[educative.io] [Design Gurus] Grokking the System Design Interview - Part 5

Andrew.Kaluba

Dec '18
Dec 2018
1 / 7
Dec 2018
22d ago

System Design Basics

Whenever we are designing a large system, we need to consider few things:

1. What are different architectural pieces that can be used?
2. How do these pieces work with each other?
3. How can we best utilize these pieces, what are the right tradeoffs?

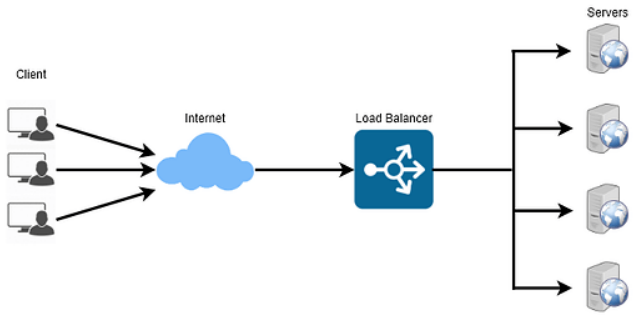
Investing in scaling before it is needed is generally not a smart business proposition; however, some forethought into the design can save valuable time and resources in the future. In the following chapters, we will try to define some of the core building blocks of scalable systems. Familiarizing these concepts would greatly benefit in understanding distributed system concepts. In the next section, we will go through Consistent Hashing, CAP Theorem, Load Balancing, Caching, Data Partitioning, Indexes, Proxies, Queues, Replication, and choosing between SQL vs. NoSQL.

Let's start with Load Balancing.

Load Balancing

Load Balancer (LB) is another critical component of any distributed system. It helps to spread the traffic across a cluster of servers to improve responsiveness and availability of applications, websites or databases. LB also keeps track of the status of all the resources while distributing requests. If a server is not available to take new requests or is not responding or has elevated error rate, LB will stop sending traffic to such a server.

Typically a load balancer sits between the client and the server accepting incoming network and application traffic and distributing the traffic across multiple backend servers using various algorithms. By balancing application requests across multiple servers, a load balancer reduces individual server load and prevents any one application server from becoming a single point of failure, thus improving overall application availability and responsiveness.



To utilize full scalability and redundancy, we can try to balance the load at each layer of the system. We can add LBs at three places:

- Between the user and the web server
- Between web servers and an internal platform layer, like application servers or cache servers
- Between internal platform layer and database.

Benefits of Load Balancing

- Users experience faster, uninterrupted service. Users won't have to wait for a single struggling server to finish its previous tasks. Instead, their requests are immediately passed on to a more readily available resource.
- Service providers experience less downtime and higher throughput. Even a full server failure won't affect the end user experience as the load balancer will simply route around it to a healthy server.
- Load balancing makes it easier for system administrators to handle incoming requests while decreasing wait time for users.
- Smart load balancers provide benefits like predictive analytics that determine traffic bottlenecks before they happen. As a result, the smart load balancer gives an organization actionable insights. These are key to automation and can help drive business decisions.
- System administrators experience fewer failed or stressed components. Instead of a single device performing a lot of work, load balancing has several devices perform a little bit of work.

Load Balancing Algorithms

How does the load balancer choose the backend server?
Load balancers consider two factors before forwarding a request to a backend server. They will first ensure that the server they can choose is actually responding appropriately to requests and then use a pre-configured algorithm to select one from the set of healthy servers. We will discuss these algorithms shortly.

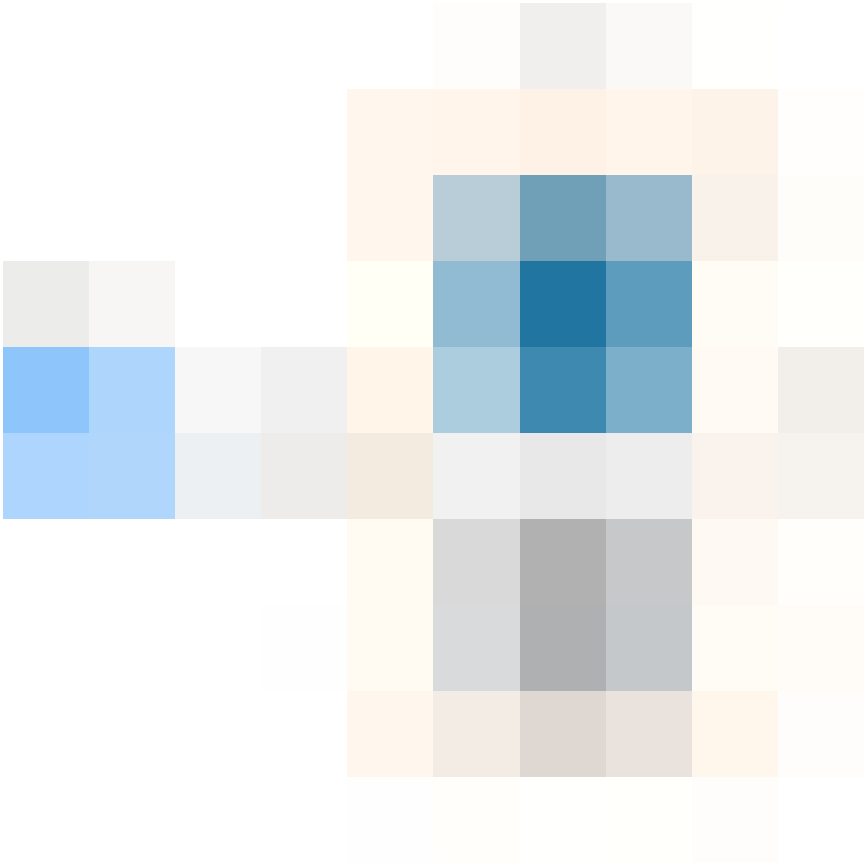
Health Checks - Load balancers should only forward traffic to "healthy" backend servers. To monitor the health of a backend server, "health checks" regularly attempt to connect to backend servers to ensure that servers are listening. If a server fails a health check, it is automatically removed from the pool, and traffic will not be forwarded to it until it responds to the health checks again.

There is a variety of load balancing methods, which use different algorithms for different needs.

- **Least Connection Method** — This method directs traffic to the server with the fewest active connections. This approach is most useful when there are a large number of persistent connections in the traffic unevenly distributed between the servers.
- **Least Response Time Method** — This algorithm directs traffic to the server with the fewest active connections and the lowest average response time.
- **Least Bandwidth Method** - This method selects the server that is currently serving the least amount of traffic, measured in megabits per second (Mbps).
- **Round Robin Method** — This method cycles through a list of servers and sends each new request to the next server. When it reaches the end of the list, it starts over at the beginning. It is most useful when the servers are of equal specification, and there are not many persistent connections.
- **Weighted Round Robin Method** — The weighted round-robin scheduling is designed to better handle servers with different processing capacities. Each server is assigned a weight, an integer value that indicates the processing capacity. Servers with higher weights receive new connections first than those with less weights, and servers with higher weights get more connections than those with less weights.
- **IP Hash** — This method calculates a hash of the IP address of the client to determine which server receives the request.

Redundant Load Balancers

The load balancer can be a single point of failure, to overcome this a second load balancer can be connected to the first to form a cluster. Each LB monitors the health of the other and since both of them are equally capable of serving traffic and failure detection, in the event the main load balancer fails, the second load balancer takes over.



There are many ways to implement load balancing.

1. Smart Clients

One way to implement load-balancing is through the client applications. Developers can add the load balancing algorithm to the application or the database client. Such a client will take a pool of service hosts and balances load across them. It also detects hosts that are not responding to avoid sending requests their way. Smart clients also have

to discover recovered hosts, deal with adding new hosts, etc. Smart clients look easy to implement and manage especially when the system is not large, but as the system grows, LBs need to be evolved into standalone servers.

2. Hardware Load Balancers

The most expensive–but very high performance–solution to load balancing is to buy a dedicated hardware load balancer (like a **Citrix NetScaler**). While they can solve a remarkable range of problems, hardware solutions are costly, and they are not trivial to configure.

As such, even large companies with large budgets will often avoid using dedicated hardware for all their load-balancing needs. Instead, they use them only as the first point of contact for user requests to their infrastructure and use other mechanisms (smart clients or the hybrid approach discussed in the next section) for load-balancing for traffic within their network.

3. Software Load Balancers

If we want to avoid the pain of creating a smart client, and since purchasing dedicated hardware is expensive, we can adopt a hybrid approach, called software load-balancers.

HAProxy is one of the popular open source software LB. The load balancer can be placed between the client and the server or between two server-side layers. If we can control the machine where the client is running, HAProxy could be running on the same machine. Each service we want to load balance can have a locally bound port (e.g., localhost:9000) on that machine, and the client will use this port to connect to the server. This port is, actually, managed by HAProxy; every client request on this port will be received by the proxy and then passed to the backend service in an efficient way (distributing load). If we can't manage the client's machine, HAProxy can run on an intermediate server. Similarly, we can have proxies running between different server-side components. HAProxy manages health checks and will remove or add servers to those pools. It also balances requests across all the servers in those pools.

For most systems, we should start with a software load balancer and move to smart clients or hardware load balancing as the need arises.

Caching

Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have, as well as making otherwise unattainable product requirements feasible. Caches take advantage of the locality of reference principle: recently requested data is likely to be requested again. They are used in almost every layer of computing: hardware, operating systems, web browsers, web applications and more. A cache is like short-term memory: it has a limited amount of space, but is typically faster than the original data source and contains the most recently accessed items. Caches can exist at all levels in architecture but are often found at the level nearest to the front end, where they are implemented to return data quickly without taxing downstream levels.

1. Application server cache

Placing a cache directly on a request layer node enables the local storage of response data. Each time a request is made to the service, the node will quickly return local, cached data if it exists. If it is not in the cache, the requesting node will query the data from disk. The cache on one request layer node could also be located both in memory (which is very fast) and on the node's local disk (faster than going to network storage).

What happens when you expand this to many nodes? If the request layer is expanded to multiple nodes, it's still quite possible to have each node host its own cache. However, if your load balancer randomly distributes requests across the nodes, the same request will go to different nodes, thus increasing cache misses. Two choices for overcoming this hurdle are global caches and distributed caches.

2. Distributed cache

In a distributed cache, each of its nodes own part of the cached data. Typically, the cache is divided up using a consistent hashing function, such that if a request node is looking for a certain piece of data, it can quickly know where to look within the distributed cache to determine if that data is available. In this case, each node has a small piece of the cache, and will then send a request to another node for the data before going to the origin. Therefore, one of the advantages of a distributed cache is the ease by which we can increase the cache space, which can be achieved just by adding nodes to the request pool.

A disadvantage of distributed caching is resolving a missing node. Some distributed caches get around this by storing multiple copies of the data on different nodes; however, you can imagine how this logic can get complicated quickly, especially when you add or remove nodes from the request layer. Although even if a node disappears and part of the cache is lost, the requests will just pull from the origin—so it isn't necessarily catastrophic!

3. Global Cache

A global cache is just as it sounds: all the nodes use the same single cache space. This involves adding a server, or file store of some sort, faster than your original store and accessible by all the request layer nodes. Each of the request nodes queries the cache in the same way it would a local one. This kind of caching scheme can get a bit complicated because it is very easy to overwhelm a single cache as the number of clients and requests increase, but is very effective in some architectures (particularly ones with specialized hardware that make this global cache very fast, or that have a fixed dataset that needs to be cached).

There are two common forms of global caches depicted in the following diagram. First, when a cached response is not found in the cache, the cache itself becomes responsible for retrieving the missing piece of data from the underlying store. Second, it is the responsibility of request nodes to retrieve any data that is not found in the cache.

Most applications leveraging global caches tend to use the first type, where the cache itself manages eviction and fetching data to prevent a flood of requests for the same data from the clients. However, there are some cases where the second implementation makes more sense. For example, if the cache is being used for very large files, a low cache hit percentage would cause the cache buffer to become overwhelmed with cache misses; in this situation, it helps to

have a large percentage of the total data set (or hot data set) in the cache. Another example is an architecture where the files stored in the cache are static and shouldn't be evicted. (This could be because of application requirements around that data latency—certain pieces of data might need to be very fast for large data sets—where the application logic understands the eviction strategy or hot spots better than the cache.)

4. Content Distribution Network (CDN)

CDNs are a kind of cache that comes into play for sites serving large amounts of static media. In a typical CDN setup, a request will first ask the CDN for a piece of static media; the CDN will serve that content if it has it locally available. If it isn't available, the CDN will query the back-end servers for the file and then cache it locally and serve it to the requesting user.

If the system we are building isn't yet large enough to have its own CDN, we can ease a future transition by serving the static media off a separate subdomain (e.g. **static.yourservice.com**) using a lightweight HTTP server like Nginx, and cutover the DNS from your servers to a CDN later.

Cache Invalidation

While caching is fantastic, it does require some maintenance for keeping cache coherent with the source of truth (e.g., database). If the data is modified in the database, it should be invalidated in the cache, if not, this can cause inconsistent application behavior.

Solving this problem is known as cache invalidation, there are three main schemes that are used:

Write-through cache: Under this scheme data is written into the cache and the corresponding database at the same time. The cached data allows for fast retrieval, and since the same data gets written in the permanent storage, we will have complete data consistency between cache and storage. Also, this scheme ensures that nothing will get lost in case of a crash, power failure, or other system disruptions.

Although write through minimizes the risk of data loss, since every write operation must be done twice before returning success to the client, this scheme has the disadvantage of higher latency for write operations.

Write-around cache: This technique is similar to write through cache, but data is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write operations that will not subsequently be re-read, but has the disadvantage that a read request for recently written data will create a "cache miss" and must be read from slower back-end storage and experience higher latency.

Write-back cache: Under this scheme, data is written to cache alone, and completion is immediately confirmed to the client. The write to the permanent storage is done after specified intervals or under certain conditions. This results in low latency and high throughput for write-intensive applications, however, this speed comes with the risk of data loss in case of a crash or other adverse event because the only copy of the written data is in the cache.

Cache eviction policies

Following are some of the most common cache eviction policies:

1. First In First Out (FIFO): The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
2. Last In First Out (LIFO): The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
3. Least Recently Used (LRU): Discards the least recently used items first.
4. Most Recently Used (MRU): Discards, in contrast to LRU, the most recently used items first.
5. Least Frequently Used (LFU): Counts how often an item is needed. Those that are used least often are discarded first.
6. Random Replacement (RR): Randomly selects a candidate item and discards it to make space when necessary.

Sharding or Data Partitioning

Data partitioning (also known as sharding) is a technique to break up a big database (DB) into many smaller parts. It is the process of splitting up a DB/table across multiple machines to improve the manageability, performance, availability and load balancing of an application. The justification for data sharding is that, after a certain scale point, it is cheaper and more feasible to scale horizontally by adding more machines than to grow it vertically by adding beefier servers.

1. Partitioning Methods

There are many different schemes one could use to decide how to break up an application database into multiple smaller DBs. Below are three of the most popular schemes used by various large scale applications.

a. Horizontal partitioning: In this scheme, we put different rows into different tables. For example, if we are storing different places in a table, we can decide that locations with ZIP codes less than 10000 are stored in one table, and places with ZIP codes greater than 10000 are stored in a separate table. This is also called a range based sharding, as we are storing different ranges of data in separate tables.

The key problem with this approach is that if the value whose range is used for sharding isn't chosen carefully, then the partitioning scheme will lead to unbalanced servers. In the previous example, splitting location based on their zip codes assumes that places will be evenly distributed across the different zip codes. This assumption is not valid as there will be a lot of places in a thickly populated area like Manhattan compared to its suburb cities.

b. Vertical Partitioning: In this scheme, we divide our data to store tables related to a specific feature to their own server. For example, if we are building Instagram like application, where we need to store data related to users, all the photos they upload and people they follow, we can decide to place user profile information on one DB server, friend lists on another and photos on a third server.

Vertical partitioning is straightforward to implement and has a low impact on the application. The main problem with this approach is that if our application experiences additional growth, then it may be necessary to further partition a feature specific DB across various servers (e.g. it would not be possible for a single server to handle all the metadata queries for 10 billion photos by 140 million users).

c. Directory Based Partitioning: A loosely coupled approach to work around issues mentioned in above schemes is to create a lookup service which knows your current partitioning scheme and abstracts it away from the DB access code. So, to find out where does a particular data entity resides, we query our directory server that holds the mapping between each tuple key to its DB server. This loosely coupled approach means we can perform tasks like adding servers to the DB pool or change our partitioning scheme without having to impact your application.

2. Partitioning Criteria

a. Key or Hash-based partitioning: Under this scheme, we apply a hash function to some key attribute of the entity we are storing, that yields the partition number. For example, if we have 100 DB servers and our ID is a numeric value that gets incremented by one, each time a new record is inserted. In this example, the hash function could be 'ID % 100', which will give us the server number where we can store/read that record. This approach should ensure a uniform allocation of data among servers. The fundamental problem with this approach is that it effectively fixes the total number of DB servers, since adding new servers means changing the hash function which would require redistribution of data and downtime for the service. A workaround for this problem is to use Consistent Hashing.

b. List partitioning: In this scheme, each partition is assigned a list of values, so whenever we want to insert a new record, we will see which partition contains our key and then store it there. For example, we can decide all users living in Iceland, Norway, Sweden, Finland or Denmark will be stored in a partition for the Nordic countries.

c. Round-robin partitioning: This is a very simple strategy that ensures uniform data distribution. With 'n' partitions, the 'i' tuple is assigned to partition (i mod n).

d. Composite partitioning: Under this scheme, we combine any of above partitioning schemes to devise a new scheme. For example, first applying a list partitioning and then a hash based partitioning. Consistent hashing could be considered a composite of hash and list partitioning where the hash reduces the key space to a size that can be listed.

3. Common Problems of Sharding

On a sharded database, there are certain extra constraints on the different operations that can be performed. Most of these constraints are due to the fact that, operations across multiple tables or multiple rows in the same table, will no longer run on the same server. Below are some of the constraints and additional complexities introduced by sharding:

a. Joins and Denormalization: Performing joins on a database which is running on one server is straightforward, but once a database is partitioned and spread across multiple machines it is often not feasible to perform joins that span database shards. Such joins will not be performance efficient since data has to be compiled from multiple servers. A common workaround for this problem is to denormalize the database so that queries that previously required joins can be performed from a single table. Of course, the service now has to deal with all the perils of denormalization such as data inconsistency.

b. Referential integrity: As we saw that performing a cross-shard query on a partitioned database is not feasible, similarly trying to enforce data integrity constraints such as foreign keys in a sharded database can be extremely difficult.

Most of RDBMS do not support foreign keys constraints across databases on different database servers. Which means that applications that require referential integrity on sharded databases often have to enforce it in application code. Often in such cases, applications have to run regular SQL jobs to clean up dangling references.

c. Rebalancing: There could be many reasons we have to change our sharding scheme:

1. The data distribution is not uniform, e.g., there are a lot of places for a particular ZIP code, that cannot fit into one database partition.
2. There are a lot of load on a shard, e.g., there are too many requests being handled by the DB shard dedicated to user photos.

In such cases, either we have to create more DB shards or have to rebalance existing shards, which means the partitioning scheme changed and all existing data moved to new locations. Doing this without incurring downtime is extremely difficult. Using a scheme like directory based partitioning does make rebalancing a more palatable experience at the cost of increasing the complexity of the system and creating a new single point of failure (i.e. the lookup service/database).

Indexes

Indexes are well known when it comes to databases. Sooner or later there comes a time when database performance is no longer satisfactory. One of the very first things you should turn to when that happens is database indexing.

The goal of creating an index on a particular table in a database is to make it faster to search through the table and find the row or rows that we want. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

Example: A library catalog

A library catalog is a register that contains the list of books found in a library. The catalog is organized like a database table generally with four columns: book title, writer, subject and date of publication. There are usually two such catalogs, one sorted by the book title and one sorted by the writer name. That way you can either think of a writer you want to read and then look through their books, or look up a specific book title you know you want to read or in case you don’t know the writer’s name. These catalogs are like indexes for the database of books. They provide a sorted list of data that is easily searchable by relevant information.

Simply saying, an index is a data structure that can be perceived as a table of contents that points us to the location where actual data lives. So when we create an index on a column of a table, we store that column and a pointer to the whole row in the index. Let's assume a table containing a list of books, following diagram shows how an index on 'Title' column looks like:

Just as to a traditional relational data store, we can also apply this concept to larger datasets. The trick with indexes is that we must carefully consider how users will access the data. In the case of data sets that are many terabytes in size but with very small payloads (e.g., 1 KB), indexes are a necessity for optimizing data access. Finding a small payload in such a large dataset can be a real challenge since we can't possibly iterate over that much data in any reasonable time. Furthermore, it is very likely that such a large data set is spread over several physical devices—this means we need some way to find the correct physical location of the desired data. Indexes are the best way to do this.

How do Indexes decrease write performance?

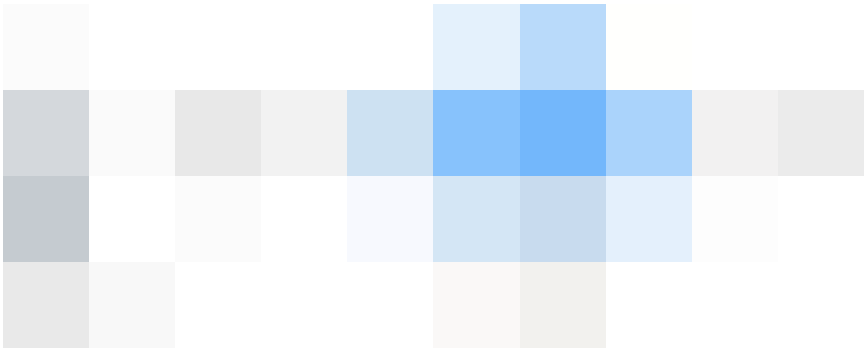
The downside is that indexes make it slower to add rows or make updates to existing rows for that table since we not only have to write the data but also have to update the index. So adding indexes can increase the read performance, but at the same time, decrease the write performance.

Adding a new row to a table without indexes is simple. The database finds the next available space in the table to add the new entry and inserts it there. However, when adding a new row to a table with one or more indexes, the database not only have to add the new entry to the table but also have to add a new entry into each index on that table, making sure to insert the entry into the correct spot in the index to ensure that the data is sorted correctly. This performance degradation applies to all insert, update, and delete operations for the table. For this reason, adding unnecessary indexes on tables should be avoided, and indexes that are no longer used should be removed. To reiterate, adding indexes is about improving the performance of search queries. If the goal of the database is to provide a data store that is often written to and rarely read from, in that case, decreasing the performance of the more common operation, which is writing, is probably not worth the increase in performance we get from reading.

For more details, see **Database Indexes** .

Proxies

A proxy server is an intermediary piece of hardware/software that sits between the client and the back-end server. It receives requests from clients and relays them to the origin servers. Typically, proxies are used to filter requests or log requests, or sometimes transform requests (by adding/removing headers, encrypting/decrypting, or compression). Another advantage of a proxy server is that its cache can serve a lot of requests. If multiple clients access a particular resource, the proxy server can cache it and serve all clients without going to the remote server.



Proxies are also extremely helpful when coordinating requests from multiple servers and can be used to optimize request traffic from a system-wide perspective. For example, we can collapse the same (or similar) data access requests into one request and then return the single result to the user; this scheme is called collapsed forwarding.

Imagine there is a request for the same data across several nodes, and that piece of data is not in the cache. If these requests are routed through the proxy, then all them can be collapsed into one, which means we will be reading the required data from the disk only once.

Another great way to use the proxy is to collapse requests for data that is spatially close together in the storage (consecutively on disk). This strategy will result in decreasing request latency. For example, let's say a bunch of servers request parts of file: part1, part2, part3, etc. We can set up our proxy in such a way that it can recognize the spatial locality of the individual requests, thus collapsing them into a single request and reading complete file, which will greatly minimize the reads from the data origin. Such scheme makes a big difference in request time when we are doing random accesses across TBs of data. Proxies are particularly useful under high load situations, or when we have limited caching since proxies can mostly batch several requests into one.

Queues

Queues are used to effectively manage requests in a large-scale distributed system. In small systems with minimal processing loads and small databases, writes can be predictably fast; however, in more complex and large systems writes can take an almost non-deterministically long time. For example, data may have to be written in different places on different servers or indices, or the system could simply be under high load. In such cases where individual writes (or

tasks) may take a long time, achieving high performance and availability requires different components of the system to work in an asynchronous way; a common way to do that is with queues.

Let's assume a system where each client is requesting a task to be processed on a remote server. Each of these clients sends their requests to the server, and the server tries to finish the tasks as quickly as possible to return the results to the respective clients. In small systems where one server can handle incoming requests just as fast as they come, this kind of situation should work just fine. However, when the server gets more requests than it can handle, then each client is forced to wait for other clients' requests to finish before a response can be generated.

This kind of synchronous behavior can severely degrade client's performance; the client is forced to wait, effectively doing zero work, until its request can be responded. Adding extra servers to address high load does not solve the problem either; even with effective load balancing in place, it is very difficult to ensure the fair and balanced distribution of work required to maximize client performance. Further, if the server processing the requests is unavailable, or fails, then the clients upstream will fail too. Solving this problem effectively requires building an abstraction between the client's request and the actual work performed to service it.

A processing queue is as simple as it sounds: all incoming tasks are added to the queue, and as soon as any worker has the capacity to process, they can pick up a task from the queue. These tasks could represent a simple write to a database, or something as complex as generating a thumbnail preview image for a document.

Queues are implemented on the asynchronous communication protocol, meaning when a client submits a task to a queue they are no longer required to wait for the results; instead, they need only acknowledgment that the request was properly received. This acknowledgment can later serve as a reference for the results of the work when the client requires it. Queues have implicit or explicit limits on the size of data that may be transmitted in a single request and the number of requests that may remain outstanding on the queue.

Queues are also used for fault tolerance as they can provide some protection from service outages and failures. For example, we can create a highly robust queue that can retry service requests that have failed due to transient system failures. It is preferable to use a queue to enforce quality-of-service guarantees than to expose clients directly to intermittent service outages, requiring complicated and often inconsistent client-side error handling.

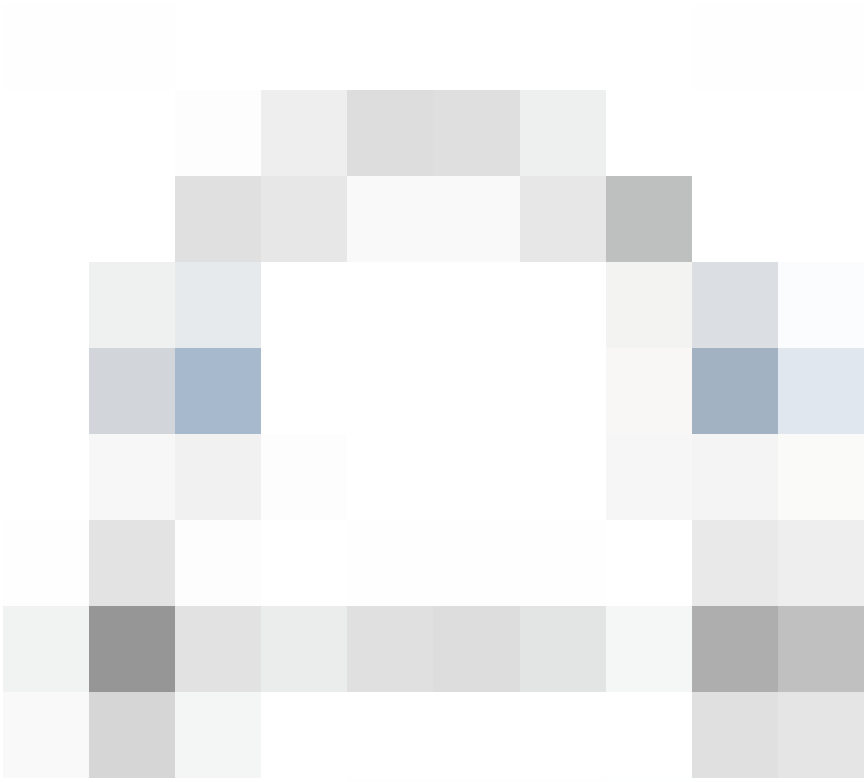
Queues play a vital role in managing distributed communication between different parts of any large-scale distributed system. There are a lot of ways to implement them and quite a few open source implementations of queues available like RabbitMQ, ZeroMQ, ActiveMQ, and BeanstalkD.

Redundancy and Replication

Redundancy means duplication of critical data or services with the intention of increased reliability of the system. For example, if there is only one copy of a file stored on a single server, then losing that server means losing the file. Since losing data is seldom a good thing, we can create duplicate or redundant copies of the file to solve this problem.

This same principle applies to services too. If we have a critical service in our system, ensuring that multiple copies or versions of it are running simultaneously can secure against the failure of a single node.

Creating redundancy in a system can remove single points of failure and provide backups if needed in a crisis. For example, if we have two instances of a service running in production, and if one fails or degrades, the system can failover to the other one. These failovers can happen automatically or can be done manually.



Another important part of service redundancy is to create a shared-nothing architecture, where each node can operate independently of one another. There should not be any central service managing state or orchestrating activities for the other nodes. This helps a lot with scalability since new servers can be added without special conditions or knowledge and most importantly, such systems are more resilient to failure as there is no single point of failure.

SQL vs. NoSQL

In the world of databases, there are two main types of solutions: SQL and NoSQL - or relational databases and non-relational databases. Both of them differ in the way they were built, the kind of information they store, and how they store it.

Relational databases are structured and have predefined schemas, like phone books that store phone numbers and addresses. Non-relational databases are unstructured, distributed and have a dynamic schema, like file folders that hold everything from a person’s address and phone number to their Facebook ‘likes’ and online shopping preferences.

SQL

Relational databases store data in rows and columns. Each row contains all the information about one entity, and columns are all the separate data points. Some of the most popular relational databases are MySQL, Oracle, MS SQL Server, SQLite, Postgres and MariaDB.

NoSQL

Following are most common types of NoSQL:

Key-Value Stores: Data is stored in an array of key-value pairs. The 'key' is an attribute name, which is linked to a 'value'. Well-known key value stores include Redis, Voldemort and Dynamo.

Document Databases: In these databases data is stored in documents, instead of rows and columns in a table, and these documents are grouped together in collections. Each document can have an entirely different structure. Document databases include the CouchDB and MongoDB.

Wide-Column Databases: Instead of 'tables,' in columnar databases we have column families, which are containers for rows. Unlike relational databases, we don't need to know all the columns up front, and each row doesn't have to have the same number of columns. Columnar databases are best suited for analyzing large datasets - big names include Cassandra and HBase.

Graph Databases: These databases are used to store data whose relations are best represented in a graph. Data is saved in graph structures with nodes (entities), properties (information about the entities) and lines (connections between the entities). Examples of graph database include Neo4J and InfiniteGraph.

High level differences between SQL and NoSQL

Storage: SQL stores data in tables, where each row represents an entity, and each column represents a data point about that entity; for example, if we are storing a car entity in a table, different columns could be 'Color', 'Make', 'Model', and so on.

NoSQL databases have different data storage models. The main ones are key-value, document, graph and columnar. We will discuss differences between these databases below.

Schema: In SQL, each record conforms to a fixed schema, meaning the columns must be decided and chosen before data entry and each row must have data for each column. The schema can be altered later, but it involves modifying the whole database and going offline.

Whereas in NoSQL, schemas are dynamic. Columns can be added on the fly, and each 'row' (or equivalent) doesn't have to contain data for each 'column.'

Querying: SQL databases uses SQL (structured query language) for defining and manipulating the data, which is very powerful. In NoSQL database, queries are focused on a collection of documents. Sometimes it is also called UnQL (Unstructured Query Language). Different databases have different syntax for using UnQL.

Scalability: In most common situations, SQL databases are vertically scalable, i.e., by increasing the horsepower (higher Memory, CPU, etc.) of the hardware, which can get very expensive. It is possible to scale a relational database across multiple servers, but this is a challenging and time-consuming process.

On the other hand, NoSQL databases are horizontally scalable, meaning we can add more servers easily in our NoSQL database infrastructure to handle large traffic. Any cheap commodity hardware or cloud instances can host NoSQL databases, thus making it a lot more cost-effective than vertical scaling. A lot of NoSQL technologies also distribute data across servers automatically.

Reliability or ACID Compliancy (Atomicity, Consistency, Isolation, Durability): The vast majority of relational databases are ACID compliant. So, when it comes to data reliability and safe guarantee of performing transactions, SQL databases are still the better bet.

Most of the NoSQL solutions sacrifice ACID compliance for performance and scalability.

SQL VS. NoSQL - Which one to use?

When it comes to database technology, there's no one-size-fits-all solution. That's why many businesses rely on both relational and non-relational databases for different needs. Even as NoSQL databases are gaining popularity for their speed and scalability, there are still situations where a highly structured SQL database may perform better; choosing the right technology hinges on the use case.

Reasons to use SQL database

Here are a few reasons to choose a SQL database:

1. We need to ensure ACID compliance. ACID compliance reduces anomalies and protects the integrity of your database by prescribing exactly how transactions interact with the database. Generally, NoSQL databases sacrifice ACID compliance for scalability and processing speed, but for many e-commerce and financial applications, an ACID-compliant database remains the preferred option.
2. Your data is structured and unchanging. If your business is not experiencing massive growth that would require more servers and if you're only working with data that's consistent, then there may be no reason to use a system designed to support a variety of data types and high traffic volume.

Reasons to use NoSQL database

When all the other components of our application are fast and seamless, NoSQL databases prevent data from being the bottleneck. Big data is contributing to a large success for NoSQL databases, mainly because it handles data differently than the traditional relational databases. A few popular examples of NoSQL databases are MongoDB, CouchDB, Cassandra, and HBase.

1. Storing large volumes of data that often have little to no structure. A NoSQL database sets no limits on the types of data we can store together and allows us to add different new types as the need changes. With document-based databases, you can store data in one place without having to define what "types" of data those are in advance.
2. Making the most of cloud computing and storage. Cloud-based storage is an excellent cost-saving solution but requires data to be easily spread across multiple servers to scale up. Using commodity (affordable, smaller) hardware on-site or in the cloud saves you the hassle of additional software, and NoSQL databases like Cassandra are designed to be scaled across multiple data centers out of the box without a lot of headaches.
3. Rapid development. NoSQL is extremely useful for rapid development as it doesn't need to be prepped ahead of time. If you're working on quick iterations of your system which require making frequent updates to the data structure without a lot of downtime between versions, a relational database will slow you down.

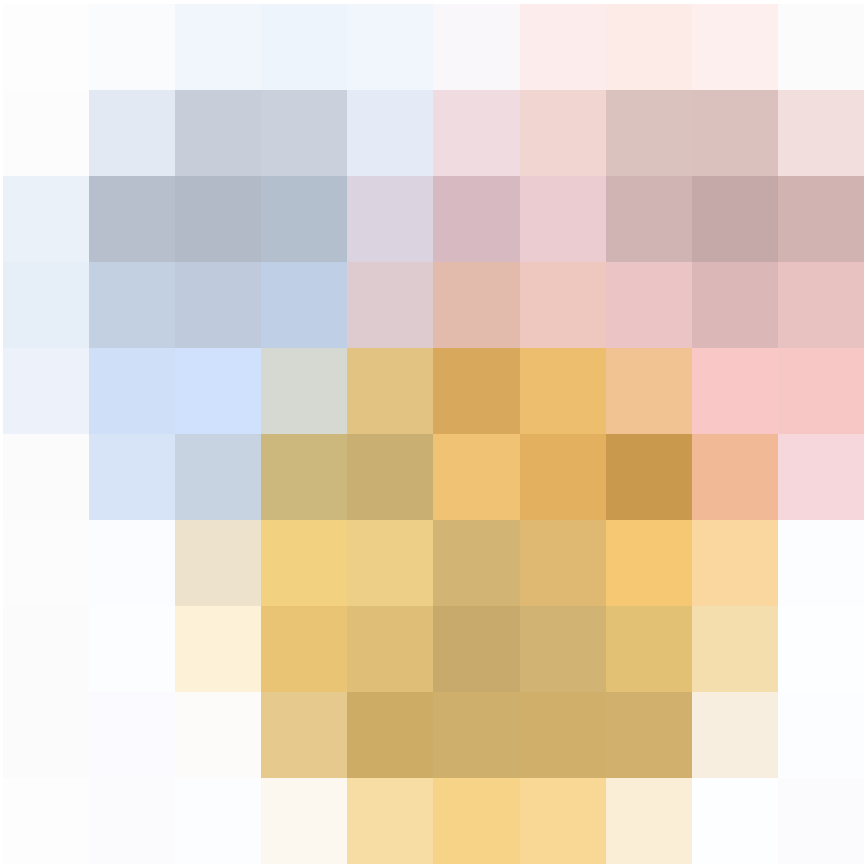
CAP Theorem

CAP theorem states that it is impossible for a distributed software system to simultaneously provide more than two out of three of the following guarantees (CAP): Consistency, Availability and Partition tolerance. When we design a distributed system, trading off among CAP is almost the first thing we want to consider. CAP theorem says while designing a distributed system we can pick only two of:

Consistency: All nodes see the same data at the same time. Consistency is achieved by updating several nodes before allowing further reads.

Availability: Every request gets a response on success/failure. Availability is achieved by replicating the data across different servers.

Partition tolerance: System continues to work despite message loss or partial failure. A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network. Data is sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.



We cannot build a general data store that is continually available, sequentially consistent and tolerant to any partition failures. We can only build a system that has any two of these three properties. Because, to be consistent, all nodes should see the same set of updates in the same order. But if the network suffers a partition, updates in one partition might not make it to the other partitions before a client reads from the out-of-date partition after having read from the up-to-date one. The only thing that can be done to cope with this possibility is to stop serving requests from the out-of-date partition, but then the service is no longer 100% available.

Consistent Hashing

Distributed Hash Table (DHT) is one of the fundamental component used in distributed scalable systems. Hash Tables need key, value and a hash function, where hash function maps the key to a location where the value is stored.

index = hash_function(key)

Suppose we are designing a distributed caching system. Given ‘n’ cache servers, an intuitive hash function would be ‘key % n’. It is simple and commonly used. But it has two major drawbacks:

1. It is NOT horizontally scalable. Whenever a new cache host is added to the system, all existing mappings are broken. It will be a pain point in maintenance if the caching system contains lots of data. Practically it becomes difficult to schedule a downtime to update all caching mappings.
2. It may NOT be load balanced, especially for non-uniformly distributed data. In practice, it can be easily assumed that the data will not be distributed uniformly. For the caching system, it translates into some caches becoming hot and saturated while the others idle and almost empty.

In such situations, consistent hashing is a good way to improve the caching system.

What is Consistent Hashing?

Consistent hashing is a very useful strategy for distributed caching system and DHTs. It allows distributing data across a cluster in such a way that will minimize reorganization when nodes are added or removed. Hence, making the caching system easier to scale up or scale down.

In Consistent Hashing when the hash table is resized (e.g. a new cache host is added to the system), only k/n keys need to be remapped, where k is the total number of keys and n is the total number of servers. Recall that in a caching system using the 'mod' as the hash function, all keys need to be remapped.

In consistent hashing objects are mapped to the same host if possible. When a host is removed from the system, the objects on that host are shared by other hosts; and when a new host is added, it takes its share from a few hosts without touching other's shares.

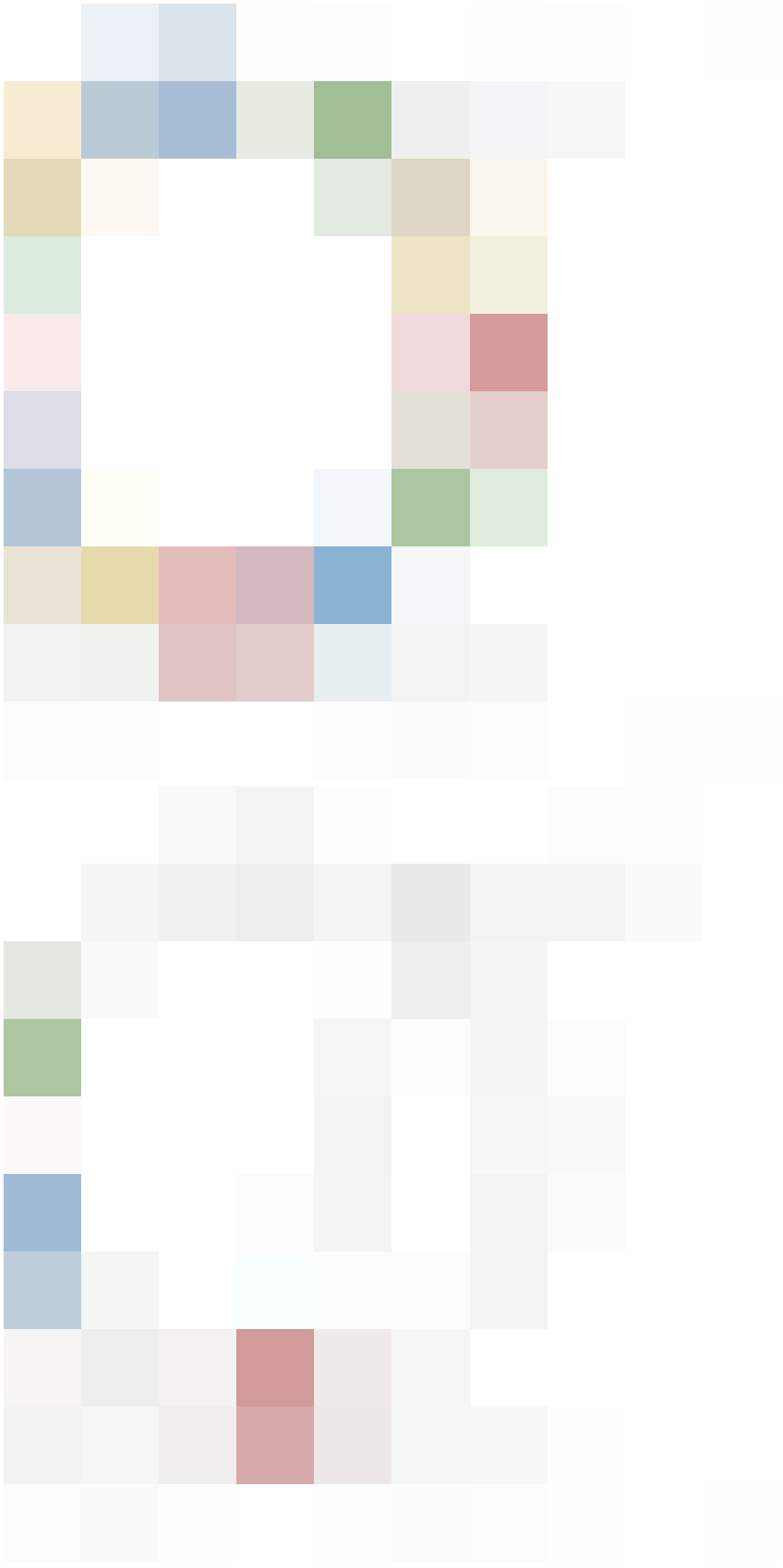
How it works?

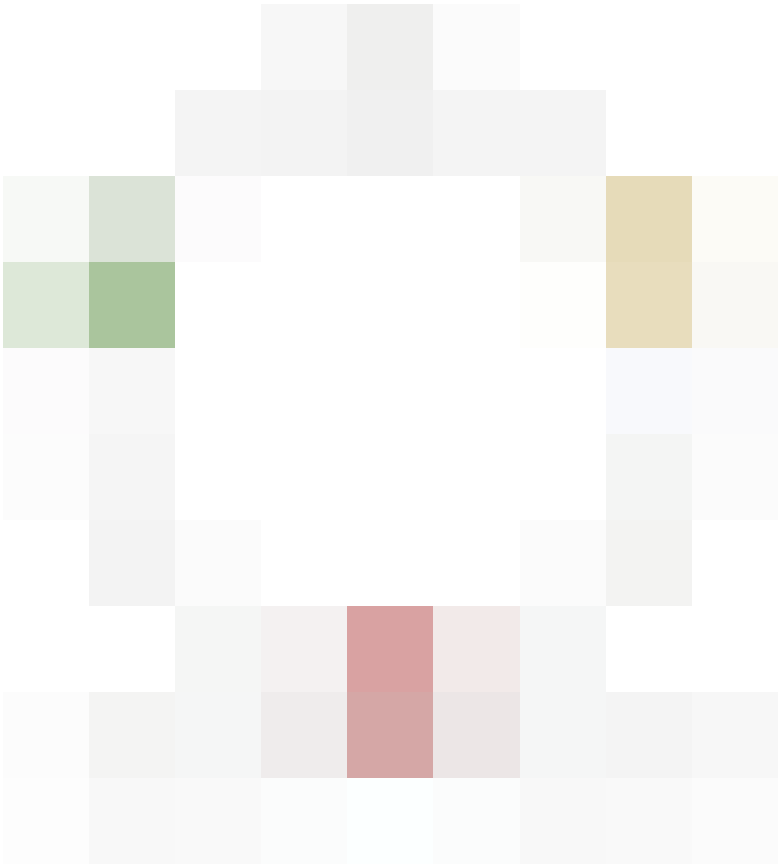
As a typical hash function, consistent hashing maps a key to an integer. Suppose the output of the hash function is in the range of [0, 256). Imagine that the integers in the range are placed on a ring such that the values are wrapped around.

Here's how consistent hashing works:

1. Given a list of cache servers, hash them to integers in the range.
2. To map a key to a server,

- Hash it to a single integer.
- Move clockwise on the ring until finding the first cache it encounters.
- That cache is the one that contains the key. See animation below as an example: key1 maps to cache A; key2 maps to cache C.





To add a new server, say D, keys that were originally residing at C will be split. Some of them will be shifted to D, while other keys will not be touched.

To remove a cache or if a cache failed, say A, all keys that were originally mapping to A will fall into B, and only those keys need to be moved to B, other keys will not be affected.

For load balancing, as we discussed in the beginning, the real data is essentially randomly distributed and thus may not be uniform. It may make the keys on caches unbalanced.

To handle this issue, we add “virtual replicas” for caches. Instead of mapping each cache to a single point on the ring, we map it to multiple points on the ring, i.e. replicas. This way, each cache is associated with multiple portions of the ring.

If the hash function is “mixes well,” as the number of replicas increases, the keys will be more balanced.

Long-Polling vs WebSockets vs Server-Sent Events

What is the difference between Long-Polling, WebSockets and Server-Sent Events?

Long-Polling, WebSockets, and Server-Sent Events are popular communication protocols between a client like a web browser and a web server. First, let’s start with understanding what a standard HTTP web request looks like. Following are a sequence of events for regular HTTP request:

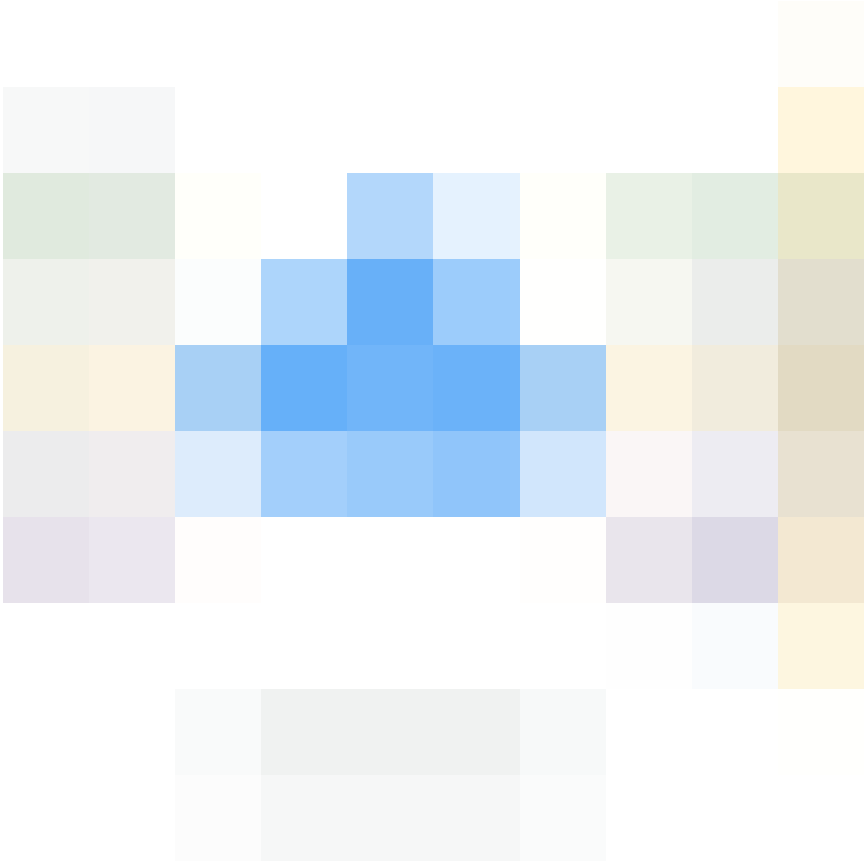
1. Client opens a connection and requests data from the server.
2. The server calculates the response.
3. The server sends the response back to the client on the opened request.

Ajax Polling

Polling is a standard technique used by the vast majority of AJAX applications. The basic idea is that the client repeatedly polls (or requests) a server for data. The client makes a request and waits for the server to respond with data. If no data is available, an empty response is returned.

1. Client opens a connection and requests data from the server using regular HTTP.
2. The requested webpage sends requests to the server at regular intervals (e.g., 0.5 seconds).
3. The server calculates the response and sends it back, just like regular HTTP traffic.
4. Client repeats the above three steps periodically to get updates from the server.

Problem with Polling is that the client has to keep asking the server for any new data. As a result, a lot of responses are empty creating HTTP overhead.



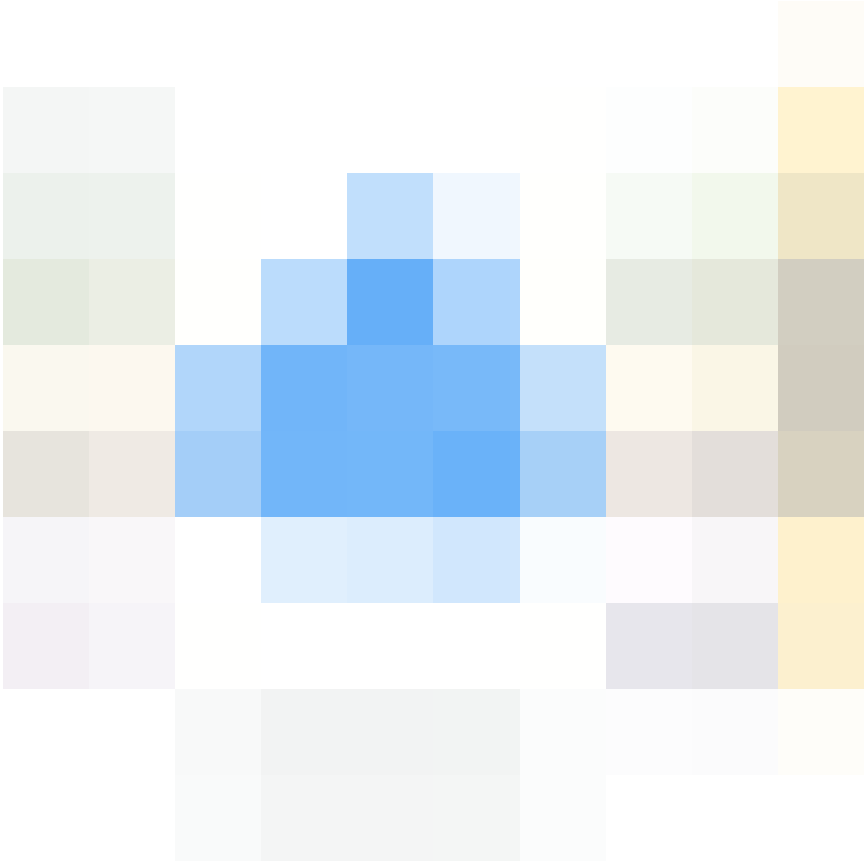
HTTP Long-Polling

A variation of the traditional polling technique that allows the server to push information to a client, whenever the data is available. With Long-Polling, the client requests information from the server exactly as in normal polling, but with the expectation that the server may not respond immediately. That's why this technique is sometimes referred to as a "Hanging GET".

- If the server does not have any data available for the client, instead of sending an empty response, the server holds the request and waits until some data becomes available.
- Once the data becomes available, a full response is sent to the client. The client then immediately re-request information from the server so that the server will almost always have an available waiting request that it can use to deliver data in response to an event.

The basic life cycle of an application using HTTP Long-Polling is as follows:

1. The client makes an initial request using regular HTTP and then waits for a response.
2. The server delays its response until an update is available, or until a timeout has occurred.
3. When an update is available, the server sends a full response to the client.
4. The client typically sends a new long-poll request, either immediately upon receiving a response or after a pause to allow an acceptable latency period.
5. Each Long-Poll request has a timeout. The client has to reconnect periodically after the connection is closed, due to timeouts.



WebSockets

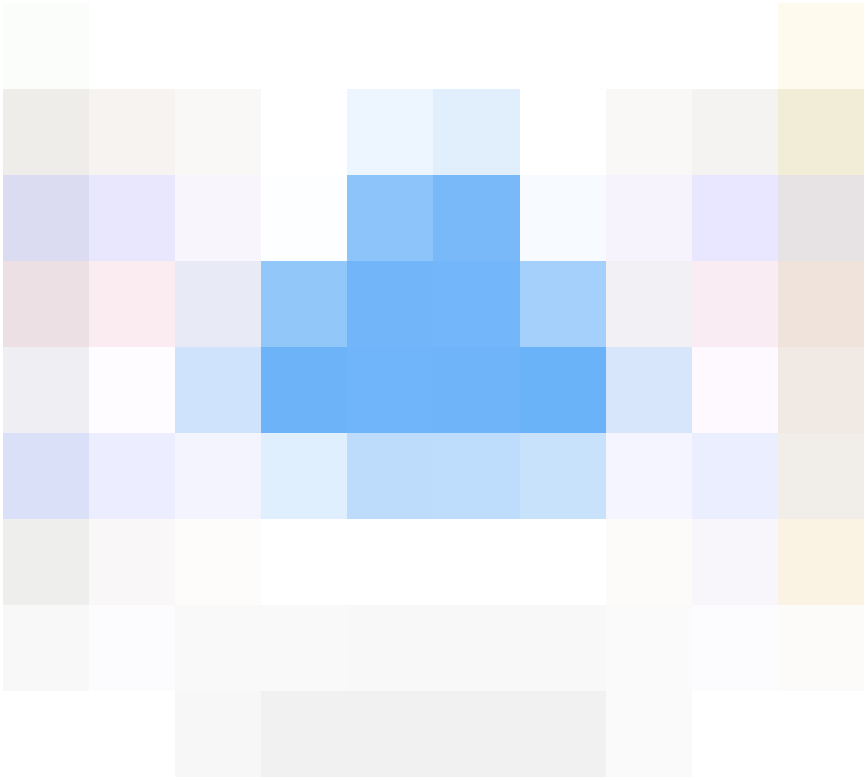
WebSocket provides **Full duplex** communication channels over a single TCP connection. It provides a persistent connection between a client and a server that both parties can use to start sending data at any time. The client establishes a WebSocket connection through a process known as the WebSocket handshake. If the process succeeds, then the server and client can exchange data in both directions at any time. The WebSocket protocol enables communication between a client and a server with lower overheads, facilitating real-time data transfer from and to the server. This is made possible by providing a standardized way for the server to send content to the browser without being asked by the client, and allowing for messages to be passed back and forth while keeping the connection open. In this way, a two-way (bi-directional) ongoing conversation can take place between a client and a server.

Server-Sent Events (SSEs)

Under SSEs the client establishes a persistent and long-term connection with the server. The server uses this connection to send data to a client. If the client wants to send data to the server, it would require the use of another technology/protocol to do so.

1. Client requests data from a server using regular HTTP.
2. The requested webpage opens a connection to the server.
3. The server sends the data to the client whenever there's new information available.

SSEs are best when we need real-time traffic from the server to the client or if the server is generating data in a loop and will be sending multiple events to the client.



Key Characteristics of Distributed Systems

Key characteristics of a distributed systems include Scalability, Reliability, Availability, Efficiency, and Manageability. Let's briefly review them:

Scalability

Scalability is the capability of a system, process or a network to grow and manage increased demand. Any distributed system that can continuously evolve in order to support the growing amount of work is considered to be scalable.

A system may have to scale because of many reasons like increased data volume or increased amount of work, e.g., number of transactions. A scalable system would like to achieve this scaling without performance loss.

Generally, the performance of a system, although designed (or claimed) to be scalable, declines with the system size, due to the management or environment cost. For instance, network speed may become slower because machines tend to be far apart from one another. More generally, some tasks may not be distributed, either because of their inherent atomic nature or because of some flaw in the system design. At some point, such tasks would limit the speed-up obtained by distribution. A scalable architecture avoids this situation and attempts to balance the load on all the participating nodes evenly.

Horizontal vs. Vertical Scaling: Horizontal scaling means that you scale by adding more servers into your pool of resources whereas Vertical scaling means that you scale by adding more power (CPU, RAM, Storage, etc.) to an existing server.

With horizontal-scaling it is often easier to scale dynamically by adding more machines into the existing pool; Vertical-scaling is usually limited to the capacity of a single server, scaling beyond that capacity often involves downtime and comes with an upper limit.

Good examples of horizontal scaling are **Cassandra** and **MongoDB**, as they both provide an easy way to scale horizontally by adding more machines to meet growing needs. Similarly a good example of vertical scaling is MySQL

as it allows for an easy way to scale vertically by switching from small to bigger machines. However, this process often involves downtime.

Reliability

By definition, reliability is the probability a system will fail in a given period. In simple terms, a distributed system is considered reliable if it keeps delivering its services even when one or several of its software or hardware components fail. Reliability represents one of the main characteristics of any distributed system, as in such systems any failing machine can always be replaced by another healthy one, ensuring the completion of the requested task.

Take the example of a large electronic commerce store (like **Amazon**), where one of the primary requirement is that any user transaction should never be canceled due to a failure of the machine that is running that transaction. For instance, if a user has added an item to their shopping cart, the system is expected not to lose it. A reliable distributed system achieves this through redundancy of both the software components and data. If the server carrying the user's shopping cart fails, another server that has the exact replica of the shopping cart should replace it.

Obviously, redundancy has a cost, and a reliable system has to pay that to achieve such resilience for services by eliminating every single point of failure.

Availability

By definition, availability is the time a system remains operational to perform its required function, in a specific period. It is a simple measure of the percentage of time that a system, service, or a machine remains operational under normal conditions. An aircraft that can be flown for many hours a month without much downtime can be said to have a high availability. Availability takes into account maintainability, repair time, spares availability, and other logistics considerations. If an aircraft is down for maintenance, it is considered not available during that time.

Reliability is availability over time considering the full range of possible real-world conditions that can occur. An aircraft that can make it through any possible weather safely is more reliable than one that has vulnerabilities to possible conditions.

Reliability Vs. Availability

If a system is reliable, it is available. However, if it is available, it is not necessarily reliable. In other words, high reliability contributes to high availability, but it is possible to achieve a high availability even with an unreliable product by minimizing repair time and ensuring that spares are always available when they are needed. Let's take the example of an online retails store that has 99.99% availability for the first two years after its launch. However, the system was launched without any information security testing. The customers were happy with the system, but they don't realize that it isn't very reliable as it is vulnerable to likely risks. In the third year, the system experiences a series of information security incidents that suddenly result in extremely low availability for extended periods of time. This results in reputational and financial damage to the customers.

Efficiency

To understand how to measure the efficiency of a distributed system, let's assume an operation that runs in a distributed manner, and delivers a set of items as result. Two standard measures of its efficiency are the response time (or latency) that denotes the delay to obtain the first item, and the throughput (or bandwidth) which denotes the number of items delivered in a given time unit (e.g., a second). The two measures correspond to the following unit costs:

- Number of messages globally sent by the nodes of the system, regardless of the message size.
- Size of messages representing the volume of data exchanges.

The complexity of operations supported by distributed data structures (e.g., searching for a specific key in a distributed index) can be characterized as a function of one of these cost units. Generally speaking, the analysis of a distributed structure in terms of 'number of messages' is over-simplistic. It ignores the impact of many aspects, including the network topology, the network load and its variation, the possible heterogeneity of the software and hardware components involved in data processing and routing, etc. However, it is quite difficult to develop a precise cost model that would accurately take into account all these performance factors; therefore we've to live with rough but robust estimates of the system behavior.

Serviceability or Manageability

Another important consideration while designing a distributed system is how easy it is to operate and maintain. Serviceability or manageability is the simplicity and speed with which a system can be repaired or maintained; if the time to fix a failed system increases, then availability will decrease. Things to consider for manageability are the ease of diagnosing and understanding problems when they occur, ease of making updates or modifications, and how simple the system is to operate (i.e., does it routinely operate without failure or exceptions?).

Early detection of faults can decrease or avoid system downtime. For example, some enterprise systems can automatically call a service center (without human intervention) when the system experiences a system fault.

Why System Design Interviews?

1. What are system design interviews?

In system design interviews, candidates are required to show their ability to develop a high-level architecture of a large system. Designing software systems is a very broad topic and even a software engineer having years of experience at a top software company may not claim to be an expert on system design. During such interviews, candidates have 30 to 40 minutes to answer questions like, "How to design a cloud storage system like Dropbox?" or "How to design a search engine" etc. In real life, companies spend not weeks but months and hire a big team of software engineers to build such systems. Given this, then how can a person answer such a question in 40 minutes? Moreover, there is no set pattern of such questions. Questions are flexible, unpredictable, usually open-ended, and have no standard or squarely correct answer.

Unlike coding interviews where problem-solving ability of the candidates is evaluated, design interviews consist of complicated and fuzzy questions which aim at testing the candidate's ability to analyze a vague and complicated

problem, their compatibility with building large systems and how do they present their solution. Such interviews also look into a candidate's competence in guiding and moving the conversation forward.

These days, companies are least bothered about your pedigree, where you studied or where you come from but surely concerned about what you can do on the job. For them, the most important thing is your thought process and your mindset to look into and handle problems. For these counts, candidates are generally scared of the design interviews. But despite all this, I believe there is no need of scaring off. You need to get into what the companies want to know about you during these 40 minutes, which is basically "your approach and strategy to handle a problem" and how organized, disciplined, systematic, and professional you are at solving it. What is your capacity to analyze an issue and your level of professional mechanics to solve it step by step?

In short, system design interview is, just understanding it from interviewer's perspective. During the whole process, it is your discussion with the interviewer that is of core importance.

2. How to give system design interview?

There is no strictly defined process to system design interview. Secondly, there are so many things inherently unclear about large systems that without clarifying at least a few of them in the beginning, it would be impossible to go for a solution. Any candidate who does not realize this fact would fall into the trap of quickly jumping onto finding a solution.

For instance, the questions can be like:

- Design a URL shortening service like TinyURL.
- How would you build a social network like Facebook and implement a feature where one user receives notifications when their friends 'like' the same things as they do?
- How to design a ride-sharing service like Uber, which connects passengers who need a ride with drivers who have a car.

Any candidate who does not have experience in building systems might think such question grossly unfair. On top of that, there isn't one correct answer to such questions. The way you are answering the question would sufficiently tell upon your professional competence and background experience. That is the thing which the interviewer will evaluate you on.

Since the questions are intentionally weakly defined, jumping onto designing the solution immediately without fully understanding is liable to get you in trouble. Spend a few minutes questioning the interviewer to comprehend the full scope of the system. Never assume things that are not explicitly stated. For instance, the "URL shortening service" could be serving just a few thousand users, but each could be sharing millions of URLs. It could also mean to handle millions of clicks on the shortened URLs or just a few dozens. The service may also require providing extensive statistics about each shortened URL (which will increase our data size), or statistics may not be a requirement at all. Therefore, don't forget to make sure you gather all the requirements as the interviewer will not be listing them for you.

The main difference between design interviews and the rest is that you are not given the full detail of the problem, rather you are required to scale the breadth and depth of a blurred problem. You are supposed to take the details and figure out the issue by asking probing questions. Your questions for clarifying the problem reflects your evaluating ability and competence, which would be an asset to the company.

In design and architecture interviews the problems presented are quite significant. They definitely cannot be solved in 40 minutes' time implying that the objective is to test the technical depth and diversity the interviewee invokes during the interview. That also speaks strongly for your would be 'level' in the company. And your level in the company should come from your analytical ability to sort out the problem besides your ability to work in a team (your behavioral and background side of the interview), and your capacity to perform as a strong technical leader. In a nutshell, the basic idea of hiring at a level is to scale a person's ability to contribute value to the company's wants and needs. For that, you must exhibit your strengths by showing reasonable technical breadth.

Try to learn from the existing systems: How have these been designed? Another critical point to be kept in mind is that the interviewer expects that candidate's analytical ability and questioning on the problem must comparable to

his/her experience. If you have a few years of software development experience, you are expected to have certain knowledge and should avoid divulging into asking basic questions that might have been appropriate coming from a fresh graduate. For that, you should prepare sufficiently ahead of time. Try to go through real projects and practices well in advance of the interview as most questions are based on real-life products, issues, and challenges.

Leading the conversation: It is not the ultimate solution to the problem, instead the discussion process itself that is important in the interview. And it is the candidate who should lead the conversation going both broad and deep into the components of the problem. Hence, take the interviewer along with you during the course of solving the problem by communicating with him/her step by step as you move along

Solving by breaking down: Design questions at first might look complicated and intimidating. But whatever the complexity level of the problem, a top-down and modularization approach can help a lot in solving the problem. Therefore, you should break the problem into modules and then tackle each of them independently. Subsequently, each component can be explained as a sub-problem by reducing it to the level of a known algorithm. This strategy will not only make the design much clearer to you and your interviewer but make evaluation much easier for the interviewer. However, while doing so, keep this thing in mind that mostly the problems presented in high skill design interviews don’t have the solutions. The most important thing is the way how you make progress tackling the problem and the strategies you adopt.

Dealing with the bottlenecks: Working on the solution, you might confront some bottlenecks. This is very normal. While resolving bottlenecks, your system might require a load balancer with many machines behind it to handle the user requests or the data might be so huge that you need to distribute your database on multiple servers. It might also be possible that the interviewer wants to take the interview in a particular direction. If that is the case, you are supposed to move in that direction and should go deep leaving everything else aside. If you feel stuck somewhere, you can ask for a hint so that you may keep going. Keep in mind that each solution is a kind of trade-off; hence, changing something may worsen something else. Here, the important thing is your ability to talk about these trade-offs and to measure their impact on the system keeping all the constraints and use cases in mind. After finishing your high-level design and making sure that the interviewer is ok with it, you can go for making it more detailed. Usually, that means making your system scale.

3. Summary

Solving system design questions could be broken down into three steps:

- **Scoping the problem:** Don’t make assumptions; Ask clarifying questions to understand the constraints and use cases.
- **Sketching up an abstract design** Illustrating the building blocks of the system and the relationships between them.
- **Identifying and addressing the bottlenecks** by using the fundamental principles of scalable system design.

4. Conclusion

Design interviews are formidable, open-ended problems that cannot be solved in the allotted time. Therefore, you should try to understand what your interviewer intends to focus on and spend sufficient time on it. Be well aware of the fact that the discussion on system design problem could go in different directions depending on the preferences of the interviewer. The interviewers might be unwilling to see how you create a high-level architecture covering all aspects of the system or they could be interested in looking for specific areas and diving deep into them. This means that you must deal with the situation strategically as there are chances of even the good candidates failing the interview, not because they don’t have the knowledge, but because they lack the ability to focus on the right things while discussing the problem.

If you have no idea how to solve these kinds of problems, you can familiarize yourself with the typical patterns of system design by reading diversely from the blogs, watching videos of tech talks from conferences. It is also advisable to arrange discussions and even mock interviews with experienced engineers at big tech companies.

Remember there is no ONE right answer to the question because any system can be built in different ways. **The only thing that is going to be looked into is your ability to rationalize ideas and inputs.**