CAP Theorem

What is it?

In distributed systems where multiple nodes (machines) are responsible for the read/write of a service, there is an inherent conflict on what can be guaranteed for the data being read. The CAP theorem states that in a distributed system you can only satisfy 2 out of the following 3.

- *Consistency*: A read is guaranteed to return the most recent write or an error
- *Availability*: A read is guaranteed to return a response, although it doesn't guarantee it's the most recent write
- *Partition Tolerance*: The system is tolerant and continues to operate despite network failures (arbitrary partitioning)

In modern distributed systems, fault tolerance to network failures is typically a strong requirement, hence we need to choose between *Consistency* and *Availability*.

Trade-offs, caveats and alternatives

- CP (Consistency, Partition Tolerance): Wait response from the partitioned node with the latest write, potentially resulting in an error (e.g. timeout). Useful for systems that require strong atomic guarantees on reads/writes (e.g. financial transaction services).
- AP (Availability, Partition Tolerance): Return the most recent "available" version of the data despite it being possibly stale. Useful for systems that benefit from returning something recent over an error (e.g. posts on social network feed).
- A majority of the system design questions these days favor towards building AP systems, with eventual or weak consistency.

Use it for…

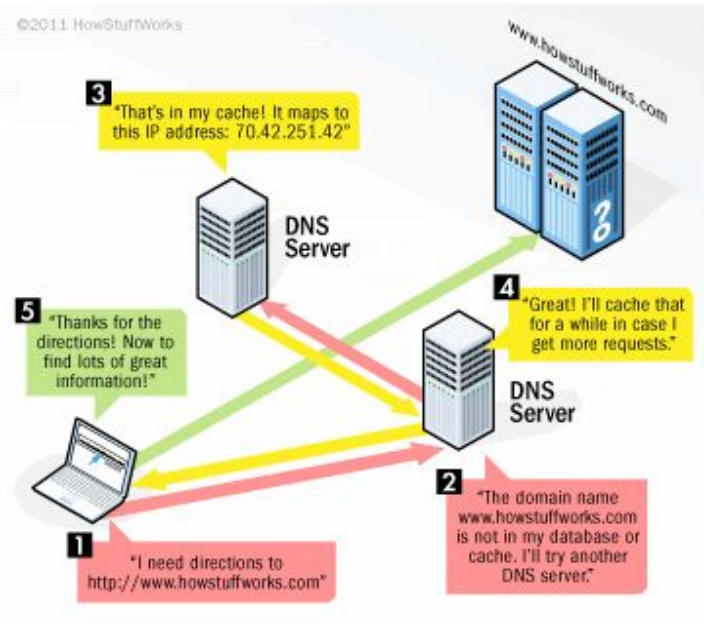
- When outlining goals at the start of the interview, the CAP theorem can be mentioned to discuss the high level guarantee the system is aiming for.
- Database choices are influenced by your CAP choices, so you can touch on this while going over your database. For example, RDBMS databases with synchronized replication are typically used for CP whereas NoSQL can be a good choice for AP when you prioritize high scaling and availability over consistency.
- Almost any problem that touches scalability is relevant for discussing the CAP theorem

++

- In practice, consistency and availability is a spectrum. You can have weak consistency (memcached, key-value stores like DynamoDB) or eventual availability (DNS, email) systems, whereas services like Zookeeper (service coordination) will gear towards strong consistency.
- High availability systems are supported via fail-over (active-passive, active-active) and/or replication (master-slave, master-master).
- SLA (Service Level Agreement) is used in modern systems to quantify and communicate availability guarantees (e.g. Google Map SLA)
- Further reading: 1, 2, 3

Domain Name Service (DNS)

What is it?



(*Image source*)

A DNS maps a domain name (e.g. www.google.com) to an IP address. DNS lookup is hierarchical with your router or ISP (and even your browser & OS) caching common domains for fast lookup at the lower level, leading upto root name servers at the top level.

Cached DNS results typically have an invalidation mechanism (e.g. TTL) so that they can be updated.

Trade-offs, caveats and alternatives

- DNS is a critical part of routing any web traffic that begins with a domain name. This creates a single point of failure where DDOS attacks are commonly targeted towards.
- DNS lookup adds a round trip delay although this is typically negligible given caching at local levels (browser, OS, router).

Use it for…

- Since a DNS is the backbone of every modern web service, it's generally good to mention it when drawing your overall architecture. You won't necessarily need to go into the details unless otherwise prompted by your interviewer.

++

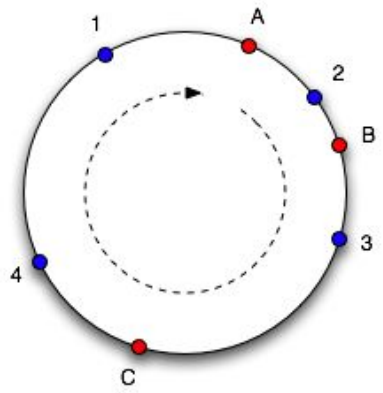
- For multiple web server IP addresses, the DNS server can route traffic to the appropriate IP using (weighted) round-robin, geographic proximity or some other criteria.
- Services like Cloudflare DNS and Route 53 provide DNS registration with additional features like security protection (DDOS, spoofing), monitoring and access control.
- Common DNS record types: NS (nameserver: The DNS server I have to look up to get details of my domain), A (address: The IP address for my domain), CNAME (canonical: alias that points to another domain. e.g. google.com → www.google.com), MX (mail exchange: Points to a mail server)
- Further reading: 1, 2

Consistent Hashing

What is it?

Hashing is the mapping of an unbounded sequence of characters (e.g. a string) to a fixed-sized value (i.e. the hash). Hashing is used in cryptography (obfuscate the original message), database indexing (fast lookup of string by mapping input to a fixed-size key) and routing (consistent hashing for node selection).

Consistent hashing is a technique used to route a request to a machine with the objective of 1) sending a specific request to a certain machine (e.g. sharding) while 2) it being tolerant to node failures. A common use case is when caching data across multiple servers. Where should one route the data when a server is down?



Consistent hashing accomplishes this by hashing both the server and the data separately and mapping their keys on an imaginary circular ring. In the image on the left, imagine the blue circles (1, 2, 3, 4) are the data being cached, and the red circles (A, B, C) are the machines, in which the data will be cached. The algorithm selects the first machine to the right of the keyed data position (2 goes to B). If a node goes down (B), the data is simply sent to the next machine (C). If a new node is set up, it is placed on the hash ring according to its hash value and immediately starts serving data requests.

(Image source)

Trade-offs, caveats and alternatives

- When sharding data that can't afford a miss (e.g. non-cache), consistent hashing has to be paired with replication which adds additional complexity for communication.
- Hash collision is less of an issue when routing requests but is problematic for encryption and can be a source of attack even for popular hash functions like MD5, SHA-1

Use it for…

- Consistent hashing can be used when discussing how to shard data when it no longer scales to store all the data in one node, but you still want it to be tolerant to machine failures.
- Routing requests for load balancing purposes can also use consistent hashing where the machine selection is dependent on the request or the processing of a request is stateful (i.e., subsequent requests need to go to the same server).

++

- Popular storage systems (e.g. DynamoDB, Cassandra, memcached) use consistent hashing to distribute the data
- Other hashing methods (Jump Hash, Multi-probe CH) further improves on CH.
- Further reading: 1, 2, 3, 4

Load Balancing

What is it?

Load balancing is used to distribute incoming traffic to multiple backend servers. It ranges from simple round robin algorithms to using knowledge such as server response times, least connections, or cookies to route the traffic to a backend node. There are generally 2 types of load balancers and Layer 4 (TCP-level) and Layer 7 (Application-level) load balancers.

- Layer 4: Routes traffic at individual packet level and does not inspect the traffic beyond relying on the IP address and TCP/UDP ports. It applies a simple round robin algorithm or information not specific to the request (e.g. server response time) to route the traffic. It is generally fast as it does not inspect the traffic
- Layer 7: Routes traffic by inspecting requests, such as the HTTP header, information of the request type (video, text), cookie (to route to the same server that handled the initial traffic). Is able to inspect specifics of the request to make a more informed decision on node selection at the cost of additional time/complexity required to route each traffic.

Trade-offs, caveats and alternatives

- Load balancers are used as a way to achieve scaling horizontally (add more machines) as opposed to vertically (increase power of machines). Horizontal scaling is cheaper to scale but adds complexity of maintaining such additional services to maintain and orchestrate multiple nodes.
- The load balancer itself can act as a single point of failure if there is only one load balancer. Adding additional load balancers adds complexity and can be managed by either routing at the DNS level or utilizing floating IPs.

Use it for…

- When you discuss scaling by cloning instances, you will have to start considering adding a load balancer. Be aware of the trade-offs of adding a load balancer so try to understand which service endpoints will have the most traffic requests before scaling horizontally and adding load balancers.

++

- Load balancers can have additional purposes beyond its core functionality of distributing incoming requests. It can act as a reverse proxy by adding caching and SSL termination.
- Load balancing can be done using dedicated hardware with specialized OS. This was common for Layer 4 load balancers, prior to commodity hardware being as powerful as they are today. Today it's common to use specialized software (e.g. HAProxy, NGINX) that make it easy to set up a load balancer. Using software-based load balancers has the benefit of it being flexible to configure and cheaper than hardware-based load balancers at the cost of lower performance.
- Further reading: 1, 2, 3

Reverse Proxy

What is it?

A reverse proxy acts as an intermediate interface between the public user and web servers. All requests targeted for the service initially hit the reverse proxy which then forwards the requests to the server to handle. Responses are forwarded back to the reverse proxy which then sends it back to the client. There are various benefits of adding this additional layer.

- Security: Backend web server IPs and configurations are hidden from the public making it more difficult for targeted attacks like DDOS.
- Flexibility: Configuration of backend services are hidden making it easy to reconfigure and scale as necessary without affecting the interface with public clients.
- Caching and static content: Responses can be cached for common requests. Static HTML content can also be directly served from the reverse proxy.
- SSL encryption/decryption: Can offload SSL termination from backend servers.
- Load balancing: Reverse proxies can be used as a load balancer when interfacing with multiple service clones. However, it doesn't necessarily have to act as a load balancer for it to be useful (as shown in the features above).

Trade-offs, caveats and alternatives

- Adding a reverse proxy adds additional complexity of maintaining another service. Breaking the handling of static vs dynamic content into separate services also adds complexity internally.
- The proxy now is a single point of failure which can cause outages without proper failover set in place. Malicious attacks that gain access to the reverse proxy will expose information of backend services.
- Features such as caching, compression, SSL termination can be handled by web servers themselves until the complexity and scale of those services start requiring a dedicated service to handle such operations.

Use it for…

- A reverse proxy can be useful when the web service has complicated functionality and you want to offload other operations such as caching, SSL termination in a separate service.
- If your system requires limiting exposure to your backend services, either for protection or for ease of reconfiguration.

++

- The difference between a forward proxy and a reverse proxy is that a forward proxy is used as an interface for the client and obfuscates details about the client whereas a reverse proxy acts as an interface for the backend server and hides details about the server to the client.
- Further reading: 1, 2

Content Delivery Network (CDN)

What is it?

A CDN is a type of caching mechanism to host commonly accessed static files (images, videos, JS/HTML) geographically closer to the users through a globally distributed network of servers. Backend services will maintain a URL to the static content and return this pointer to the client. The client then independently connects to the CDN host using this URL to access the content. There are two mechanisms in which your service can maintain the freshness of your content with the CDN

- Pull: When a user requests content from your servers, it is then placed to a CDN and the corresponding URL for that file is updated to point to the CDN host. A TTL is placed to invalidate the cache and the next request would be served directly from the server. Good for heavy traffic across a wide variety of content as it minimizes the amount of content required on the CDN
- Push: Content is pushed to the CDN when content changes, and the service owner (you!) is responsible for uploading it to the CDN and changing the URLs accordingly. Good for periodically updated services (e.g. blogs)

Trade-offs, caveats and alternatives

- Cost and complexity of updating and maintaining pointers to content served on CDNs can be high.
- The perceived delay of the content being downloaded might not be that significant compared to what it would be should it be downloaded directly from your servers. It's always good to check the tradeoffs of adding this engineering complexity vs the enhanced product experience by monitoring the load times across different environments.

Use it for…

- Services that require high bandwidth (e.g. video streaming, high-res photos) would greatly benefit from a locally accessible CDN.
- It also acts as a dedicated service to handle hosting of static content that decouples your main servers from its core business logic, so even though your service doesn't require high bandwidth a CDN could be useful.
- If your users' behavior is sensitive to website load time (e.g. monitoring suggests conversion drops on regions with slower network connectivity) consider adding a CDN.

++

- High bandwidth services (e.g. Netflix) require significant resources on CDNs. They have their own dedicated CDN service partnering directly with ISPs to deliver their video content as fast as possible across all global users.
- There are CDNs dedicated to serving images as image hosting is one of the core usages of CDNs. The most popular providers (Cloudflare, Akamai, Fastly) all have solutions that optimize service of images via compression, dynamic sizing, format support (e.g. WebP) and end-device adaptation.
- Further reading: 1, 2

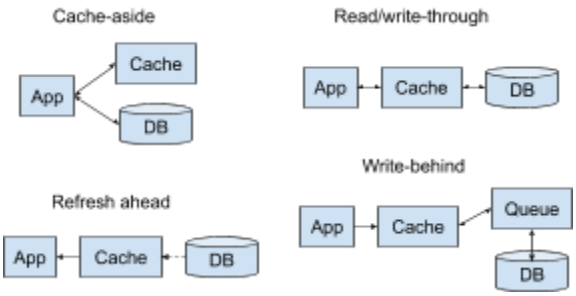
Caching

What is it?

A cache is an intermediary data storage that reduces delay upon a request and balances load on backend services by inspecting the cache before hitting the backend servers. Caching happens at the client level (OS, browser) or server side. On the server side there are:

- Reverse-proxy: Reverse proxies can act as a cache where common responses are cached and returned without hitting backend services.
- CDN: CDNs are a type of cache where frequent static files are stored and served.
- Database: Databases have caching options available to save disk lookup.
- Remote cache: Dedicated servers with memory-based key values stores (Redis, memcached) save load on database servers by caching frequently accessed data.

Methods of updating the cache are:



Cache-aside: Application is responsible for checking the set-aside cache before reading DB
Read/write-through: Application only interacts with the cache and the cache is responsible for synchronously updating the DB in the back.
Write-behind: The cache does the write to DB asynchronously saving time for the user
Refresh-ahead: The cache automatically fetches the latest data from the DB before cache expires.

Trade-offs, caveats and alternatives

- Understand where the traffic load is prior to adding the cache. Optimization can be done at various stages of the request (reverse proxy prior to hitting the servers, application level, database level, etc.) and varies depending on the data you want to cache.
- Synchronization of cache and the database is important, and various cache invalidation mechanisms exist to determine when to evict/refresh the data.

Use it for…

- Use key-value based memory cache for storing essential metadata that is frequently assessed (e.g. tweet ids of a person's timeline)
- For BLOBs that require high bandwidth and faster load times, leverage CDNs

++

- Caching can be done at the query level or the object level. Query level caches the request to the database and stores the results. Downside is that it can be difficult for fuzzy matching (slight change in query that results in same results) and one change in a DB field will require invalidating all relevant caches. Object level cache stores the response as a whole. User sessions, rendered web pages are such examples.
- Redis has options to store data structures (lists, sets) that can be helpful beyond storing simple key-value pairs.
- Further reading: 1, 2

Relational Database Management System (RDBMS)

What is it?

A database with a schema that guides the relationship across tables. This predefined schema allows one to use SQL like queries to quickly lookup and join data that might span across multiple tables.

Scaling of RDBMS can be done using one of the following approaches

- Replication: Master-slave replication uses one master to handle read/write whereas the other slave clones are used for read-only. Requires logic to promote slave to master if master goes down. Master-master replication is where all replicas handle both read & write. Requires logic to load balance write requests across instances and handle write race conditions.
- Federation: Splits up database tables by function (e.g. user, groups, posts) such that each DB server receives a portion of the traffic. Joining the data (e.g. all posts from a user from a specific group) now requires handling from the application side.
- Sharding: Hashes the request (e.g. user_id) and deterministically forwards the data to a specific database instance. Joining results from multiple shards can be more complex. Consistent hashing can be used to be tolerant to DB shard failures.

Trade-offs, caveats and alternatives

- Replication adds complexity of synchronizing the data at the cost of fault tolerance. The traffic load on a replica is a function of the number of replicas and write traffic. Additional logic is required to handle race conditions and failures during replication.
- Federation & sharding downsides are that it puts the load and complexity of joining the data from the database servers to the application servers

Use it for…

- RDBMS are good when your application requires complicated queries or you can't foresee what type of queries will be required in the future. SQL provides a battle-tested and flexible way to query your data. NoSQL alternatives require complicated data manipulation at the application level which can get difficult to maintain over time.
- Remember to mention how you would scale (replication, federation/sharding) if required.

++

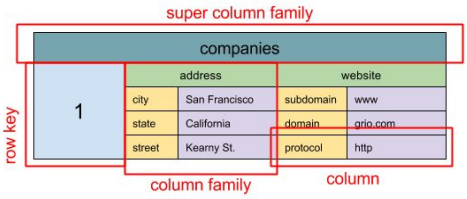
- ACID is a set of properties for database transactions: Atomicity (a transaction fails or succeeds, not in between), Consistency (a transaction brings the database to one valid state to another), Isolation (Concurrent operations results as if operations were executed sequentially), Durability (Data will persist even in the case of machine failure).
- Denormalization "flattens" the data by replicating fields to avoid expensive joins during read time. It trades off write compute and storage at the expense of fast reads.
- SQL optimization are methods to fine tune your SQL queries, tighten up your schema fields and better table partitioning. It's important to monitor and profile your traffic to identify slow queries.
- Further reading: 1, 2

NoSQL

What is it?

NoSQL denormalizes the data and stores them without a strict underlying schema. It generally lacks strong ACID guarantees and optimizes for availability over consistency as the data scales. A few popular NoSQL databases are:

- Key-value store: Simple stores backed by memory (Redis, memcached). Guarantees high availability and fast lookup compared to disk-based storage systems.
- Document store: Stores XML/JSON type data structures and provides a flexible way of querying them using simple SQL-like queries (MongoDB, Firestore). More complex joins require handling on the application side.
- Graph database: Stores a node and it's relationships as arcs of the node. Beneficial for storing data with intertwined relationships (e.g. social networks)
- Wide-column store: Inspired by Google's Bigtable, wide column stores can be viewed as two-dimensional key-value stores, with the basic representation being a "column" as a key-value pair, and a "column family" being a group of columns. The flexibility of representing complex relationships while still being able to scale makes it a popular choice (HBase, Cassandra)



(*Image source*)

Trade-offs, caveats and alternatives

- NoSQL makes it easy to scale and flexible to change, but comes with drawbacks such as follows:
  - As your application complexity grows and you require more joining of data across data stores your application code complexity will increase, whereas SQL is a very efficient way to query complex joins that is not tied to any one application.
  - Lookups are not as fast as indexed cells in RDBMS.
  - They don't guarantee strong consistency which might be crucial depending on your application (e.g. financial transactions)

Use it for…

- If you have a system that has a few of the following characteristics, a NoSQL database can be a good option: Large scale (PB of data), Flexible schema, No complex joins required, intensive workload (high read traffic).

++

- Operates on BASE (Basically available, Soft state, eventually consistent) property unlike ACID for RDBMS
- The ease of use to developers using familiar format (JSON) and simple schema is a big reason for NoSQL popularity. Being able to scale up storage without hiring a specialized DBA is an advantage for organizations.
- Further reading: 1, 2, 3, 4

Messaging

What is it?

Messaging is a form of event-driven architecture where messages are passed across services asynchronously using queues. This provides a flexible interface for services to communicate without forcing explicit contracts across services. The asynchronous nature also allows for services to respond back without waiting for subsequent services to finish (e.g. email delivery). Separating the business logic that requires offline processing also modularizes the system so that it's easier to scale each component independently. There are two forms of messaging:

- Point-to-point: There is one explicit connection between services where a middle layer queue acts as an intermediary buffer while events get processed.
- Publish-subscribe: There are multiple consumers of the message, and each subscriber "subscribes" to a topic. Once an event of that topic is produced the subscribers are notified and messages are pushed.

Trade-offs, caveats and alternatives

- Queues can be overloaded with events if not processed in time (consumer machines are overloaded or down). Backpressure techniques are applied to prevent the queue from being overloaded and the overall throughput remains consistent.
- Messaging requires additional services that act as brokers with additional logic to acknowledge the acceptance of a message. It also adds delay for the final job (on the consumer) to finish. Simple tasks can be better off using synchronous communication protocols (REST, RPC, etc.)
- There are other methods that facilitate asynchronous communication across services without involving an explicit message broker such as webhooks, websockets (if an explicit open communication channel is possible), SSE.

Use it for…

- Whenever there is a process that requires time that you don't want to incur to your client (e.g. respond back immediately), leveraging a queue to handle the request asynchronously can be beneficial.
- Depending on the application, it can be helpful to discuss how you would return a response back to the user after the consumer has worked on the task. (e.g. a confirmation email for a bug report). Essentially the response back to the user can be split into tiers (immediate response followed by final response).

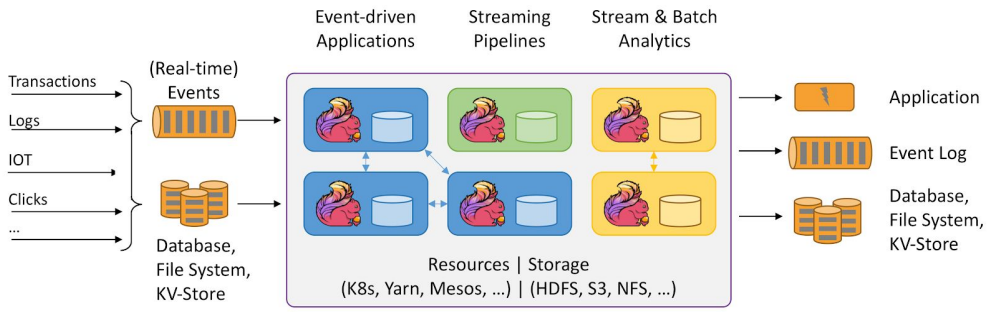
++

- Protocols such as AMQP, STOMP, MQTT exist to define contracts between services and are available on popular messaging services such as RabbitMQ, while other services such as Kafka, AmazonSQS are not restricted by such protocols.
- Depending on your need, a simple Redis List might suffice as a highly available message broker without incurring complexity. If you need more complex features (e.g. message receipt, flexible routing) a proper message broker would be needed.
- Further reading: 1, 2, 3

Streaming

What is it?

Stream processing can be seen as an evolution of messaging based event-driven architectures, where multiple events are processed by multiple consumers across an infinite time horizon.



(Image source)

Unlike messaging queues, streaming processors can store the data allowing consumers to pull data from the history and do complex operations across different streams and time frames. Streaming can also be seen as batch-processing with a smaller time window and continuous processing. Frameworks such as Spark and Flink can be used for both batch and stream processing.

Batch processing evolved from frameworks like MapReduce and Hadoop where it was primarily used for offline data processing/analytics, but the line between batch processing frameworks and streaming processors have merged with on-the-fly mini-batch processing.

Trade-offs, caveats and alternatives

- One should consider the tradeoff of adding a real time stream processing system compared with offline batch processing. Offline batch processing allows one to process data at larger scales with complex operations so for detailed analytics streaming might not be appropriate. Whereas for monitoring and adding alerts, a streaming pipeline can be appropriate.

Use it for…

- Stream processing can be used to handle real time processing of data either for analytical purposes (live monitoring system) or asynchronous processing across microservices.
- Topics such as real-time processing of financial transactions, tracking logistic operations, analyzing sensor data from IoT devices in real time are good use cases of using a stream processing architecture to handle incoming events.

++

- Further reading: 1, 2

Communication Protocols

What is it?

Communication protocols are used for two nodes to communicate across each other. For generic system design interview purposes, there are a couple of protocols across the OSI 7 layer model that are worth understanding.

- HTTP (Hyper-text Transfer Protocol): Backbone of the WWW where clients can send requests to fetch data (HTML, JS, images, etc.) from a server using a specialized set of request methods (GET, POST, PUT, DELETE, etc.), and the server can respond back with the data and a response code. This is an application layer (Layer 7) protocol.
- TCP (Transmission Control Protocol): Connection-oriented layer 4 protocol that establishes and terminates connections between nodes via handshakes. It optimizes for maintaining a stable connection via ack messages, checksums, flow/congestion controls to guarantee order of packet delivery (first sent first arrive).
- UDP (User Datagram Protocol): UDP is a layer 4 protocol that optimizes for speed of delivery over consistency. It doesn't have any congestion control and does not guarantee delivery of all datagrams (packets).

Trade-offs, caveats and alternatives

- TCP vs UDP: TCP is optimal for services that require guarantee of delivery (web services), whereas UDP is optimal where some data loss is acceptable at the cost of speed (video streaming, multiplayer games).
- UDP might require its own application specific error handling to deal with packet/datagram loss.

Use it for…

- UDP can be a consideration if the service you are building is sensitive to response time and latency. Discuss that you would need some error handling mechanism specific to your application.
- REST vs RPC can be a choice on how your internal services communicate with each other. RPCs can generally be a good choice for internal service communications with tight coupling whereas RESTful APIs over HTTP provide flexibility.

++

- REST is an architectural framework between a client and server (typically over HTTP), and has core principles such as requiring each request to be stateless and responses explicitly labeled for cacheability.
- RPC is an inter-process communication that enables execution of a process in another address space (typically a remote machine). This enables one to write code that can be executed on the same machine or a separate node as the project scales. RPCs are popularly used in companies like Google (Protobuf) and Facebook (Thrift).
- Further reading: 1, 2, 3, 4