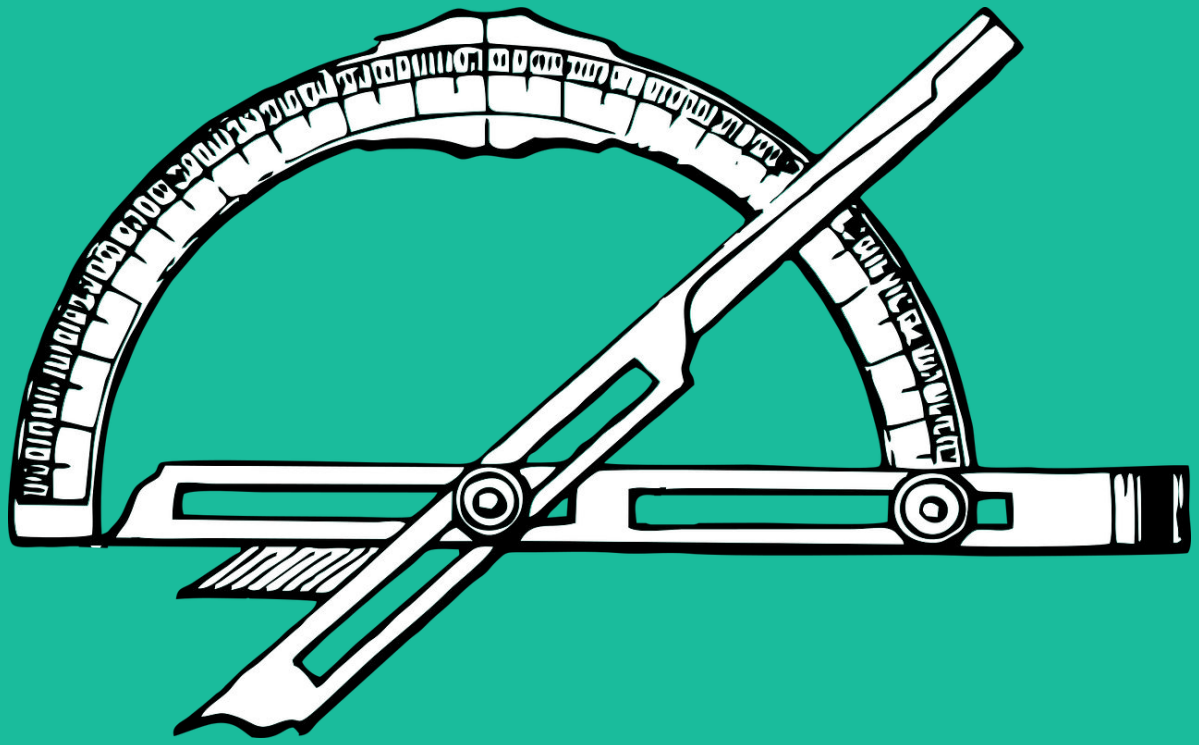
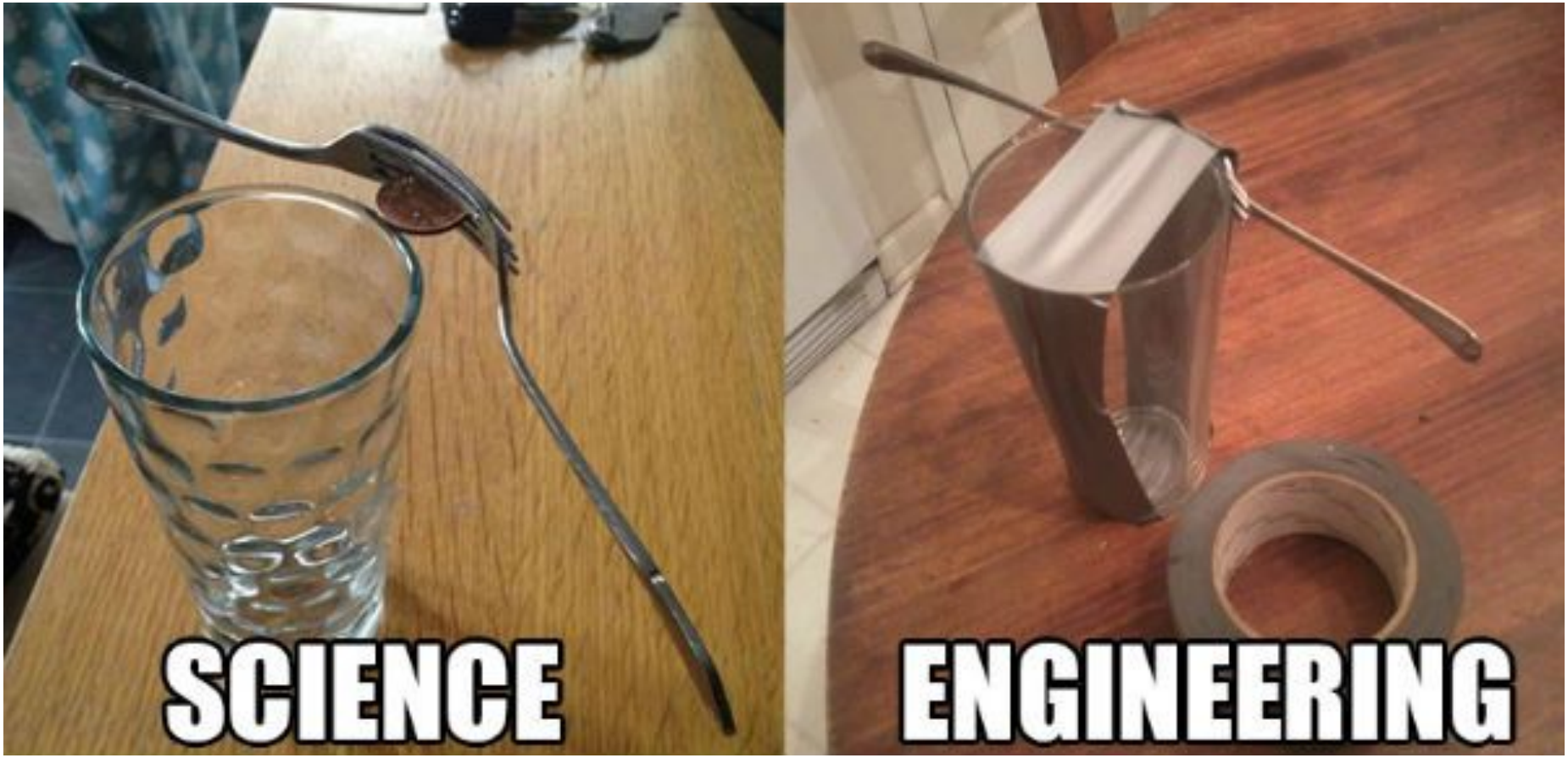


# Chapter 4. Designing Systems for Scale

Bilkent University | CS443 | 2020, Spring | Dr. Orçun Dayıbaş

# Introduction

- Everything in large-scale system design is **a trade-off**



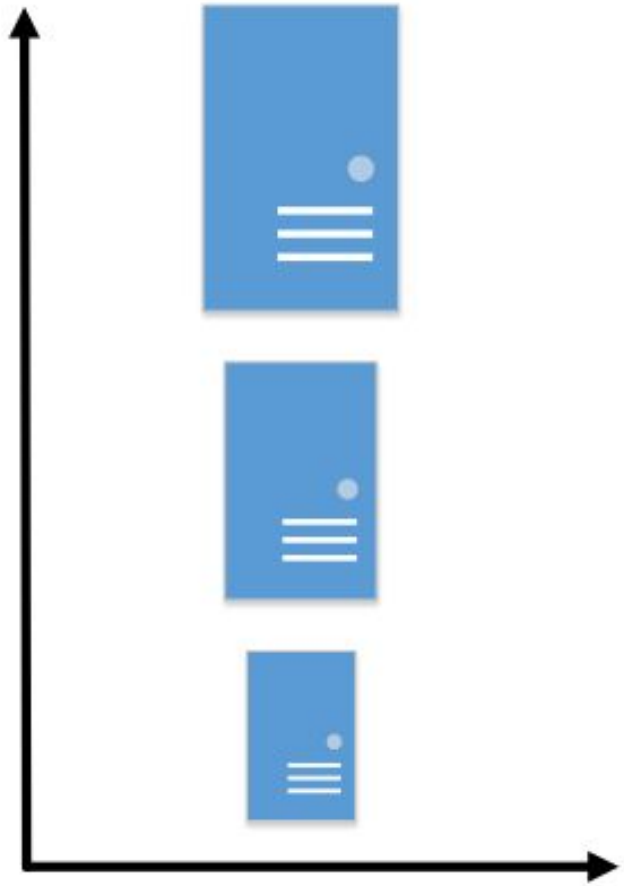

**Source:** WeKnowMemes

# Introduction

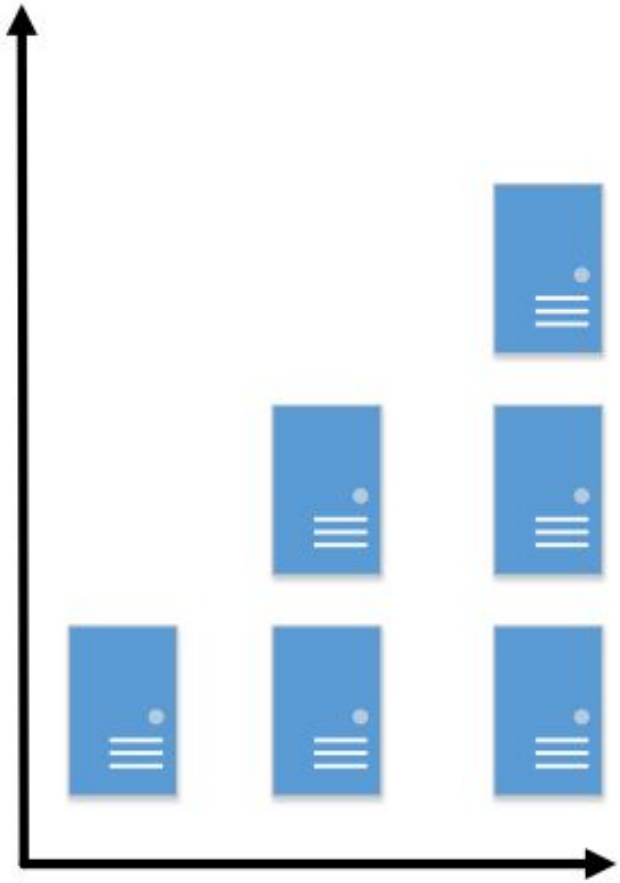
- **Performance vs. Scalability**
  - Performance problem: slow for a single user
  - Scalability problem: slow under heavy load
  - Do you really need a distributed one?
- **Latency vs. Throughput**
  - Latency: time to produce a result
  - Throughput: # of results/actions per unit of time
  - Maximize throughput with acceptable latency
- **Availability vs. Consistency**
  - Availability: Every request receives a response
  - Consistency: Every request receives the most recent data

# Scalability

- **Vertical vs. Horizontal**



scale up

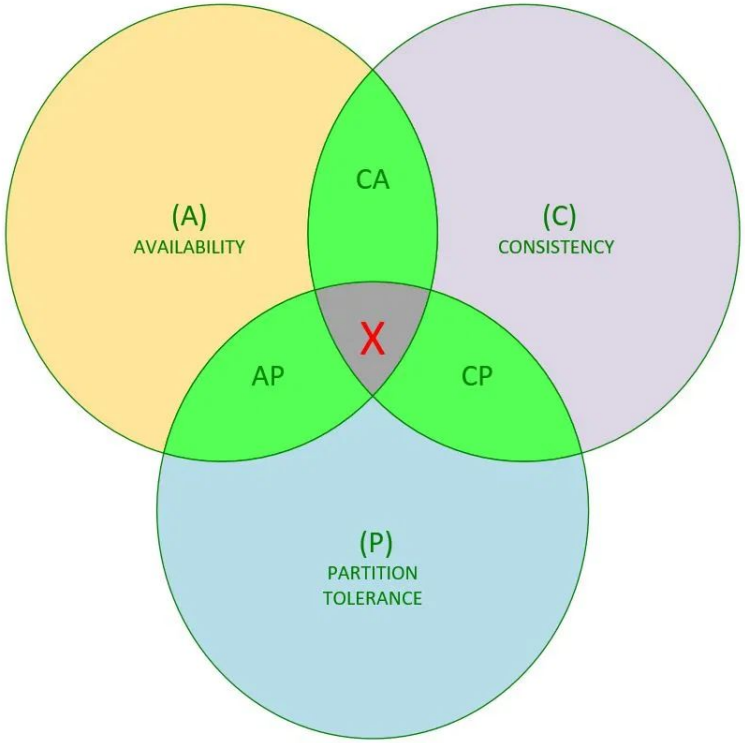


scale out

# CAP Theorem

- **Definition:** a distributed system can deliver only two of three desired characteristics: consistency, availability, and partition tolerance.

All clients can access data even in presence of failure

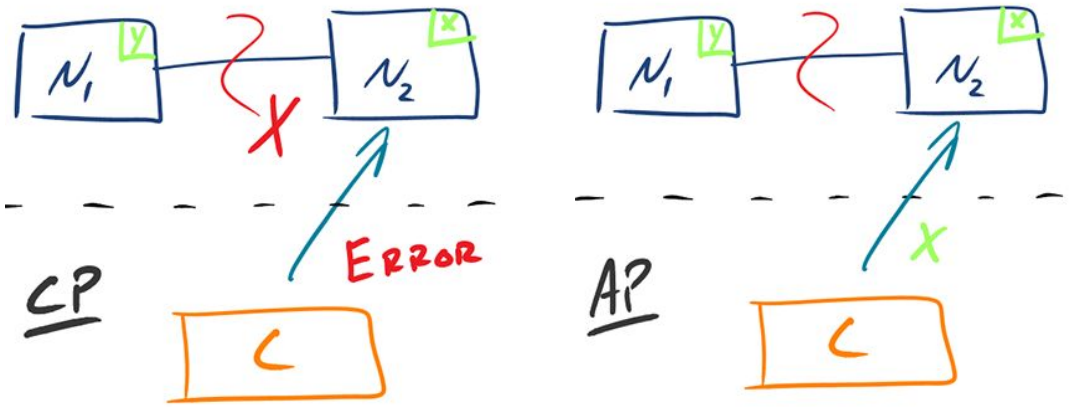



All clients can access same view, even presence of updates

The system property holds even if network partitioned

# CAP Theorem (Revisited)

- **Trade-off analysis for a distributed system**
  - Reality: network is not reliable [Link]


**Source:** http://robertgreiner.com/2014/08/cap-theorem-revisited/

# Consistency Patterns

- Weak Consistency
  - After a write, reads may or may not see it (**best effort**)
  - Works well in **real time** use cases (VoIP, Multiplayer games)
- Eventual Consistency
  - After a write, reads will eventually see it (Data is replicated **asynchronously**)
  - Works well in **highly available** systems (Email, DNS, etc.)
- Strong Consistency
  - After a write, reads will see it (Data is replicated **synchronously**)
  - Works well in systems that need **transactions** (File system, RDBMS, etc.)

# Availability Patterns

- The table of nines
  - Measure of availability (e.g. “a 4-nines solution”)

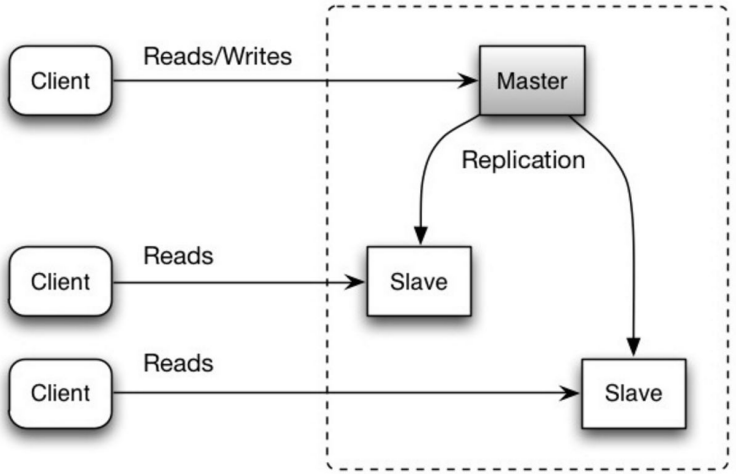
| Level | Percentage | Downtime per Year |
|---|---|---|
| 1 Nine | 90% | 36 days 12 hours |
| 2 Nines | 99% | 3 days 15 hours 36 minutes |
| 3 Nines | 99.9% | 8 hours 45 minutes |
| 4 Nines | 99.99% | 52 minutes 36 seconds |
| 5 Nines | 99.999% | 5 minutes 15 seconds |
| 6 Nines | 99.9999% | 32 seconds |

# Availability Patterns

- **Fail-over**
  - Active-Passive (Master-Slave)
    - Heartbeats are sent between the active and the passive server on standby
    - If the heartbeat is interrupted, the passive server takes over the active's IP address and resumes service
    - The length of downtime is determined by whether the passive server is already running in "hot" standby or not ("cold" one).
  - Active-Active (Master-Master)
    - Both servers are managing traffic, spreading the load between them.
    - If the servers are public-facing, the DNS would need to know about the public IPs of both servers. If the servers are internal-facing, application logic would need to know about both servers.
  - Cons
    - Fail-over adds more hardware and additional complexity.
    - There is a potential for loss of data

# Availability Patterns

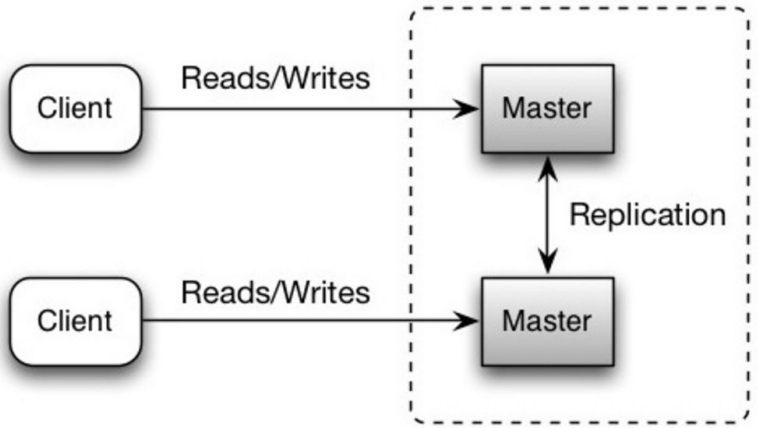
- Replication
  - Master-slave
    - The master serves reads and writes, replicating writes to one or more slaves, which serve only reads.
    - If the master goes offline, the system can continue to operate in read-only mode until a slave is promoted to a master.
    - Cons
      - Additional logic is needed to promote slave/master
      - Potential for loss of data if the master fails before any newly written replicated
      - The more read slaves, the more you have to replicate, which leads to greater replication lag
      - Replication adds more hardware and additional complexity




**Source:** http://www.slideshare.net/jboner/scalability-availability-stability-patterns/

# Availability Patterns

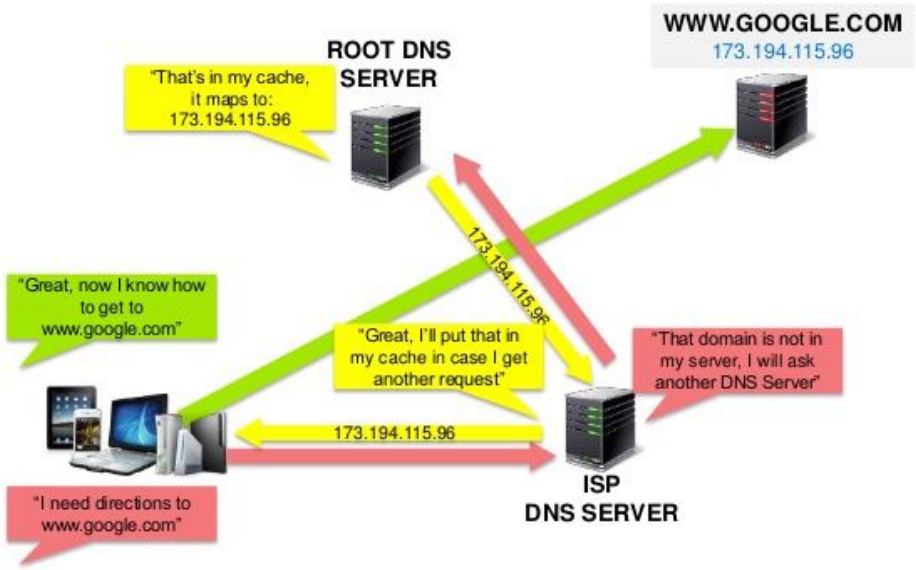
- Replication
  - Master-master
    - Both masters serve reads and writes and coordinate with each other on writes.
    - Cons
      - You'll need a load balancer or you'll need to make changes to your application logic to determine where to write
      - Potential for loss of data if the master fails before any newly written replicated
      - Most master-master systems are either weak consistent or have **increased write latency** due to synchronization: **Conflict resolution** comes more into play
      - Replication adds more hardware and additional complexity




**Source:** http://www.slideshare.net/jboner/scalability-availability-stability-patterns/

# DNS (Domain Name System)

- **How does DNS work?**
  - NS record (name server)
    - Specifies the DNS servers for your domain/subdomain.
  - MX record (mail exchange)
    - Specifies the mail servers for accepting messages.
  - A record (address)
    - Points a name to an IP address.
  - CNAME (canonical)
    - Points a name to another name or CNAME (example.com to www.example.com) or to an A record.




**Source:** https://www.slideshare.net/srikrupa5/dns-security-presentation-issa

# DNS (Domain Name System)

- **Cons**
  - Accessing a DNS server introduces a slight delay, although mitigated by caching.
  - DNS server management could be complex, although they are generally managed by governments, ISPs, and large companies.
  - DNS services have recently come under DDoS attack, preventing users from accessing websites without knowing the IP address(es).
- **DNS-SD**
  - DNS Service Discovery is a way of using standard DNS programming interfaces, servers, and packet formats to browse the network for services.

# CDN (Content Delivery Network)

- Globally distributed network of proxy servers, serving content from locations closer to the user.
- The site's DNS resolution will tell clients which server to contact.
- Serving content from CDNs can significantly improve performance in two ways:
  - Users receive content at data centers close to them
  - Your servers do not have to serve requests that the CDN fulfills
- Two types: Push CDNs and Pull CDNs

# CDN (Content Delivery Network)

- **Push CDNs**
  - Receive new content whenever changes occur on your server
  - You take full responsibility for providing content, uploading directly to the CDN and rewriting URLs to point to the CDN
  - You can configure when content expires and when it is updated. Content is uploaded only when it is new or changed, **minimizing traffic, but maximizing storage**
  - Sites with a **small amount of traffic** or sites with more **static content** work well with push CDNs.
  - Content is placed on the CDNs once, instead of being re-pulled at regular intervals.

# CDN (Content Delivery Network)

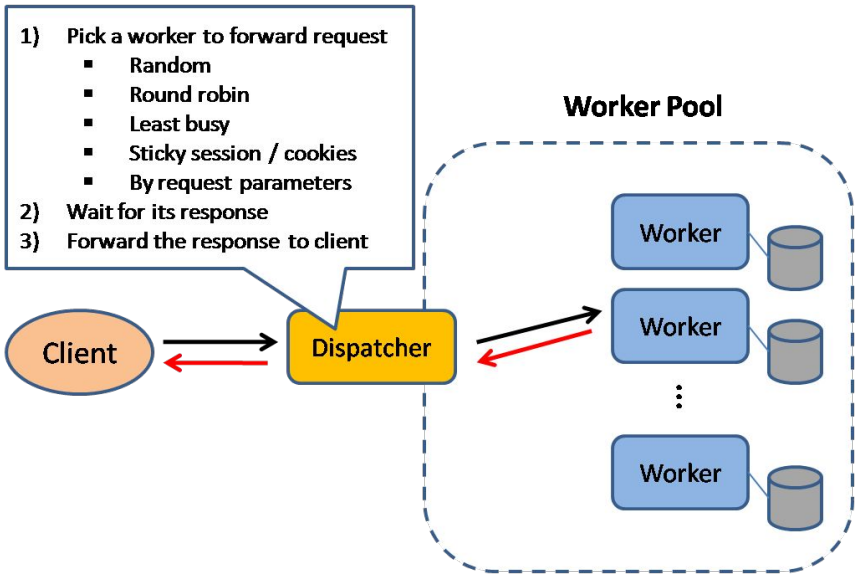
- **Pull CDNs**
  - Pull CDNs grab new content from your server when the first user requests the content
  - You leave the content on your server and rewrite URLs to point to the CDN. This results in a **slower request until the content is cached** on the CDN
  - A time-to-live (TTL) determines how long content is cached.
  - Pull CDNs **minimize storage** space on the CDN, but can create **redundant traffic** if files expire and are pulled before they have actually changed.
  - Sites with **heavy traffic** work well with pull CDNs, as traffic is spread out more evenly with only recently-requested content remaining on the CDN.

# CDN (Content Delivery Network)

- **Pull CDNs**
  - Cons
    - CDN costs could be significant depending on traffic, although this should be weighted with additional costs you would incur not using a CDN.
    - Content might be stale if it is updated before the TTL expires it.
    - CDNs require changing URLs for static content to point to the CDN.

# Load Balancer

- Load balancers distribute incoming requests to computing resources
- Load balancers are effective at:
  - Preventing requests from going to unhealthy servers
  - Preventing overloading resources
  - Helping eliminate single points of failure



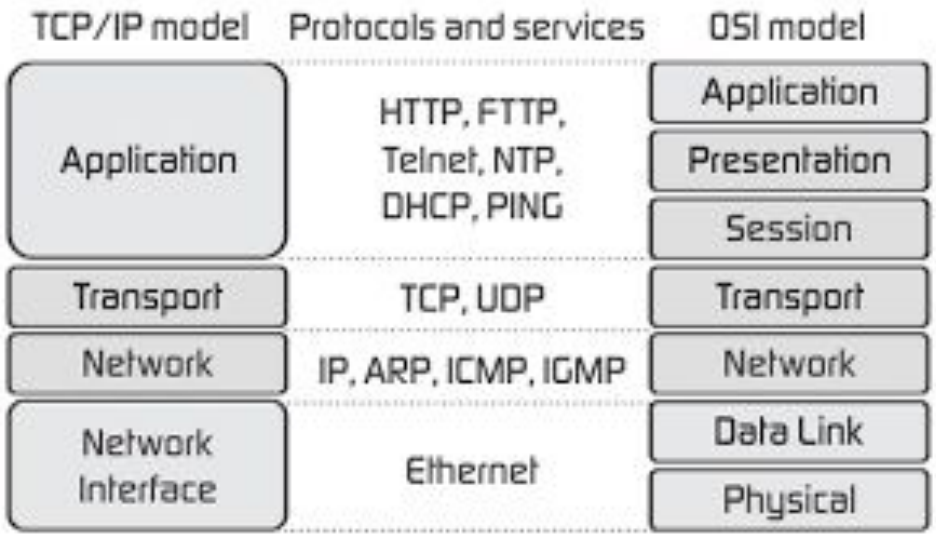

**Source:** http://horicky.blogspot.com.tr/2010/10/scalable-system-design-patterns.html

# Load Balancer

- LBs can be implemented with
  - Hardware (Cisco, F5, Citrix, Kemp, etc.)
  - Software (HAProxy, Traefik, Envoy, Kemp, LVS, etc.)
- Additional benefits
  - SSL termination
  - Session persistence
- It's common to replicate LBs (Active-active, Active-passive)
- LBs can route traffic based on: Random, Least loaded, Session/cookies, Round robin, Layer 4/7

# Load Balancer

- Layer 4 LBs
  - Forward packets to/from the upstream server performing NAT (Network Address Translation)
- Layer 7 LBs
  - Look at the application layer to decide
  - Header, message and cookies
- Horizontal scaling
- Ponder this
  - HW vs SW LB?
  - L7 vs L4 LB?

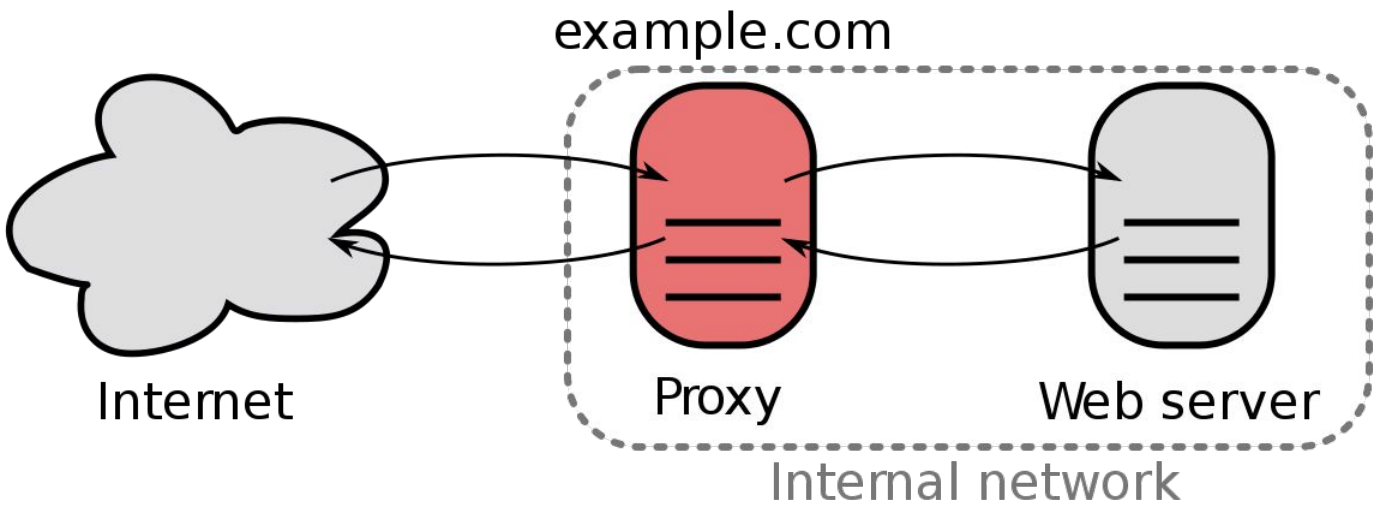


# Load Balancer

- **Cons**
  - The load balancer can become a performance **bottleneck** if it does not have enough resources or if it is not configured properly.
  - Introducing a load balancer to help eliminate single points of failure results in increased **complexity**.
  - A single load balancer is **a single point of failure**, configuring multiple load balancers further increases complexity.

# Reverse Proxy

- A web server that centralizes internal services and provides unified interfaces to the public.
- Requests from clients are forwarded to a server that can fulfill it before the reverse proxy returns the server's response to the client.




**Source:** Wikipedia

# Reverse Proxy

- **Additional Benefits**
  - Security (hides backend servers, limit conn./client, etc.)
  - Flexibility (clients only see the RP's IP)
  - SSL Termination
  - Compression/Caching
  - Serve static content directly
- **Cons**
  - Similar to LB (complexity, single point of failure, etc.)
- **Load Balancer vs. Reverse Proxy**
  - LB is useful when you have multiple servers
  - RP can be useful even with just one server

# Application Layer

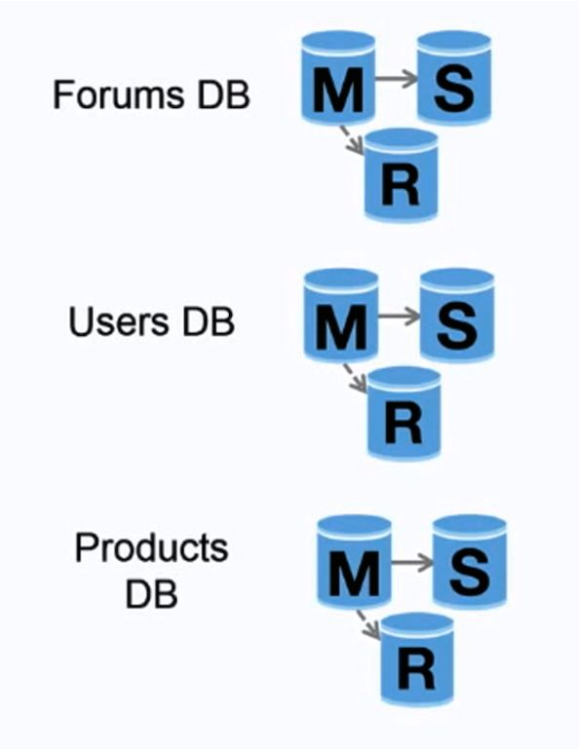
- Separating out the web layer from the application layer (also known as platform layer) allows you to scale and configure both layers independently
- Adding a new API results in adding application servers without necessarily adding additional web servers
- Microservices
  - Independently deployable, small, modular services
  - Modern interpretation of service-oriented architecture
  - Loose Coupling & High Cohesion
  - The Bounded Context (DDD)
  - (more on this later)

# Database

- Relational Database Management Systems (RDBMS)
  - Data items are organized in tables
  - ACID
    - Atomicity: Each transaction is all or nothing
    - Consistency: Trans. will bring the DB from one valid state to another
    - Isolation: Concurrent/serial trans. exec. produce same results
    - Durability: Once trans. has been committed, it will remain so
  - Techniques to scale a RDBMS
    - Replication (master-slave, master-master)
    - Federation
    - Sharding
    - Denormalization
    - SQL Tuning

# Database

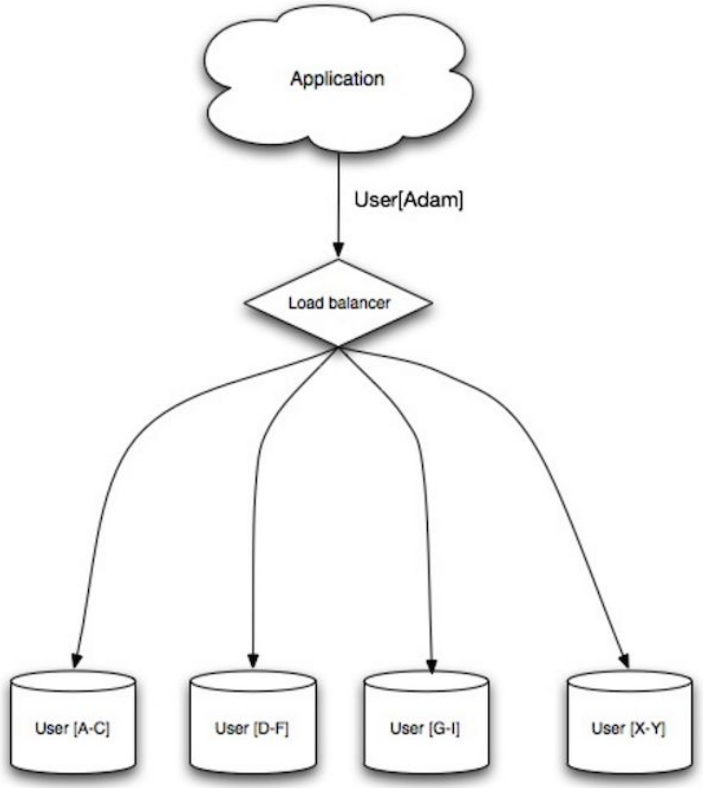
- **Federation**
  - Splits up database by function.
  - Less read/write traffic --> less replication lag
  - Smaller DB --> more cache hits (improved cache locality)
  - Cons
    - Federation is not effective if your schema requires huge tables
    - Requires to update the application logic
    - Joining data from two DBs is more complex
    - Hardware++, Complexity++




**Source:** https://www.youtube.com/watch?v=vg5onp8TU6Q

# Database

- **Sharding**
  - Distributes data across different databases such that each database can only manage a subset of the data
  - Less read/write traffic --> less replication lag
  - Smaller DB --> more cache hits (improved cache locality)
  - Cons
    - Data distribution can be lopsided
    - Requires to update the application logic
    - Joining data from shards is more complex
    - Hardware++, Complexity++




**Source:** http://www.slideshare.net/jboner/scalability-availability-stability-patterns/

# Database

- **Denormalization**
  - Attempts to improve read performance at the expense of some write performance
  - Some RDBMS support materialized views (storing redundant information)
  - In most systems, reads can heavily out number writes 100:1 or more
  - Cons
    - Data is dublicated
    - Stay in sync --> complexity++
- **SQL Tuning**
  - Very broad topic (CHAR vs VARCHAR <--> Table parti.)
  - Benchmark (Simulate high-load) & Profile (Track)

# Database

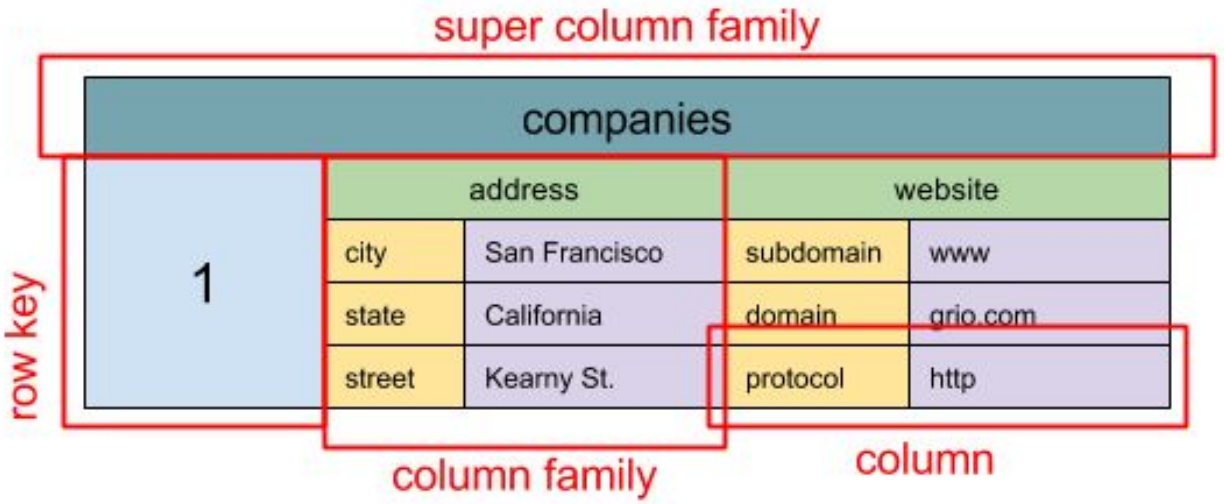
- **NoSQL**
  - Data items represented in a key-value store, document-store, wide column store, or a graph database
  - Data is denormalized, and joins are generally done in the application code. Most NoSQL stores lack true ACID transactions and favor eventual consistency
  - BASE
    - Basically Available: 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 doesn't receive input during that period

# Database

- **Key-value store**
  - Allows for O(1) reads and writes and is often backed by memory or SSD
  - KV stores provide high performance and are often used for simple data models or for rapidly-changing data
  - Offer limited operations: complexity shifted to app. layer
  - Ex: Redis, DynamoDB, Berkeley DB
- **Document store**
  - Centered around documents (XML, JSON, etc.)
  - Provides API/QL to query on the internal str. of the doc.
  - Document stores provide high flexibility and often used for working with occasionally changing data
  - Ex: MongoDB, DynamoDB,CouchDB, Elasticsearch

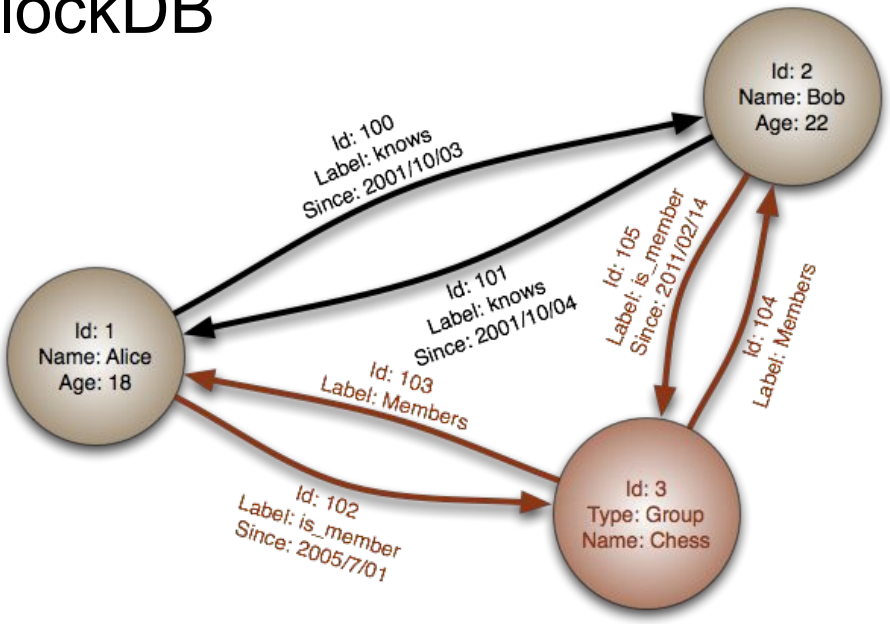
# Database

- **Wide column store**
  - The basic unit of data is a column that can be grouped in column families (roughly an SQL table) and super columns families
  - You can access each column independently with a row key, and columns with the same row key form a row
  - Ex: Cassandra, HBase, Accumulo



# Database

- **Graph**
  - Each node (vertex) is a record and each arc (edge) is a relationship between two nodes
  - Optimized to represent complex relationships with many foreign keys or many-to-many relationships
  - Ex: Neo4j, FlockDB



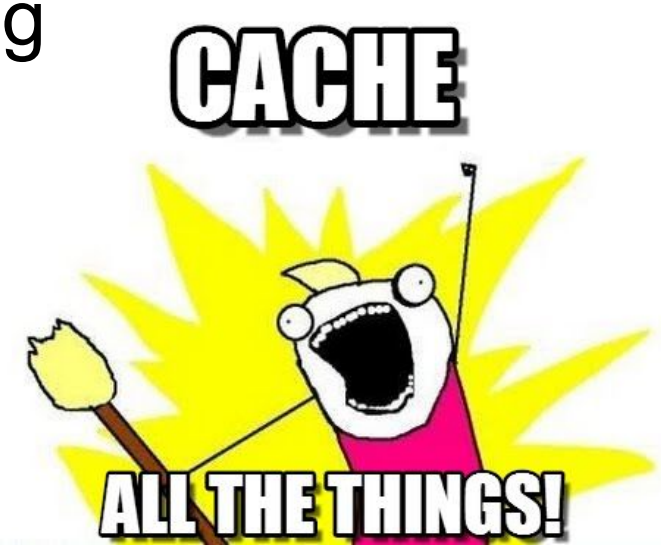

**Source:** Wikipedia

# Database

| SQL (RDBMS) | NoSQL |
|---|---|
| Rigid | Flexible |
| Relational | Object-oriented |
| Mature | Emerging |
| ACID (Sync) | BASE (Async) |
| Query Optimizer | Developer/Pattern |
| Complex Joins are OK | Handle them on App. level |
| Lookups by index are fast | High throughput for IOPS |
| Stable | Scalable |
| Structured | Semi-structured |

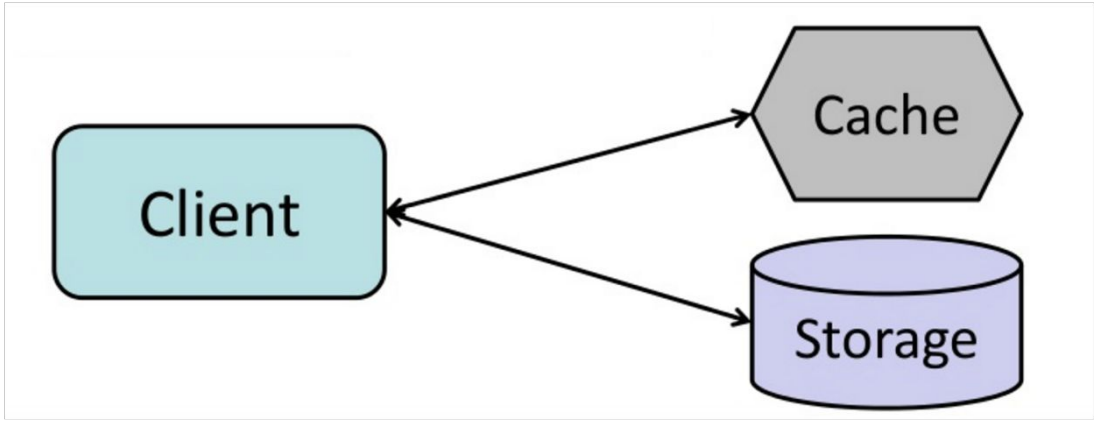
# Cache

- Caching improves load times and can reduce the load on the servers
- Putting a cache in front of a database can help absorb uneven loads and spikes in traffic
- Client caching (OS/Browser), CDN caching, Web server caching, Application caching (e.g. Redis, Memcached), Database caching
- Questions
  - What to cache?
  - When to update?



# Cache

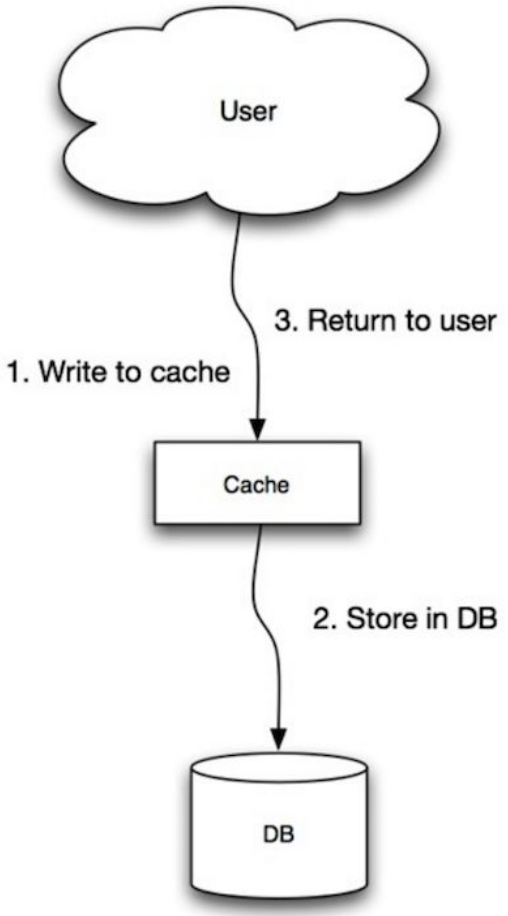
- **Cache update strategies**
  - **Cache-aside**
    - The application is responsible for reading and writing from storage
    - The cache does not interact with storage directly
    - Cons
      - Each cache miss results in three trips: delay
      - Data can become stale if it's updated in the storage (TTL?)




**Source:** https://www.slideshare.net/tmatyashovsky/from-cache-to-in-memory-data-grid-introduction-to-hazelcast

# Cache

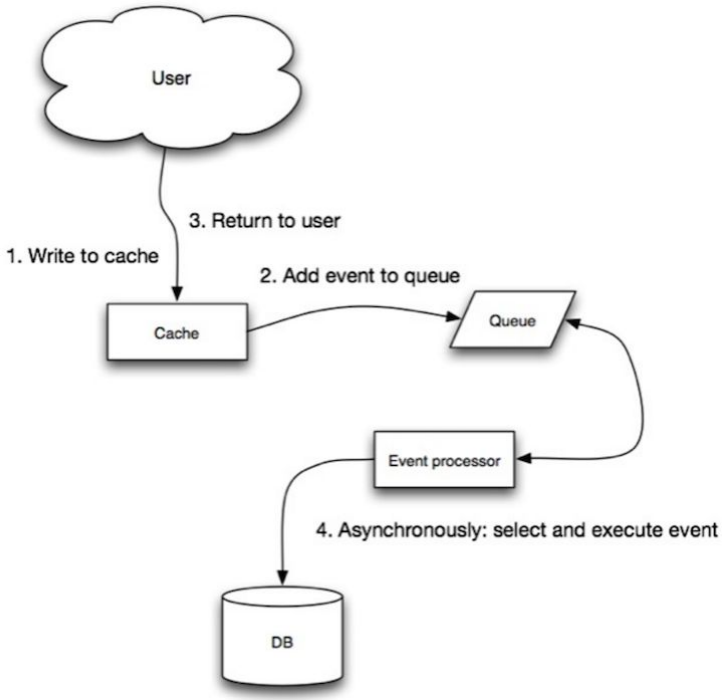
- **Cache update strategies**
  - **Write-through**
    - Cache is responsible for reading & writing to DB
    - Application adds/updates entry in cache and cache synchronously writes it to DB
    - One slow write for fast subsequent reads of  just written data
    - Cons
      - When a new node created for fail-over/scaling, the new node will not cache any entry until it updated in the DB
      - Most data written might never read
    - Cache-aside + Write-through?




**Source:** https://www.slideshare.net/jboner/scalability-availability-stability-patterns

# Cache

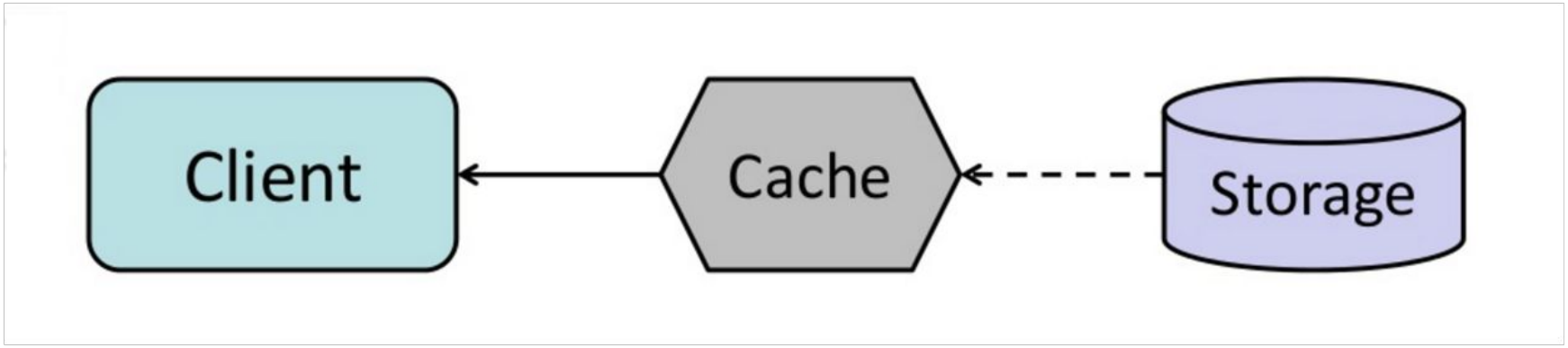
- **Cache update strategies**
  - **Write-back (Write-behind)**
    - Asynchronously write entry to the data store, improving write performance
    - Cons
      - There could be data loss if the cache goes down before DB write
      - More complex to implement than cache-aside or write-through




**Source:** https://www.slideshare.net/jboner/scalability-availability-stability-patterns

# Cache

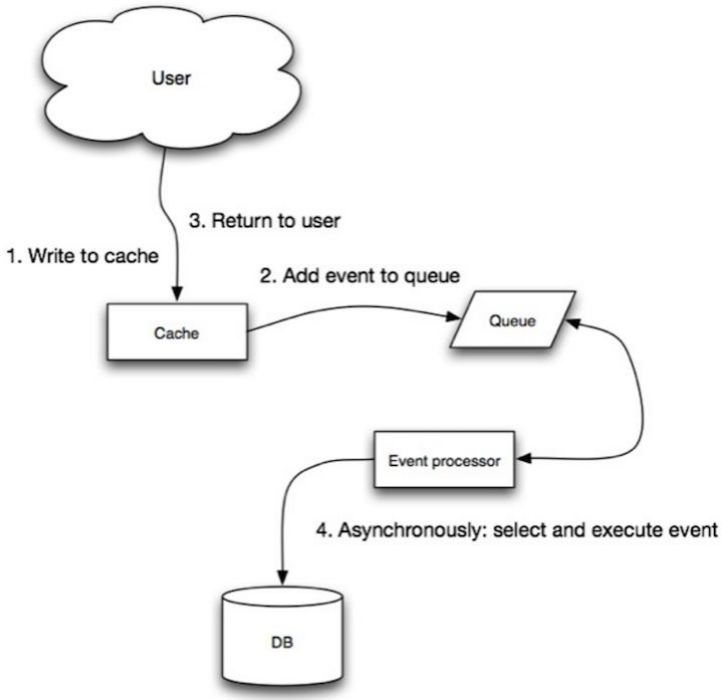
- **Cache update strategies**
  - **Refresh-ahead**
    - Cache automatically (async.) refresh any recently accessed cache entry prior to its expiration
    - Can reduce latency vs read-through if it predicts well
    - Cons
      - Not accurately predicting which items are likely to be needed in the future can result in reduced performance




**Source:** https://www.slideshare.net/tmatyashovsky/from-cache-to-in-memory-data-grid-introduction-to-hazelcast

# Cache

- **Cache update strategies**
  - **Write-back (Write-behind)**
    - Asynchronously write entry to the data store, improving write performance
    - Cons
      - There could be data loss if the cache goes down before DB write
      - More complex to implement than cache-aside or write-through



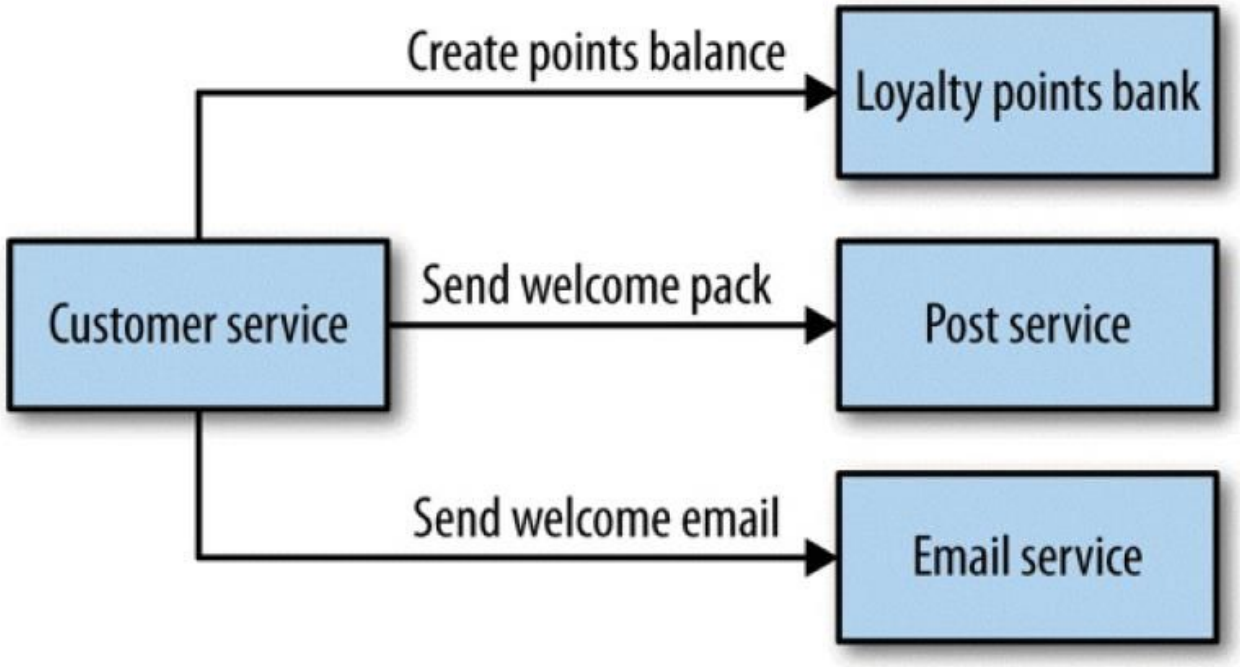

**Source:** https://www.slideshare.net/jboner/scalability-availability-stability-patterns

# Cache

- **Cache Types**
  - Local Cache (e.g. EhCache, Google Guava)
    - Pros: simple, no serialization overhead (performance)
    - Cons: Not a fault-tolerant, not scalable
  - Replicated Cache (e.g. Infinispan, Oracle Coherence)
    - Pros: best read performance, fault-tolerant
    - Cons: poor write performance, additional NW & memory load
  - Distributed Cache (e.g. Redis, Hazelcast, Terracota)
    - Pros: linear perf. scalability for R&W, fault-tolerant
    - Cons: increased latency of reads (NW round-trip & de-serialization)
  - Majority of replicated/distributed caches support two modes: Embedded, Client-server (location of the cache mng. instance)

# Asynchronism

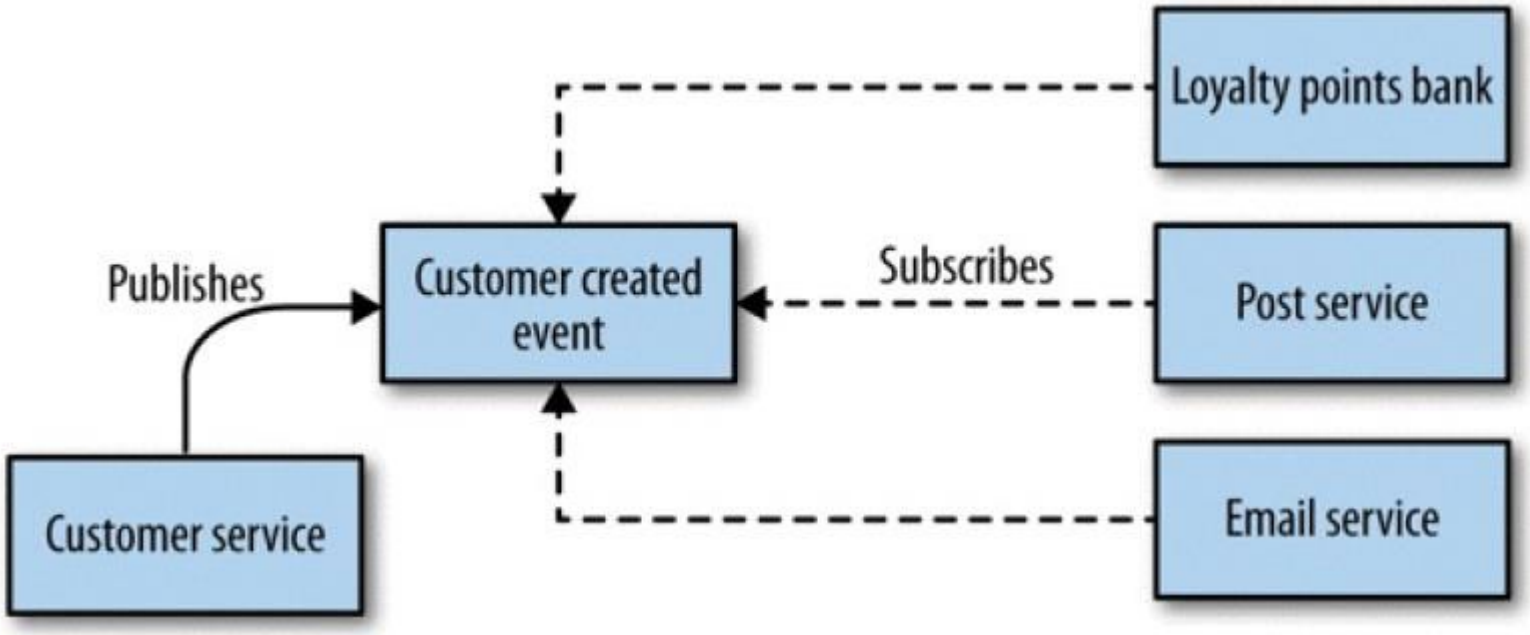
- Synchronous vs. Asynchronous communication
  - Request / response (syn. or quasi async.)
  - Event-based
- **Orchestration** vs. Choreography




**Source:** Building Microservices, Sam Newman / O'Reilly

# Asynchronism

- Synchronous vs. Asynchronous communication
  - Request / response (syn. or quasi async.)
  - Event-based
- Orchestration vs. **Choreography**




**Source:** Building Microservices, Sam Newman / O'Reilly

# Asynchronism

- Message Queue (MQ)
  - Receive, hold and deliver messages
  - Message broker + Protocol
  - Exp: ZeroMQ, RabbitMQ, Amazon SQS
- Task Queue
  - MQ + Scheduling
  - Exp: Celery, Huey
- Back pressure
  - A method to prevent the system from becoming completely overwhelmed and unavailable (by monitoring sys. res.)
  - Limiting the queue size, to maintain a high throughput rate and good response times for jobs already in the queue (Once the queue fills up, clients get a server busy)

# Communication

OSI (Open Source Interconnection) 7 Layer Model

| Layer | Application/Example | | Central Device/ Protocols | | | DOD4 Model |
|---|---|---|---|---|---|---|
| **Application (7)**<br>Serves as the window for users and application processes to access the network services. | **End User layer** Program that opens what was sent or creates what is to be sent<br>Resource sharing • Remote file access • Remote printer access • Directory services • Network management | | **User Applications**<br>SMTP | | GATEWAY<br>Can be used on all layers | Process |
| **Presentation (6)**<br>Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network. | **Syntax layer** encrypt & decrypt (if needed)<br>Character code translation • Data conversion • Data compression • Data encryption • **Character Set Translation** | | JPEG/ASCII<br>EBDIC/TIFF/GIF<br>PICT | | | |
| **Session (5)**<br>Allows session establishment between processes running on different stations. | **Synch & send to ports** (logical ports)<br>Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc. | | **Logical Ports**<br>RPC/SQL/NFS<br>NetBIOS names | | | |
| **Transport (4)**<br>Ensures that messages are delivered error-free, in sequence, and with no losses or duplications. | **TCP** Host to Host, Flow Control<br>Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing | PACKET FILTERING | TCP/SPX/UDP | | | Host to Host |
| **Network (3)**<br>Controls the operations of the subnet, deciding which physical path the data takes. | **Packets** ("letter", contains IP address)<br>Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting | | **Routers**<br>IP/IPX/ICMP | | | Internet |
| **Data Link (2)**<br>Provides error-free transfer of data frames from one node to another over the Physical layer. | **Frames** ("envelopes", contains MAC address)<br>[NIC card — Switch — NIC card] (end to end)<br>Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control | | **Switch Bridge WAP**<br>PPP/SLIP | Land Based Layers | | Network |
| **Physical (1)**<br>Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium. | **Physical structure** Cables, hubs, etc.<br>Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts | | **Hub** | | | |

**Source:** http://www.escotal.com/osilayer.html

# Communication

- TCP vs. UDP

| TCP | UDP |
|---|---|
| Slower but reliable | Fast but unreliable (best-effort) |
| Full-featured protocol | Simple |
| Connection-oriented | Data is sent without setup |
| Delivery of all data is managed | No retransmission (handle it at a higher level) |
| Segment sequencing | No sequencing |
| Overhead: 20 bytes | Overhead: 8 bytes |
| Web, email, file transfer | Video streaming, VoIP |

- Use UDP over TCP when late data is worse than lost data

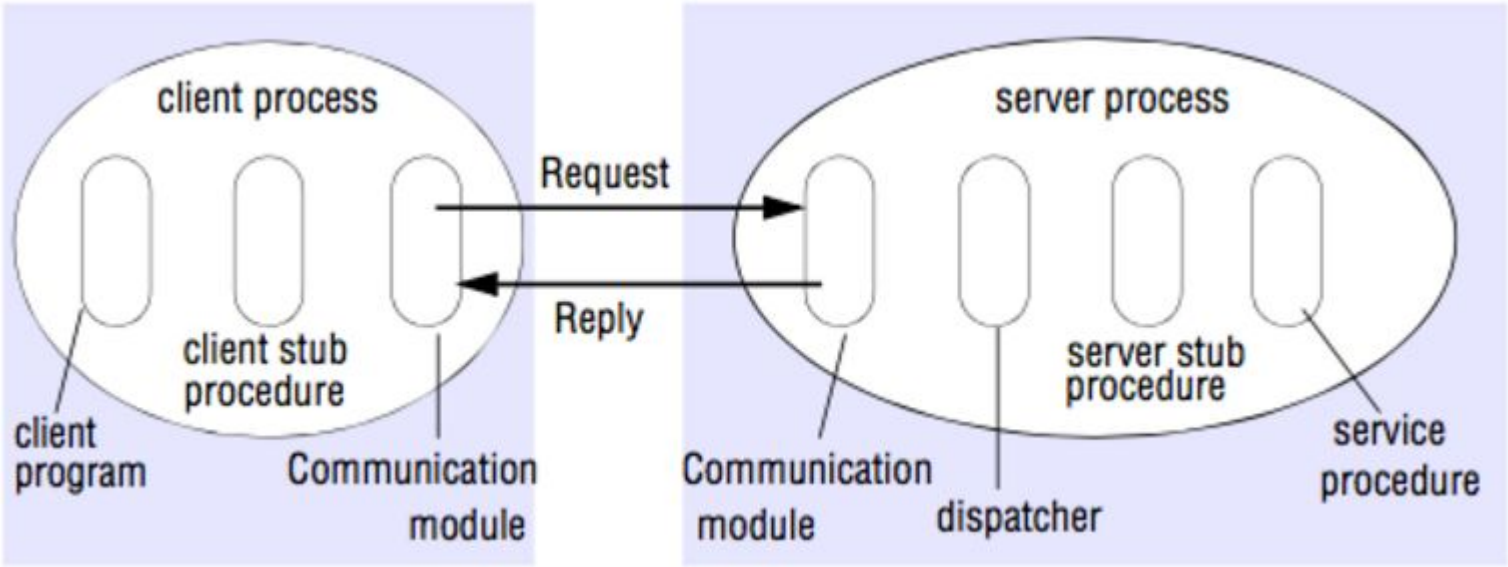
# Communication

- Understanding communication levels
  - Ponder this: how to check your service's availability?
- HTTP (Hypertext Transfer Protocol)
  - HTTP is a method for encoding and transporting data between a client and a server
  - It is a request/response protocol

| Verb | Function | Idempotent | Safe | Cacheable |
|---|---|---|---|---|
| GET | Reads a resource | Yes | Yes | Yes |
| POST | Creates a resource | No | No | Yes (if resp. contains freshness info) |
| PUT | Creates or replace a res. | Yes | No | No |
| PATCH | Partially updates a res. | No | No | Yes (if resp. contains freshness info) |
| DELETE | Deletes a resource | Yes | No | No |

# Communication

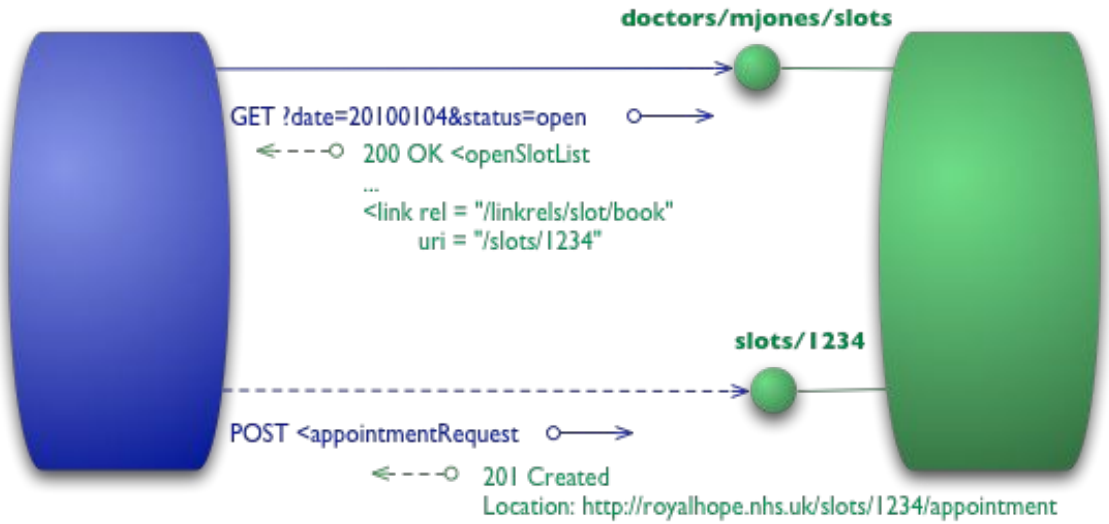
- **Remote Procedure Call (RPC)**
  - An application layer protocol for inter-process comm.
  - Ex: Google Protobuf, Facebook Thrift, Apache Avro
  - Generally RPC is internally used (harder to debug, not very flexible but performant)

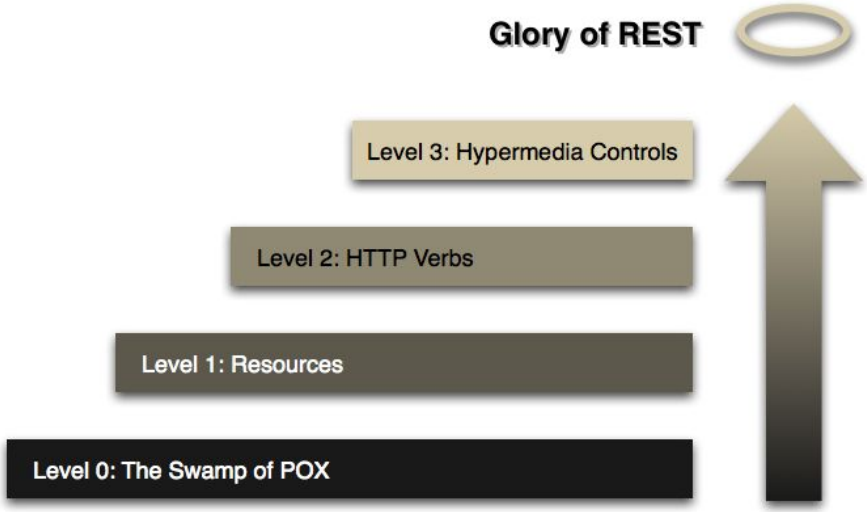



**Source:** http://www.puncsky.com/blog/2016/02/14/crack-the-system-design-interview/

# Communication

- **Representational State Transfer (REST)**
  - REST is an architectural style enforcing a client/server model where the client acts on a set of resources managed by the server
  - All communication must be stateless and cacheable
  - De facto standard for external APIs




**Source:** http://martinfowler.com/articles/richardsonMaturityModel.html

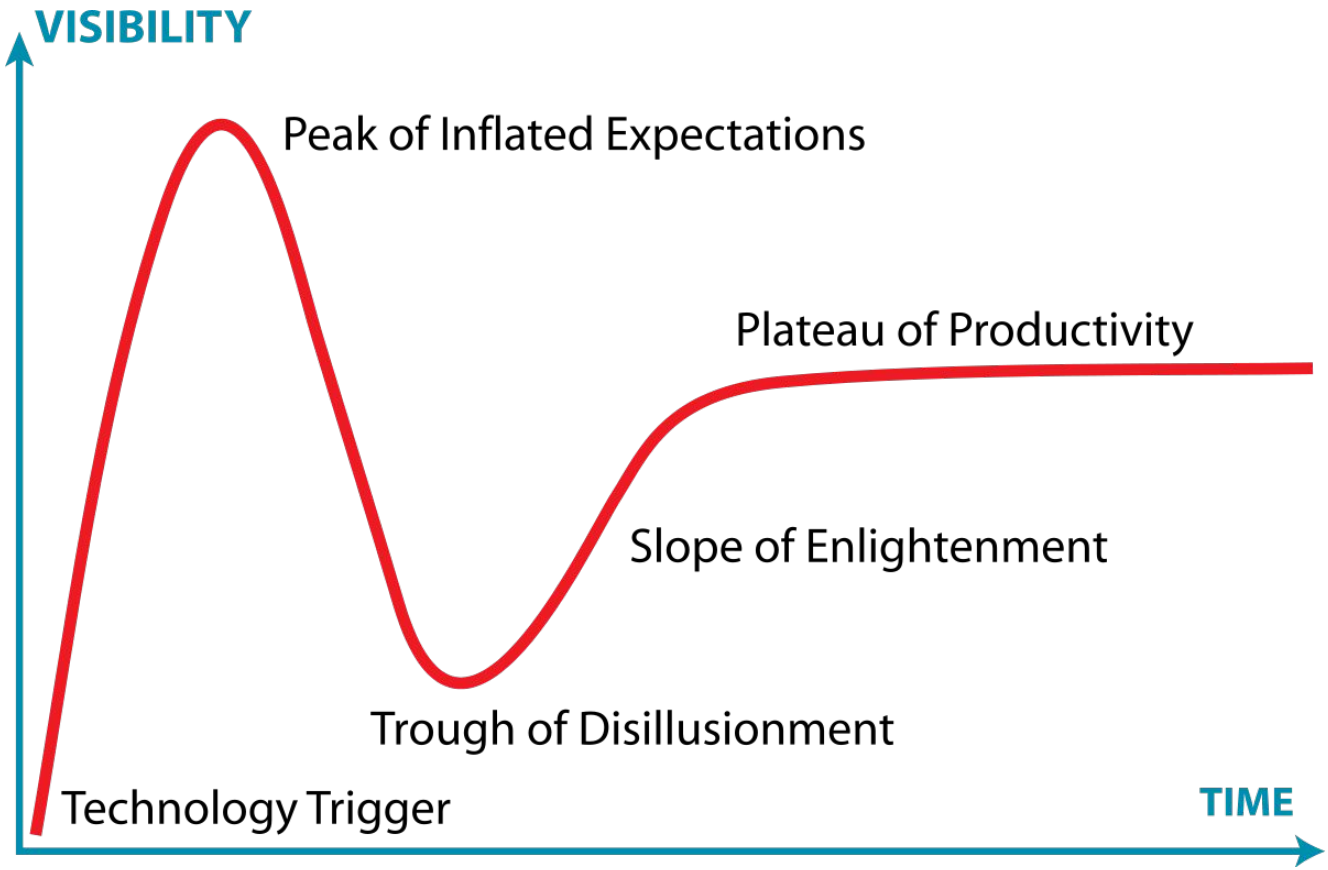
# Security

- **OWASP (Open Web Application Security Project)**
  - Core Purpose: *"Be the thriving global community that drives visibility and evolution in the safety and security of the world's software"*
  - OWASP TOP 10: A list of the 10 Most Critical Web Application Security Risks: https://owasp.org/www-project-top-ten/

| OWASP Top 10 - 2013 | → | OWASP Top 10 - 2017 |
|---|---|---|
| A1 – Injection | → | A1:2017-Injection |
| A2 – Broken Authentication and Session Management | → | A2:2017-Broken Authentication |
| A3 – Cross-Site Scripting (XSS) | ↘ | A3:2017-Sensitive Data Exposure |
| A4 – Insecure Direct Object References [Merged+A7] | U | A4:2017-XML External Entities (XXE) [NEW] |
| A5 – Security Misconfiguration | ↘ | A5:2017-Broken Access Control [Merged] |
| A6 – Sensitive Data Exposure | ↗ | A6:2017-Security Misconfiguration |
| A7 – Missing Function Level Access Contr [Merged+A4] | U | A7:2017-Cross-Site Scripting (XSS) |
| A8 – Cross-Site Request Forgery (CSRF) | ☒ | A8:2017-Insecure Deserialization [NEW, Community] |
| A9 – Using Components with Known Vulnerabilities | → | A9:2017-Using Components with Known Vulnerabilities |
| A10 – Unvalidated Redirects and Forwards | ☒ | A10:2017-Insufficient Logging&Monitoring [NEW,Comm.] |

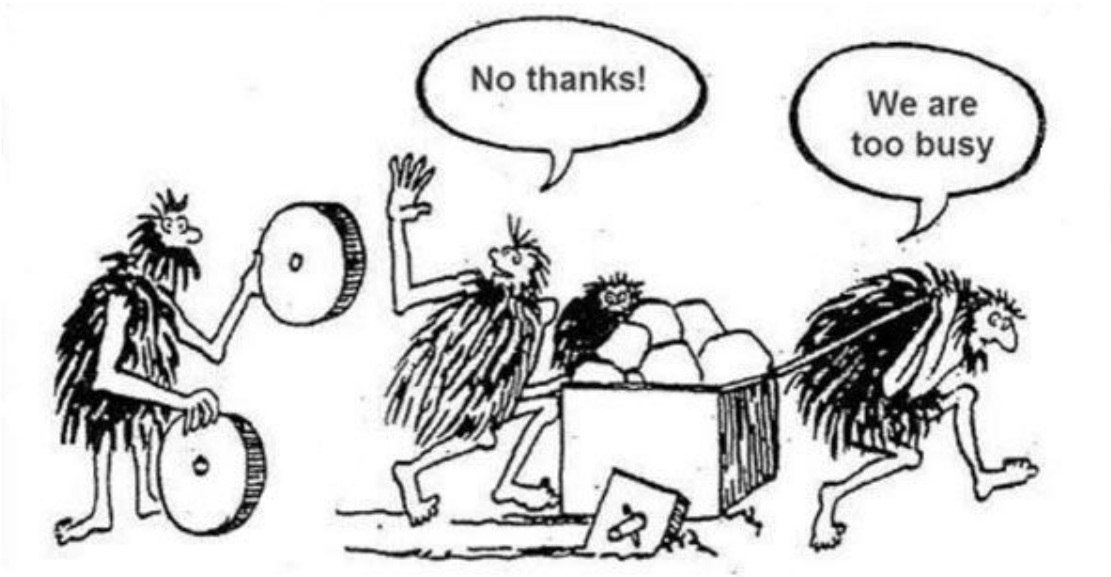
# Recap

- Everything in large-scale system design is **a trade-off**
  - Do not buy the hype & choose wisely



# Recap

- Do not reinvent the wheel
  - Patterns (Availability, Consistency, etc. )
  - Protocols (esp. application level ones such as RPC)




**Source:** Pinterest

# Q/A