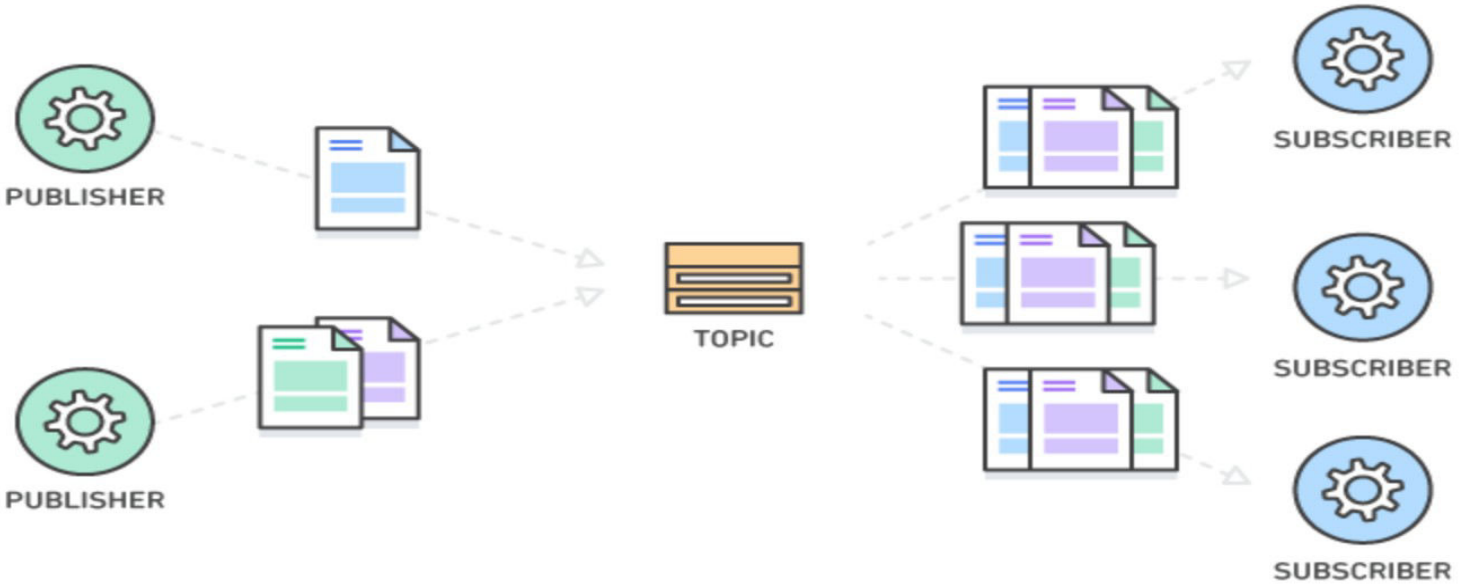
System Design

- Vedant Dhiman

Schedule

- Content Delivery Networks (CDN)
- Introduction
- Popular CDNs
- How CDNs Work?
  - Push Mechanism
  - Pull Mechanism
- When Not To Use CDNs
- Caching
  - Why We Need Caching?
  - How Caching Works?
  - Caching Policies
  - Cache Coherence
- Practice – System Design Problems

Pub/Sub Model



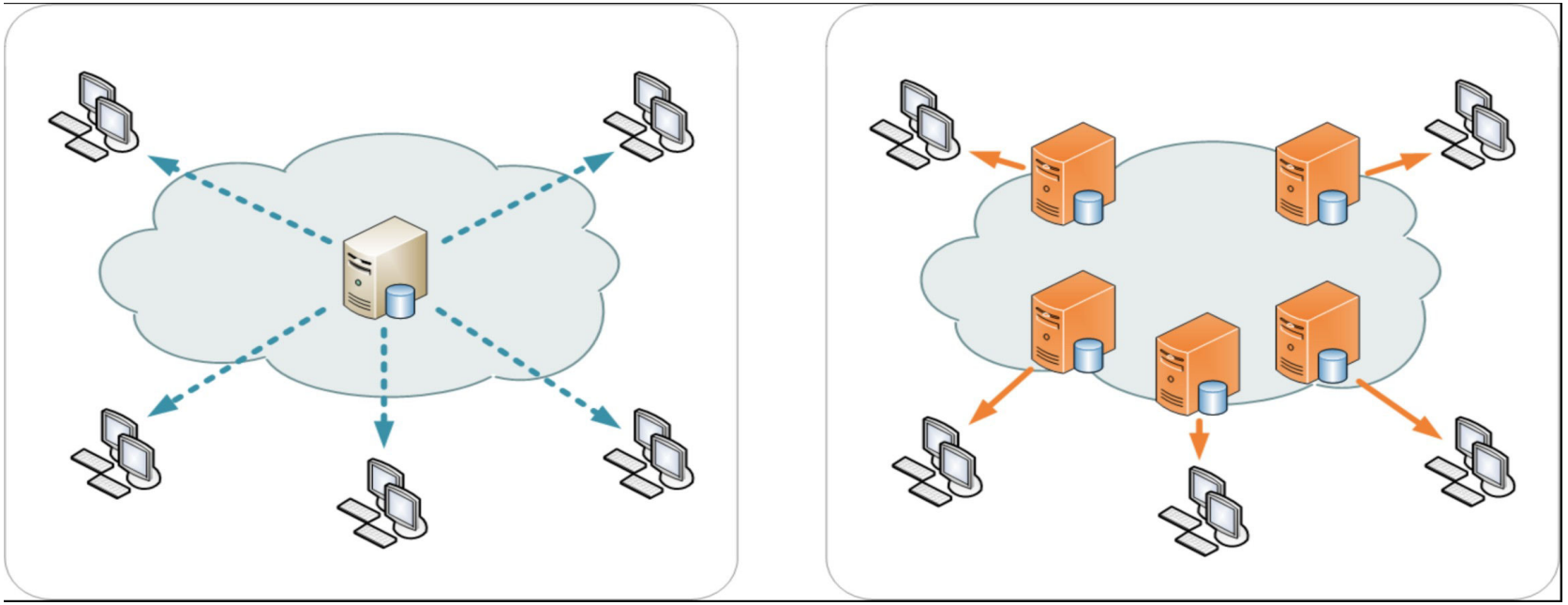
Content Delivery Network (CDN)

How to reduce request latency ?

Content Delivery Network (CDN)

- CDN is a very powerful way to reduce request latency when fetching static assets from a server
- Ideal CDN is a group of servers spread out globally
- Instead of fetching static messages such as images / videos / HTML / CSS / Javascript files from the origin server, users can fetch cached copies of these files from the nearest CDN more quickly

Content Delivery Network (CDN)

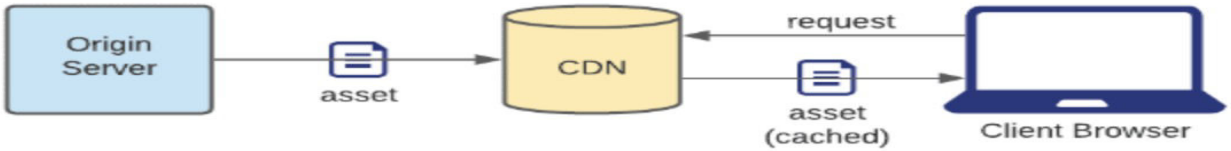


Popular CDNs

How CDNs Work?

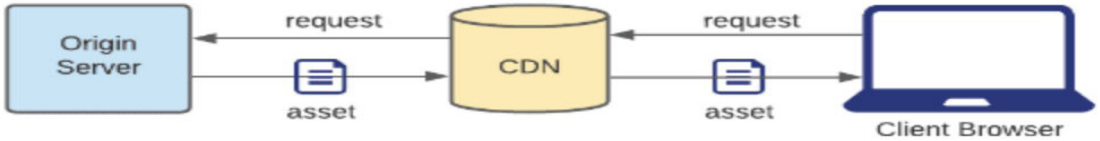
There are 2 primary ways for a CDN cache to be populated:

Push CDN: The engineers push new/updated files to the CDN, which would then propagate them to all of the CDN server caches



Pull CDN

When user sends a static request to the CDN server, if the CDN doesn't have it, it'll fetch the asset from the origin server, populate its cache with the asset, and then asset to the user



Advantages & Disadvantages

- Push CDN
  - More engineering work for the developers to make sure the CDN assets are up to date
  - When asset is created/updated, developers have to make sure to push it, otherwise clients won’t be able to fetch the new assets
  - Users see the most-recent version as intended by the developers/company policies
- Pull CDN
  - Require less maintenance
  - CDN will automatically fetch the assets that are not in it’s cache
  - Downside is that if the assets are already cached and developers have deployed a new version, the users may still see a slightly older version of the app for some time
  - Downside is that for the first request to a Pull CDN will always take a while as the Pull CDN has to fetch the asset from the origin server

Pull CDNs

- Pull CDNs are more popular than Push CDNs as they are easier to maintain
- Pull CDNs assign a timestamp to an asset when cached
- It typically caches an asset for 24 hours by default.
- If the user makes a request for an expired asset, then the CDN will re-fetch the asset from the origin server, and get an updated asset if there is one
- Pull CDNs usually support Cache-Control response headers offering
    - Flexibility with regards to caching policy
    - Cached assets can be fetched every 5 minutes, when there’s a new release version
- **Cache Busting**: CDN caches assets with a hash or etag that is unique compared to previous asset versions

When not to use CDN ?

- If you service’s target audience are in a specific region, then there won’t be any benefit of adding a CDN
- You can just host your origin servers in that specific region instead
- CDNs are not a good idea when the assets are dynamic and sensitive
- You don’t want to serve stale data for sensitive applications such as financial/government services

Caching

How to retrieve data quickly?

Cache

- A cache is any data store that can store and retrieve data quickly for faster use
- Cache enables faster response time
- Cache decreases load on other parts of the system

Why we need caching ?

- Without caching, applications will be really slow due to access time of retrieving data at every step
- Caches store data closer to where it is likely to be needed
- In large-scale applications, caching makes data retrieval efficient by:
  - Reducing repeated calculations
  - Reducing database queries
  - Reducing requests to other services
- Caching is storage of pre-computed data like personalized newsfeed or analytics report

How caching works ?

- Caching can be implemented in the following ways:
  - In-memory application cache
  - Distributed in-memory cache
  - Database cache
  - File System Cache

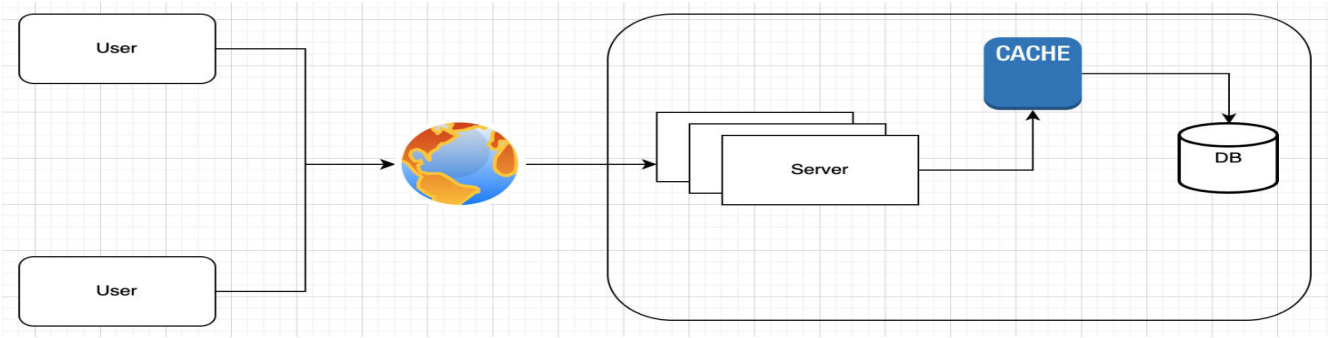
In-memory and database cache

- **In-memory application cache**:
- Storing data directly in the application’s memory is a fast and simple option
- Each server must contain it’s own cache
- It increases overall memory demands and cost of the system

- **Database cache:**
- A traditional database can be used to cache commonly requested data
- Store the results of pre-computed operations for retrieval later

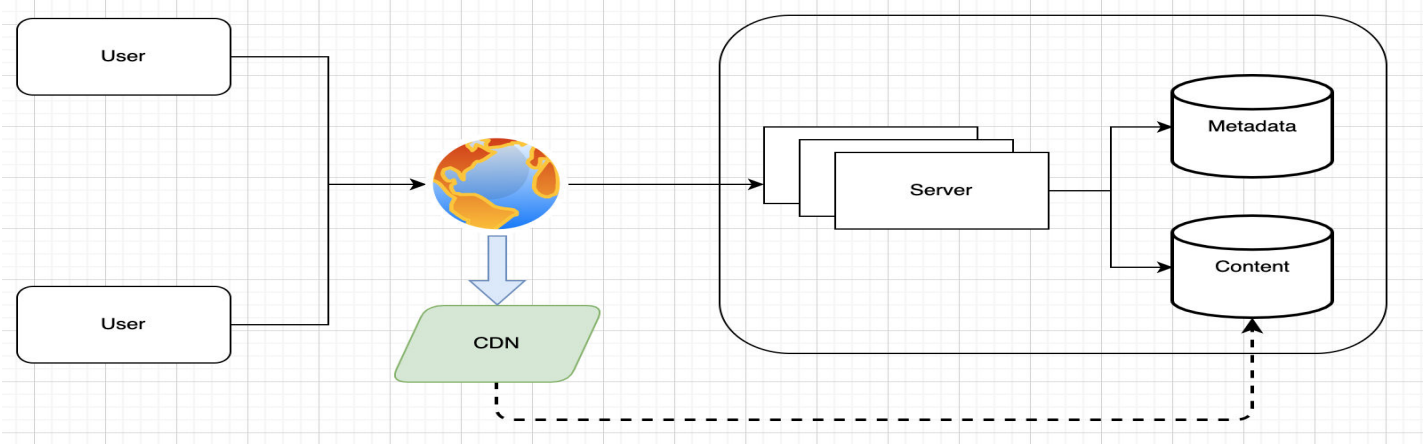
Distributed In-Memory Cache

- **Distributed in-memory cache:**
- A separate caching server can be used to store data
- Multiple servers can read and write from the same cache
- Examples: Redis, Memcached



File System Cache

- A file system can also be used to store commonly accessed files
- CDNs are one example of a distributed file system
- CDNs take advantage of the geographic locality



Why not Cache everything ?

- Caching everything is not a good idea for the following two reasons:
- Cost
- Accuracy
- Caching is meant to be fast and temporary
- It is often implemented with more expensive and less resilient hardware
- Caches are typically smaller than the primary data store
- Engineers must selectively choose which data to keep and remove (evict)
- **Caching Policy** or the selection process helps free up space for the most relevant data to be fetched in the near future

Caching Policies

- **First-In First-Out (FIFO):** Similar to a queue, this policy evicts whichever item was added longest ago and keeps the most recently added items.
- **Least Recently Used (LRU):** This policy keeps track of when items were last retrieved and evicts whichever item has not been accessed recently
- **Least Frequently Used (LFU):** This policy keeps track of how often items are retrieved and evicts whichever item is used least frequently, regardless of when it was last accessed.
- Caches can quickly become out of date from the actual state of the system
- The speed vs accuracy trade-off can be reduced by implementing an **eviction policy**
- Eviction policy is also known as **Limiting the Time To Live**
- Eviction policy of each cache entry before or after database writes (write-through vs write-behind)

Cache Coherence

- How to ensure appropriate cache consistence?
- A **write-through cache** updates the cache and main memory simultaneously, meaning there's no chance either can go out of date. It also simplifies the system.
- In a **write-behind cache,** memory updates occur asynchronously. This may lead to inconsistency, but it speeds things up significantly.
- **Cache-Aside / Lazy Loading:** Data is loaded into the cache on demand
- First, the application checks the cache for the requested data
- If it’s not there (called a **cache-miss**), the application fetches the data from the data store and updates the cache
- It keeps the data in the cache relatively relevant - as long as you choose a cache eviction policy and limited TTL combination that matches data access patterns.

Cache Policy - Comparisons

Cache Policy	Pros	Cons
Write-Through	Ensures consistency. A good option if your application requires more reads than writes	Writes are slow
Write-Behind	Speed (both read and write)	Risky. You are more susceptible to issues with consistency and you may lose data in a crash
Cache-aside / Lazy Loading	Simplicity and reliability. Also, only requested data is cached	Cache missed causes delays. May result in stale data

Additional Reading Material

- Facebook engineers published Scaling Memcache at Facebook, a now-famous paper detailing improvements to *Memcached*. The concept of *leases*, a Facebook innovation addressing the speed/accuracy problem with caching, is introduced here.
- This blog series written by Stack Overflow's Chief Architect on the company's approach to caching (and many other architectural topics) is both informative and fun.

Design Problem

Design Instagram