# System Design

## 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.

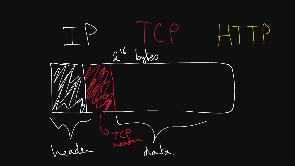
## Network Protocols

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).

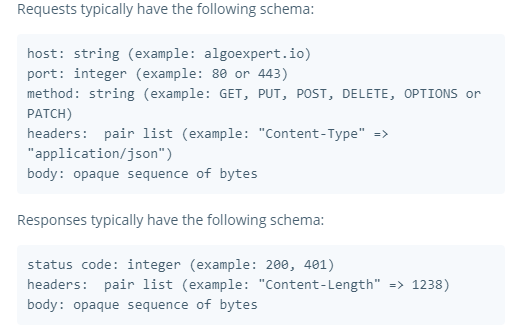
To overcome this, we then apply the TCP header which takes up more bytes in the packet. This will include information about the ordering of packets. The core idea about TCP is that when first accessing as server you will create a TCP connection (Handshake), a special TCP interaction where the two services 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.



**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.

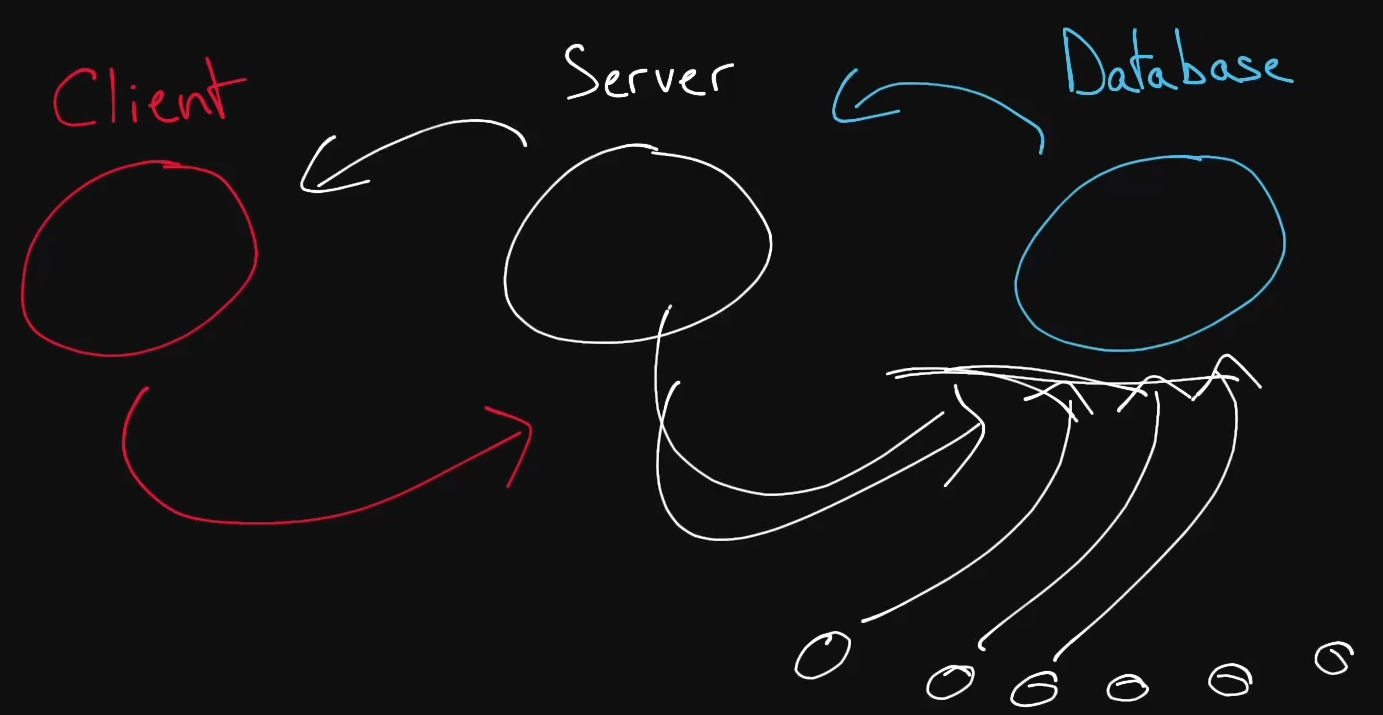
**Cache Hit:** When the requested data is found in the cache.

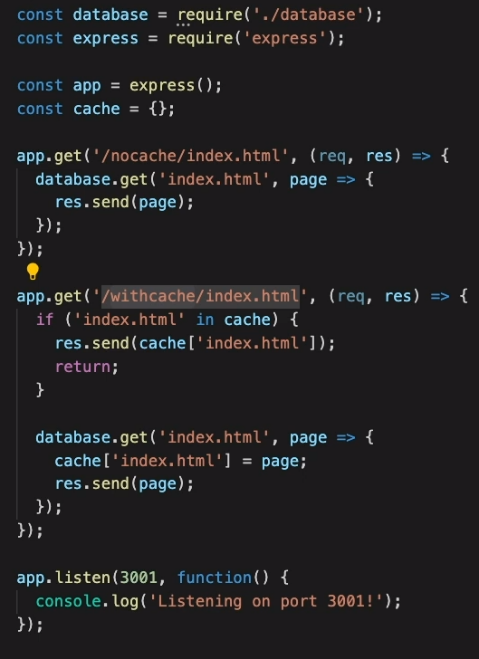
**Cache Miss:** When requested data could have been found in a cache but isn’t. This is typically used to refer to a negative consequence of a system failure or of a poor design choice. For example: “if a server goes down, our loud balancer will have to forward requests to a new server, which will result 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

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).

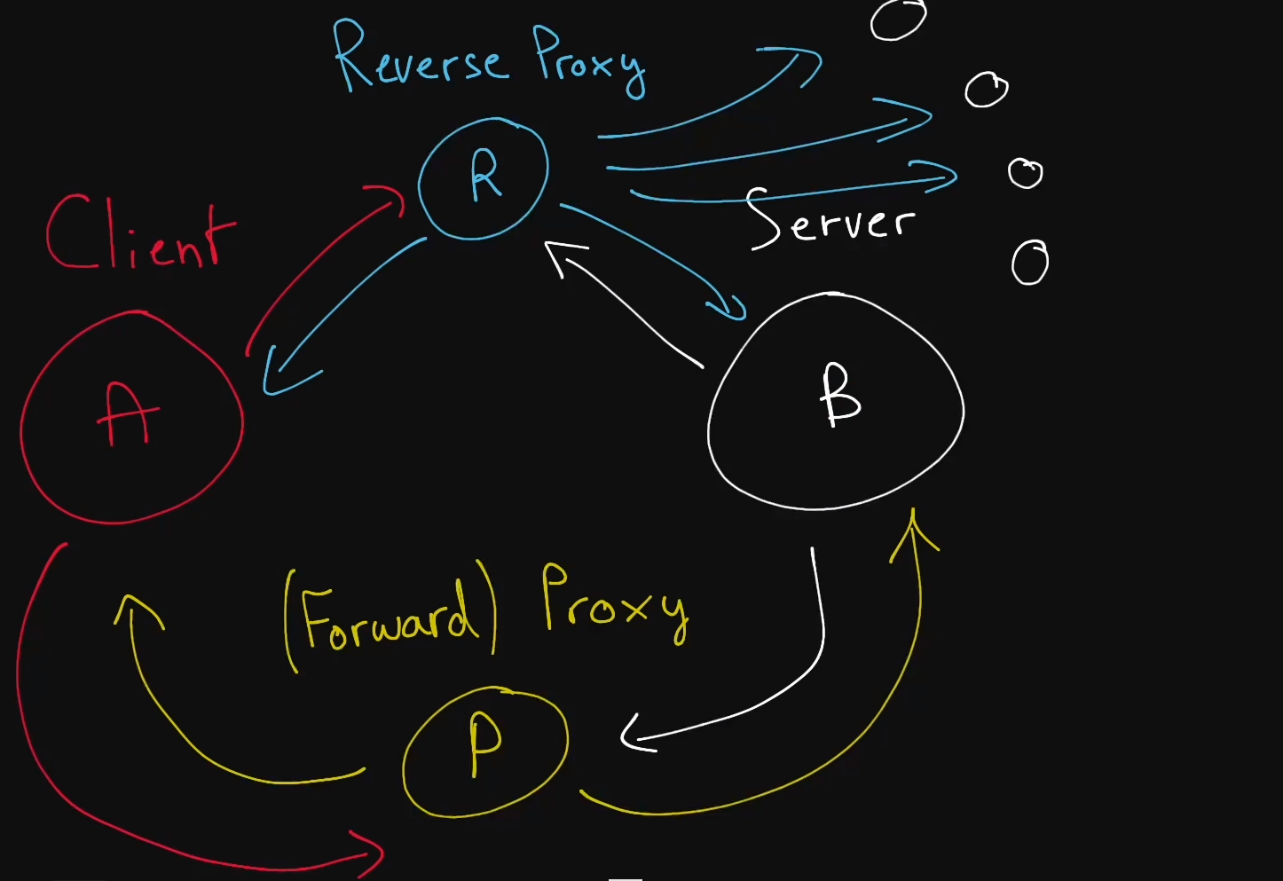
### Proxies

**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.

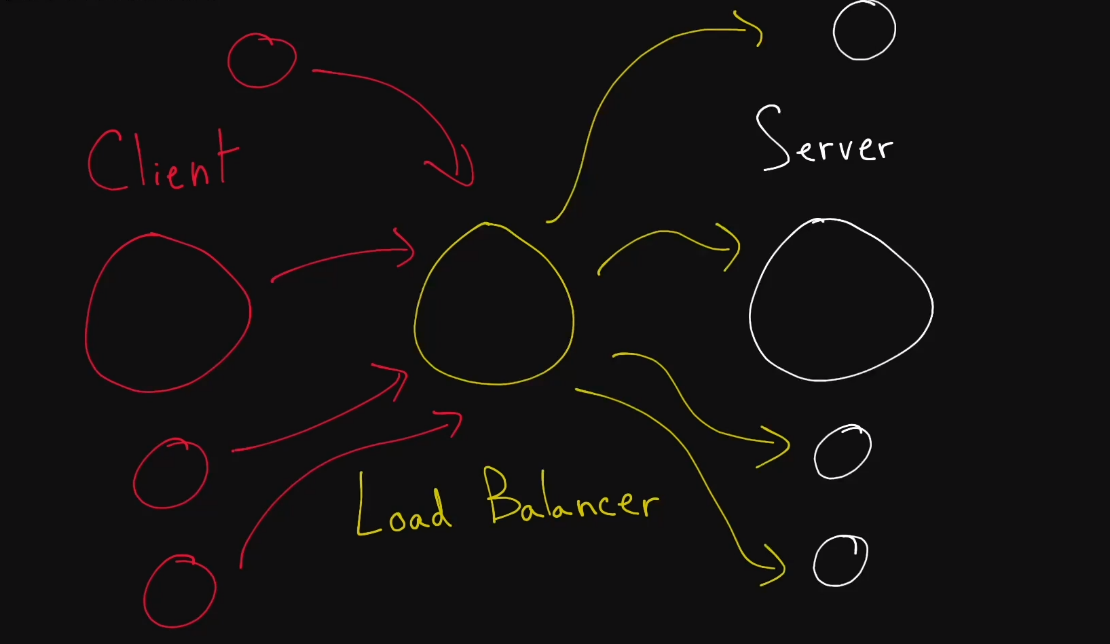
**Video Notes:** When a client requests something from a server it will first go to the (forward) Proxy which will then forward to the server and vice-versa. (basically how VPN’s work in a simplified way. Reverse proxies are a little more complicated and act on behalf of the server. It makes the client 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.

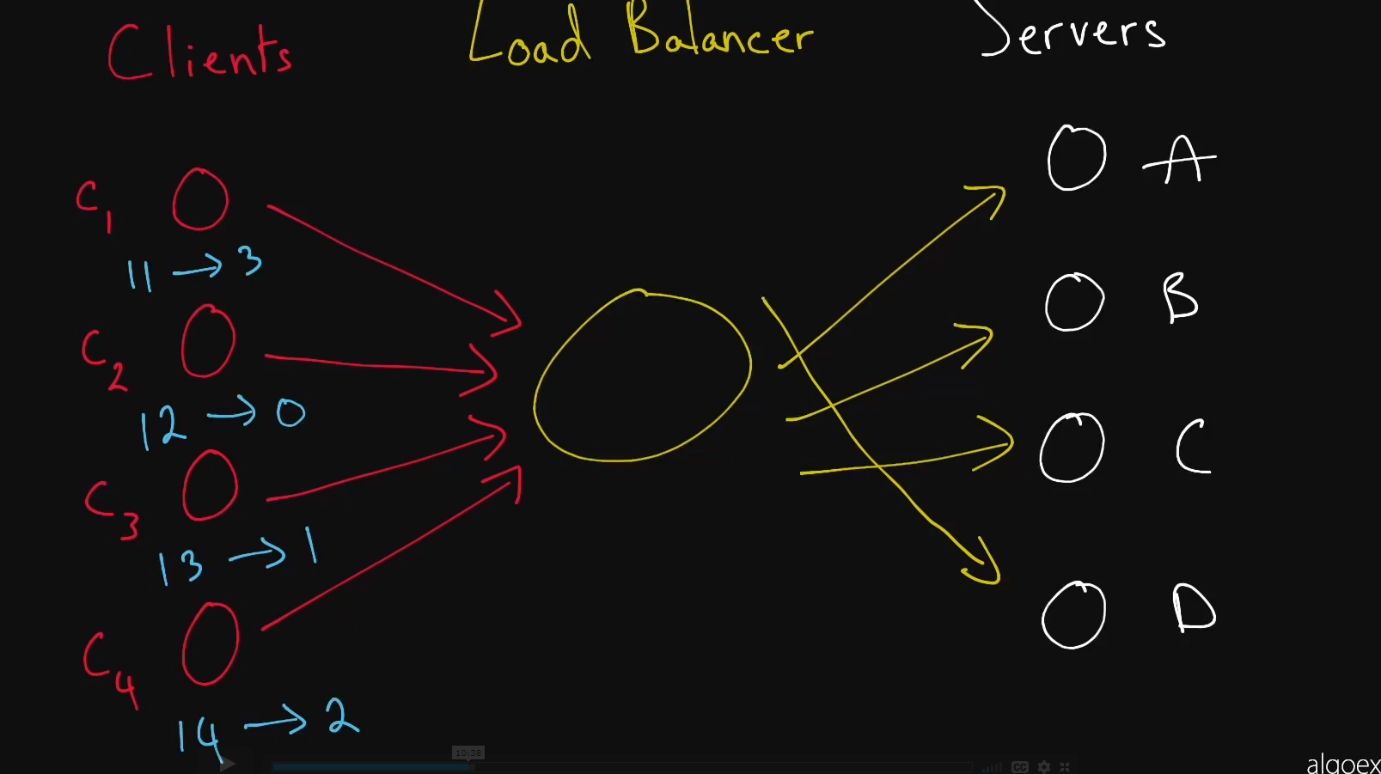
**Hot Spot:** When distributing a workload across a set of servers, that workload might be spread unevenly. This can happen if your **sharding key** or your hashing function are suboptimal, or if your workload is naturally skewed: some severs will receive a lot more traffic than other’s, thus creating a “hot-spot”.

## Hashing

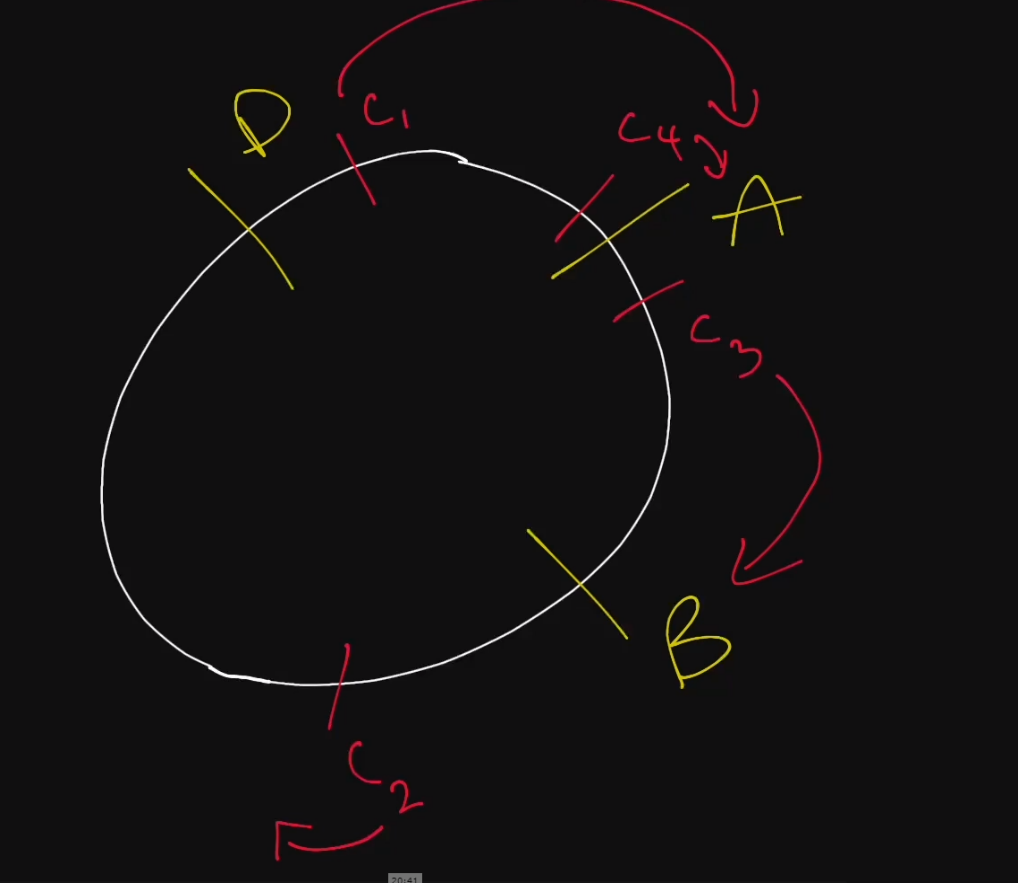
**Hashing Function:** A function that takes in a specific data type (such as a string or an identifier) and outputs a different number. Different inputs may have the same output, but a good hashing function attempts to minimize those hashing collisions (which his equivalent to maximising uniformity).

**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.

