**Grokking and SystemExpert/AlgoExpert**

Base64 encoding – uses 64 characters [A-Z],[a-z],[0-9],+ (plus) ,/ (slash)

26+26+10+1+1 = 64.

**Capacity estimation and constraints** / also called **Back of the envelope** calculations

1. Traffic estimates – segregate read traffic from the write traffic with some ratio. Say 5:1 or 10:1.

So for something like pastebin service, if there are 1M writes per day, there would be 5M reads.

After that you can do per second calculations – 1M / (24 \* 3600) ~~ 12 paste writes/second.

1. Storage estimates – For something like pastebin, average paste would be 10KB (with a max paste size capped to 10M).

Storage requirement per day – 10KB \* 1M = 10GB per day

If we store this data for 10 years, we would need 36500GB or 36TB.

Storing of urls (this is common with tinyUrl design too) – *If we use base64 encoding and reserve 6 characters for unique urls, total number of of urls that can be generated are 64 ^ 6 = 68.7 billion.*

Since we have 1M writes (1M new urls are to be saved every day). Each url is 6 characters long and assuming each character would take 1byte, we will store 6M \* 365 \* 10 years ~ 22GB. This is negligible compared to storage space needed for pastes (36TB).

1. Bandwidth estimates – Do along similar lines. You already have traffic estimates per second from point 1.
2. Memory estimates – This will needed for deciding the cache size. Do a 80-20 rule. i.e. 80% of the traffic is generated from 20% of the pastes. Since we are reading 5M pastes per day, we can have a cache size of 0.2 \* 5M \* 10KB = 10GB.

Q-) How would design a REST API for searching (say searching restaurants in Zomato – where you are search by “name” , “rating” etc)

A-) Use a Lucene based search engine like ElasticSearch. The API signature would be like following –

GET /restaurants?q=name contains “xyz” and rating > 3.5

The API layer would do the validation and call the elasticsearch component.

More research required in this field.

Design Google drive from AlgoExpert –

1. Blobs will be stored separately and metadata will be stored differently.
2. Blobs will be stored in some general “blob store” and metadata will be stored in KeyValue stores (HA can be implemented using etcd and ZooKeaper).
3. Metadata should be stored in a tree like data structure with linking to its parent too. (Remember when it comes to files like structure, tress are best)

{

Id :

Name :

Children: []

Parent:

IsFolder:

Blobs:[] //splitted blobs reference from a given file/blob

}

1. When a file/blob is getting uploaded, it will be split into multiple blobs and then stored. Each mini blob should be immutable and should be repeated, if already present. This can be implemented by keeping a hash of the miniblob and checking if that is already present, then don’t re-add it.
2. Rename, Delete, Move – This can be simply done accessing the Key-Value stores only.
3. With the above type of system, the mini-blobs can never be deleted, so we will have to implement a garbage-collector system (GC). GC will be constantly checking the key values stores and see if there is any “living reference” to a mini-blob. If not, it can be marked for deletion.

**Challenges with database sharding (Horizontal scaling)**

Sharding is a great technique to scale the database but it is far from a perfect solution. It introduces complexities and new challenges to the system:

**Resharding data**: Resharding data is needed when 1) a single shard could no longer hold more data due to rapid growth. 2) Certain shards might experience shard exhaustion faster than others due to uneven data distribution. When shard exhaustion happens, it requires updating the sharding function and moving data around. **Consistent hashing** is a commonly used technique to solve this problem.

**Celebrity problem**: This is also called a hotspot key problem. Excessive access to a specific shard could cause server overload. Imagine data for Katy Perry, Justin Bieber, and Lady Gaga all end up on the same shard. For social applications, that shard will be overwhelmed with read operations. To solve this problem, we may need to allocate a shard for each celebrity. Each shard might even require further partition.

**Join and de-normalization**: Once a database has been sharded across multiple servers, it is hard to perform join operations across database shards. A common workaround is to de-normalize the database so that queries can be performed in a single table.

**Design WhatsApp/Chat application -** [**https://bytebytego.com/courses/system-design-interview/design-a-chat-system**](https://bytebytego.com/courses/system-design-interview/design-a-chat-system)

1. Sender to ChatService. HTTP connection initiated by client. **Keep-Alive** HTTP header enabled (to reduce multiple TCP handshakes).
2. ChatService to Receiver – HTTP connection cannot be initiated by the service. So the client either does **polling, long-polling or websockets** are the options.

In **long polling**, a client holds the connection open until there are actually new messages available or a **timeout** threshold has been reached. Once the client receives new messages, it immediately sends another request to the server, restarting the process. Long polling has a few drawbacks:

* Sender and receiver may not connect to the same chat server. HTTP based servers are usually stateless. If you use round robin for load balancing, the server that receives the message might not have a long-polling connection with the client who receives the message.
* A server has no good way to tell if a client is disconnected.
* It is inefficient. If a user does not chat much, long polling still makes periodic connections after timeouts.

Since **WebSocket** connections are persistent, efficient connection management is critical on the server-side. Note web sockets are not magic. They are initiated using HTTP only but later upgraded to a persistent connection.

**DataStore (Hint – There is no single solution for all kind of data).**

Users, Friends list, Profiles, Settings can be stored in relational datastore, while chat messages needs to be stored in KV stores for horizontal scaling. (Ques – Why cant horizontal scaling happen in RDMS. Short Ans – It can happen but there is a huge maintenance cost for normalization. Refer - <https://stackoverflow.com/questions/27157227/can-relational-database-scale-horizontally>

**MessageId Generation**

1. Should be unique
2. Should be sortable by time

This in itself is a system design question. Ans – Twitter Snowflake.

For this problem – local sequence generator (local to channel/group) will work.

**Service discovery (Apache zookeeper – Think coordination in a distributed world).**

Responsibility – Once a user logs in – which ChatServer should be persistent connection (websocket) be established with ?

**Small Group Chat**

The same message is copied to multiple queues (for each member of the group). This is expensive as the same message is copied. So applicable for small group only.

**Online-Offline**

Client sends heartbeat to “Presence Servers” which is persisted in KV stores.

**Online status Fanout**

Client sends “online status msg” to the “Presence servers topics” to which different other clients/users subscribe.

**Design YouTube -** [**https://bytebytego.com/courses/system-design-interview/design-youtube**](https://bytebytego.com/courses/system-design-interview/design-youtube)

1. Note – Not everything has to be built from scratch.
2. You can use Cloud services like CDN and blob storage.

Super high-level diagram (everything-else 😉)

Diagram

Description automatically generated

To be cotd.

**Difference between YouTube design vs NetFlix design** – The ratio of UserMetadata content size : Video content size would be way higher in Netflix as compared to that of Youtube.

**Airbnb**

For searching, fetch the data from ElasticSearch cluster (based on Apache Lucene platform) . Apart from performance, you will also get fuzzy search capabilities. Maldives ~ Maaldives [Edit distance 1]

To be cotd..

**Cross-cutting in all most System Design**

1. Capturing metrics. [region, ipAddress, userAgent, platform… and other attributes from http headers] – These metrics can be sent to Kafka (or any other streaming platform). From their it could be sent to other consumers subscribed or Hadoop.

As an optimization – instead of sending every metric (as an event) to kafka, it could be batched locally and send to Kafka every x minutes or when the local queue size crosses a certain threshold. This approach ensures local server resources are optimized. However in cases of failures, the entire batch will fail. Since these are just metrics and not sensitive data. It is ok if a batch fails.

1. Monitoring / Alerting / Grafana