html', page => {
    cache['index.html'] = page;
    res.send(page);
  });
});

app.listen(3001, function() {
  console.log('Listening on port 3001!');
});
```

This example shows how a simple application can retrieve information from a database if it has been cached. The first time we request the data we store the key for that data in a JavaScript object (cache). The next time we load the page, the function will look to see if our cache contains that key. If it does, we just return that value to the user.

However, what about if a user wants to write posts etc? You might want to add the post to the server and database. If you're editing a post how do you know whether to update the cache or the database?

There are two popular types of caching to deal with this. The first is called a **write-through** cache. When you write a piece of information it will be stored in both the database and the cache

at the same time. If you wanted to edit that post, the request would first be sent to the cache were the current object is overwritten and then that would send the request to the database where it is overridden (The downside of this is that you still have to go to the database). The second type of cache is called the **write-back** cache. Instead when a user edits a post it will be updated in the cache and sent back to the user (this leaves the cache and database out of sync). Behind the scenes the system will asynchronously update the values in the database with the values in the cache (every 1minute etc (you choose the schedule)). The downside to this method is that if you somehow lost the information in the database may not be up-to date, luckily there are ways to mitigate this (later).

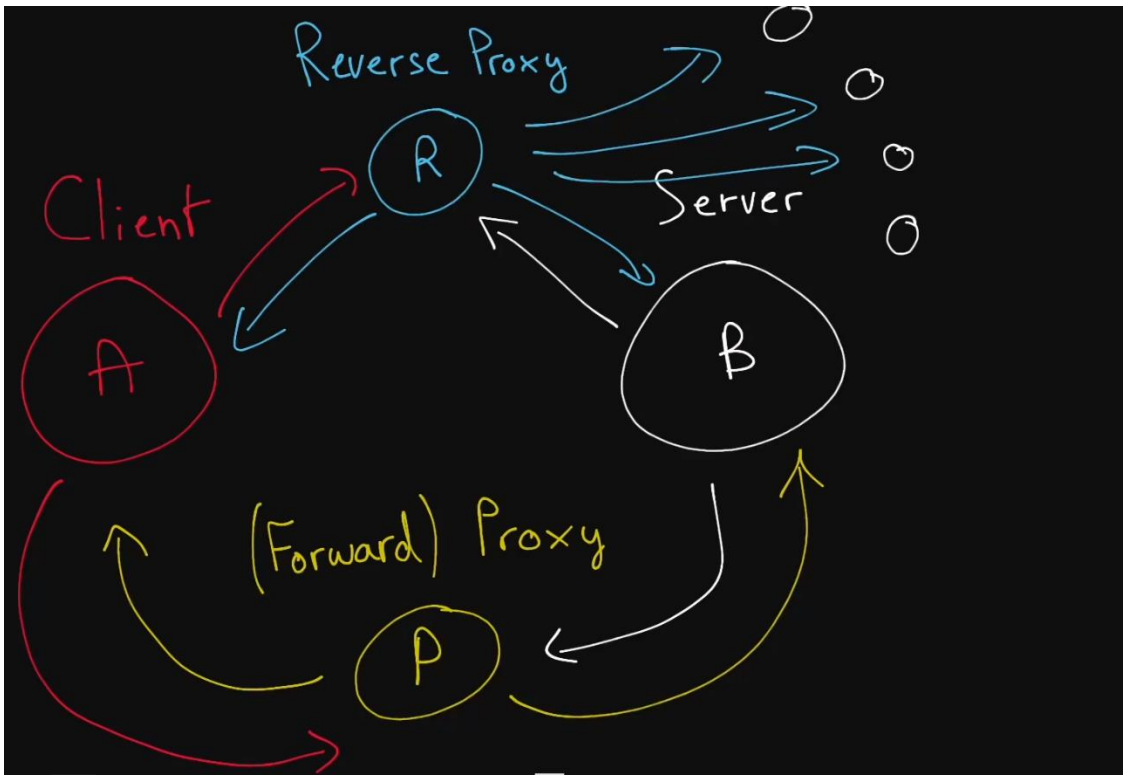
Proxies

Forward Proxy: A Sever 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 just referred to as proxies.

Reverse Proxy: A Sever that sits between clients and servers and acts on behalf of the servers, typically used for logging, load balancing and caching.

Nginx: Pronouced "Engine X" is a very popular webserver that is used as a reverse proxy/ load balancer.

Video Notes: When a client requests something from a server it will first go to the (forward) Proxy which will then forward to the server and vice-versa. (basically how VPN's work in a simplified way. Reverse proxies are a little more complicated and act on behalf of the server. It makes the client think they're interacting with a server where in fact they are interacting with the reverse proxy. You should always want to implement a reverse proxy as it can (Log, cache and prevent certain queries). You can also use a reverse proxy as a load balancer. A load balancer is a server that can distribute a request load between a bunch of servers.

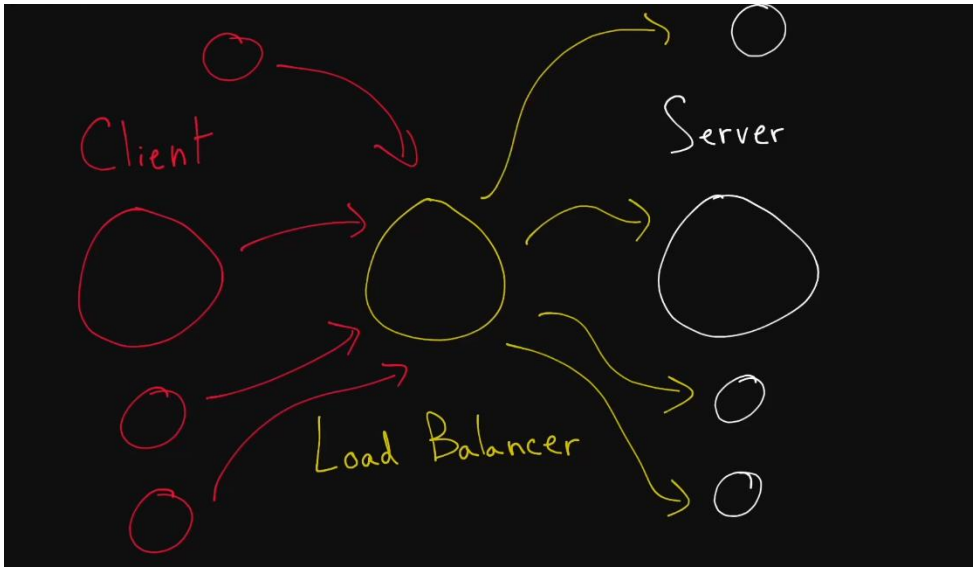


Load Balancers

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 best performance metrices, 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 severs will receive a lot more traffic than other's, thus creating a "hot-spot".



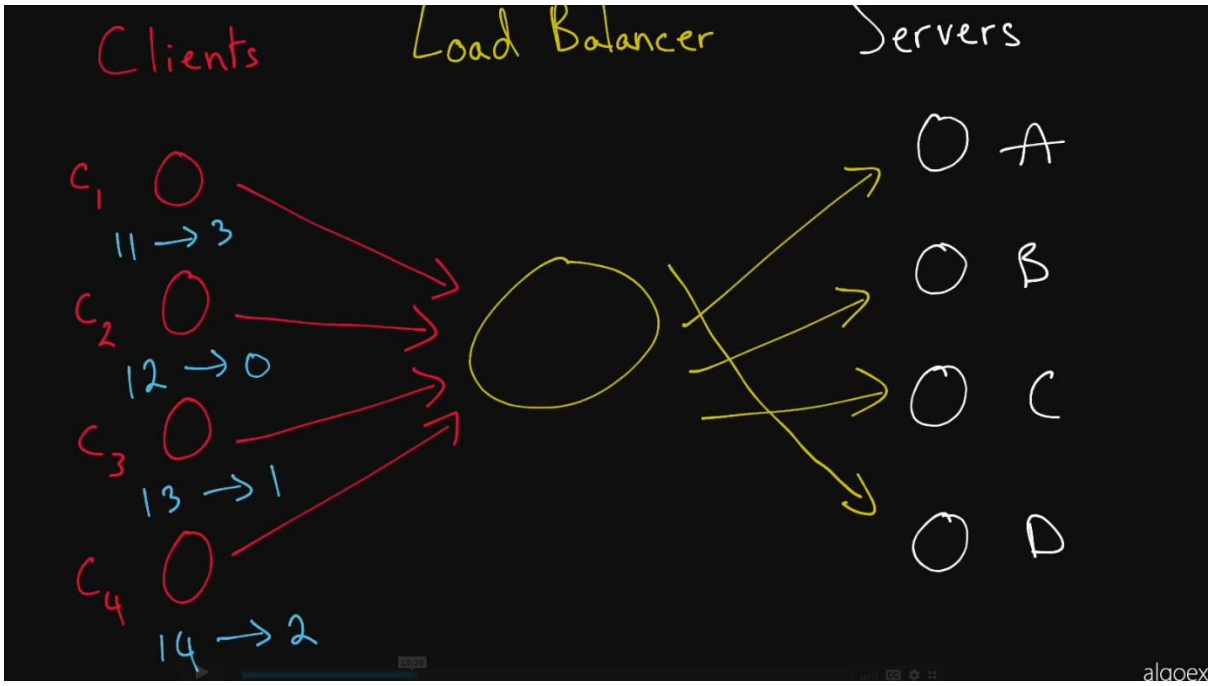
Hashing

Hashing Function: A function that takes in a specific data type (such as a string or an identifier) and outputs a different number. Different inputs may have the same output, but a good hashing function attempts to minimize those hashing collisions (which his equivalent to maximising uniformity).

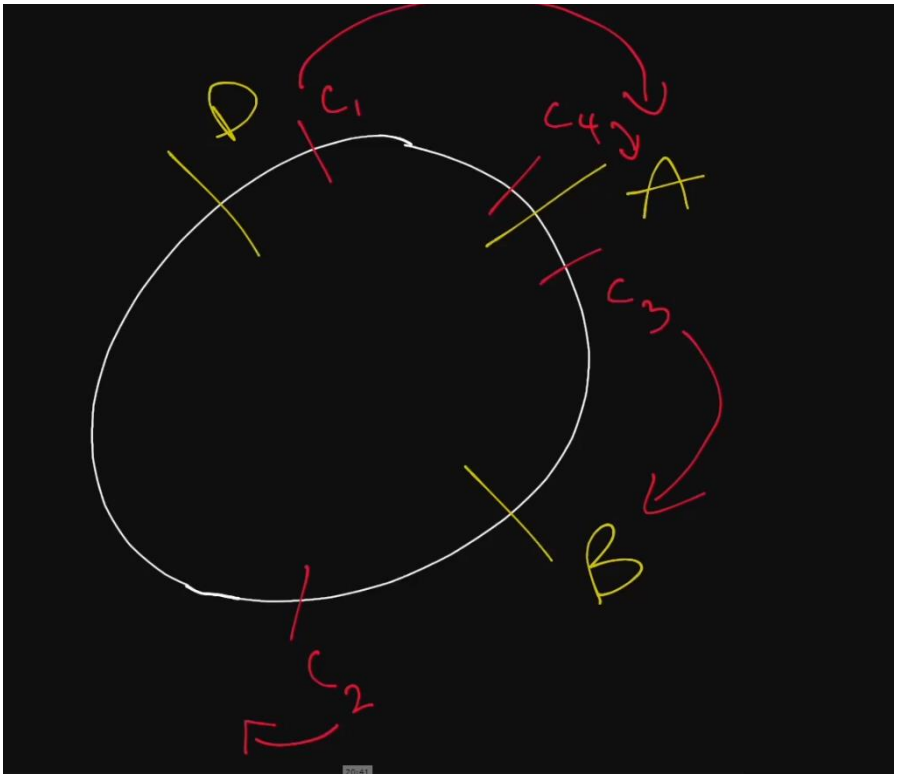
Consistent Hashing: A type of hashing that minimizes the number of keys that need to be remapped when a hash table gets resized. It's often used by load balancers to distribute traffic to servers; it minimizes the number of requests that get forwarded to different servers when new severs are added or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



In this simple example we’re hashing the clients name and then taking the mod of the number of servers to see which client points to which server. In the case of client one we get 11%4 which is 3. This means that client one will point to server D. This way the cache stored in serve D will always be relevant to client one and will prevent cache misses. This is just as example as more complex systems will use common hashing functions such as SHA. We can experience problems with this concept however, for example what if a server dies or we need to add a new server for example E and then our pointers would no longer work so we need to come up with a better solution (this is where different types of hashing come in)



This is known as consistent hashing. Instead of imagining our clients and servers are acting in a line like before we instead imagine it as a circle. We then hash the servers and place them at their specified location round the circle. When clients are hashing depending on what direction you choose (clockwise in this case) they point to the nearest server in that direction. For example, say server C dies which in this case it has. Instead of having to redo the entire hashing function we can simply point those clients to the next nearest server (D in this case). The same logic applies to adding new servers. By using this technique, we can minimise the number of redirecting we have to do when making changes to the system.

You could add more slots for the same servers. For example, server A could appear at different points on the circle as its more powerful than the rest and would therefore be more likely to be hit.

Relational Databases

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

Non-Relational Database: Is a type of database that is free of the imposed tabular-like structure and are often referred to as NoSQL databases (Used this for my dissertation Docs/pages).

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 qualities (will probably be relevant to work at BNY since its finance based).

- **Atomicity:** The operations that constitute to the transaction will either all succeed, or all fail. There is no in-between state. Imagine you're making a money transaction from one bank account to another. You need to request money from one bank and increase in another. If one of these fail then they all fail (rolled back).
- **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 (essentially as if they had been put in a queue).
- **Durability:** Any commit transaction is written to non-volatile storage. It will not be undone by a crash, power-loss, or network partition.

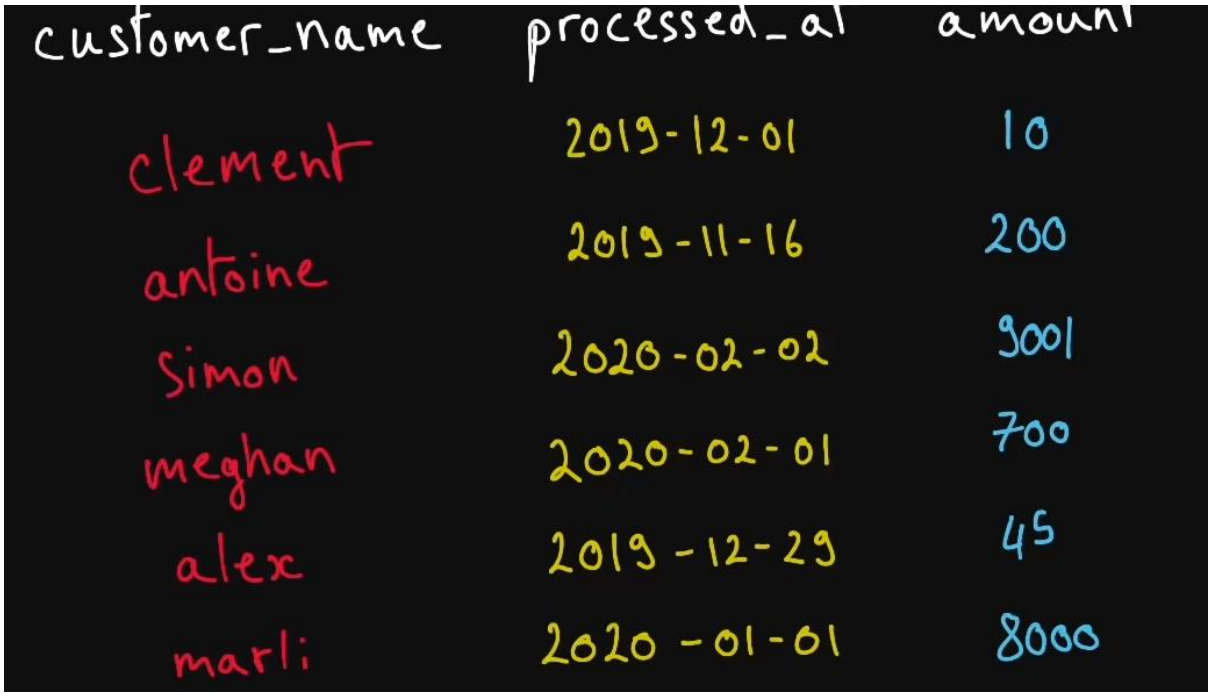
Database Index: A special auxiliary data structure that allows your database to perform certain queries much faster. Indexes can typically only exist to reference structured data, like data stored in relational databases. In practice, you create an index on one or multiple columns in your database to greatly speed up read queries that you run very often, with the downside of slightly longer writes to your database, since writes have to ask 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 period (could be 10seconds, or minutes

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

Notes: A SQL database must make use of Acid transactions. Database indexes are very complicated, essentially, they allow you to create a data structure to speed up searches. Think about it as a table of contents. For example, we could have a database index that stores the amounts in descending order, where each of these amounts would point to the relevant row in the actual database (bring a linear search down to a constant time operation etc). The trade of with a database index is that you're going to take up more space and the write operations will take longer as you're writing twice.



Key-Value Stores

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 Zookeepr.

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 strong consistent, high available key-value store. It's often used to store important configuration or to perform leader election.

Specialised Storage Paradigms

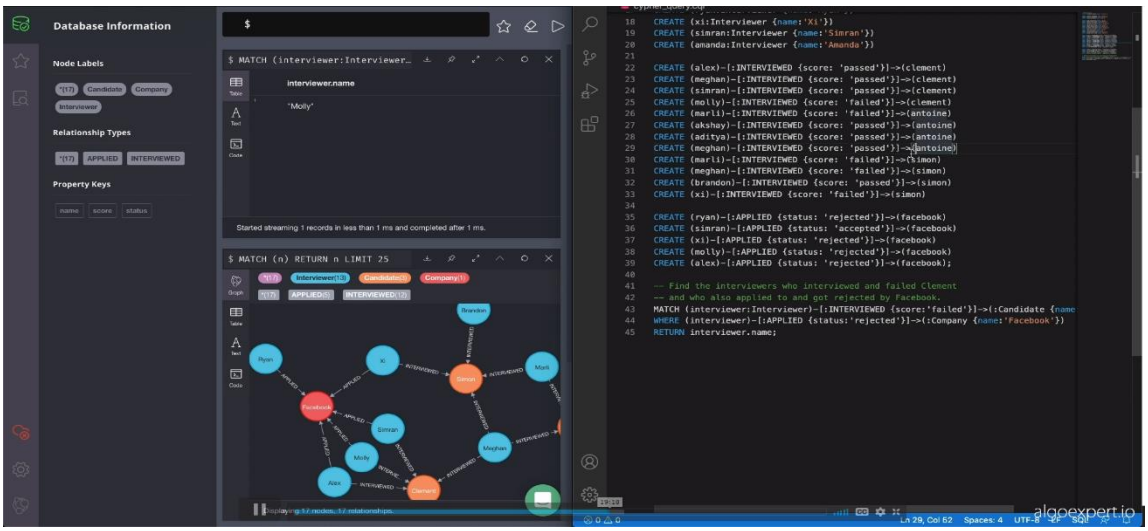
Blob Storage: Widely used kind of storage, in small- and large-scale systems. They don't really count as databases per se, partially because they only allow the user to store and retrieve data based on the name of the blob. This is sort of like a key-value store but usually blob stores have different 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 binaries, database snapshots or images and other static asses that a website might have.

Blob storage is rather complicated to have on premise, and only giant companies like Google and Amazon that infrastructure that supports it. So usually in the context of a System Design interview 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 analysing 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.



Cypher: A graph query language that was originally developed for the Neo4j graph database, but that has since been standardised 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 a 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 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 special region.

- 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, it's corresponding rectangle is subdivided into four quadrants (the corresponding node in the quadtree is given four children)
- 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 the perfect quadtree is an extremely fast operation that runs in log4(x) time (where x is the total number of locations) since a quadtree has four children nodes.

Google Cloud Storage: GCS is a blob storage service provided by Google.

S3: S3 is a blob storage service provided by Amazon through AWS

InfluxDB: A popular open-source time series database

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

Noe4j: A popular graph database that consists of nodes, relationships, properties and labels

Replication & Sharding

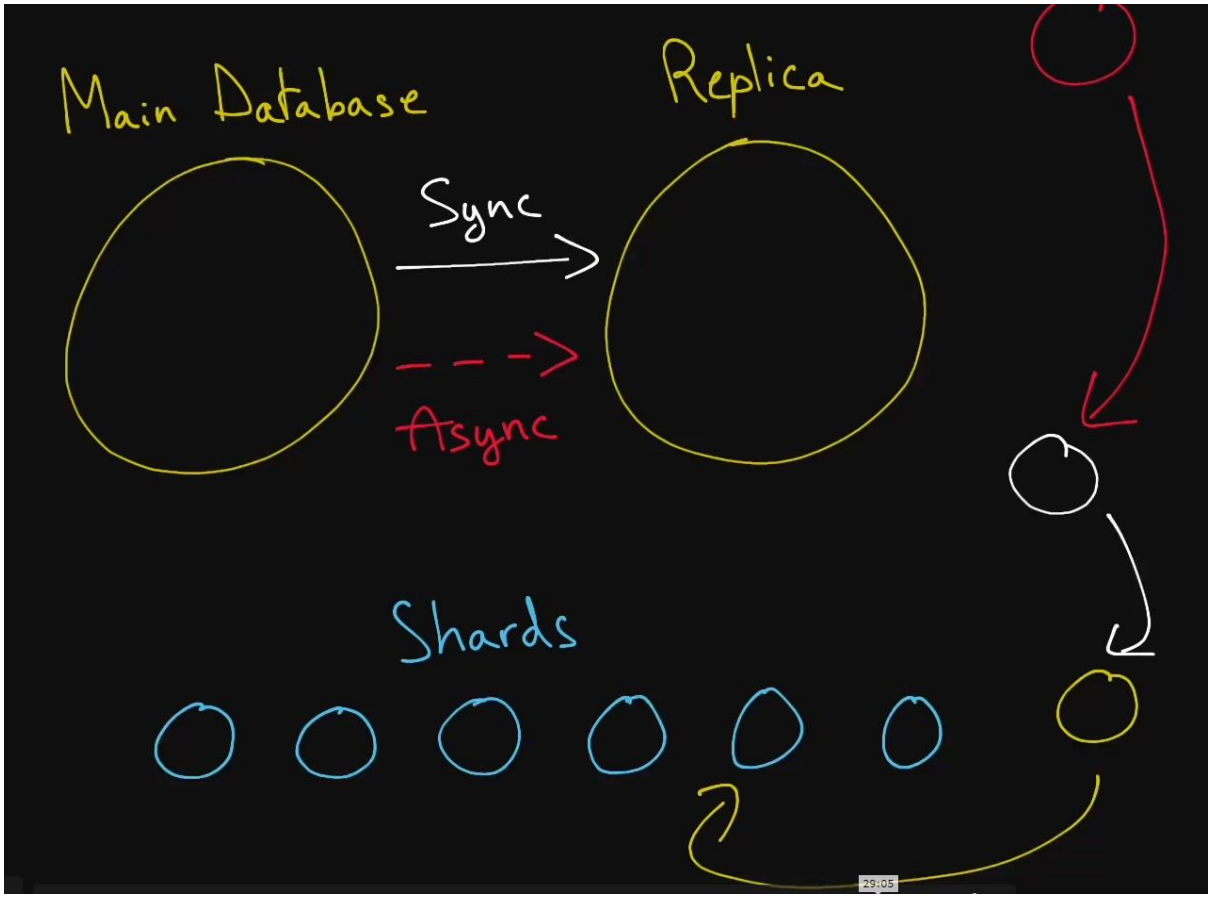
Replication: The act of duplicating the data from one database server to others. This is sometimes used to increase the redundancy of your system and tolerate regional failures for instance. Other times you can use replication to move data closer to your clients, thus decreasing latency of accessing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

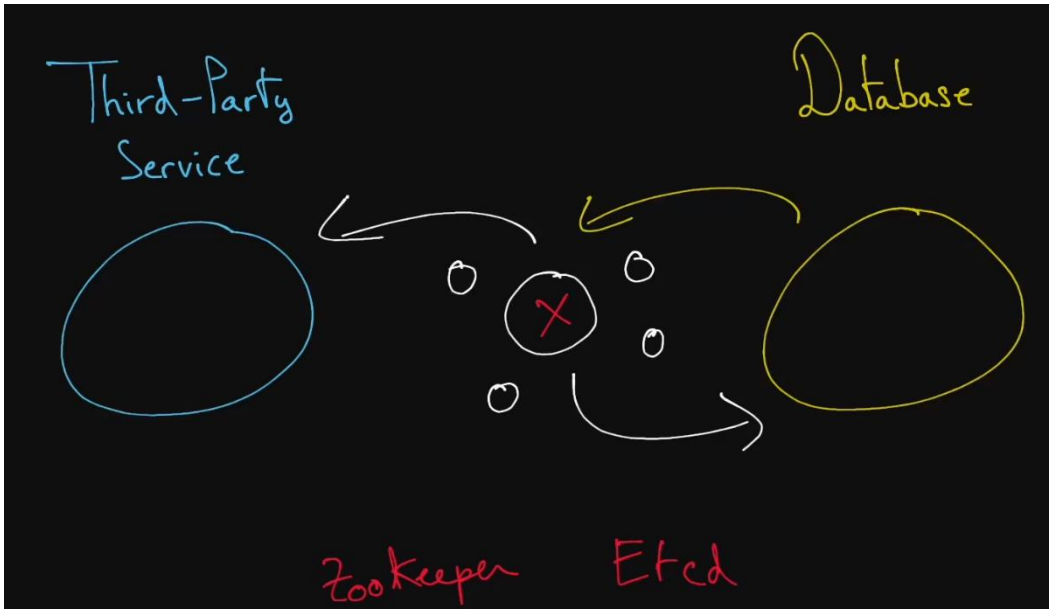
Notes: We need to ensure that the replica is synchronously updated at the same time as the main database, we also want it to be able to takeover the main database encase it goes down. Imagine the system is getting thousands of requests per seconds, a good way to fix this after looking at vertical scaling/horizontal scaling is to split up the data onto "shards" (bunch of little databases), this increases throughput and avoids duplication.



Leader Election

Leader Election: The process by which nodes in a cluster (for instance, servers in as et of servers) elect a so called "leader" amongst them, responsible for the primary operations of the service that the 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.



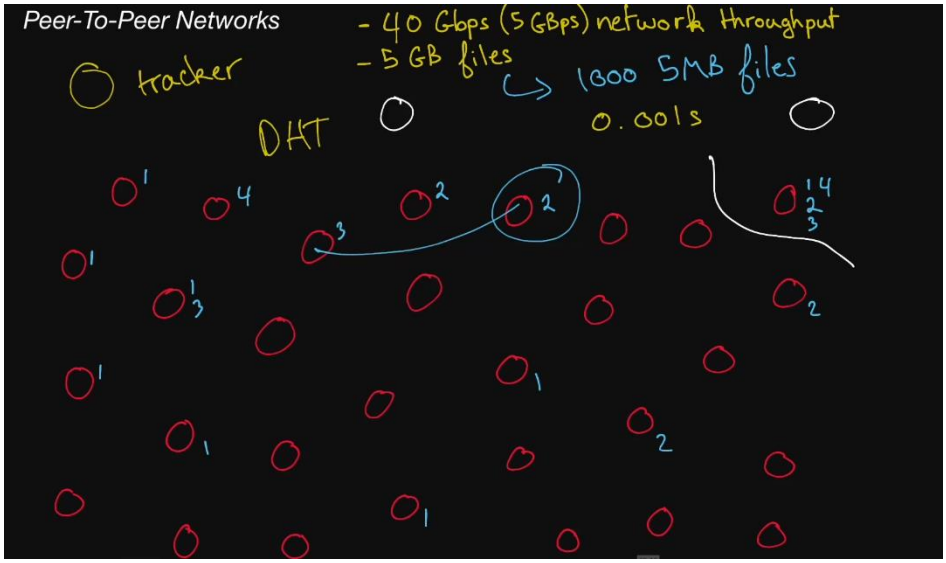
Paxos & Raft: Two consensus algorithms that, when implemented correctly, allow for the synchronization of certain operations, even a distributed setting.

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 been possible. Peer-to-peer networks are often used to file-distribution systems.

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

Notes: In this example we want to transfer 5gb files to 1000 machines (red(Peers)). We can split up our 5gb file into 1000 5mb files and send each tiny package to one of the peers. Each machine would now need the other 999 other files. This would take .999 seconds assuming we're transferring (5gbps). Torrenting works in the exact same way.

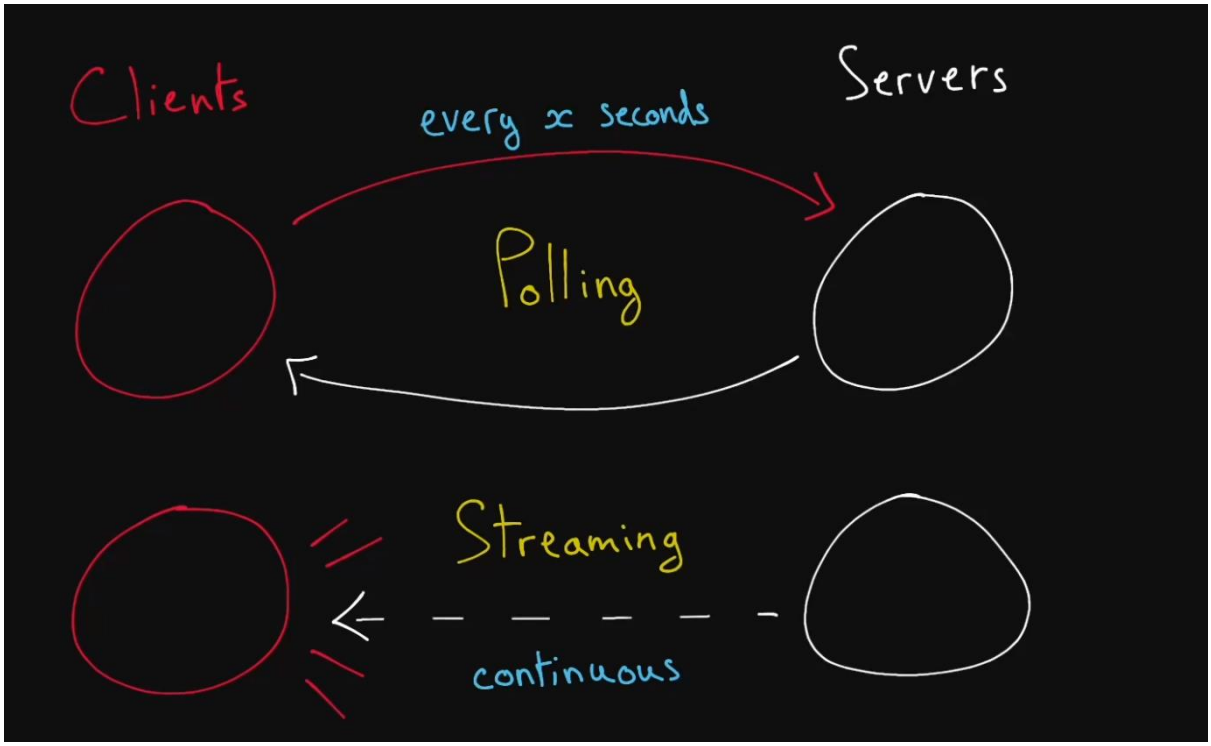


Polling and Streaming

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.

Notes: What if we're designing a system where a client wants a piece of information that is updated very regularly (i.e. the temperature outside). For this example, lets imagine a chat app were the client needs to be updated in real-time. We could use polling for the first issue but not for the second as we want to receive messages from other people instantly. Streaming would work here as it allows a client to get continuous data from the server. In this case the client will get data whenever the server has some that is relevant.



Configuration

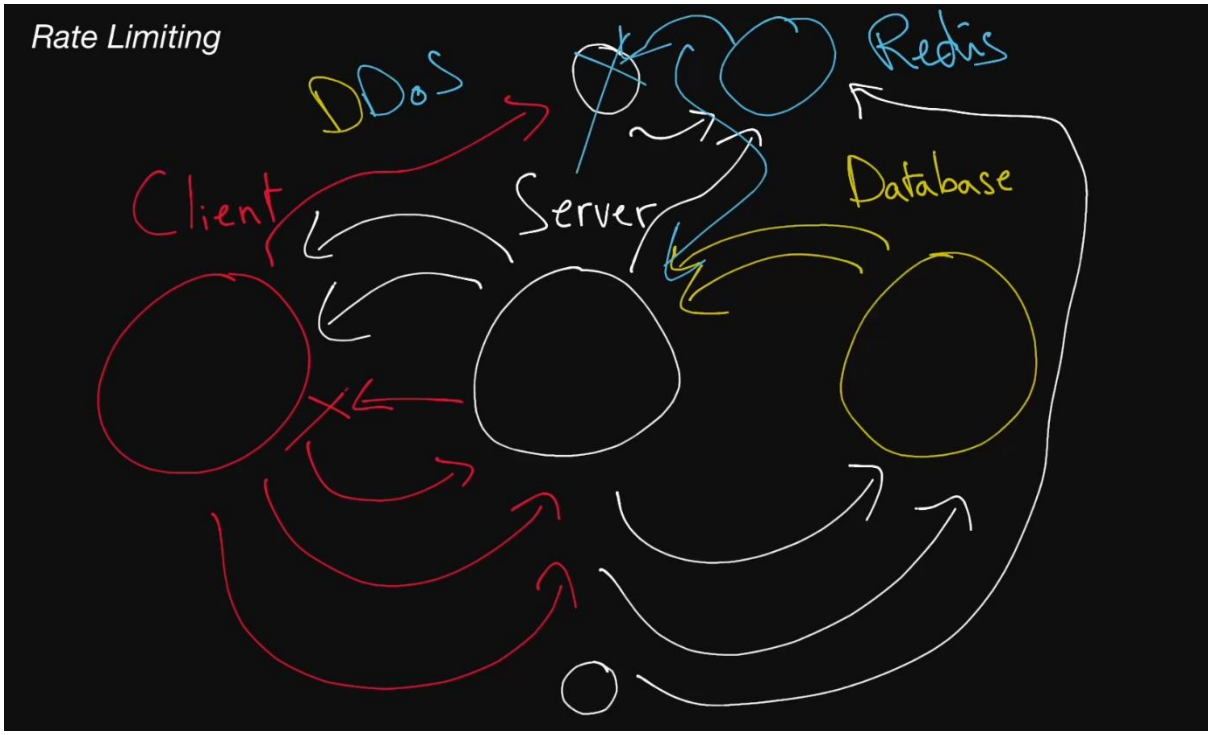
Configuration: A set of parameters or constants that are critical to a system. Configuration is typically written in JSON or YAML and can either **static**, meaning its hard coded in and shipped with your systems application code (like frontend code, for instance), or **dynamic** meaning that it livers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

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 regional 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 second and 10 per minute etc.

Dos Attack: Short for (Denial of service), 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), a DDoS attack in which the traffic flooding the system comes from many different sources (like thousands of machines), making it much harder to defend against.



Notes: Redis can be implemented to check that the number of requests are within a reasonable level. The servers interact with the Redis database first to check the rate limiting. Its hard to check rate limiting when the sources are coming from lots of machines!

Logging and Monitoring

Logging: The act of collecting the storing logs – useful information about events in your system. Typically, your programs will output log message to its STDOUT or STDEER pipes, which will automatically get aggregated into a **centralised logging solution.**

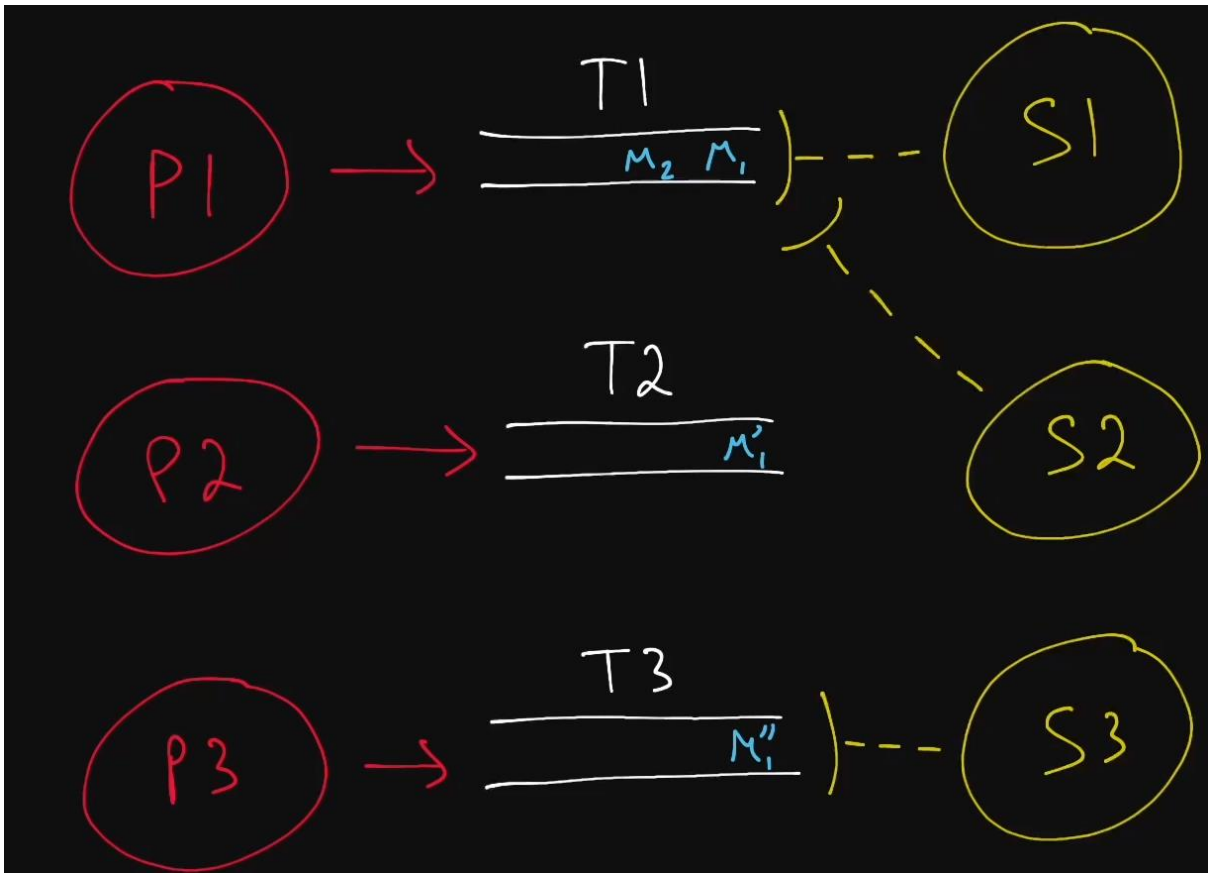
Monitoring: The process of having visibility into a system's key metrics, monitoring is typically implemented by collecting important events in a system and aggregating them in human-readable charts.

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

Publish/Subscribe Pattern

Publish/Subscribe Pattern: Often shortened to 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** or messages and **replay ability** of messages.



Notes: There are 4 key entities in a pub/sub system. **Publishers** are effectively servers that publish data to the topics. **Topics** are kind of like channels of specific information such as T1 could publish photos, T2 messages etc. **Subscribers** are our clients who subscribe to topics to receive the data they want. They do not communicate with the servers directly, more so through the topics (they don't know about each other). The fourth and final entity is the messages which is a sort of data that is relevant to the subscribers. These messages can really vary from a message to a chat application to an operation to execute some kind of trade. Every message will have sound kind of tracking system that keeps track of every subscribe listening to that topic.

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 it's 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 consumes 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 has performed only once. Conversely, setting a value to "complete" is an idempotent operation, since repeating this operation will always yield that same results: 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

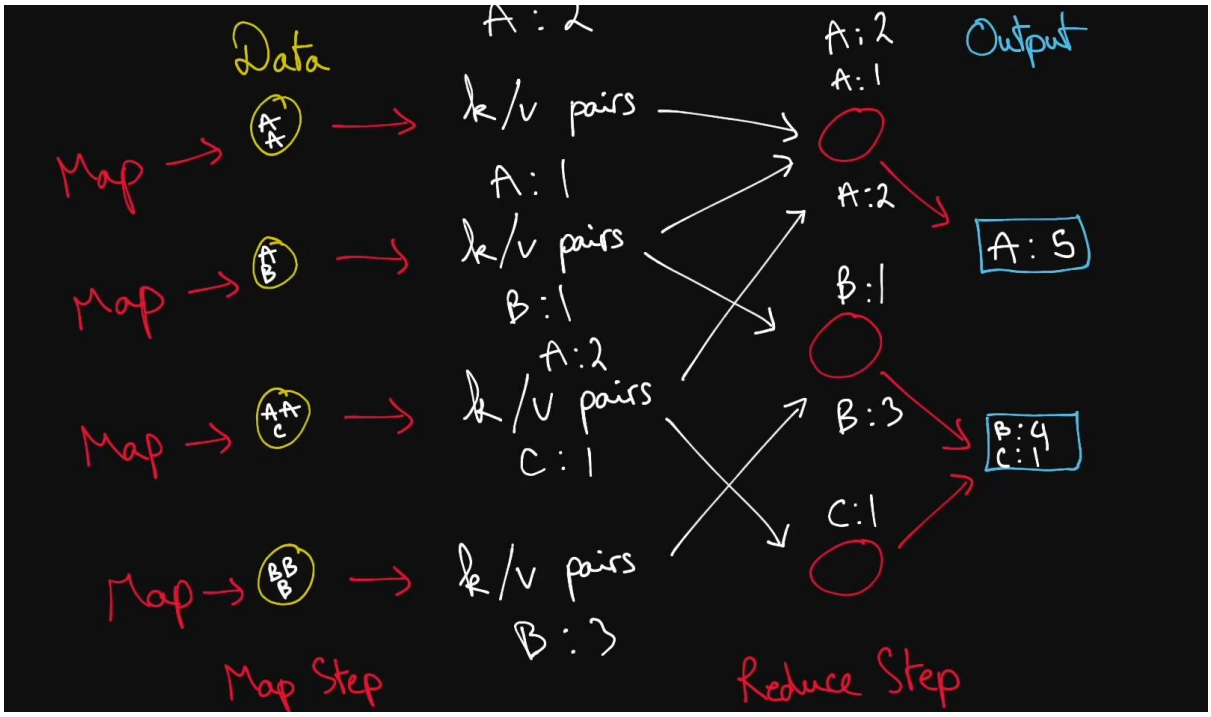
MapReduce is a programming model for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

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

- 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 reorganises 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 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 parallelisation of tasks and the fault-toleration 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 larger) cluster of machines that allows them to act like on 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 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 certain size

(4MB or 64MB, for instance) and those chunks are sharded across larger 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 **HDFS**, on top of which other technologies have been developed.

Security & HTTPS

Man-in the-Middle Attack: An Attack in which the attack 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 packed on its way from a client to a server it would be considered a Man in the middle attack.

MITM attacks the primary threat that encryption and HTTPS aims 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 parties at one point or another.

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

The most widely used symmetric 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 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 using 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 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 communications online. It requires servers to have trusted certificates (usually SSL certificated) and uses 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 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 defence 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 certificated, which contains its public key.
- The Client verifies that the certificate was issued by the certificate authority and sends a premaster secret, yet another string of random bytes, this time encrypted with the server's public key.
- The client and 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 or the connection.

API Design

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 they 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 a mobile device due to the lower network speeds and simply wouldn't be optimal, since most users will only ever scroll through the fist 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 the that API response is paginated.

CRUD Operations: stand 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.

Notes: Look at a few apis and see how they work (Had some experience with previous tests in which I was asked to consume that API).

Stripe API (https://stripe.com/docs/api)

Google Cloud API (https://cloud.google.com/apis)

Twitter API (https://developer.twitter.com/en/docs/twitter-api)