-------------------------------------------

Youtube, Instagram,Twitter

-------------------------------------------

**Non-Functional Requirements:**

1. The system should be **highly reliable, any video uploaded should not be lost**.
2. The system should be highly available.**Consistency can take a hit (in the interest of availability); if a user doesn’t see a video for a while, it should be fine.**
3. Users should have a real time experience **while watching videos and should not feel any lag.**

CAP theorem Application changes for different system

------------------------------------------

Facebook Messenger

1. Users should have **real-time chat experience with minimum laten**cy. Long poll ensures minimum latency, no handshake overheads.. 😀.
2. Our system should be highly consistent; users should be able to see the **same chat history on all their device**s. Hbase used.. 😀
3. Messenger’s high availability is desirable; we can't tolerate lower availability in the interest of consistency.
4. Should be reliable also. Chat messages should

           not be lost.

Storage -

Storing Large sized objects

-----------------------------------------

Youtube

------------

**Where would videos be stored?** Videos can be stored in a distributed file storage system like [HDFS](https://en.wikipedia.org/wiki/Apache_Hadoop#HDFS) or [GlusterFS](https://en.wikipedia.org/wiki/GlusterFS). 🤨

For videos - S3 is not preferable..???? 😜

Image

----------

We can store photos in a distributed file storage like [HDFS](https://en.wikipedia.org/wiki/Apache_Hadoop) or [S3](https://en.wikipedia.org/wiki/Amazon_S3). 😀

**Storing Metadata and General Storage**

**-------------------------------------------------------**

**Youtube**

**How should we efficiently manage read traffic?** We should segregate our read traffic from write traffic. Since we will have multiple copies of each video, we can distribute our read traffic on different servers. For metadata, we can have master-slave configurations where writes will go to master first and then gets applied at all the slaves. Such configurations can cause some staleness in data, e.g., **when a new video is added, its metadata would be inserted in the master first and before it gets applied to the slave our slaves would not be able to see it; and therefore it will be returning stale results to the user. This staleness might be acceptable in our system as it would be very short-lived and the user would be able to see the new videos after a few milliseconds.**

**Consistency not important. 😀**

**Special Case of Thumbnails**

**---------------------------------------**

**Where would thumbnails be stored?** There will be a lot more thumbnails than videos. If we assume that every video will have five thumbnails, we need to have a **very efficient storage system that can serve a huge read traffic**. There will be two consideration before deciding which storage system should be used for thumbnails:

1. Thumbnails are small files with, say, a maximum 5KB each.
2. **Read traffic for thumbnails will be huge compared to videos. Users will be watching one video at a time, but they might be looking at a page that has 20 thumbnails of other videos.**

Let’s evaluate storing **all the thumbnails on a disk**. Given that we have a huge number of files, we have to perform a lot of seeks to different locations on the disk to read these files. This is quite inefficient and will result in higher latencies.

[**Bigtable**](https://en.wikipedia.org/wiki/Bigtable)**can be a reasonable choice here as it combines multiple files into one block to store on the disk and is very efficient in reading a small amount of data. Both of these are the two most significant requirements of our service.** 😀

Keeping hot thumbnails in the cache will also help in improving the latencies and, given that thumbnails files are small in size, we can easily cache a large number of such files in memory.😀

Files small in size frequently to be accesssed, good idea to cache.

Another example of this is Captcha images.. pregenerated.. small in size don't have to generate, fetch again. Images were getting stored in Mysql datastore.

Used LRU cache for storing image data corresponding to cache. hot images applicable here..

😇

**Messenger**

**Which storage system we should use?** We need to have a database that can support a very high rate of small updates and also fetch a range of records quickly. This is required because we have a huge number of small messages that need to be inserted in the database and, while querying, a **user is mostly interested in sequentially accessing the messages.**

We cannot use RDBMS like MySQL or NoSQL like MongoDB because **we cannot afford to read/write a row from the database every time a user receives/sends a message. This will not only make the basic operations of our service run with high latency but also create a huge load on databases.**

Both of our requirements can be easily met with a wide-column database solution like [HBase](https://en.wikipedia.org/wiki/Apache_HBase). HBase is a **column-oriented key-value NoSQL database that can store multiple values against** **one key into multiple columns.** HBase is modeled after Google’s [BigTable](https://en.wikipedia.org/wiki/Bigtable) and runs on top of Hadoop Distributed File System ([HDFS](https://en.wikipedia.org/wiki/Apache_Hadoop)). HBase **groups data together to store new data in a memory buffer and, once the buffer is full, it dumps the data to the disk. This way of storage not only helps to store a lot of small data quickly but also fetching rows by the key or** **scanning ranges of rows. HBase is also an efficient database to** **store variable sized data, which is also required by our service.**

**Instagram**

A **straightforward a**pproach for **storing the above schema would be to use an RDBMS like MySQL since we require joins. But relational databases come with their challenges, especially when we need to scale them**. For details, please take a look at [SQL vs. NoSQL](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5728116278296576/).

We can store the above schema in a distributed key-value store to enjoy the benefits offered by NoSQL. All the **metadata related to photos can go to a table where the ‘key’ would be the ‘PhotoID’ ankd the ‘value’ would be an object containing PhotoLocation, UserLocation, CreationTimestamp, etc.**

We need to store r**elationships between users and photos, to know who owns which photo. We also need to store the list of people a user follows**. For both of these tables, we can use a **wide-column datastore like**[**Cassandra**](https://en.wikipedia.org/wiki/Apache_Cassandra)**. For the ‘UserPhoto’ table, the ‘key’ would be ‘UserID’ and the ‘value’ would be the list of ‘PhotoIDs’ the user owns, stored in different columns. We will have a similar scheme for the ‘UserFollow’ table.**

**Cassandra list data structure using collection for photos ids.**

Cassandra or key-value stores in general, always maintain a certain number of replicas to offer reliability. Also, in such d**ata stores, deletes don’t get applied instantly, data is retained for certain days (to support undeleting) before getting removed from the system permanently.**

Delete - records are tombstoned. Gets deleted after compaction.

**Separation of read /write Traffic**

**------------------------------------------------**

Photo uploads (or writes) can be **slow as they have to go to the disk,** whereas **reads will be faster, especially if they are being served from cache.**

Uploading users can consume all the available connections, as uploading is a slow process. This means that ‘reads’ cannot be served if the system gets busy with all the write requests. We should keep in mind that web servers have a connection limit before designing our system. If we assume that a web server can have a maximum of 500 connections at any time, then it can’t have more than 500 concurrent uploads or reads. To handle this bottleneck we can split reads and writes into separate services. We will have **dedicated servers for reads and different servers for writes to ensure that uploads don’t hog the system.**

Separating photos’ read and write requests will also allow us to scale and optimize each of these operations independently.

Image storage - s3, Google cloud, hdfs

A screen shot of a computer

Description automatically generated with low confidence

**Fault Tolerance - Replication of Services/Storage**

--------------------------------------------------------------------------------

**Losing files is not an option for our service. Therefore, we will store multiple copies of each file so that if one storage server dies we can retrieve the photo from the other copy present on a different storage server.**

This same principle also applies to other components of the system. If we want to have h**igh availability of the system, we need to have multiple replicas of services running in the system, s**o that if a few services die down the system still remains available and running. Redundancy removes the single point of failure in the system.

If o**nly one instance of a service is required to run at any point, we can run a redundant secondary copy of the service that is not serving any traffic, but it can take control after the failover when primary has a problem.**

**Replication as discussed in microservices course. Starting spring boot based app on different ports. Using ribbon for load balancing and eureka for naming. Zipkin for distributed log monitoring.**

Creating redundancy in a system can remove single points of failure and provide a **backup or spare functionality if needed in a crisis**. For example, if there are two instances of the same service running in production and one fails or degrades, the system can failover to the healthy copy. Failover can happen loautomatically or require manual intervention.

**Metadata Sharding**

**--------------------------**

Let’s discuss different schemes for metadata sharding:

**Instagram/Youtube/Tweets**

**a.** **Partitioning based on UserID** Let’s assume we shard based on the ‘UserID’ so that we can keep all photos of a user on the same shard. If one DB shard is 1TB, we will need four shards to store 3.7TB of data. Let’s assume for better performance and scalability we keep 10 shards.

So we’ll find the shard number by UserID % 10 and then store the data there. To uniquely identify any photo in our system, we can append **shard number with each PhotoID.**

**How can we generate PhotoIDs?** Each **DB shard can have its own auto-increment sequence for PhotoIDs and since we will append ShardID with each PhotoID, i**t will make it **unique throughout our system.**

**No need fore distributed uuid - each shard has its own uuid generator**

**What are the different issues with this partitioning scheme?**

1. How would we handle **hot users? Several people follow such hot users and a lot of other people s**ee any photo they upload.
2. Some users will have a **lot of photos compared to others, thus making a non-uniform distribution of storage.**
3. What if we cannot store a**ll pictures of a user on one shard? If we distribute photos of a user onto multiple shards will it cause higher latencies?**
4. Storing all photos of a user on one shard can cause issues like **unavailability of all of the user’s data if that shard is down or higher latency** if it is serving high load etc.

**b.** **Partitioning based on PhotoID** If we can generate unique PhotoIDs first and then find a shard number through “PhotoID % 10”, the above problems will have been solved. We would not need to append ShardID with PhotoID in this case as PhotoID will itself be unique throughout the system.

**How can we generate PhotoIDs?** Here we cannot have an auto-incrementing sequence in each shard to define PhotoID because we need to know PhotoID first to find the shard where it will be stored. One solution could be that we **dedicate a separate database instance to generate auto-incrementing IDs. If our PhotoID can fit into 64 bits, we can define a table containing only a 64 bit ID field. So whenever we would like to add a photo in our system, we can insert a new row in this table and take that ID to be our PhotoID of the new photo.**

**Wouldn’t this key generating DB be a single point of failure?** Yes, it would be. A workaround for that could be defining two such databases with one generating even numbered IDs and the other odd numbered. For the MySQL, the following script can define such sequences:

KeyGeneratingServer1

auto-increment-increment = 2

auto-increment-offset = 1

KeyGeneratingServer2:

auto-increment-increment = 2

auto-increment-offset = 2

We can put a **load balancer in front of both of these databases to round robin between them and to deal with downtime. Both these servers could be out of sync with one generating more keys than the other, but this will not cause any issue in our system.** **We can extend this design by defining separate ID tables for Users, Photo-Comments, or other objects present in our system.**

**Alternately,** we can **implement a ‘key’ egeneration scheme similar to what we have** discussed in [Designing a URL Shortening service like TinyURL](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5668600916475904).

**Keeping Shards on same server initially, Separater Servers Later without loss of hash location Logic**

**--------------------------------------------------------------------------------------------------------------------------------------------**

**How can we plan for the future growth of our system?** We can have a large number of **logical partitions to accommodate future data growth, such that in the beginning, multiple logical partitions reside on a single physical database server. Sin**ce each database server can have multiple database instances on it, we can have **separate databases for each logical partition on any server**. So whenever we feel that a particular database server has a lot of data, we can migrate some logical partitions from it to another server. We can maintain a config file (or a separate database) that can map our logical partitions to database servers; this will enable us to move partitions around easily. Whenever we want to move a partition, we only have to update the config file to announce the change.

Using autosharding of hbase.

Elasticsearch shards

**Feed Generation  - Instagram Sample**

**--------------------------------------------------**

To create the News Feed for any given user, we need to fetch the latest, most popular and relevant photos of the people the user follows.

For simplicity, let’s assume we **need to fetch top 100 photos for a user’s News Feed**. Our application server will first get a list of people the user follows and then fetch **metadata info of latest 100 photos from each user. I**n the final step, the server will submit all these photos to our ranking algorithm which will **determine the top 100 photos (based on recency, likeness, etc.) and return them to the user**. A possible problem with this approach would be higher latency as we have to query multiple tables and perform sorting/merging/ranking on the results. To improve the efficiency, we can pre-generate the News Feed and store it in a separate table.

Possible use of Kafka streams / spark streams in Instagram feeds.

**Pre-generating the News Feed:** We can have dedicated servers tha**t are continuously generating users’ News Feeds and destoring them in a ‘UserNewsFeed’ table. So whenever any user needs the latest photos for their News Feed, we will simply query this table and return the results to the user.**

Whenever these servers need to generate the News Feed of a user, they will first query the UserNewsFeed table to find the last time the News Feed was generated for that user. Then, new **News Feed data will be generated from that time onwards (following the steps mentioned above).**

**Times at which feeds are generated for different users. Depending on their browsing behaviour. Certain fixed times + when their visiting tendency is high. This tendency may change.**

**What are the different approaches for sending News Feed contents to the users?**

**1. Pull:** Clients can pull the News Feed contents from the server on a regular basis or manually whenever they need it. Possible problems with this approach are a) **New data might not be shown to the users until clients issue a pull request** b) Most of the time pull requests **will result in an empty response if there is no new data.**

**2. Push:** Servers can push new data to the users as soon as it is available. To efficiently manage this, users have to maintain a [Long Poll](https://en.wikipedia.org/wiki/Push_technology#Long_polling) request with the server for receiving the updates. A possible problem with this approach is, a user who follows a lot of people or a celebrity user who has millions of followers; in this case, the s**erver has to push updates quite frequently.**

**Updates are being pushed frequently as lot of users are posting as user follows a large number of people.**

**Celebrity user - millions of followers. Push updates frequently as different people will be online at different times.**

**3. Hybrid:** We can adopt a hybrid approach. We can move all the users who have a high number of follows to a pull-based model and only push data to those users who have a few hundred (or thousand) follows. Another approach could be that the server pushes updates to all the users not more than a certain frequency, letting users with a lot of follows/updates to regularly pull data.

For a detailed discussion about News Feed generation, take a look at [Designing Facebook’s Newsfeed](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5641332169113600).

Pull feeds from cached for the first time. Feed may be outdated, not present in cache, can this scenario arise

Automatic updates in background iusing long poll.

When user comes online, pulls  feeds automatically, feed updates keep on happening in background via long poll.

Won't have to poll server for friends updates,  keeps on syncing online while user is doing other work.

User pulls feed from the cache,feed generated again, gets inserted in table. Server pushes new feed periodically at a certain frequency.

**Messages Handling - Chat Server Sample**

---------------------------------------------------------------------------

**How would we efficiently send/receive messages?** To send messages, a user needs to connect to the server and post messages for the other users. To get a message from the server, the user has two options:

1. **Pull model:** Users can periodically ask the server if there are any new messages for them.
2. **Push model:** Users can keep a connection open with the server and can depend upon the server to notify them whenever there are new messages.

If we go with our first approach, then the **server needs to keep track of messages that are still waiting to be delivered, a**nd as soon as **the receiving user connects to the server to ask for any new message,** **the server can return all the pending messages.** **To minimize latency for the user, they have to check the server quite frequently, and most of the time they will be getting an empty response if there are no pending message. This will waste a lot of resources and does not look like an efficient solution.**

If we go with our second approach, where all the active users keep a connection open with the server, then as soon as the server receives a message it can immediately pass the message to the intended user. This way, the **server does not need to keep track of the pending messages,** and **we will have minimum latency, as the messages are delivered instantly on the opened connection.**

**How will clients maintain an open connection with the server?** We can use HTTP [**Long Polling**](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5715426797420544)**or**[**WebSockets**](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5715426797420544). In long polling, clients can request information from the server with the expectation that the server may not respond immediately. If the server has no new data for the client when the poll is received, instead of sending an empty response, the server holds the request open and waits for response information to become available. Once it does have new informationnn, the server immediately sends the response to the client, completing the open request. Upon receipt of the server response, the client can immediately issue another server request for future updates. This gives a lot of improvements in latencies, throughputs, and performance. The long polling request can timeout or can receive a disconnect from the server, in that case, the client has to open a new request.

**Database Storage in case of Mesenger - In case of other systems, how metadata is ingested in Nosql Storage.**

**Any processing of data strema is required beceofre ingestion in  no sql storage.**

**Storing and retrieving the messages from the database**

Whenever the chat server receives a new message, it needs to store it in the database. To do so, we have two options:

1. **Start a separate thread, which will work with the database to store the message.**
2. **Send an asynchronous request to the database to store the message.**

We have to keep certain things in mind while designing our database:

1. How to efficiently work with the database connection pool.
2. How to retry failed requests.
3. Where to log those requests that failed even after some retries**. --> Trying to store data in database, database row, element could not be inserted, had to implement retries for that.**
4. How to retry these logged requests (that failed after the retry) when all the issues have resolved.

Amount of data to keep in feeds - Dep  User browsing behaviour - how much feed objects should be there for a user.

Amazon sqs - distributed queue over network.

Used by dropbox - over network.

Blocking queue used internally by application for example by chat application for storing messages in database.

**Caching-**

We can introduce a cache for database servers to cache hot tweets and users. We can use an off-the-shelf solution like Memcache that can store the whole tweet objects. Application servers, before hitting database, can quickly check if the cache has desired tweets. Based on clients’ usage patterns we can determine how many cache servers we need.

**Which cache replacement policy would best fit our needs?** When the cache is full and we want to replace a tweet with a newer/hotter tweet, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system. Under this policy, we discard the least recently viewed tweet first.

**How can we have a more intelligent cache?** If we go with 80-20 rule, that is 20% of tweets generating 80% of read traffic which means that certain tweets are so popular that a majority of people read them. This dictates that we can try to cache 20% of daily read volume from each shard.

For each shard - 20% of traffic.

**What if we cache the latest data?** Our service can benefit from this approach. Let’s say if 80% of our users see **tweets from the past three days only; we can try to cache all the tweets from the past three days. Let’s say we have dedicated cache servers that cache all the tweets from all the users from the past three days.**

Recency + views combination

Stored in ehcache using consistent hashing.

Different expiration times based on recency.. + view limit.. called cache regions. Data stored and fetched from cache via http rest apis. Server capacity can alter based on cache region and data and latency requirements.

As estimated above, **we are getting 100 million new tweets or 30GB of new data every day (without photos and videos). If we want to store all the tweets from last three days, we will need less than 100GB of memory.** This data can easily fit into one server, but we should replicate it onto multiple servers to distribute all the read traffic to reduce the load on cache servers. So whenever we are generating a user’s timeline, we can ask the cache servers if they have all the recent tweets for that user. If yes, we can simply return all the data from the cache.

If we don’t have enough tweets in the cache, we have to query the backend server to fetch that data. On a similar design, we can try caching photos and videos from the last three days.

Our cache would be like a hash table where ‘key’ would be ‘OwnerID’ and ‘value’ would be a doubly linked list containing all the tweets from that user in the past three days. Since we want to retrieve the most recent data

first, we can always insert new tweets at the head of the linked list, which means all the older tweets will be near the tail of the linked list. Therefore, we can remove tweets from the tail to make space for newer tweets.

**We can have dedicated servers that are continuously generating users’ newsfeed and storing them in memory. So, whenever a user requests for the new posts for their feed, we can simply serve it from the pre-generated, stored location. Using this scheme, user’s newsfeed is not compiled on load, but rather on a regular basis and returned to users whenever they request for it.**

**Structure of  Feeds**

Whenever these servers need to generate the feed for a user, they will first query to see what was the last time the feed was generated for that user. Then, new feed data would be generated from that time onwards. We can store this data in a hash table where the “key” would be UserID and “value” would be a STRUCT like this:

Struct {

    LinkedHashMap<FeedItemID, FeedItem> feedItems;

    DateTime lastGenerated;

}

**How user interact with Feeds to get incremental Data**

**We can store FeedItemIDs in a data structure similar to**[**Linked HashMap**](https://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html)**or**[**TreeMap**](https://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html)**, which can allow us to not only jump to any feed item but also iterate through the map easily. Whenever users want to fetch more feed items, they can send the last FeedItemID they currently see in their newsfeed, we can then jump to that FeedItemID in our hash-map and return next batch/page of feed items from there.**

**How many feed items should we store in memory for a user’s feed?** I**nitially, we can decide to store 500 feed items per user, but this number can be adjusted later based on the usage pattern. For example, if we assume that one page of a user’s feed has 20 posts and most of the users never browse more than ten pages of their feed, we can decide to store only 200 posts per user.** For any user who wants to see more posts (more than what is stored in memory), we can always query backend servers.

**Should we generate (and keep in memory) newsfeeds for all users?** There will be a lot of users that don’t login frequently. Here are a few things we can do to handle this;

People who don't login frequently will be evicted by LRU Policy

**1) a more straightforward approach could be, to use a LRU based cache that can remove users from memory that haven’t accessed their newsfeed for a long time**

**When to generate Feed for users based on Login pattern**

**2)a smarter solution can figure out the login pattern of users to pre-generate their newsfeed, e.g., at what time of the day am user is active and which days of the week does a user access their newsfeed? etc.**

Send emails to user when user feed is generated. Mechanism to avoid customer churn.

**Attack to evict Frequent entries by recent burst of recent queries - avoided using 2Q and Arch Cache**

**Feed Caching Format  -**

How does server combined metadata with photos,videos to generate tweet.

Feeds are available in json, xml format.

Photos and videos comes as url, this full JSON, XML is cached corresponding to userIds.

Video player, Images in iframes. Tweet templates are available rendered from json,xml.

Use of protocol buffers for faster feed generation.

Less data transferred over network, faster feeds generation.

Reason #3: Less Boilerplate Code

**In messenger caching chat history is not a concern - size of cache is on that large**

We can c**ache a few recent messages (say last 15) in a few recent conversations that are visible in a user’s viewport (say last 5)**. **(caching latest messages just like caching recent tweets)** Since we decided to store all of the user’s messages on one shard, the cache for a user should entirely reside on one machine too.

**Why use gRPC?**

In the microservice world at WePay, every microservice was using REST with JSON payload as a standard way of communicating with one another. REST has its advantages for service-to-service communication:

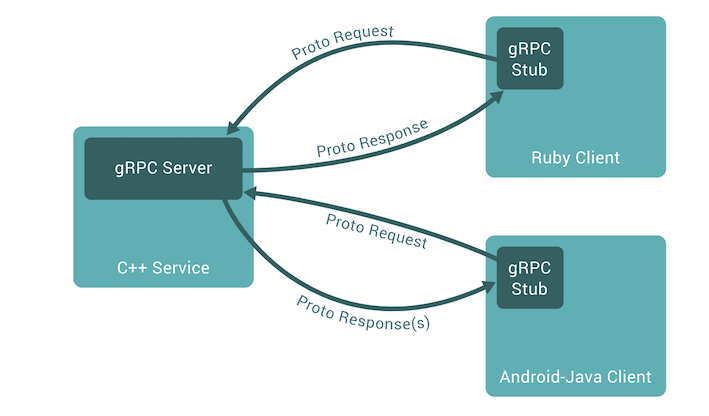
* JSON payloads are easy to understand.
* As a mature serialization format, there are lots of frameworks available for building services using REST.
* REST is a very popular standard and is language and platform agnostic.

As the number of microservices increase in the infrastructure, the service graph becomes more complex, and as a result, there are pain points and limitations in using REST for communications:

* For example, in Figure 1, When a client makes a call to service X, in turn, service X could make calls to other services to provide a response to the client.
* For every REST call between the services, a new connection is established and there is the overhead of SSL handshake. This would cause an increase in overall latency.
* The client has to be implemented for each service in every language required and also has to be updated whenever there is a change in the API definition.
* JSON payloads are simple messages that have relatively slower serialization and deserialization performance.

With these pain points in mind, gRPC seemed to be a good option for us to improve our microservices communications in our infrastructure.

In gRPC, a client application can directly call methods on a server application on a different machine as if it were a local object, making it easier to create distributed services.

Diagram

Description automatically generated

*Figure 3: gRPC clients sending the proto request to gRPC service and receives the proto response (*[*source*](https://grpc.io/docs/guides/)*)*

gRPC has its own advantages:

* Similar to RPC protocols, gRPC is based on the idea of defining a service and its methods that can be called remotely with their parameters and return types.
* Building low latency and highly scalable microservice call graphs
* gRPC uses [Protocol Buffers](https://developers.google.com/protocol-buffers/docs/overview) by default, which provide better performance when compared to JSON.
* Ability to auto-generate client stubs in several languages, which reduces the responsibility of building and maintaining client libraries.

**Influx DB**

*An open-source distributed time series database with no external dependencies*. InfluxDB is a scalable datastore for **metrics, events, and real-time analytics**

**System APIs**

**Back of the Envelope calculation**

**Youtube**

**Video** **deduplication**

**---------------------------**

With a huge number of users uploading a massive amount of video data our service will have to deal with widespread video duplication. Duplicate videos often differ in aspect ratios or encodings, can contain overlays or additional borders, or can be excerpts from a longer original video. The proliferation of duplicate videos can have an impact on many levels:

1. Data Storage: We could be wasting storage space by keeping multiple copies of the same video.
2. Caching: Duplicate videos would result in degraded cache efficiency by taking up space that could be used for unique content.
3. Network usage: Duplicate videos will also increase the amount of data that must be sent over the network to in-network caching systems.
4. Energy consumption: Higher storage, inefficient cache, and network usage could result in energy wastage.

For the end user, these inefficiencies will be realized in the form of duplicate search results, longer video startup times, and interrupted streaming.

For our service, deduplication makes most sense early; when a user is uploading a video as compared to post-processing it to find duplicate videos later. I**nline deduplication will save us a lot of resources that can be used to encode, transfer, and store the duplicate copy of the video.** As soon as any user starts uploading a video, our service can run video matching algorithms (e.g., [Block Matching](https://en.wikipedia.org/wiki/Block-matching_algorithm), [Phase Correlation](https://en.wikipedia.org/wiki/Phase_correlation), etc.) to find duplications. If we already have a copy of the video being uploaded, we can either stop the upload and use the existing copy or continue the upload and use the newly uploaded video if it is of higher quality. If the newly uploaded video is a subpart of an existing video or, vice versa, we can intelligently divide the video into smaller chunks so that we only upload the parts that are missing.

**DropBox**

**8. Data Deduplication**

Data deduplication is a technique used for eliminating duplicate copies of data to improve storage utilization. It can also be applied to network data transfers to reduce the number of bytes that must be sent. For each new incoming chunk, we can calculate a hash of it and compare that hash with all the hashes of the existing chunks to see if we already have the same chunk present in our storage.

We can implement deduplication in two ways in our system:

**a. Post-process deduplication**

With post-process deduplication, new chunks are first stored on the storage device and later some process analyzes the data looking for duplication. The benefit is that clients will not need to wait for the hash calculation or lookup to complete before storing the data, thereby ensuring that there is no degradation in storage performance. Drawbacks of this approach are 1) We will unnecessarily be storing duplicate data, though for a short time, 2) Duplicate data will be transferred consuming bandwidth.

**b. In-line deduplication**

**Alternatively, deduplication hash calculations can be done in real-time as the clients are entering data on their device. If** our system identifies a chunk that it has already stored, only a reference to the existing chunk will be added in the metadata, rather than a full copy of the chunk. This approach will give us optimal network and storage usage.

**API rate Limiter -**

**Race conditions in reading same data concurrently, value of counter resulting in correct capping, serving extra requests.**

**Usage of write back cache, data is being written to write back cache, written to permanent storage periodically at fixed intervals, flush to disk.**

A**pplication servers can quickly check if the cache has the desired record before hitting backend servers. Our rate limiter can significantly benefit from the Write-back cache by updating all counters and timestamps in cache only. The write to the permanent storage can be done at fixed intervals. This way we can ensure minimum latency added to the user’s requests by the rate limiter. The reads can always hit the cache first; which will be extremely useful once the user has hit their maximum limit and the rate limiter will only be reading data without any updates.**

**Least Recently Used (LRU) can be a reasonable cache eviction policy for our system.**

**Server Monitoring Health checking components and Alerts  - Zabbix**

**Sharding of data based on creation time. - 5 minute -1 minute shards granularity.**

**60 shards for per minute sharding, 12 shards for 5 minute sharding**

**Log monitoring Systems - Apache web logs, data points of tweets streamed to Metadata Storage.**

**Scaling out AWS service in general**

**---------------------------------------------------**

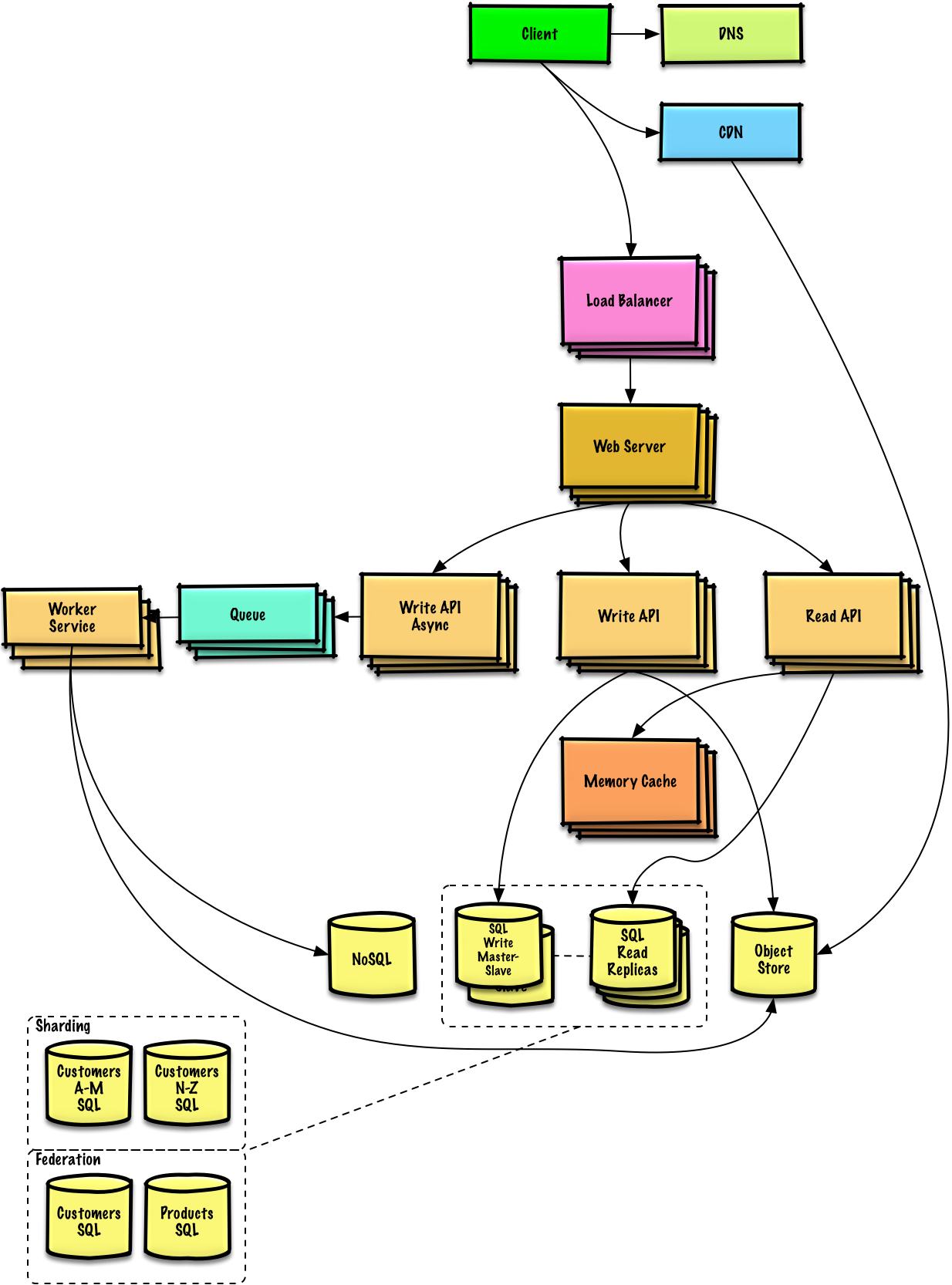
We can further separate out our [Application Servers](https://github.com/donnemartin/system-design-primer#application-layer) to allow for independent scaling. **Batch processes or computations that do not need to be done in real-time can be done**[**Asynchronously**](https://github.com/donnemartin/system-design-primer#asynchronism)**with Queues and Workers:**

* For example, in a photo service, the photo upload and the thumbnail creation can be separated:
  + Client uploads photo
  + Application Server puts a job in a Queue such as SQS.
  + **DIfferent worker threads probably an executor pool consuming the item from queue.**
  + **The Worker Service on EC2 or Lambda pulls work off the Queue then:**
    - Creates a thumbnail
    - Updates a Database
    - Stores the thumbnail in the Object Store
* If our MySQL Database starts to grow too large, we might consider only storing a limited time period of data in the database, while storing the rest in a data warehouse such as Redshift
  + A data warehouse such as Redshift can comfortably handle the constraint of 1 TB of new content per month
* With 40,000 average read requests per second, read traffic for popular content can be addressed by scaling the **Memory Cache, which is also useful for handling the unevenly distributed traffic and traffic spikes**
  + The**SQL Read Replicas might have trouble handling the cache misses, we'll probably need to employ additional SQL scaling patterns**
* 400 average writes per second (with presumably significantly higher peaks) might be tough for a single SQL Write Master-Slave, also pointing to a need for additional scaling techniques

SQL scaling patterns include:

* [**Federation**](https://github.com/donnemartin/system-design-primer#federation)
* [**Sharding**](https://github.com/donnemartin/system-design-primer#sharding)
* [**Denormalization**](https://github.com/donnemartin/system-design-primer#denormalization)
* [**SQL Tuning**](https://github.com/donnemartin/system-design-primer#sql-tuning)

To further address the **high read and write requests, we should also consider moving appropriate data to a**[**NoSQL Database**](https://github.com/donnemartin/system-design-primer#nosql)**such as DynamoDB.**

Graphical user interface

Description automatically generated

* **In addition to adding and scaling a Memory Cache, MySQL Read Replicas can also help relieve load on the MySQL Write Master**
* Add logic to **Web Server to separate out writes and reads**
* Add **Load Balancers in front of MySQL Read Replicas** (not pictured to reduce clutter)
* **Most services are read-heavy vs write-heavy**
* Move the following data to a [Memory Cache](https://github.com/donnemartin/system-design-primer#cache) such as Elasticache to reduce load and latency:
  + Frequently accessed content from MySQL
    - First, try to configure the MySQL Database cache to see if that is sufficient to relieve the bottleneck before implementing a Memory Cache
  + Session data from the Web Servers
    - The Web Servers become stateless, allowing for Autoscaling
  + Reading 1 MB sequentially from **memory takes about 250 microseconds**, **while reading from SSD takes 4x** and **from disk takes 80x longer.**[**1**](https://github.com/donnemartin/system-design-primer#latency-numbers-every-programmer-should-know)
* Add [MySQL Read Replicas](https://github.com/donnemartin/system-design-primer#master-slave-replication) to reduce load on the write master
* Add more Web Servers and Application Servers to improve responsiveness

Use [Horizontal Scaling](https://github.com/donnemartin/system-design-primer#horizontal-scaling) to handle increasing loads and to address single points of failure

* Add a [**Load Balancer**](https://github.com/donnemartin/system-design-primer#load-balancer)**such as Amazon's ELB or HAProxy**
  + ELB is highly available
  + If you are configuring your own Load Balancer, setting up multiple servers in [active-active](https://github.com/donnemartin/system-design-primer#active-active) or [active-passive](https://github.com/donnemartin/system-design-primer#active-passive) in multiple atreemapvailability zones will improve availability
  + **Terminate SSL on the Load Balancer to reduce computational load on backend servers and to simplify certificate administration**
* Use **multiple Web Servers spread out over multiple availability zones**
* Use multiple MySQL instances in [Master-Slave Failover](https://github.com/donnemartin/system-design-primer#master-slave-replication) mode across multiple availability zones to improve redundancy.

**Important Numbers**

* **Read sequentially from disk at 30 MB/s**
* **Read sequentially from 1 Gbps Ethernet at 100 MB/s**
* **Read sequentially from SSD at 1 GB/s**
* **Read sequentially from main memory at 4 GB/s**
* 6-7 world-wide round trips per second
* 2,000 round trips per second within a data center

Security is a broad topic. Unless you have considerable experience, a security background, or are applying for a position that requires knowledge of security, you probably won't need to know more than the basics:

* **Encrypt in transit and at rest.**
* **Sanitize all user inputs or any input parameters exposed to user to prevent**[**XSS**](https://en.wikipedia.org/wiki/Cross-site_scripting)**and**[**SQL injection**](https://en.wikipedia.org/wiki/SQL_injection)**.**

**Security of APIs**

* **Limit requests (Throttling) to avoid DDoS / brute-force attacks.**
* **Use HTTPS on server side to avoid MITM (Man in the Middle Attack).**
* Don't auto-increment IDs. Use UUID instead.

**Processing**

* **Check if all the endpoints are protected behind authentication to avoid broken authentication process.**
* **User own resource ID should be avoided. Use /me/orders instead of /user/654321/orders.**
* **Don't auto-increment IDs. Use UUID instead.**
* **If you are parsing XML files, make sure entity parsing is not enabled to avoid XXE (XML external entity attack).**
* If you are parsing XML files, make sure entity expansion is not enabled to avoid Billion Laughs/XML bomb via exponential entity expansion attack.
* Use a CDN for file uploads.
* If you are dealing with huge amount of data, use Workers and Queues to process as much as possible in background and return response fast to avoid HTTP Blocking.

**Task queues**

Tasks queues receive tasks and their related data, runs them, then delivers their results. They can support scheduling and can be used to run computationally-intensive jobs in the background.

Celery has support for scheduling and primarily has python support.

**All task queue tasks are performed asynchronously. The application that creates the task hands it off to the queue. The originating application is not notified whether or not the task completes, or if it was successful.**

**If a worker fails to process a task, the Task Queue service provides the queue with a retry mechanism, so the task can be retried a finite number of times.**

**Back-pressure**

[Back pressure](http://mechanical-sympathy.blogspot.com/2012/05/apply-back-pressure-when-overloaded.html) can help by limiting the queue size, thereby maintaining a high throughput rate and good response times for jobs already in the queue. Once the queue fills up, clients get a server busy or HTTP 503 status code to try again later. Clients can retry the request at a later time, perhaps with [exponential backoff](https://en.wikipedia.org/wiki/Exponential_backoff).

**Using machine learning for caching objects - For example specific category, tag based tweets.**

**Consistent hashing implementatiom.**

[https://www.acodersjourney.com/system-design-interview-consistent-hashing/](https://www.acodersjourney.com/system-design-interview-consistent-hashing)

[https://medium.com/@dgryski/consistent-hashing-algorithmic-tradeoffs-ef6b8e2fcae8](https://medium.com/%40dgryski/consistent-hashing-algorithmic-tradeoffs-ef6b8e2fcae8)

Hash function keys -  generates a uniform distribution of keys across a circle. Ring hashing better than [hashkey%N ]  Adding/removing a server changes entire key distribution.

Use sorted map first function to locate server node corresponding to the key.

Locate nearest position in a circle.

public static boolean isConnected(String uri)  {

    boolean isConnected = false;

    try {

    //isConnected =  invoke(targetURL);

    HttpURLConnection  httpURLConnection = get(uri);

    //httpURLConnection.

} catch (Exception e) {

// TODO: handle exception

e.printStackTrace();

}

    return isConnected;

    }

public static HttpURLConnection get(String uri) throws IOException, ParserConfigurationException, SAXException {

HttpURLConnection urlConnection = null;

try {

URL u = new URL(uri);

urlConnection = (HttpURLConnection) u.openConnection();

urlConnection.setUseCaches(false);

urlConnection.setRequestMethod("GET");

urlConnection.setReadTimeout(ehCacheReadTimeOut);

urlConnection.setConnectTimeout(ehCacheConnectionTimeOut);

int status = urlConnection.getResponseCode();

} catch (Exception e) {

e.printStackTrace();

return null;

}

return urlConnection;

}

private static final Logger log = LoggerFactory.getLogger(ConsistentHashingProcessor.class);

private HashingAlgo hashingAlgorithm = null;

private int numberOfReplicas;

private String serverIndices;

// private final SortedMap<Integer, T> circle = new TreeMap<Integer, T>();

private final SortedMap<Long, T> circle = new TreeMap<Long, T>();

/\*public ConsistentHashingProcessor(HashingAlgo hashFunction, int numberOfReplicas, Collection<T> nodes) {

this.hashingAlgorithm = hashFunction;

this.numberOfReplicas = numberOfReplicas;

for (T node : nodes) {

add(node);

}

}\*/

/\*\*

\* This method called from ServerNodeLocator to initialise but need to figure out how to handle.

\*

\* @param nodes

\* @throws Exception

\*/

public void spreadNodeOnCircle(Collection<T> nodes) throws Exception {

log.debug("ConsistentHashing.init method to put server in  ..");

if (nodes != null) {

for (T node : nodes) {

add(node);

}

} else {

throw new IllegalArgumentException("Servers list can't be blank. ");

}

}

/\*\*

\* Release the resources that occupied.

\*

\* @throws Exception

\*/

public void destroy() throws Exception {

// TODO : release the resources that occupied.

log.debug("Resources released..... ");

}

/\*\*

\* This method is used to add the server node to the cache server cluster,

\* so that while getting of server index consider and get serverNode for added.

\*

\* @param node

\*/

public void add(T node) {

//if(log.isDebugEnabled())log.debug("adding into Circle dot....");

for (int i = 0; i < numberOfReplicas; i++) {

//if (log.isDebugEnabled()) log.debug(hashingAlgorithm.getHashCode(node.toString() + i ) + "");

circle.put(hashingAlgorithm.getHashCode(node.toString() + i), node);

}

}

/\*\*

\* Thais method is used to remove the server node from the cluster so that

\* while getting of server index do not return the server index by the key.

\*

\* @param node

\*/

public void remove(T node) {

//if(log.isDebugEnabled()) {log.debug("REMOVING SERVER NODE: ");}

for (int i = 0; i < numberOfReplicas; i++) {

//if (log.isDebugEnabled()) log.debug("Node Removed : " + hashingAlgorithm.getHashCode(node.toString() + i));

circle.remove(hashingAlgorithm.getHashCode(node.toString() + i));

}

}

/\*\*

\* This method is used to get the SeverNode by key.

\*

\* @param key

\* @return

\*/

public T get(String key) {

if (circle.isEmpty()) {

return null;

} else {

// log.debug("size of Circle : " + circle.size());

}

long hash = hashingAlgorithm.getHashCode(key);

if (!circle.containsKey(hash)) {

// SortedMap<Integer, T> tailMap = circle.tailMap(hash);

SortedMap<Long, T> tailMap = circle.tailMap(hash);

hash = tailMap.isEmpty() ? circle.firstKey() : tailMap.firstKey();

}

return circle.get(hash);

}

/\*\*

\* @return the hashingAlgorithm

\*/

public HashingAlgo getHashingAlgorithm() {

return hashingAlgorithm;

}

/\*\*

\* @param hashingAlgorithm the hashingAlgorithm to set

\*/

public void setHashingAlgorithm(HashingAlgo hashingAlgorithm) {

this.hashingAlgorithm = hashingAlgorithm;

}

/\*\*

\* @return the numberOfReplicas

\*/

public int getNumberOfReplicas() {

return numberOfReplicas;

}

/\*\*

\* @param numberOfReplicas the numberOfReplicas to set

\*/

public void setNumberOfReplicas(int numberOfReplicas) {

this.numberOfReplicas = numberOfReplicas;

}

/\*\*

\* @return the serverIndices

\*/

public String getServerIndices() {

return serverIndices;

}

/\*\*

\* @param serverIndices the serverIndices to set

\*/

public void setServerIndices(String serverIndices) {

this.serverIndices = serverIndices;

}

public SortedMap<Long, T> getCircleNodes() {

return circle;

}

/\*\*

\* This method is used to call by timer to check whether any down server up or not, if up then add into

\* consistentHashing algorithm list of servers so that it can be operational. This method will call by timer

\* which will manage internally.

\*

\* @param host

\*/

private void addNodeInConsistentHashing(String host) {

long startTime = System.currentTimeMillis();

long executionTime = 0;

try {

executionTime = System.currentTimeMillis() - startTime;

Map<String, Integer> downServerMap = TapServerConnection.downServersMap;

if (downServerMap != null && downServerMap.size() > 0) {

for (Map.Entry<String, Integer> entry : downServerMap.entrySet()) {

if (entry.getKey() != null) {

if(logger.isDebugEnabled()){logger.debug(" check server connnection and add server into Cluster, serverIP =" + host);};

boolean isConnected = HttpUtil.isConnected(TapServerConnection.PREFIX\_URL + host + TapServerConnection.SUFFIX\_URL);

if (isConnected) {

ehCacheService.addServerNode(host);

TapServerConnection.downServersMap.remove(host);

} else {

logger.debug("connection failed serverIp: " +  host);

}

}

}

}

} catch (Exception e) {

logger.error("exception caught inside pushing nodeServer in consistentHashing in addNodeInConsistentHashing method, for hostid = " + host + " , msg : " + e.getMessage());

} finally {

try {

stopAddNodeTimer(host);

// create another timer if downserverMap contains that  host.

if (TapServerConnection.downServersMap.containsKey(host)) {

timerMap.put(host, new Timer());

initAddNodeTimer(host);

}

} catch (Exception e) {

logger.error("exception caught while re-creating timer with period ZERO and with delaytime Can in cache for host= " + host

+ " , nodeServer pushing executionTime in ms = " + executionTime + " msg : " + e.getMessage());

}

}

}

/\*\*

\* This method is used to initialize the Timer and schedule the timerTask to add the node into consistent hashing circle.

\*

\* @param host

\* @param delayTime

\* @throws Exception

\*/

private void initAddNodeTimer(final String host) throws Exception {

if (logger.isDebugEnabled()) logger.debug("initAddNodeTimer method start...");

try {

timerMap.get(host).schedule(new TimerTask() {

public void run() {

logger.debug("addNodeTimer invoking service repeatedly. for host=" + host);

addNodeInConsistentHashing(host);

}

}, delayTime);

} catch (Exception e) {

//e.printStackTrace();

logger.error("Exception caught during initializing content timer for the host=" + host);

}

}

**Use of schedular service in consistent Hashing ??**

**GRPC - Server streaming big data - chat, live stream**

**GRPC - Client streaming big data - uploading, server processing is expensive, don't really epxect response from server.**

**GRPC - Bi directional streaming data - client server, need to communicate asychronously. Example chat**

<https://medium.com/congruence-labs/back-pressure-implementation-aws-sqs-polling-from-a-sharded-akka-cluster-running-on-kubernetes-56ee8c67efb>

<https://medium.com/netflix-techblog/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40>

**Autoscale worker threads depending on worker load and queue depth.**

<https://dzone.com/articles/websockets-vs-long-polling>

<https://www.callicoder.com/spring-boot-websocket-chat-example/>

**WebSockets: Pros and Cons**

**Pros**

* WebSockets keeps a unique connection open while eliminating latency problems that arise with Long Polling.
* WebSockets generally do not use XMLHttpRequest, and, as such, headers are not sent every-time we need to get more information from the server. This, in turn, reduces the expensive data loads being sent to the server.

**Cons**

* WebSockets don’t automatically recover when connections are terminated — this is something you need to implement yourself, and is part of the reason why there are many[client-side libraries](https://www.ably.io/download) in existence.
* Browsers older than 2011 aren’t able to support WebSocket connections — but this is increasingly less relevant.

**Why the WebSocket Protocol Is the Better Choice**

Generally, WebSockets will be the better choice.

Long polling is much more resource intensive on servers whereas WebSockets have an extremely lightweight footprint on servers. Long polling also requires many hops between servers and devices. And these gateways often have different ideas of how long a typical connection is allowed to stay open. If it stays open too long something may kill it, maybe even when it was doing something important.

Why you should build with WebSockets:

* Full-duplex asynchronous messaging. In other words, both the client and the server can stream messages to each other independently.
* WebSockets pass through most firewalls without any reconfiguration.
* Good security model (origin-based security model).

<https://stackoverflow.com/questions/9985471/how-efficient-is-apache-tomcat-for-long-polling>

Recent versions of apache tomcat support [comet](http://www.ibm.com/developerworks/web/library/wa-cometjava/) which allows non blocking IO to allow tomcat to scale to a large number of requests.

<https://github.com/scotch-io/go-realtime-chat/blob/master/src/main.go>

Can dropbox use websockets..

webhooks are used for server to server communication.. for example if a user unsubscribes from a service, a server side event occurs and all the rest of the services get notified via webhook.

Server communicates occasionaly use rest calls, frequent bidirection communication use websockets.

Don't use polling for change in data or state.

amazon sqs duplex stream

<https://www.digitalcitizen.life/what-is-p2p-peer-to-peer>

if you download the same file through a peer-to-peer network, using a BitTorrent website as a starting point, the download is performed differently. The file is downloaded to your computer in bits and parts that come from many other computers in the P2P network that already have that file. At the same time, the file is also sent (uploaded) from your computer to others which ask for it. This situation is similar to a two-way road: the file is like multiple small cars that come to your PC but also leave to others when they are requested.

<https://medium.com/textileio/build-a-decentralized-chat-app-with-knockout-and-ipfs-fccf11e8ce7b>

Distributed Decentralized systems :)

----------------------------------------------------

<https://github.com/croqaz/awesome-decentralized>

<https://blockply.com/advertisement/basic-attention-token>

<https://www.guidingtech.com/blockchain-based-chat-messenger-apps/>

Dust offering rewards for data !!

Basic atttention token, advertising, cryptocurrency

<https://github.com/loomnetwork/etherboy-core/blob/master/tools/cli/indexer/etherboyindexer.go>

websockets + publish - subscribe + elasticsearch + redis + queue

Simple Social Network SMart contract

pragma solidity ^0.4.24;

contract SimpleSocialNetwork {

    struct Comment {

        string text;

    }

    struct Post {

        string text;

    }

    mapping (address => uint[]) public postsFromAccount;

    mapping (uint => uint[]) public commentsFromPost;

    mapping (uint => address) public commentFromAccount;

    Post[] public posts;

    Comment[] public comments;

    event NewPostAdded(uint postId, uint commentId, address owner);

    constructor () public {

        // created the first post and comment with ID

        // IDs 0 are invalid

        newPost("");

        newComment(0, "");

    }

    function hasPosts() public view returns(bool \_hasPosts) {

        \_hasPosts = posts.length > 0;

    }

    function newPost(string \_text) public {

        Post memory post = Post(\_text);

        uint postId = posts.push(post) - 1;

        postsFromAccount[msg.sender].push(postId);

        emit NewPostAdded(postId, 0, msg.sender);

    }

    function newComment(uint \_postId, string \_text) public {

        Comment memory comment = Comment(\_text);

        uint commentId = comments.push(comment) - 1;

        commentsFromPost[\_postId].push(commentId);

        commentFromAccount[commentId] = msg.sender;

        emit NewPostAdded(\_postId, commentId, msg.sender);

    }

}

Post, comments  data stored in loomchain.

CLI to query post, comments for user Id will go to user smart contract account  post key, comment key form where posts can be  retrieved.

Post struct protobuf

Comments struct protobuf.

Always use a loom chain which can do go smart contracts..  because of technical advantages.. as compared to solidity...

Generate events for smart contract.

Subscribe to event data corresponding to different topics.

Use RPC client to subscribe.

If contract written in solidity, use loom-js event listeners to listen to event.

<https://github.com/bee-queue/bee-queue>

Spark tutorial -

<https://www.javatpoint.com/apache-spark-rdd>

<https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/sql/Dataset.html>

<https://stackoverflow.com/questions/46004290/will-async-await-block-a-thread-node-js>

<https://blog.vanila.io/handling-concurrency-with-async-await-in-javascript-8ec2e185f9b4>

Backwards compatibility - java 8

<https://www.quora.com/Is-Java-8-backward-compatible-with-Java-7>

<https://www.baeldung.com/jvm-garbage-collectors>

G1 Garbage collection - jar  application pauses

According to Java docs, by default, the JVM is configured to throw this error if the Java process spends more than 98% of its time doing GC and when only less than 2% of the heap is recovered in each run. In other words, this means that our application has exhausted nearly all the available memory and the Garbage Collector has spent too much time trying to clean it and failed repeatedly.

<https://www.baeldung.com/java-gc-overhead-limit-exceeded>

Url scraper - metadata based urls , format urls before crawling, remove UTM Tags, request parameters

Crawl - popular urls, count > 3 say certain threshold ==> than unique urls

Kafka stream examples -

<https://github.com/confluentinc/kafka-streams-examples/tree/5.4.0-post/>

Elastic search complete guide -

<https://github.com/codingexplained/complete-guide-to-elasticsearch>

<https://angel.co/r/remote-friendly/distributed-systems-developer/jobs>