System Design

Thomas Monson

Contents

1 System Design Interview Process

You are given a *problem statement*.

1.1 Requirements Clarification

This shows the interview how you deal with ambiguity. It also reduces the scope of the problem to something that can be discussed within the time constraints of an interview.

1.1.1 Ask Questions

- Users/customers
  - Who will use the system?
  - How will the system be used?
- Scale (read/write)
  - How many read queries per second?
  - How many write queries per second?
  - How much data is queried per request?
  - Can there be spikes in traffic?
- Performance
  - What is the expected write-to-read data delay? (batch processing vs. stream processing)
  - How fast must data be read from the system? (p99 (worst-case) latency)

    $$\text{as fast as possible} \rightarrow \text{perform complex operations on write}$$
- Cost
  - Should the design minimize development cost? (open-source vs. proprietary)
  - Should the design minimize maintenance cost? (public cloud services vs. on-prem)

Requirements clarification gets us closer to defining functional and non-functional requirements.

1.1.2 Define Functional Requirements

Functional requirements specify system behavior in terms of APIs, a set of operations that the system will support.

Write down something that the system must do, convert that into a function, and then generalize as necessary. For example:

> "The system has to **count video view events**."
>
> `countViewEvent(videoId)`
>
> `countEvent(videoId, eventType)`
>
> `processEvent(videoId, eventType, functionType)`
>
> `processEvents(listOfEvents)` (Event is an object here)

1.1.3 Define Non-functional Requirements

Non-functional requirements specify system qualities (e.g. fast, fault-tolerant, secure).

Possible non-functional requirements include: scalability, availability, consistency, performance, durability, maintainability, inexpensive. For example, a system might have the following requirements:

- Scalable: Tens of thousands of video views per second
- Performant: A few tens of milliseconds to return total view count for a video
- Available: Survives hardware and network failures, no single point of failure

1.2 High-Level Design

Think about how data gets into the system and how it gets out and draw some components.

1.3 Detailed Design

The first thing to understand is what data we need to store, where we should store it, and how we should store it. We need to define a *data model*.

1.3.1 What to Store

To count video views, we can store individual events (every click) or we can aggregate data (e.g. per minute) in real-time.

Individual events:

- Fast writes (get the whole event, push it to the database)
- Can slice and dice data as needed
- Can recalculate numbers in reports (because we have all the original data)
- Slow reads (need to count every event to count views)
- Needs a lot of storage, expensive

Aggregated data:

- Fast reads (direct access)
- Data is ready for decision making
- Can query only the way the data was aggregated (details lost due to aggregation)
- Requires data aggregation pipeline
- Hard or impossible to fix calculation errors (because we do not have all the original data)

If the system must process events within no more than a few minutes after they occur, then we should store individual events (stream processing). If events can be processed hours later, then we can store aggregated data (batch processing).

You could also choose both options. For example, store individual events for several days or weeks and then delete them. Also, calculate and store numbers in real-time. This way, you get fast reads, the ability to aggregate data differently, and the ability to fix errors by referencing individual events. The downside to this approach is that the system becomes more complex and expensive.

1.3.2 Where to Store

Decide between SQL or NoSQL.

1.3.3 How to Store

Blah blah

1.3.4 Processing and Querying Details

1.3.5 Technology Stack

1.4 Bottlenecks and Tradeoffs

How to identify bottlenecks: load testing, stress testing, soak testing (Apache JMeter can produce a desired load)

Monitoring, metrics, dashboards

How to make sure the system is correct: audit systems

2 Components

2.1 Domain Name System (DNS) Server

A *DNS server* is a server that translates hostnames to IP addresses.

When a client sends a request to a DNS server for an IP address, it typically caches that IP so it doesn't have to query the DNS server again. If that IP is a server instead of a load balancer, and a client sends all of their requests to that one IP, and those requests are hard, then that server will be overloaded.

2.2 Content Delivery Network (CDN)

A *content delivery network* is a geographically distributed network of servers whose purpose is to store data assets (e.g. text, images, video, files, applications) spatially closer to users for high availability and performance.

2.3 Load Balancer

A (hardware or software) component that distributes requests from clients across N servers

Instead of DNS returning the IP of a single server, it can return the IP of the load balancer, which will choose which server to send the request to. This is helpful for *horizontal scaling.*

The load balancer can route traffic using various strategies:

- **Random**
- **Least-loaded**
- **Round-robin:** first server 1, then server 2, ..., back to server 1 (by chance, one server could receive the hardest requests and be overloaded)
- **Session/cookies:** send a user to the same server every time (store a hash in a cookie)
- **Layer 4 (Transport):** manage traffic based on network information such as source and destination IP addresses and TCP or UDP ports

- **Layer 7 (Application):** manage traffic based on HTTP/SMTP information like the content of headers, message bodies, and cookies (e.g. send requests for HTML to dedicated HTML server, send requests for images to dedicated image server)

If the load balancer sends a client to server 1, and server 1 stores a session for that client, and then the load balancer sends that client to server 2, then server 2 will not have that client's session (problem of *sticky sessions*). This is why it is recommended to store all user-related data (such as sessions) in a centralized data store (*shared storage*).

Multiple load balancers are often used for redundancy. They can be configured as active/passive (active dies, passive promotes itself to active by taking over active IP) or active/active. They check on each other by sending heartbeat packets. This configuration results in *high availability* (same concept exists for databases).

A Level 7 load balancer is a reverse proxy.

2.4 Reverse Proxy

A web server that acts as an intermediary that forwards client requests to backend servers and forwards server responses to clients

Benefits include:

- **Increased security:** hide information about backend servers, blacklist IPs, limit number of connections per client
- **Increased scalability and flexibility:** clients only see reverse proxy's IP, allowing you to scale servers or change their configuration
- **SSL termination:** decrypt incoming requests or encrypt server responses so backend servers do not have to perform these potentially expensive operations (no need to install X.509 certificates on each server)
- **Compression:** compress server responses
- **Caching:** return a cached response
- **Static content:** serve static content directly (HTML, images, videos)

2.5 Relational Database

Relational databases traditionally have *ACID transactions* and favor consistency over availability.

- **Atomicity:** Each transaction is all or nothing.
- **Consistency:** Any transaction will bring the database from one valid state to another.
- **Isolation:** Executing transactions concurrently has the same results as if the transactions were executed serially.
- **Durability:** Once a transaction has been committed, it will remain so.

2.5.1 Replication

Primary-replica Replication

The *primary* serves reads and writes, replicating writes to one or more *replicas*, which serve only reads. Replicas can also replicate to additional replicas in a tree-like fashion. If the primary goes offline, the system can continue to operate in read-only mode until a replica is promoted to a primary or a new primary is provisioned.

Additional logic is needed to promote a replica to primary.

Multi-primary Replication

Primaries serve reads and writes and coordinate with each other on writes. If any primary goes down, the system can continue to operate with both reads and writes.

A load balancer or additional logic is needed to determine where to write.

Most multi-primary systems are either loosely consistent (non-ACID) or have increased write latency due to synchronization.

Write conflict resolution is needed more often as more write nodes are added and latency increases.

Disadvantages

- Data loss, if a primary fails before newly written data can be replicated
- Lots of writes → replicas may get overloaded → unable to serve reads effectively
- More replicas → more replication → more replication lag
- More hardware and additional complexity

2.5.2 Federation (Functional Partitioning)

Federation (or functional partitioning) splits up databases by function. For example, instead of a single, monolithic database, you could have three databases: *forums*, *users*, and *products*.

Advantages

- Less read and write traffic
- Less replication and replication lag
- More cache hits (smaller database → more of database can fit in memory → improved cache locality)
- No central primary serializing writes → write in parallel with increased throughput

Disadvantages

- Not effective if your schema requires huge functions or tables
- Need to update application logic to determine which database to read-/write
- Joining data from multiple databases is more complex (server link).
- More hardware and additional complexity

2.5.3 Sharding

Sharding splits up databases by data value, such that each database manages only a subset of the data. For example, a users database could be split into shards for last names A–M and N–Z. Sharding by geographic location is also common practice.

Sharding is a form of *horizontal partitioning*.

Advantages

- Less read and write traffic
- Less replication and replication lag
- More cache hits (smaller database → more of database can fit in memory → improved cache locality)
- Reduced index size (improves performance with faster queries)
- If one shard fails, the other shards remain operational (replication needed to avoid data loss).
- No central primary serializing writes → write in parallel with increased throughput

Disadvantages

- Need to update application logic to work with shards, could result in complex SQL queries
- Load may be unbalanced across shards. Rebalancing requires additional complexity. A sharding function based on *consistent hashing* can distribute load more evenly.
- Joining data from multiple shards is more complex.
- More hardware and additional complexity

2.5.4 Denormalization

Informally, a *normalized database* is a database that is structured in accordance with a series of so-called *normal forms* in order to reduce data redundancy and improve data consistency. *Denormalization* is the process of storing redundant copies of data in tables in order to avoid expensive joins. Denormalization attempts to improve read performance at the expense of some write performance (in most systems, reads outnumber writes 100:1 or even 1000:1).

One example of denormalization is the use of *materialized views* (found in PostgreSQL and Oracle). A materialized view is the cached result of a query. (A materialized view that stores data based on remote tables is called a *snapshot*.) The RDBMS will try to keep redundant data consistent by recomputing the materialized views occasionally.

Denormalization can be helpful for federated or sharded databases, as queries across nodes in such systems can be complex.

Advantages

- Faster reads, especially for partitioned systems
- Simpler queries

Disadvantages

- Redundant data
- Data may be inconsistent.
- Additional complexity required to keep redundant data consistent
- Worse than a normalized database when under heavy write load

2.5.5 Query Tuning (SQL Tuning)

Query tuning is the iterative process of improving queries to speed up server performance. It is important to *benchmark* and *profile* the system to simulate and uncover bottlenecks:

Benchmark: simulate high-load situations with tools such as `ab`

Profile: enable tools such as MySQL's *slow query log* to track performance issues

Benchmarking and profiling may point you to the following optimizations:

Tighten up the schema

-

Use good indicies

-

Avoid expensive joins: denormalize where performance demands it.

Partition tables: break up a table by putting hot spots in a separate table to help keep it in memory.

Tune the query cache

2.6 NoSQL Database

NoSQL is an approach to database design that enables the storage and querying of data by some means other than the tabular relations characteristic of relational databases. A NoSQL database may be "non-SQL" or "not only SQL".

Data is denormalized, and joins are typically done in application code. Most NoSQL stores lack true ACID transactions and favor availability over consistency (opting for eventual consistency). *BASE* is often used to describe the properties of NoSQL databases:

- **Basically available:** the system guarantees availability.

- **Soft state:** the state of the system may change over time, even without input.
- **Eventual consistency:** the system will become consistent over a period of time, given that the system does not receive input during that period.

2.6.1 Key-Value Store

A *key-value store* is essentially a hash map stored in memory or in an SSD. It has $O(1)$ reads and writes. It can maintain keys in lexicographic order, allowing efficient retrieval of key ranges. A value in the store can have metadata stored along with it.

Key-value stores are often used for simple data models or for rapidly-changing data, such as an in-memory cache layer. Since they offer only a limited set of operations, complexity is shifted to the application layer if additional operations are needed.

A key-value store is the basis for more complex systems, such as a document store or, in some cases, a graph database.

2.6.2 Document Store

A *document store* is a key-value store where the values are documents (JSON, XML, binary, etc.). A document stores all the information for a given object.

Document stores provide APIs or a query language to query based on the internal structure of the document itself, similar to how many key-value stores provide features to work with a value's metadata. Based on the underlying implementation, documents are organized by collections, tags, metadata, or directories.

Document stores provide high flexibility and are often used when working with data that changes occasionally.

Implementations: MongoDB, CouchDB, DynamoDB (also a key-value store)

2.6.3 Wide-Column Store

The basic unit of data in a *wide-column store* is a *column*, which is a key-value pair where the key is a *column name* and the value is a *column value*:

Name
Value

Each value contains a timestamp for versioning and conflict resolution.

A *super column* is a key-value pair where the key is a *super column name* and the value is a *set of columns*:

Super Column Name		
Name	. . .	Name
Value		Value

A *column family* is a collection of key-value pairs where each key is a *row key* and each value is a *set of columns*:

Row Key	Name	. . .	Name
	Value		Value

A column family is akin to a table in a relational database. Unlike a relational database, the names and formats of the columns can vary from row to row in the same table. A wide-column store is like a two-dimensional key-value store; given a column family (table), you can access a value with a row key and a column name.

A *super column family* is a collection of key-value pairs where each key is a *row key* and each value is a *set of super columns*:

Row Key	Super Column Name			...	Super Column Name		
	Name	...	Name		Name	...	Name
	Value		Value		Value		Value

A super column family is akin to a *view* on a set of tables in a relational database.

Wide-column stores offer high availability and high scalability. They are often used for very large datasets.

Implementations: Bigtable (Google), Cassandra (Meta), HBase (open-source)

2.6.4 Graph Database

In a graph database, each node is a record and each arc is a relationship between two nodes. Graph databases are optimized to represent complex relationships with many foreign keys or many-to-many relationships.

Graphs databases offer high performance for data models with complex relationships, such as a social network.

2.7 Cache

A *cache* is a key-value store that acts as a buffer between two layers of a system, often between the application layer and the database layer. It stores data so that future requests for that data can be served faster.

Caching improves page load times and can reduce the load on your servers and databases. In this model, the dispatcher (load balancer, reverse proxy) will first lookup if the request has been made before and try to find the previous result in the cache. If it cannot find it, the dispatcher will forward the request to a worker and store the eventual result in the cache.

A database often benefits from a uniform distribution of reads and writes across its partitions. Popular items can skew the distribution, causing bottlenecks. Putting a cache in front of a database can help absorb uneven loads and spikes in traffic.

2.7.1 Client Caching

Caches can be located on the client side (OS or browser). This is the most efficient form of caching because it allows the client to access resources without communicating with a web server. Common files that are cached client-side include images, HTML, stylesheets, and JavaScript libraries.

2.7.2 CDN Caching

CDNs are considered a type of cache.

2.7.3 Web Server Caching

Reverse proxies (such as Varnish) can serve static and dynamic content directly, by retrieving result from an internal cache. Such proxies are considered *web accelerators* because they reduce website access time.

Web servers can also cache requests, returning responses without having to contact application servers.

2.7.4 Application Server Caching

In-memory caches (such as Memcached and Redis) are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk. RAM is more limited than disk, so cache invalidation algorithms such as *least recently used (LRU)* can help invalidate 'cold' entries and keep 'hot' data in RAM.

Redis has additional features like persistence and built-in data structures such as sorted sets and lists.

Generally, you should try to avoid file-based caching, as it makes cloning and auto-scaling more difficult.

Caching at the query level
Whenever you query the database, hash the query as a key and store the result to the cache. This approach suffers from expiration issues:

- Hard to delete a cached result with complex queries
- If one piece of data changes such as a table cell, you need to delete all cached queries that might include the changed cell.

Caching at the object level
See your data as an object, similar to what you do with your application code. Have your application assemble the result from the database into a class instance or a data structure.

- Remove the object from cache if its underlying data has changed
- Allows for asynchronous processing: workers assemble objects by consuming the latest cached object

2.7.5 Database Caching

A database usually includes some level of caching in a default configuration, optimized for a generic use case. Tweaking these settings for specific usage patterns can further boost performance.

2.7.6 Cache-Aside (Lazy Loading) Update Strategy

The application is responsible for reading and writing from storage. The cache does not interact with storage directly. The application does the following:

- Look for entry in cache, resulting in a cache miss
- Load entry from the database
- Add entry to cache
- Return entry

```
def get_user(self, user_id):
    user = cache.get("user.{0}", user_id)
    if user is None:
        user = db.query("SELECT * FROM users WHERE user_id =
                                        {0}", user_id)
        if user is not None:
            key = "user.{0}".format(user_id)
            cache.set(key, json.dumps(user))
    return user
```

Memcached is generally used in this manner. Only requested data is cached, which avoids filling up the cache with data that isn't requested.

Disadvantages

- Each cache miss results in three trips.
- Data can become stale if it is updated in the database. This issue is mitigated by setting a *time-to-live (TTL)*, which forces a deletion or an update of the cache entry.
- When a cache node fails, it is replaced by a new, empty node, increasing latency.

2.7.7 Write-Through Update Strategy

The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:

- Application adds/updates entry in cache
- Cache synchronously writes entry to data store

```
# Application code
set_user(12345, {"foo":"bar"})
```

```
# Cache code
def set_user(user_id, values):
    user = db.query("UPDATE Users WHERE id = {0}", user_id,
                                values)
    cache.set(user_id, user)
```

Write-through is a slow overall operation due to the write operation, but subsequent reads of just written data are fast. Users are generally more tolerant of latency when updating data than reading data. Data in the cache is not stale.

Disadvantages

- When a new node is created due to failure or scaling, it will not cache an entry until that entry is updated in the database. That is, a read request for data not in cache will not put that data in cache. Cache-aside in conjunction with write-through can mitigate this issue.
- Most data written may never be read (define a TTL to remove these entries).

2.7.8 Write-Behind (Write-Back) Update Strategy

In write-behind, the application does the following:

- Add/update entry in cache
- Asynchronously write entry to the data store, improving write performance (put request in message queue, dequeue request with event processor, write to database)

Disadvantages

- Data loss if cache fails before its data is written to the database
- Harder to implement

2.7.9 Refresh-Ahead Update Strategy

Configure the cache to automatically refresh any recently accessed cache entry prior to its expiration.

Refresh-ahead can result in reduced latency vs read-through if the cache can accurately predict which items are likely to be needed in the future. Not accurately predicting can result in worse performance.

2.8 Queue (Asynchronism)

A *message queue* receives, holds, and delivers messages. They facilitate *asynchronous workflows*, which can reduce response time for expensive operations that would otherwise be done in-line.

A *task queue* supports scheduling and can be used to run computationally-intensive jobs in the background.

- Method 1: do time-consuming work in advance, serve finished work with low request time. For example, complicated web pages can be pre-rendered and stored as static HTML pages on every change.
- Method 2: put time-consuming work into a job queue
    1. A user starts a computationally intensive task
    2. The frontend sends the task to a job queue and informs the user that the job is being processed
    3. Workers are constantly checking the job queue for jobs. One of them picks up the new job, works on it, and sends a signal to the frontend that the job is done
    4. The frontend is constant checking for “job is done" signals. It picks up the signal and informs the user that their job is done.

While a job is being processed, a client can optionally do a small amount of work to make it seem like the task has completed. For example, when posting a tweet, the tweet could be instantly posted to your timeline, but it could take some time before your tweet is actually delivered to all of your followers.

If queues start to grow significantly, the queue size can become larger than memory, resulting in cache misses, disk reads, and even slower performance. This happens when a producer sends messages to a queue faster than the consumer can process them. A *back pressure* mechanism can help with this by slowing down the producer’s rate of sending messages to match the rate at which the consumer can process them. Once the queue gets to a fixed size, clients will get a “server busy" or HTTP 503 status code.

Queues may not be effective for inexpensive operations or real-time workflows.

Implementations: RabbitMQ, ActiveMQ, Redis, Amazon SQS, Celery (scheduling)

3 Concepts

3.1 Communication

Communication protocols enable an entity in one host to interact with a corresponding entity at the same layer in another host. At each layer N, two entities (layer N peers) exchange *protocol data units* (PDUs) by means of a layer N protocol. Each PDU contains a payload, called the *service data unit* (SDU), along with protocol-related headers or footers.

The process of communication between a transmitting device and a receiving device is as follows:

1. The data to be transmitted is composed at the topmost layer of the transmitting device (layer N) into a PDU.
2. The layer-N PDU is passed to layer $N - 1$, where it is considered an SDU.
3. At layer $N - 1$, the SDU is concatenated with a header and/or footer, producing a layer-$N - 1$ PDU. This process is known as *encapsulation*. The PDU is then passed to layer $N - 2$.
4. This process continues until the lowest level is reached, from which the data is transmitted to the receiving device.
5. At the receiving device, the data is passed from the lowest to the highest layer. Each layer receives the data as a PDU, strips the layer-specific headers/footers from the PDU to produce an SDU, and passes the SDU up a layer, where it will be considered a PDU. At the topmost layer, the data is consumed by an application.

3.1.1 The OSI Model

Application (Layer 7)

Serves as the window for users and application processes to access the network services. At this layer, a program creates data to be sent or opens data it has received.

Facilitates resource sharing, remote file access, remote printer access, directory services, and network management.

Protocols include HTTP(S), SMTP, (S)FTP, SSH, DHCP, and DNS.

Presentation (Layer 6)

Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network. Sometimes called the "syntax layer" because it relieves application-layer protocols of concern regarding syntactical differences in data representation within end-user systems.

Facilitates character code translation, data conversion, data compression, data encryption, and character set translation.

Protocols include TLS and SSL. Works with standards like JPEG, ASCII, and EBCDIC.

Session (Layer 5)

Allows session establishment between processes running on different stations.

Facilitates session establishment (log-on), maintenance, and termination (log-off), and session support (security, name recognition, logging).

The RPC protocol is executed at this layer. *Logical ports* operate at this layer.

Transport (Layer 4)

Ensures that messages are delivered error-free, in sequence, and with no losses or duplications.

Facilitates message segmentation, message acknowledgement, message traffic control, and session multiplexing.

Protocols include TCP and UDP.

Network (Layer 3)

Controls the operations of the subnet, deciding which physical path the data takes. Transfers *packets* from one node to another in a different network.

Facilitates routing, subnet traffic control, frame fragmentation, logical-physical address mapping, and subnet usage accounting.

Protocols include IP, IPX, and ICMP. Routers operate at this layer.

Data Link (Layer 2)

Provides error-free transfer of *data frames* from one node to another over the Physical layer.

Facilitates the establishment and termination of the logical link between nodes, frame traffic control, frame sequencing, frame acknowledgement, frame delimiting, frame error checking, and media access control.

Protocols include PPP, Ethernet, and Wi-Fi. *Switches*, *bridges*, and *wireless access points* (WAPs) operate at this layer.

Physical (Layer 1)

Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium.

Facilitates data encoding, physical medium attachment, transmission technique (baseband or broadband), and physical medium transmission (bits and volts).

Method	**Description**	**Idempotent**	**Safe**	**Cacheable**
GET		Yes	Yes	
POST		No	No	
PUT		Yes	No	
PATCH		No	No	
DELETE		Yes	No	
CONNECT				
TRACE				
OPTIONS				
HEAD				

Cables and hubs operate at this layer.

3.1.2 Hypertext Transfer Protocol (HTTP)

HTTP is an application layer request/response protocol for encoding and transporting data between a client and a server.
An HTTP message consists of:

- A *start-line* (either a *request-line* or a *status-line*, depending on whether the message is a request or a response)
- Zero or more *header field lines*, also known as *headers*, which are key-value pairs, each followed by a `CR-LF` character sequence
- An empty line (i.e. a line with nothing preceding the `CR-LF`), indicating the end of the header fields
- Optionally, a message body (payload)

An HTTP request-line is a space-separated sequence of the following:

- An HTTP method (e.g. `GET`, `POST`, etc.)
- A *request-target* (identifies the resource upon which to apply the request)

  Can be in any of the following formats, depending on the method used:

  - *origin form*, an absolute path. This is the most common form, used when requests are made to an origin server.

    `/pub/WWW/ThePage.html` (cannot be empty, `/` at least)

  - *absolute form*, a complete URL. Mostly used when a `GET` request is made to a proxy.

    `https://www.website.com/pub/WWW/ThePage.html`

  - *authority form*, a domain name and port. Only used with `CONNECT` when setting up an HTTP tunnel.

    `website.com:80`

  - *asterisk form*, `*` (not a resource, but the server itself). Used with `OPTIONS`.

- An HTTP version (e.g. `HTTP/1.1`)
- `CR-LF`

For example, a request-line could look like:

```
GET https://www.website.com/pub/WWW/ThePage.html HTTP/1.1
```

An HTTP status-line is a space-separated sequence of the following:

- An HTTP version
- A *status-code* (3-digit integer code describing the server's attempt to serve the client's request)
- A *reason-phrase* (a human readable description of the status-code)

For example, a status-line could look like: `HTTP/1.1 200 OK`

There are three types of HTTP headers:

- *Request header*: contains information about the resource to be accessed or about the client itself
- *Response header*: contains information about the response or about the server itself

- *Representation header*: contains metadata about the resource in the message body

Some important headers include:

- `Host`
- `Accept`
- `Authorization`
- `User-Agent`
- `Content-Type`
- `Content-Length`
- `Cookie`
- `Set-Cookie`
- `Server`

3.1.3 Transmission Control Protocol (TCP)

TCP is a connection-oriented protocol over an IP network. That is, a sender and receiver must first establish a connection based on agreed parameters, and they do this through a *three-way handshake*:

1. Alice sends Bob a synchronize message (SYN) with sequence number x
2. Bob sends Alice a synchronize-acknowledgement message (SYN-ACK) with sequence number y and acknowledgement number $x + 1$
3. Alice sends Bob an acknowledgement message (ACK) with acknowledgement number $y + 1$

All packets sent are guaranteed to reach the destination in the original order and without corruption by means of:

- Sequence numbers and checksum fields for each packet

- Acknowledgement packets and automatic retransmission

If the sender does not receive a correct response, it will resend the packets. If there are multiple timeouts, the connection is dropped. TCP also implements *flow control* and *congestion control*. These guarantees cause delays and generally result in less efficient transmission than UDP.

To ensure high throughput, web servers can keep a large number of TCP connections open, resulting in high memory usage. Connection pooling can help with this, in addition to switching to UDP where applicable.

TCP is useful for applications that require high reliability but are less time critical. Some examples include web servers, database info, SMTP, FTP, and SSH.

Use TCP over UDP when:

- You need all of the data to arrive intact.
- You want to automatically make a best estimate use of the network throughput.

Some TCP ports:

- 80: HTTP requests (can be used once an encrypted request from the Internet is past the firewall of a system)
- 443: HTTPS requests
- 22: SSH requests
- 3306: MySQL requests

3.1.4 User Datagram Protocol (UDP)

UDP is connectionless. Datagrams (analogous to packets) are guaranteed only at the datagram level. Datagrams might reach their destination out of order or not at all. UDP does not support congestion control. Without the guarantees that TCP support, UDP is generally more efficient.

UDP is less reliable but works well in real-time use cases such as VoIP, video chat, streaming, and real-time multiplayer games.
Use UDP over TCP when:

- You need the lowest latency.
- Late data is worse than loss of data.
- You want to implement your own error correction.

3.1.5 Remote Procedure Call (RPC)

An RPC is a kind of function call that, when made by a client, causes a procedure to be executed in a different address space, typically on a remote server. The call is written in code just as a local procedure call would be. That is, the programmer does not need to write out the details for remote interaction explicitly. RPCs are usually slower and less reliable than local calls, so it is helpful to distinguish them.

An RPC is made by means of a request/response protocol:

- The client program calls a (local) client stub procedure. Parameters (procedure ID and arguments) are pushed onto the stack.
- The client stub procedure marshals the parameters into a request message.
- The client communication module sends the message from the client to the server.
- The server communication module passes the incoming packets to the server stub procedure.
- The server stub procedure unmarshals the parameters from the message, passes the arguments to the server procedure matching the procedure ID, and calls that procedure.
- The server sends its response by repeating the steps above in the other direction.

RPCs expose behaviors. REST tends to be used more often for public APIs.

RPC frameworks include Protobuf, Thrift, and Avro.

Disadvantages

- RPC clients become tightly coupled to the service implementation.
- A new API must be defined for every new operation or use case.
- It can be difficult to debug RPCs.

3.1.6 Representational State Transfer (REST)

REST is a *software architectural style* that describes how a very large, distributed network of computers that share resources amongst each other—like the Internet—should behave. Because REST was designed with a global-scale system in mind, it specifies that the coupling between client and server should be as loose as possible, to facilitate large-scale adoption.

Representation state transfer means that a server will respond with a *representation of a resource* (often an HTML, JSON, or XML document), and that representation will contain hyperlinks that can be followed to make the state of the system change. A client only needs to know the identifier of the first resource it requests; all other identifiers will be discovered in responses.

The six architectural constraints of REST APIs:

- **Client/server:** Clients should be separated from servers by a well-defined interface. UI logic should not be coupled with business logic and data-storage logic.
- **Stateless:** When a client makes a request, it should include all of the information necessary for the server to fulfill the request. The server should never have to rely on information from previous requests from the client. If any such information is important, then the client should send it as part of the current request.
- **Cacheable:** Responses should indicate whether or not their data can be cached and for how long. If data is cacheable, the client should

automatically cache it and retrieve the data from the cache instead of making an HTTP request for the same information. This reduces unnecessary client-server interaction and improves the availability of servers.

- **Uniform interface:** There should be a standard and uniform way of interacting with servers. Querying resources should allow the client to request other actions and resources without knowing about them in advance.

  1. **Identification of resources:** Resources should be identified by a unique and stable indentifier. Use the URI standard to identify resources.
  2. **Manipulation of resources through representations:** Clients should get a uniform representation of a resource that contains enough information to update or delete that resource. Use the HTTP standard to describe communication.
  3. **Self-descriptive messages:** Messages sent between the client and server should contain all of information that the receiver needs to process the message successfully. That is, messages contain not only representations of data but metadata describing the payload as well. Use IANA media types (MIME types) to identify the format of representations and use RDF vocabulary to describe metadata.
  4. **Hypermedia As The Engine Of Application State (HATEOAS):** A representation of a resource that is sent to a client should have enough information, in the form of hyperlinks, to allow that client to dynamically discover other resources and drive other interactions. This allows a client to "browse" an API similar to how one would browse a webpage. Also, a client will not need to hardcode API URLs in its codebase; instead, they can simply reference a property in the representation they have received whose value is a hyperlink.

- **Layered system:** A client should not be able to tell whether it is connected directly to an end server or to an intermediary along the way. In other words, there shouldn't be a functional difference between interacting with a single server or a layered system.

- **Code-On-Demand (optional):** A server should be able to send executable code to a client to extend the functionality of that client.

REST APIs expose data. Because it is stateless, REST is great for horizontal scaling and partitioning.

Disadvantages

- Because REST is focused on exposing data, it might not be a good fit if resources are not naturally organized or accessed in a simple hierarchy. For example, returning all updated records from the past hour matching a particular set of events is not easily expressed as a path. With REST, it is likely to be implemented with a combination of URI path, query parameters, and possibly a request body.
- REST relies on HTTP methods, which may not be able to describe a use case well. For example, moving expired documents to the archive folder may not be able to be expressed with a single HTTP request.
- Fetching complicated resources with nested hierarchies requires multiple round-trips between the client and server to render single views (e.g. fetching content of a blog entry and the comments on that entry). For mobile applications operating in variable network conditions, these multiple round-trips are highly undesirable.
- Over time, more fields might be added to an API response, and older clients will receive all new data fields, even those that they do not need. As a result, payload size and latency increases.

3.1.7 GraphQL

3.1.8 WebSocket

3.1.9 Simple Object Access Protocol (SOAP)

3.2 Application Layer

You can separate your servers into *web servers* (serve static content) and *application servers* (serve dynamic content by means of business logic), and you can scale each layer independently. For example, you could add applicaition servers to support a new API without necessarily adding web servers.

Microservices are independently deployable, small, modular services. Each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.

Systems such as Consul, Etcd, and Zookeeper can help services find each other by keeping track of registered names, addresses, and ports.

3.3 Scalability

- *Vertical scaling*: building up computational resources of a single machine
- *Horizontal scaling*: acquiring more machines to serve your application
- Every server should contain exactly the same codebase and should not store any user-related data, like sessions or profile pictures, on local disc or memory.
- Sessions should be stored in a centralized data store which is accessible to all your application servers. It can be an external database or an external persistent cache, like Redis.
- You can create an *image file* from one of your application servers and use it to clone another server instance.

3.4 CAP Theorem

Would you prefer to show stale data or no data?

Any distributed data store can provide only two of the following three guarantees:

- Consistency: every read receives the most recent write or an error
- Availability: every request receives a (non-error) response, without guarantee that it contains the most recent write

- Partition tolerance: the system continues to operate despite an arbitrary number of messages being dropped (or delayed) between nodes (network failure)

You need to support partition tolerance because networks are not reliable. So you must choose between consistency and availability.

- CP: waiting for a response from the partitioned node might result in a timeout error. Good if business needs atomic reads and writes.
- AP: responses return the most readily available version of the data, which might not be the latest. Writes might take some time to propagate when the partition is resolved. Good when system needs to continue working despite external errors (eventual consistency).

Consistency patterns:

- Weak consistency: after a write, reads may or may not see it.
- Eventual consistency: after a write, reads will eventually see it.
- Strong consistency: after a write, reads will see it.

3.5 RAID (Redundant Array of Independent Disks)

- RAID0 (*striping*): two drives, write data in "stripes", effectively double read/write speeds
- RAID1 (*mirroring*): two drives, mirror data between them for redundancy
- RAID5 (*distributed parity*): N drives, system can operate with $N - 1$ drives without data loss (one failure allowed), parity information is distributed across drives
- RAID6 (*dual parity*): RAID5 + another drive, two parity blocks per stripe, two failures allowed
- RAID10 (*striping of mirrors*): four drives, RAID0 + RAID1

4 To Do

- GraphQL (can fetch multiple resources with a single request (query) without fetching data that is not needed)
- gRPC
- WebSocket (supports bidirectional communication (not request/response, client and server can exchange messages in any order))
- Storage Engines
- Object Storage
- Dead-letter queue
- API gateway
- Blocking vs. non-blocking IO
- Gossip protocol
- Service discovery (client-side and server-side), service registry (Zookeeper)

A Glossary

hypertext: electronic text with references (hyperlinks) to other electronic text that the reader can immediately access. It is "hyper" in the sense that it goes "beyond" the linear constraint of written text.