SDE Skills: System Design

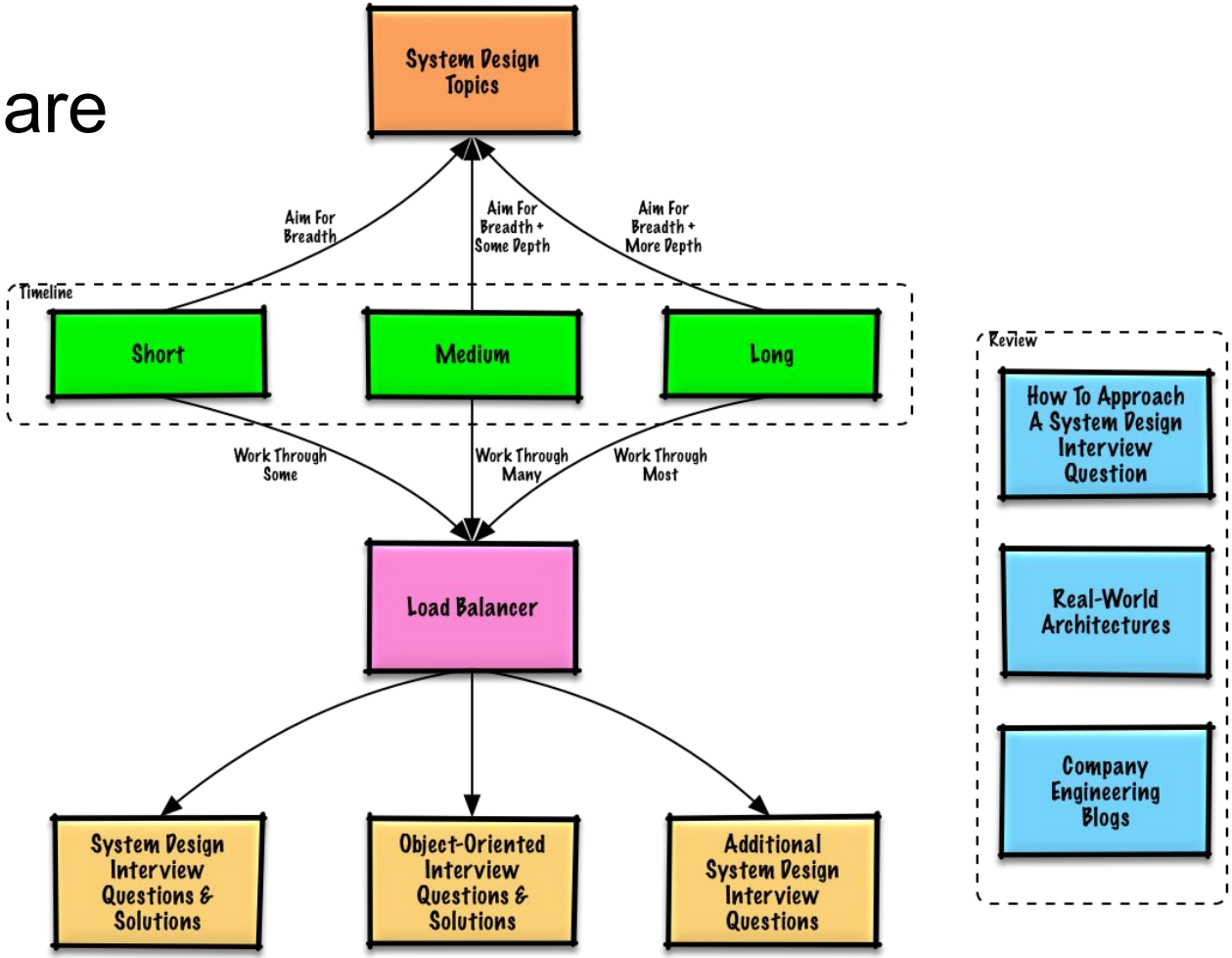
Preparation strategy, Key Concepts
Session 1 (Slides 1-20), Session 2 (Slides 21-45)

(By Naga Aerakoni)
LinkedIn twitter

Agenda

- Study Strategy- How to prepare for your system design interviews
- Key Terminology
  - Scalability vs Performance
  - Latency vs Throughput
  - Consistency / Availability
    - CAP Theorem
    - Consistency Patterns
    - Availability Patterns
- DNS, CDN
- Load Balancers
  - Types of Load Balancers
- Proxy Servers
  - Open (forward) proxy
  - Reverse Proxy (Web Server)
- Databases
  - RDBMS
  - NoSQL
  - Scaling Databases
- Caching
  - Caching mechanisms/strategies
- TCP vs UDP
- Consistent Hashing (Bonus Learning)
- Questions/References

How to prepare



Preparing for system design interviews

	Short	Medium	Long
Read through the System design topics to get a broad understanding of how systems work	👍	👍	👍
Read through a few articles in the Company engineering blogs for the companies you are interviewing with	👍	👍	👍
Read through a few Real world architectures	👍	👍	👍
Review How to approach a system design interview question	👍	👍	👍
Work through System design interview questions with solutions	Some	Many	Most
Work through Object-oriented design interview questions with solutions	Some	Many	Most
Review Additional system design interview questions	Some	Many	Most

Key Terminology - Performance vs Scalability

- Service is scalable
  - if it results in increased performance in a manner proportional to no. of resources added
- Increased performance means handling more units of work
  - It could also mean handling large units of work.

- Other way to compare
  - If you have performance problem, your system is slow for single user.
  - If you have the scalability problem, your system is fast for single user, but slow for the heavy load.

Key Terminology - Latency vs Throughput

- Latency
  - Time taken to perform some action (or) produce some result
- Throughput
  - Number of such actions (or) results per unit of time

- Generally, you should aim for maximal throughput with acceptable latency.

Key Terminology - Consistency vs Availability

- Consistency
  - Every read receives the most recent write or an error
- Availability
  - Every request receives response, without guarantee that it contains the most recent version of the information
- Partition Tolerance
  - System continues to work despite arbitrary network failures.
  - Networks aren’t reliable, so you’ll need to support Partition tolerance.

CAP Theorem

- CAP Theorem
  - Of 3 above, one of them must be sacrificed.
  - Since PT needs to be supported, you need to make tradeoff b/w consistency and availability.
- Two Options
  - Consistency / Partition Tolerance (CP)
  - Availability / Partition Tolerance (AP)
- CP
  - Waiting a response from partitioned node might result in a timeout error
  - CP is good choice
    - If your business requires atomic reads/writes
- AP
  - Responses return most recent version of data available on node, which might not be latest
  - AP is good choice
    - If your business needs allow eventual consistency
    - When system needs to continue working despite external errors

Consistency Patterns

- Weak Consistency
  - After a write, reads may (or) may not see it
  - Best effort is taken
  - Eg: Memcached, VOIP, video chat, multiplayer games
- Eventual Consistency
  - After a write, reads will eventually see it (typically within milliseconds)
  - Data is replicated asynchronously
  - Eg: DNS, email
- Strong consistency
  - After a write, reads will see it
  - Data is replicated synchronously
  - Eg: File systems, RDBMS (transaction based)

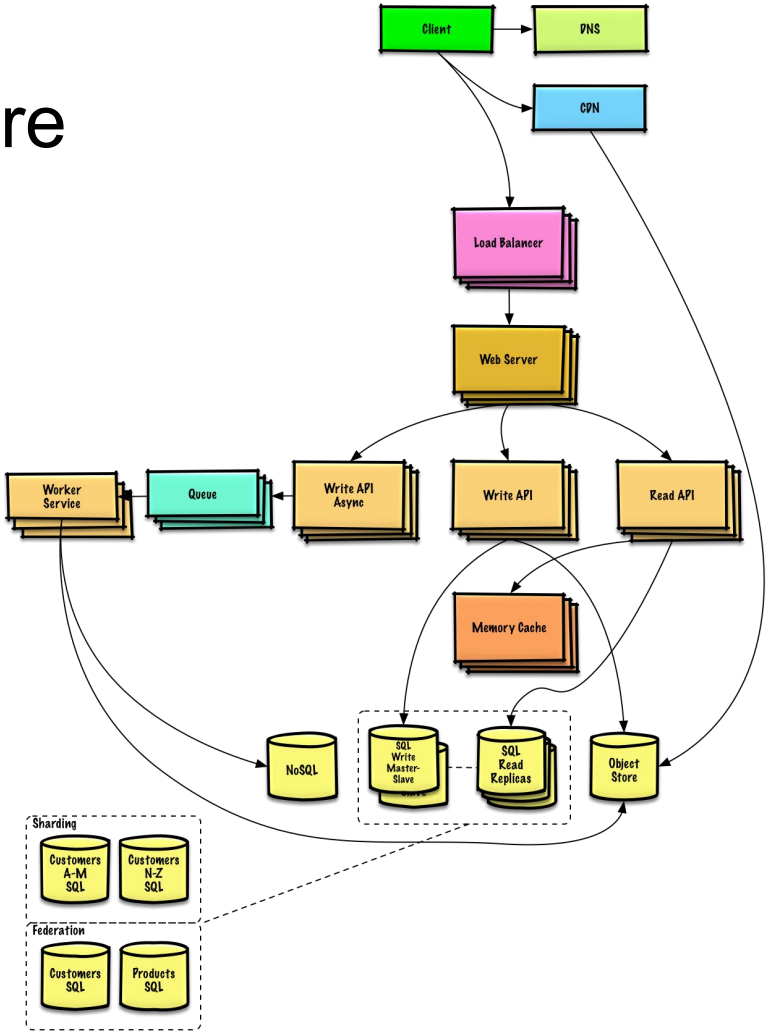
Availability Patterns

- Two main patterns
  - Fail-over
  - Replication
- Failover methods
  - Active-passive (master-slave) failover
  - Active-active (master-master) failover
- Active-passive (master-slave) failover
  - Heartbeats are sent between active & passive server on standby
  - If heartbeat is interrupted, then passive server takes over active server IP address, and resumes service
  - Length of downtime is determined by whether passive server
    - Is already running in ‘hot’ standby
    - Needs to start from ‘cold’ standby
  - Only active server handles traffic

Availability Patterns (Contd..)

- Active-active (master-master) failover
  - Both servers are managing traffic, spreading load between them
  - If servers are public facing, DNS would need to IP addresses of both
  - If servers are internal, application logic needs to know about both.
- Disadvantages (failover)
  - Adds more hardware/complexity
  - Potential loss of data if active fails before any newly written data can be replicated to passive.
- Replication
  - 2 methods (will revisit in scaling databases topic)
    - Master-slave replication
    - Master-master replication

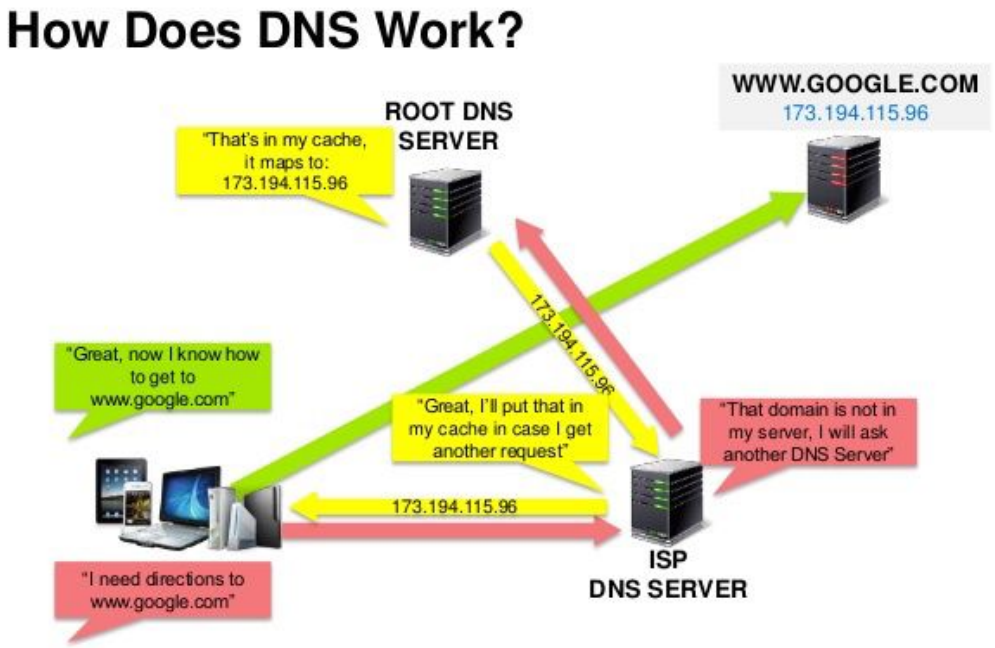
Typical Architecture



DNS (Domain Name System)

Key DNS Records:

- NS (name server)
  - List of DNS
- MX (mail exchange)
  - Mail servers
- A (alias record)
  - Name to IP addr
- CNAME (canonical name)
  - Alias names for domain.
- SOA (Start of Authority)
  - TTL (time to live),
  - owner,
  - primary DNS
  - refresh/retry interval



DNS - types of domain names

- Root domain
  - Fully qualified domain name that ends with period (.)
    - Eg: “example.microsoft.com.”
- Top-level domain
  - Name used to indicate type of organization (or) country/region
    - Eg: .com, .org, .mil, .gov, .edu, .net, .xx(2-letter country codes)
- Second level domain
  - Variable length names registered to organization or individual
    - Eg: microsoft.com
- Subdomain
  - Additional names that organization can create under main 2nd level domain
    - Eg: example.microsoft.com, docs.microsoft.com etc
- Host (or) resource name
  - Names that represent a leaf in DBS tree of names
    - Eg: host-a.example.microsoft.com

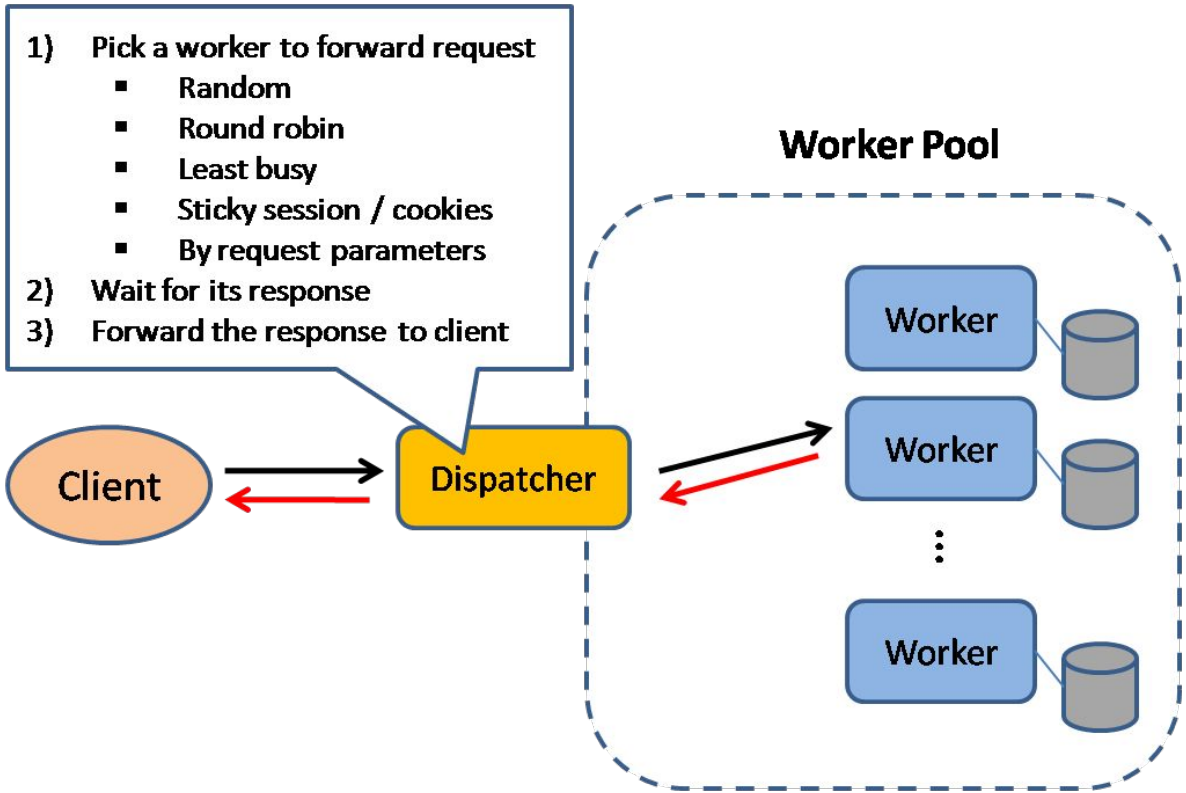
CDN (Content Delivery Network)

- Serve content close to the user
- static content is normally served
  - HTML/CSS/JS,
  - photos, and videos
- servers do not have to serve requests
  - that the CDN fulfills
- 2 categories
  - Push CDNs
  - Pull CDNs

CDN categories

- Push CDN
  - Receive content whenever changes occur on your server
  - You take full responsibility for providing content
    - Upload to CDN
    - Rewrite URL to point to CDN
  - sites that are small, or less frequently changes content works well with Push CDNs
- Pull CDN
  - Grab new content from your server when first user request comes in
  - You leave content in your server, and
    - Rewrite URL to point to CDN
    - Results in slower request until it gets cached in CDN
  - TTL determines how long its cached
  - Sites with heavy traffic works well with Pull CDNs

Load Balancers

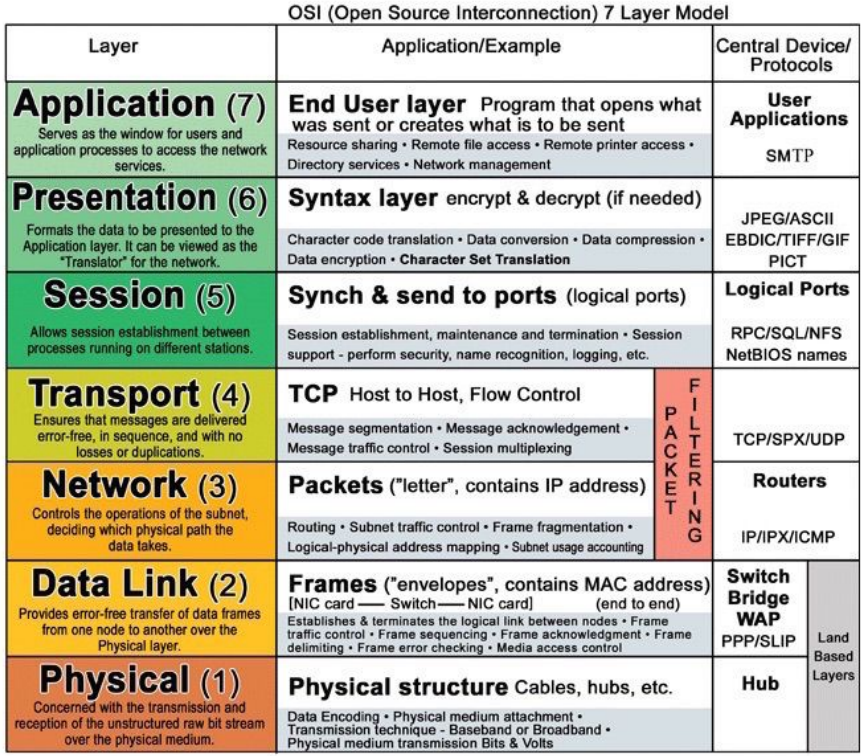


Load Balancers (Contd…)

- Benefits
  - Prevents requests from going to unhealthy servers
  - Prevents overloading resources
  - Help eliminate single point of failure
  - SSL termination: decrypt incoming requests, and encrypt outgoing responses
  - Session management: issue cookies, and route requests from same user to same server if servers not maintaining session.
- To avoid failures, it's common to setup multiple load balances (active-active, active-passive)
- Load balances can route traffic based on
  - Random
  - Least loaded
  - Session/cookies
  - Round robin or weighted round robin
  - Layer 4
  - Layer 7

Load Balancers (Contd…)

- Layer 4:
  - Look at transport layer to decide routing
  - Source, destination IP, ports
  - Performs NAT-network address translation
    - Change dest IP to target server
    - Change back to its own IP in response
- Layer 7:
  - Look at application layer to decide routing
  - Header, message, cookies
- Layer 4 vs Layer 7
  - Layer 4 requires less computing resources

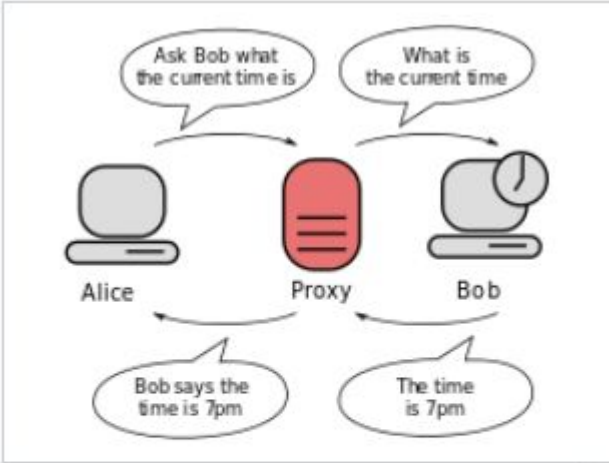


Load Balancers (Contd…)

- Horizontal Scaling
  - Load balancers help with horizontal scaling
    - Improving performance and availability
  - mostly, this is more efficient than vertical scaling (scale individual machines)
  - Disadvantages
    - Servers should be stateless - shouldn’t contain any user info
    - Sessions to be stored in centralized data store (eg: Redis/memcached)
- Disadvantages of load balancer
  - Can become performance bottleneck
    - it doesn't have enough resources (or) not configured properly
  - Single load balance is single point of failure
    - Configuring multiple increases complexity

Proxies - Proxy server

- Proxy Server
  - can reside on the client's local server (or) anywhere between the client and the remote servers
  - client connects to the proxy server, requesting some service, such as
    - a file, connection, web page, or
    - other resource available from a different server
  - add structure and encapsulation to distributed systems
- Types of proxy servers
  - Gateways (tunneling proxy)
    - passes unmodified requests and responses
  - Open (forwarding) proxy
  - Reverse proxy

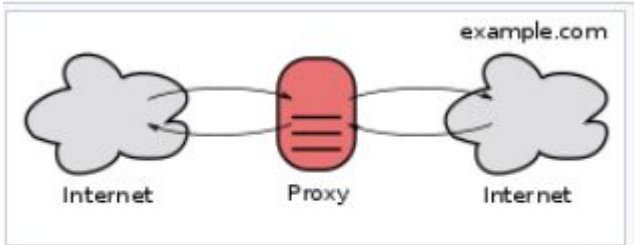


Communication between two computers (shown in grey) connected through a third computer (shown in red) acting as a proxy. Bob does not know to whom the information is going, which is why proxies can be used to protect privacy.

Proxies - Open Proxies

- Open (forwarding proxy)
  - Internet-facing proxy used to retrieve data from a wide range of sources (in most cases anywhere on the Internet)
  - Accessible any internet user.
- Types of Open Proxies
  - Anonymous Proxy
  - Transparent Proxy
- Anonymous Proxy
  - Reveals its identity as a server but does not disclose the initial IP address
  - Beneficial for some users, as it hides their IP addresses

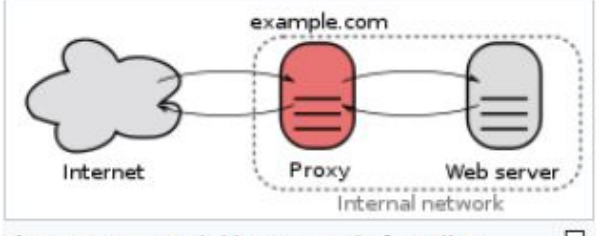
- Transparent Proxy
  - Identifies itself, and with HTTP headers support, it's first IP addresses can be seen
  - Benefit is these servers can cache websites
  - Your IP address is not hidden in server
  - Sometimes your IP may get banned for using transparent proxy



An open proxy forwarding requests from and to anywhere on the Internet.

Proxies - Reverse proxy (web server)

- proxy server that appears to clients to be an ordinary server
  - Forwards request to other servers
  - Response is returned as if it came from original server leaving clients no idea of origin servers
- Benefits
  - Provides uniform interfaces to public
  - Increased security
    - Hide backend server info, IP blacklists
    - Protects against DoS, DDoS attacks.
      - Limit no. of connections per client
    - Can add basic auth if backend servers don't
  - Increased scalability/flexibility
    - Client only sees reverse proxy's IP
    - You can scale up/down behind seens
- SSL termination - decrypt requests, encrypt responses back
- Compression - compress server responses
- Caching - caching responses, static content
- Can perform A/B testing



A reverse proxy taking requests from the Internet and forwarding them to servers in an internal network. Those making requests connect to the proxy and may not be aware of the internal network.

Load balancer vs Reverse proxy

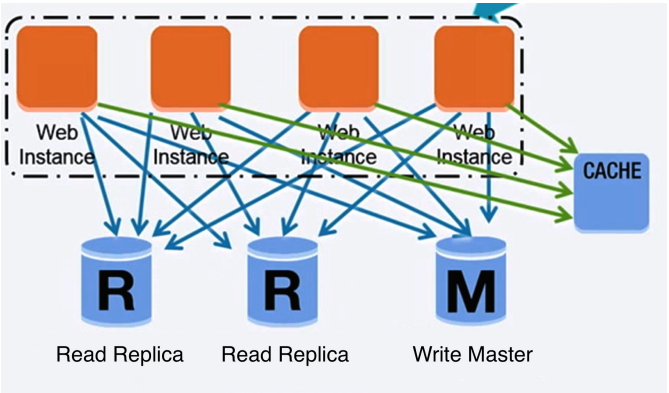
- Disadvantages of Reverse proxy
  - Introducing a reverse proxy results in increased complexity.
  - A single reverse proxy is a single point of failure, configuring multiple reverse proxies (ie a failover) further increases complexity
- Solutions such as NGINX, HAProxy supports both
  - layer 7 reverse proxying
  - Layer 7 load balancing

- Load Balancer vs Reverse Proxy
- Deploying load balancer is useful when you have multiple servers
  - Often LB route traffic to set of servers serving same function
- Reverse proxies can be useful event with just one web server
  - With benefits described previously

Databases (RDBMS)

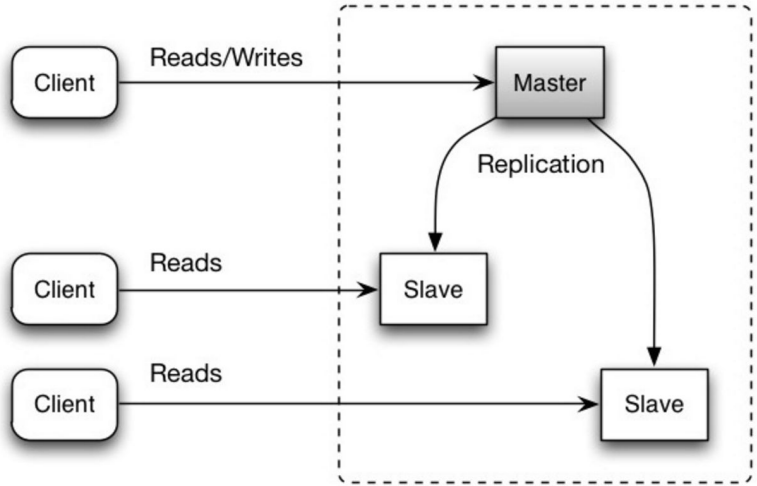
- relational database
  - collection of data items organized in tables.
- ACID properties
  - **Atomicity**: each transaction is all or nothing
  - **Consistency**: any transaction will bring database from one state to other
  - **Isolation**: Executing transactions concurrently has same effect as executing them in serial fashion.
  - **Durability**: once transaction has been committed, it will remain so

- Techniques to scale
  - Master-slave replication
  - Master-master replication
  - Federation
  - Sharding
  - Denormalization
  - SQL tuning



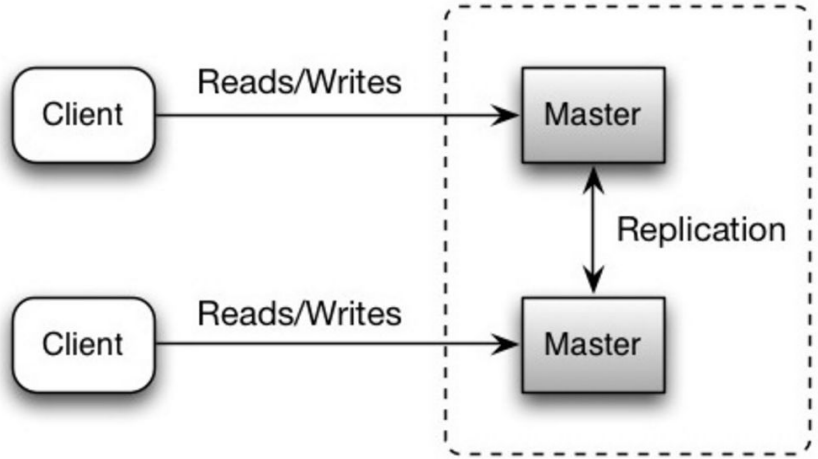
Scaling databases: Master-slave replication

- Master supports reads/writes
  - Replicates writes to all slave nodes
- Slaves supports reads only
- If master goes offline, system can continue to operate
  - Until a slave is promoted to new master
  - Or new master is provisioned
- Disadvantages:
  - Additional logic is needed to promote slave to master



Scaling databases: Master-master replication

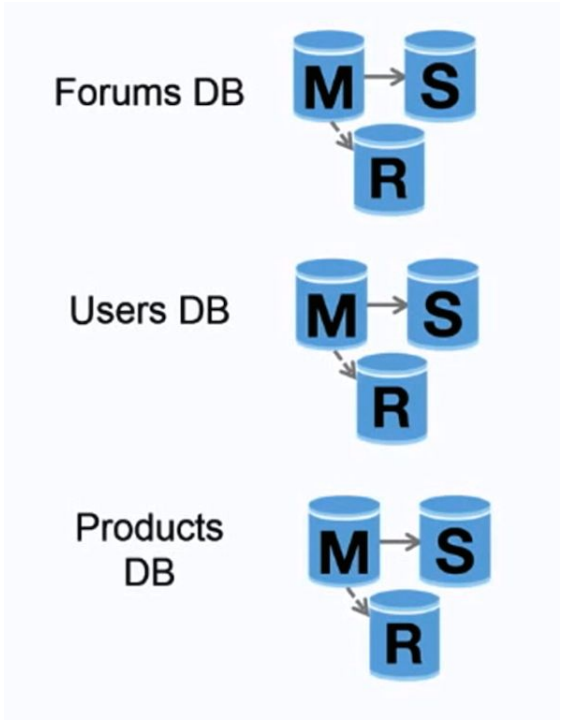
- Both masters support read/writes
  - Coordinate with each other on writes
- If either master goes down
  - System would continue to support reads/writes
- Disadvantages
  - Need LB or application logic (which master?)
  - Increased latency due to synchronization
- Disadvantages of replication
  - Potential loss of data if master fails before any
    - newly written data is replicated to other nodes
  - More read slaves, the more you need to replicate,
    - increasing latency
  - Writes are replayed to read replicas
    - If lot of writes, then it would degrade read replica performance



Scaling databases: Federation

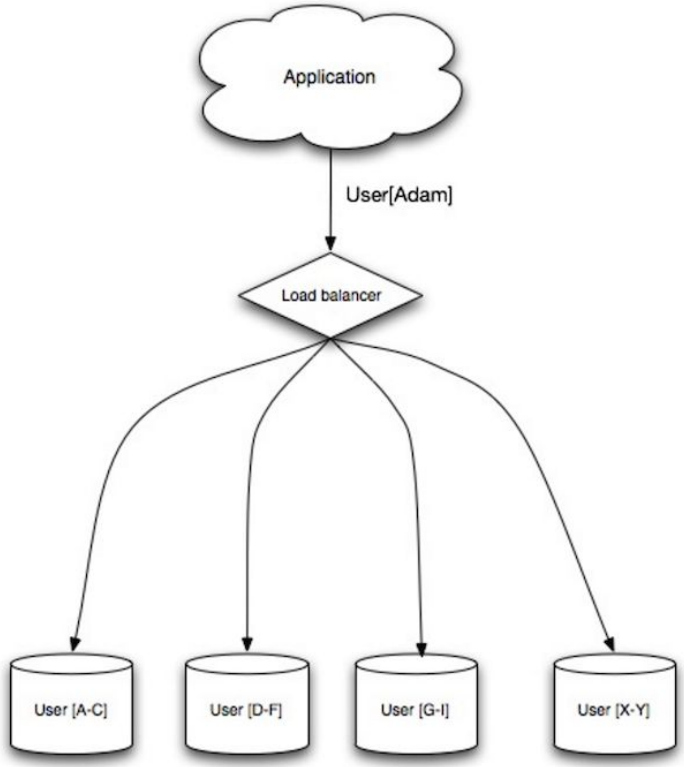
- Splits up databases by function
- Benefits
  - Results in less read/write traffic to each database
  - Less replication lag
  - More data can fit in memory
  - Writes can be parallel, increasing throughput

- Disadvantages
  - Not effective if your schema requires huge functions/tables
  - Application logic is needed (which DB to write??)
  - Joining data from multiple databases is more complex.



Scaling databases: Sharding

- Each DB handles only subset of main data
- Benefits are similar to federation.
- Common ways to shard
  - Users last name initial
  - Users geographic location
- Disadvantages
  - Update application logic to work with shards
  - Can result in complex SQL queries
  - Set of power users in one shard can result
    - Increased load in one compared to others
  - Joining data from multiple shards is complex



Scaling databases: Denormalization, SQL tuning

- **Denormalization**:
  - Attempts to improve read performance
    - At expense of some write performance
  - Redundant copies of data are written in multiple tables to avoid expensive joins
  - PostgreSQL, Oracle - support materialized views
    - Which handle storing redundant info, and keeping them consistent
  - Can be combined with federation, sharding
    - As there is need for complex joins
    - General observation: reads:writes ratio (100:1, 1000:1), so its good to denormalize data.
- **SQL Tuning**
  - Tighten up schema
    - Use char instead of varchar,
    - Use TEX for large text content like blogs etc…
    - Declaring not null wherever possible
  - Use good indices
    - Specially columns used in select, group by, order by, joins

NoSQL

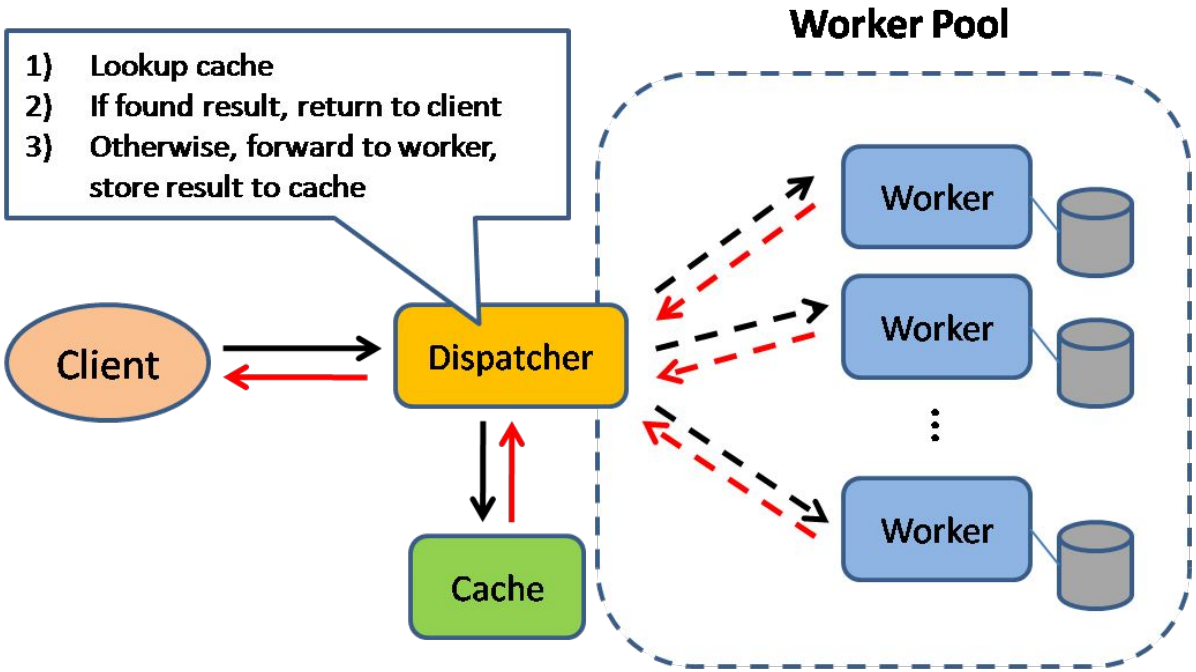
- Types of DBs
  - key-value store (Redis, memcached)
  - Document store (MongoDB, CouchDB)
  - Wide column store (Googles Bigtable, Facebook Cassandra, HBase etc)
  - Graph database (Neo4j, FlockDB)
- Most favor eventual consistency
- BASE:
  - Basically Available
    - System guarantees availability
  - Soft state
    - State of system may change over time even without input
  - Eventual consistency
    - System will become consistent over a period of time

SQL vs NoSQL

SQL	NoSQL
Structured data	Semi structured
Strict schema	Dynamic/flexible schema
Relational data	Non relational data
Need for complex joins	No need for complex joins
Transaction based systems (ACID)	Non transactional, store TB of data
Clear patterns for scaling	Very data intensive workload

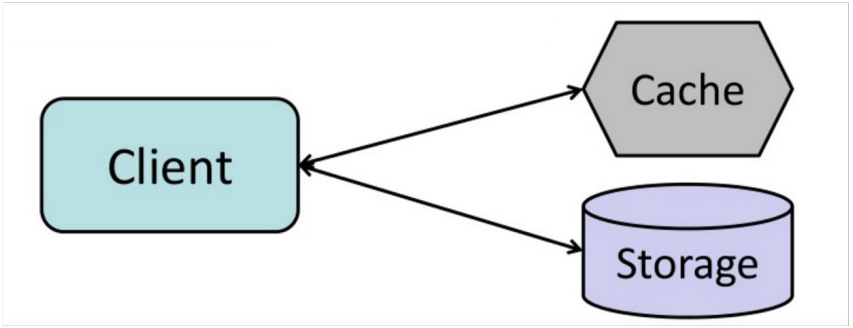
Caching mechanisms

- Caching improves page load times
- Caching reduces load on servers
- Different places where caching can be put
  - Client caching
    - OS, browser
  - CDN caching
  - Web Server (reverse proxy)
  - Database caching
  - Application caching
    - In-memory cache
  - Cache at DB query level
  - Cache at object level



Caching Strategies: Cache-aside

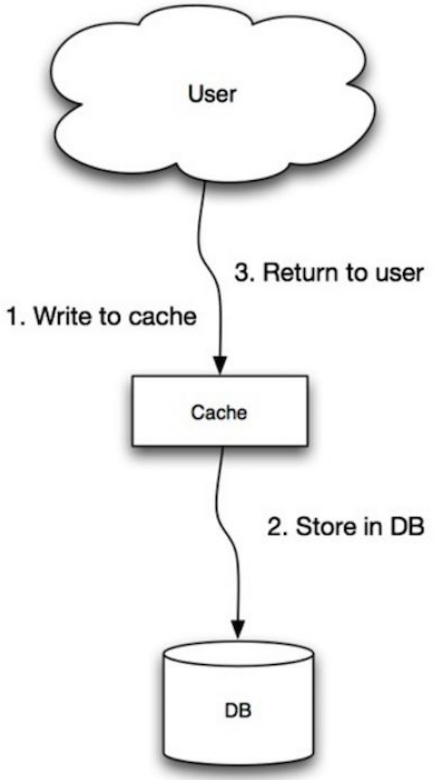
- Application does following
  - Look for an entry in cache, resulting in cache miss
  - Load entry from database
  - Add entry to cache
  - Return entry
- Application is responsible for
  - reading/writing to storage
- Cache doesn't interact with storage directly
- Disadvantages
  - Cache miss would result in 3 trips (delay)
  - Data can become stale if its updated in DB
    - Can be mitigated using TTL
  - When node fails, it's replaced by empty node causing latency



Caching strategies: Write-through

- Application uses cache as main data store
  - reading/writing data to it
- Cache is responsible for reading/writing to DB
- Slow overall operation, but subsequent reads are fast

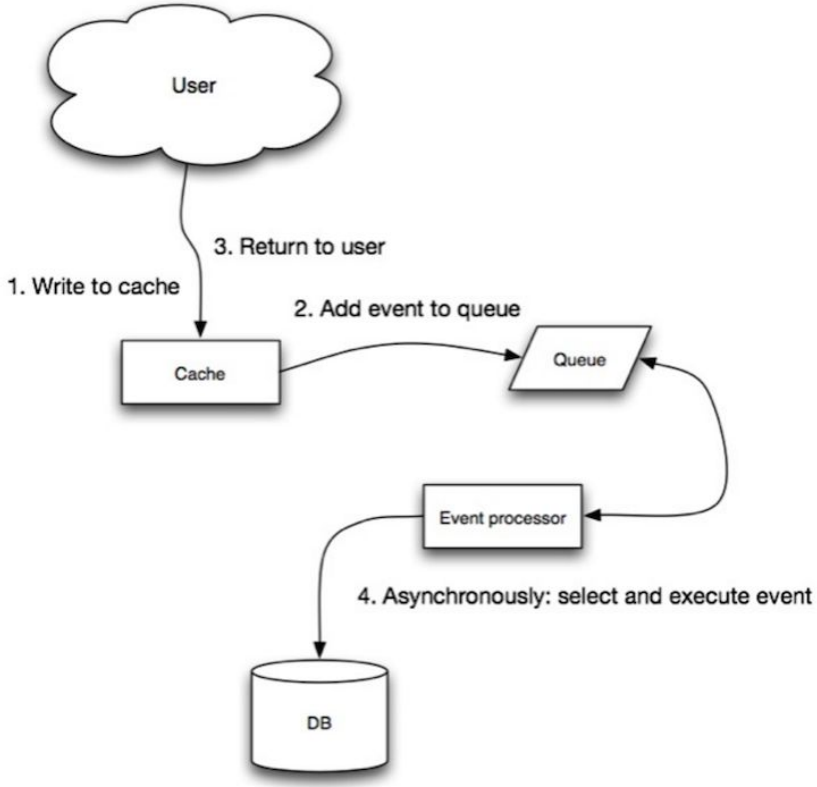
- Disadvantages
  - When a new cache node is created due to failure (or) scaling
    - It won't cache entries until its updated
      - Cache aside can be used in conjunction to mitigate this issue
  - Most data written to cache, might never be read
    - Minimized with TTL



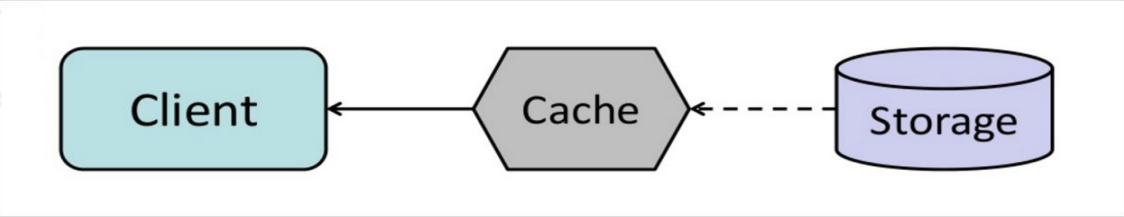
Caching strategies: write behind (write back)

- Application does following
  - add/update entry in cache
  - Asynchronously write entry to database, improving write performance
  - Response is returned to user only after adding write event to queue

- Disadvantages
  - Could be data loss if cache goes down prior to its contents hitting DB
  - More complex to implement when compared to cache aside, write-through



Caching strategies: Refresh ahead

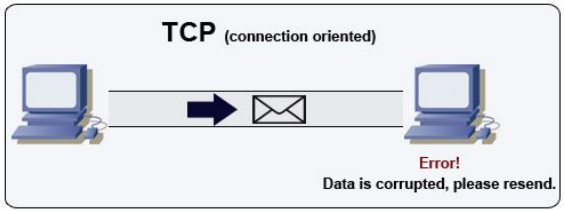


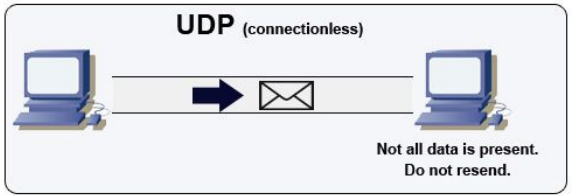
- Can configure cache to automatically refresh any recently accessed cache prior to its expiration
- Can result in reduced latency if cache can accurately predict which items are likely needed in future
- Disadvantages
  - Not accurately predicting needed items, can reduce performance

- **Disadvantages of cache (in general)**
  - Need to maintain consistency b/w cache and DB
  - Cache invalidation is difficult problem
  - Need application changes to support cache

TCP vs UDP

TCP	UDP
Connection oriented	Connectionless
All packets are guaranteed to reach in original order	Datagrams are guaranteed only at datagram level, they will reach dest out of order or not at all
Reliable - useful for applications that require high reliability	Not reliable, but generally efficient
Eg: Web servers, HTTP, SMTP, FTP, SSH	Can broadcast messages eg: Routers (DHCP) Eg: video chat, VOIP, streaming, multiplayer games
Implements congestion control	Doesn’t implement congestion control





Bonus Learning :)

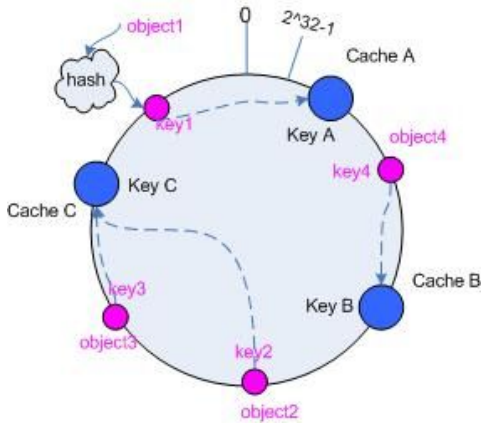
Consistent Hashing

- Special kind of hashing such that when a hash table is resized,
  - only **K/n** keys need to be remapped on average, where
    - **K** is the number of keys,
    - **n** is the number of slots
- Need for consistent hashing
  - Eg: Load balancing **'n'** cache machines
    - Common way is to put object **'obj'** in machine → ***hash (obj) % n***
    - This will not work if cache machine is added or removed
- Consistent hashing maps objects to the same cache machine, as far as possible.
  - when a cache machine is added, it takes its share of objects from all the other cache machines and
  - when it is removed, its objects are shared among the remaining machines
- very useful strategy for distributed systems (eg: distributed caching, distributed databases, distributed servers etc)

Consistent Hashing - How it works

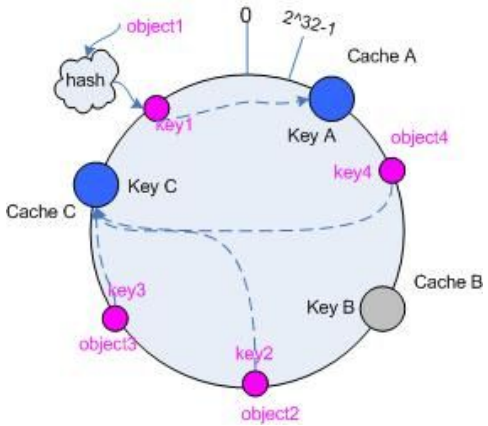
- Hashing space
  - Let's take this space 0 to 2^32 - 1
  - Hash function always maps to a value in this range
- Map all the objects into hash space
  - Use same hash function to map an object to be stored in cache
  - ```
    hash(object1) = key1;
    .....
    hash(object4) = key4;
    ```
- Map all the caches (nodes) into same hash space using same hash function
  - ```
    hash(cache A) = key A;
    hash(cache B) = key B;
    hash(cache C) = key C;
    ```

- Map objects to cache
  - Take object hash value i.e. key and see where it falls in ring, head clockwise on ring until you find a cache (or server)
  - If cache/server is down, you move to the next

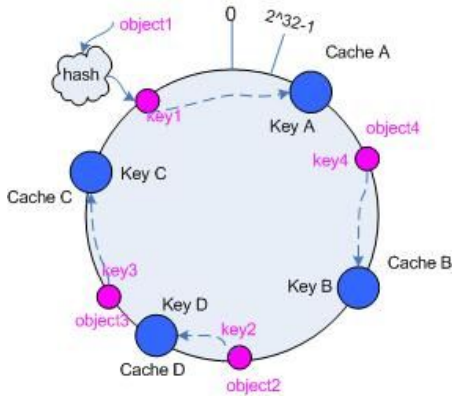


Consistent Hashing - Adding/removing a node

- If cache B is removed, then only objects stored in cache B will be rehashed and moved to cache C. (see below)



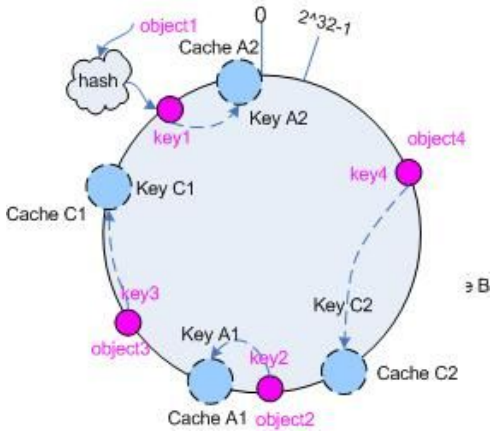
- If a new cache D is added, and D is hashed between object2 and object3 in the ring,
  - then only the objects that are between D and B will be rehashed, and moved to D. (see below)



Consistent Hashing - Virtual nodes

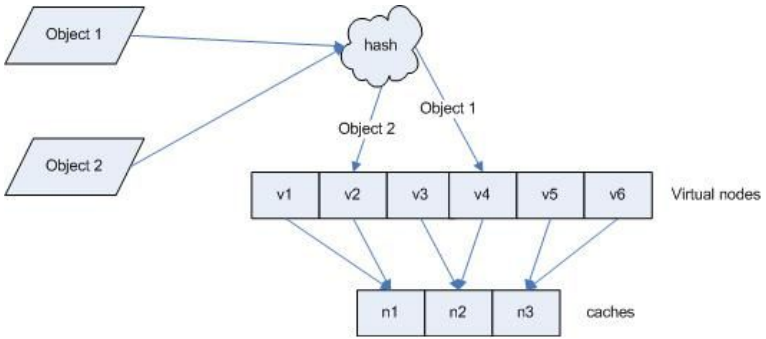
- If you don’t deploy enough caches (nodes)
  - Result in non-uniform distribution of objects into cache.
- Solution is to introduce idea of “virtual nodes”
- Virtual caches are replicas of cache points in circle (ring)
  - Each real cache corresponds to several virtual caches.
- Whenever we add a cache
  - We create number of virtual caches in circle (ring)
- Whenever cache is removed
  - We remove all of its virtual caches in circle or ring.

- Ex: lets take 2 caches A, C and 2 replica’s each (as shown below)
  - cache A1 and cache A2 represent cache A;
  - cache C1 and cache C2 represent cache C



Consistent Hashing - Complexity

- Mapping of virtual caches (nodes) to real caches (nodes)



- O(log(N)) complexity for consistent hashing comes from the fact that a
  - binary search among nodes/object keys is required to find the next node on the ring

Asymptotic time complexities for N nodes (or slots) and K keys

	Classic hash table	Consistent hashing
add a node	O(K)	O(K/N + log(N))
remove a node	O(K)	O(K/N + log(N))
add a key	O(1)	O(log(N))
remove a key	O(1)	O(log(N))

Questions

Any questions?

References:

https://github.com/donnemartin/system-design-primer

https://en.wikipedia.org/wiki/Proxy_server