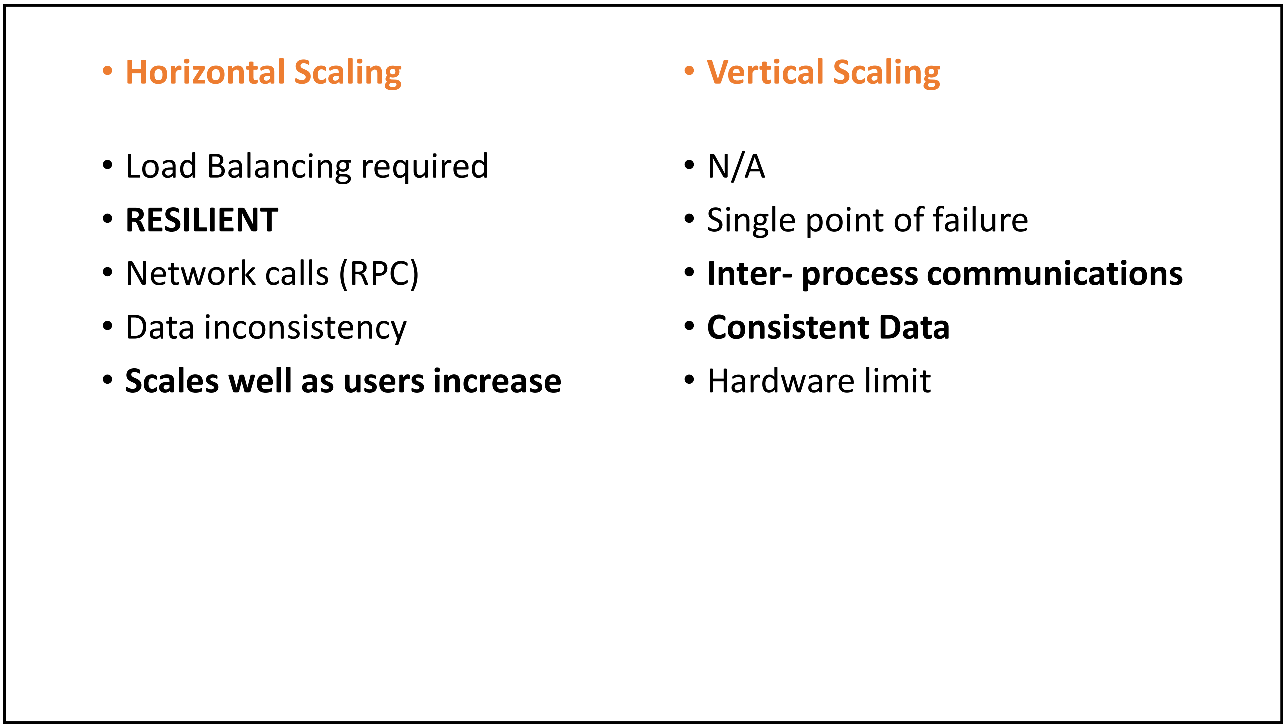
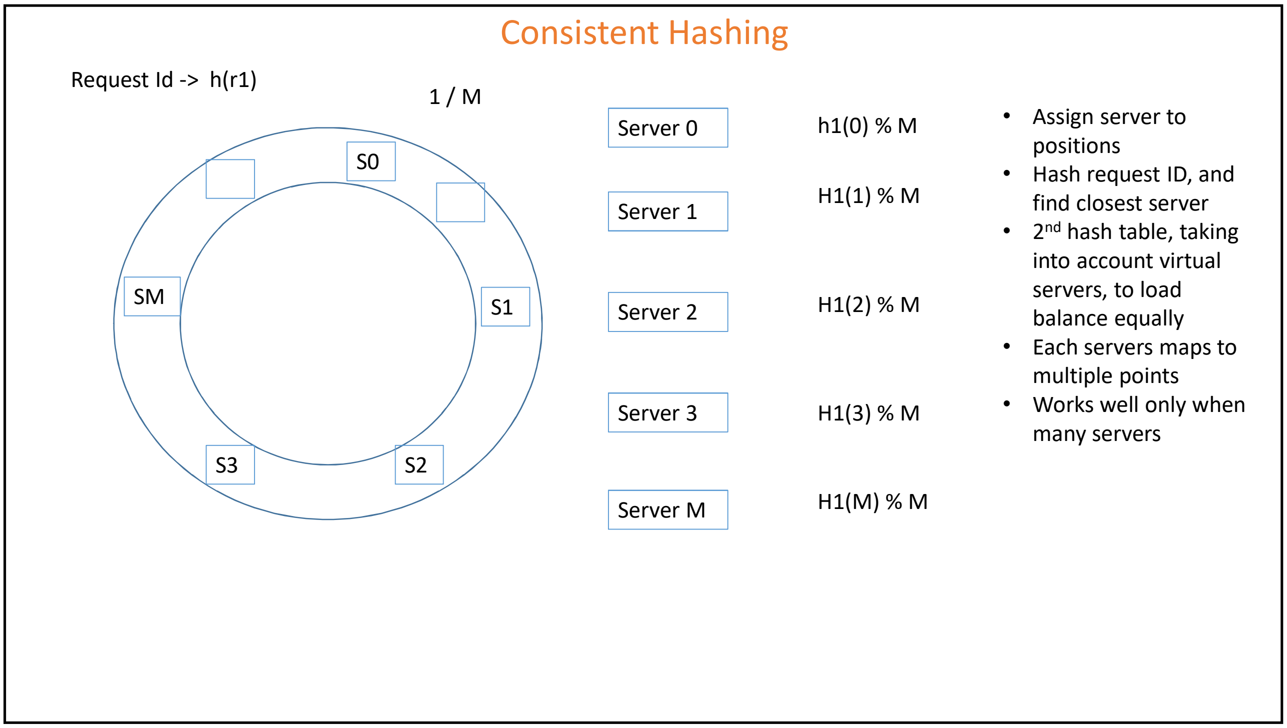


- **Horizontal Scaling**

- Load Balancing required
- **RESILIENT**
- Network calls (RPC)
- Data inconsistency
- **Scales well as users increase**

- **Vertical Scaling**

- N/A
- Single point of failure
- **Inter- process communications**
- **Consistent Data**
- Hardware limit



Consistent Hashing

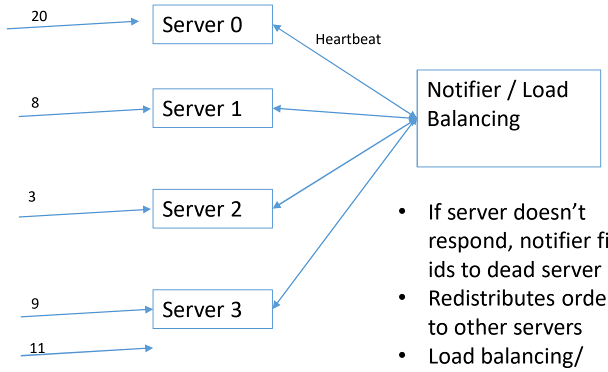
Request Id -> h(r1)

1 / M

Server 0	h1(0) % M
Server 1	H1(1) % M
Server 2	H1(2) % M
Server 3	H1(3) % M
Server M	H1(M) % M

- Assign server to positions
- Hash request ID, and find closest server
- 2nd hash table, taking into account virtual servers, to load balance equally
- Each servers maps to multiple points
- Works well only when many servers

Message/ Task Queues



- If server doesn't respond, notifier finds ids to dead server
- Redistributes orders to other servers
- Load balancing/ consistent hashing deals with duplicates, not sent to multiple servers

ID	Contents	Done
1	Ham	Y
2	Cheese	N
3	Plain	N
4	…	…

Features

1) Assignment / notification
2) Load balancing
3) Heartbeat
4) persistence

Monolithic vs MicroServices

- **Monolithic**

- Can have many machines
- Simpler to maintain
- Less Complex
- Don't need to duplicate for setting up tests, connections
- Procedure calls faster, not RPC

- Deployments are complicated, have to be monitored every time
- Single point of failure, have to restart everything instead of at few points

- **MicroService**

- Can have little machines
- Easier to scale
- Easier for new team members
- Parallel development is easier
- Fewer parts are hidden when deploying

- Not easy to design, needs smart architects

Database Sharding

- Consistency
- Availability
- Can shard by userId, location

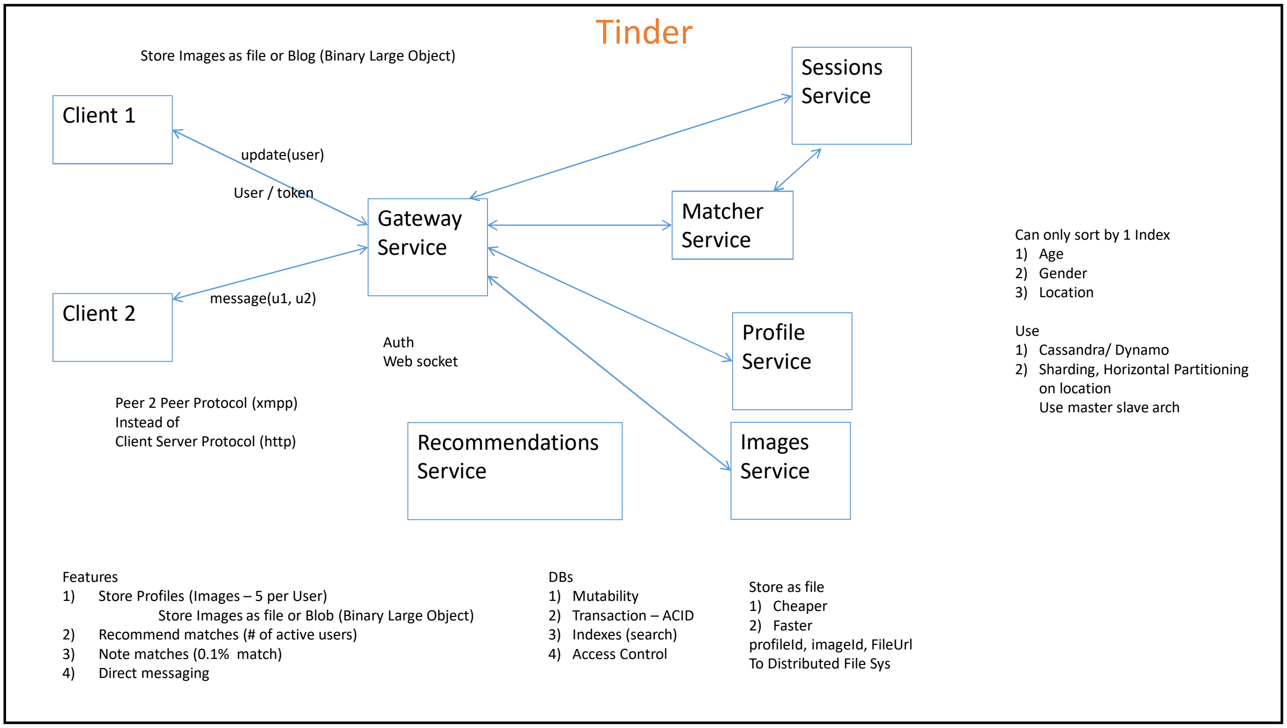
- Problems
- Joins across shards, expensive
- Shards are inflexible in number, can use consistent hashing (Memcached)

- Create index on shards
- Master -> Slave Arch
- Writes to master, reads to slaves, slave becomes master if needed

Single point of failure

- More Nodes
- Master – Slave
- Regions

- Multiple Load Balancers (Gateway)
- Client -> DNS -> LBs -> Nodes -> DBs



Tinder

Store Images as file or Blog (Binary Large Object)

Client 1

update(user)

User / token

Client 2

message(u1, u2)

Gateway Service

Auth
Web socket

Sessions Service

Matcher Service

Profile Service

Images Service

Recommendations Service

Peer 2 Peer Protocol (xmpp)
Instead of
Client Server Protocol (http)

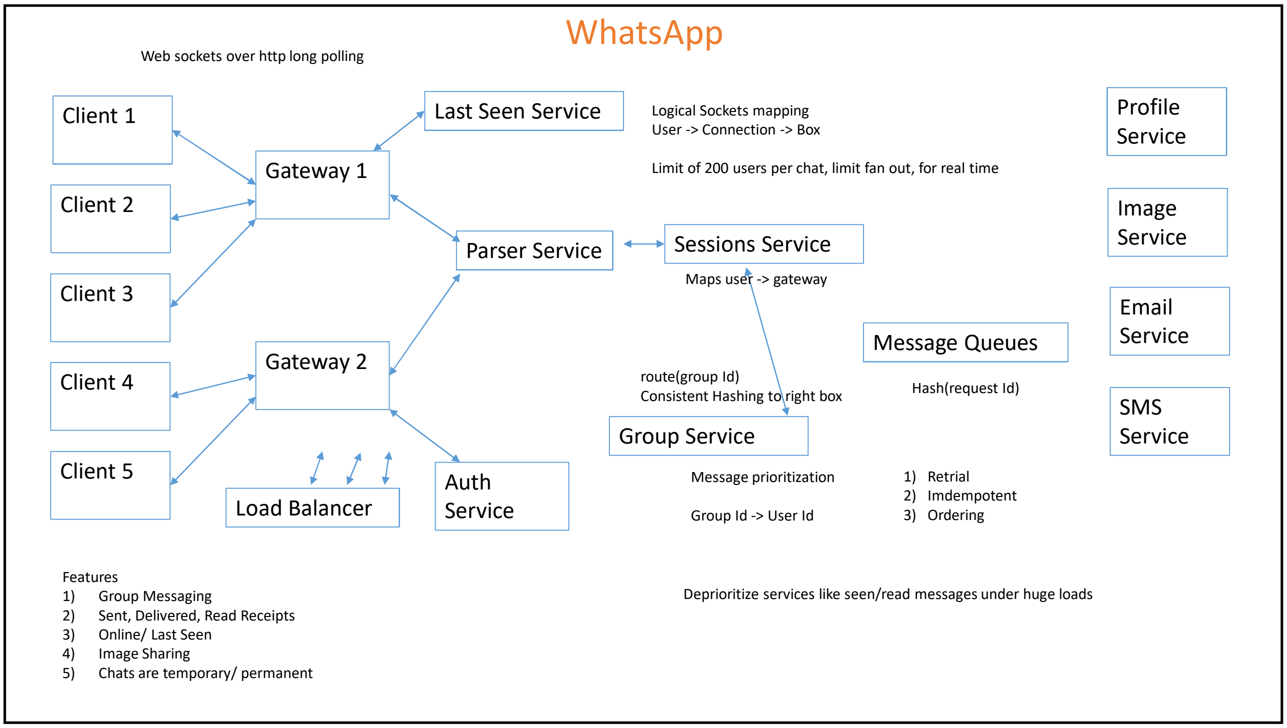
Can only sort by 1 Index
1) Age
2) Gender
3) Location

Use
1) Cassandra/ Dynamo
2) Sharding, Horizontal Partitioning on location
Use master slave arch

Features
1) Store Profiles (Images – 5 per User)
Store Images as file or Blob (Binary Large Object)
2) Recommend matches (# of active users)
3) Note matches (0.1% match)
4) Direct messaging

DBs
1) Mutability
2) Transaction – ACID
3) Indexes (search)
4) Access Control

Store as file
1) Cheaper
2) Faster
profileId, imageId, FileUrl
To Distributed File Sys



WhatsApp

Web sockets over http long polling

Client 1

Client 2

Client 3

Client 4

Client 5

Gateway 1

Gateway 2

Load Balancer

Last Seen Service

Parser Service

Auth Service

Logical Sockets mapping
User -> Connection -> Box

Limit of 200 users per chat, limit fan out, for real time

Sessions Service

Maps user -> gateway

route(group Id)
Consistent Hashing to right box

Group Service

Message prioritization

Group Id -> User Id

Message Queues

Hash(request Id)

1) Retrial
2) Imdempotent
3) Ordering

Profile Service

Image Service

Email Service

SMS Service

Features
1) Group Messaging
2) Sent, Delivered, Read Receipts
3) Online/ Last Seen
4) Image Sharing
5) Chats are temporary/ permanent

Deprioritize services like seen/read messages under huge loads

Distributed Caching

Why

1) Avoid Network Call
2) Avoid computations
3) Reduce DB load

- If cache closer to server, faster, but can be inconsistent between servers (in memory)
- Global cache (Redis), can recover when failing, and can scale independently

- Write-through
  - Update cache first, then update DB
  - Can wait and send to DB in both, for noncritical data, for saving network calls
  - Not practical for multiple caches
- Write-back (performance issues)
  - Update DB first, then make entry in cache/ or invalidate
  - When hit cache on GET, then go to db, and also update cache
  - Hit cache, invalidate entry if it is there
- Hybrid
  - If not critical info, write to cache, wait, and take entries in bulk and write to DB

1) Eviction Policy
2) Thrashing – constantly inputting and outputting to cache without using results
3) Consistency

API Design

Items of importance

1) Naming
2) Parameters deficiency
   1) More params for efficiency only
3) Response object simplicity
4) Return specific errors messages, for expected errors

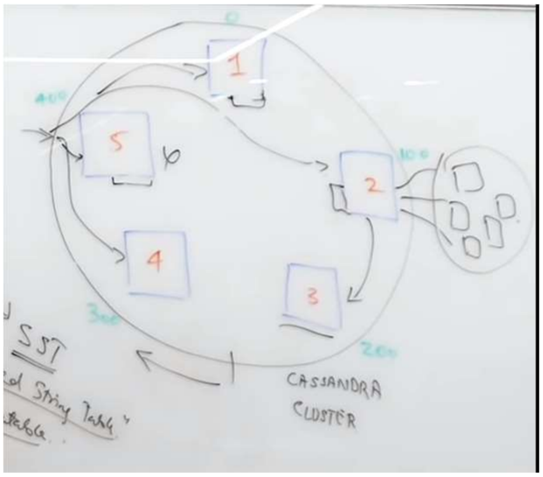
Design API request

1) www.webiste.com/chat_messaging/getAdmins
   1) POST
   2) Request Object
   3) Response
2) No side effects. If many flags, should break down into separate functions
   1) Doing everything
   2) atomicity
3) When response is huge
   1) Pagination - Break response into multiple responses, but not stateless
   2) Fragmentation

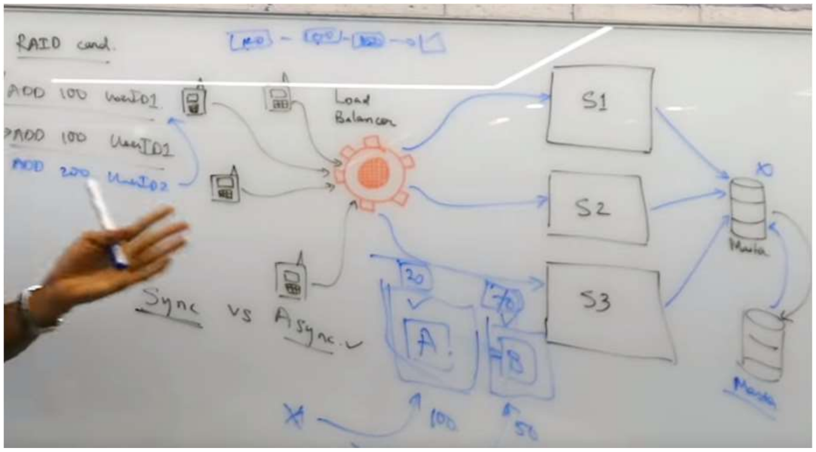
4) Do you want perfect consistency?

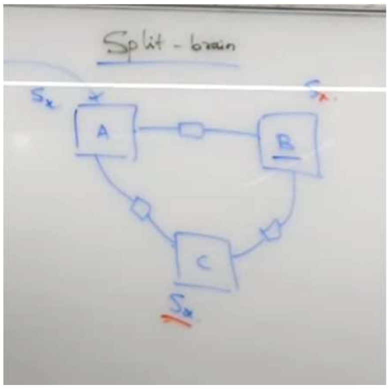
NoSQL

1) SQL
    1) requires joins which are expensive, no way to efficiently normalize
2) NoSQL
    1) Flexible schema
    2) insertions and retrievals require the whole blob
    3) Horizontal partitioning built in, build for scale
    4) Build for aggregation, finding metrics
3) NoSQL – disadvantages
    1) Not built for updates (delete and insert), Consistency, ACID not guaranteed
    2) Not read optimized
    3) Relations are not implicit
    4) Joins are hard
4) Cassandra
    1) Have n instances
    2) Writes to m servers, where pos through pos + m - 1
    3) Quorum factor of x. For reads, x servers have to return the same value



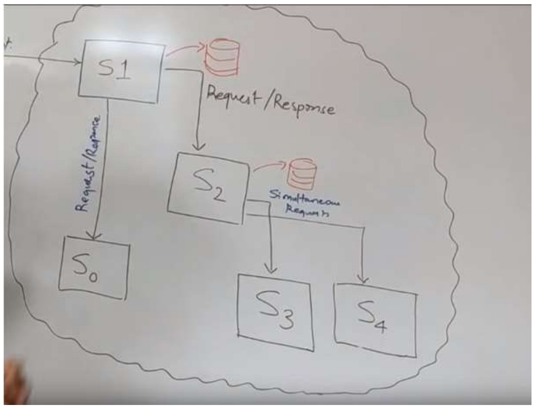
Distributed Consensus

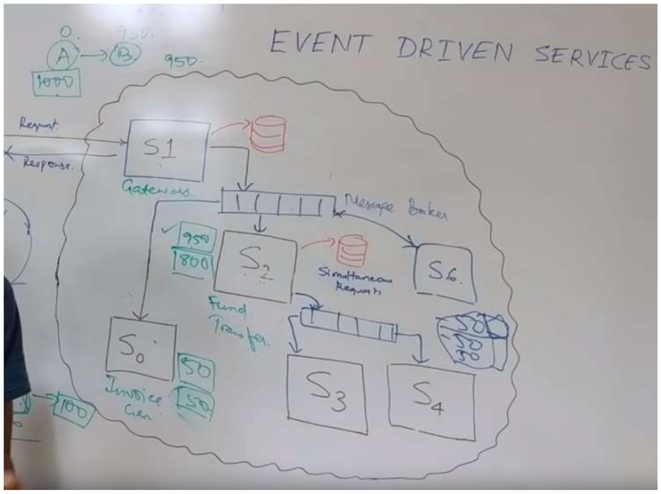




1) Bottleneck to Master DB
    1) Add slave node, prevent writes to it, only reads
    2) Or make both masters
2) Add 2 slaves
3) 2 Phase commit, 3 phase commit, MVCC, SAGA

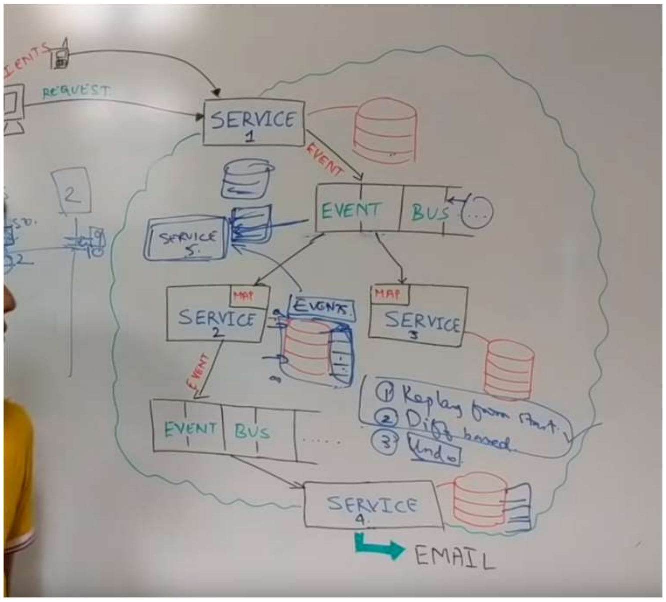
Pub Sub





1) Use queue
2) Disadvantage – not enough consistency by default

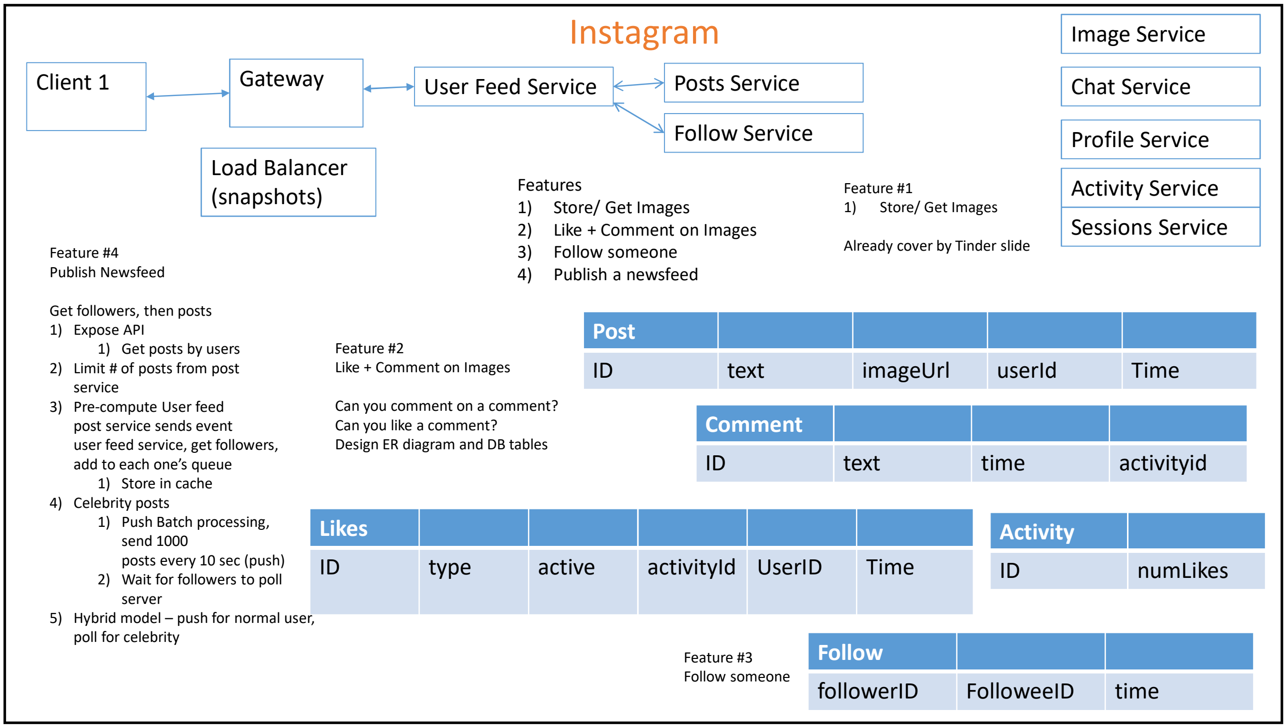
Event Driven Systems



1) Availability (but lower consistency)
2) Easy Roll- backs
3) Replacement

Problems

1) Consistency
2) N/A to Gateways
3) Less control of responses
4) Compaction
5) Hidden Slow
6) Fixing
    1) Replay from Start
    2) Diff based
    3) Undo



Instagram

Features
1) Store/ Get Images
2) Like + Comment on Images
3) Follow someone
4) Publish a newsfeed

Feature #1
1) Store/ Get Images

Already cover by Tinder slide

Feature #4
Publish Newsfeed

Get followers, then posts
1) Expose API
    1) Get posts by users
2) Limit # of posts from post service
3) Pre-compute User feed
   post service sends event
   user feed service, get followers, add to each one's queue
    1) Store in cache
4) Celebrity posts
    1) Push Batch processing, send 1000 posts every 10 sec (push)
    2) Wait for followers to poll server
5) Hybrid model – push for normal user, poll for celebrity

Feature #2
Like + Comment on Images

Can you comment on a comment?
Can you like a comment?
Design ER diagram and DB tables

Post				
ID	text	imageUrl	userId	Time

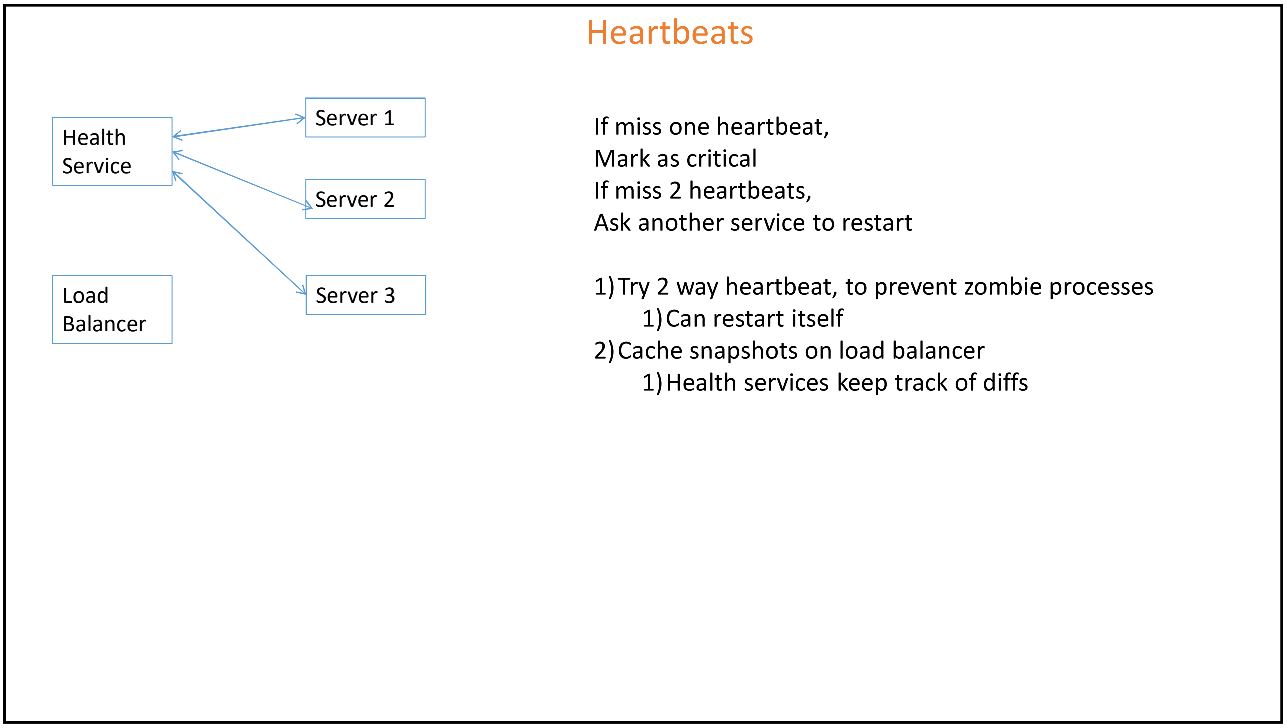
Comment			
ID	text	time	activityid

Likes					
ID	type	active	activityId	UserID	Time

Activity	
ID	numLikes

Feature #3
Follow someone

Follow		
followerID	FolloweeID	time

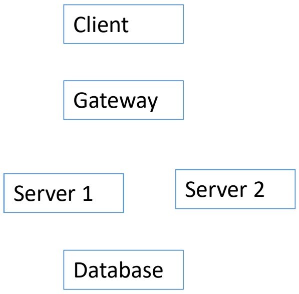


Heartbeats

If miss one heartbeat,
Mark as critical
If miss 2 heartbeats,
Ask another service to restart

1) Try 2 way heartbeat, to prevent zombie processes
    1) Can restart itself
2) Cache snapshots on load balancer
    1) Health services keep track of diffs

Tips



1) Don’t go into detail prematurely
    1) Look for first point to go into detail
    2) Database -> ER diagrams

2) Do not have a set architecture in mind

3) KISS
    1) Keep It Simple Stupid

4) Form your thoughs
    1) Make points without justifications
    2) i.e. choosing a SQL database

5) Be Tech Aware