Messaging/Chat Service

Design A Messaging/Chat Service

(e.g. WhatsApp, Facebook messenger)

Barry Steyn

Software Engineer, LinkedIn

Messaging/Chat Service

Step 1

Functional Requirements and Design Constraints

Interview Kickstart

Functional Requirements

Collect functional requirements from your interviewer

- Ask clarifying questions
- Aim is to show you can communicate
- Objective → to be able to visualize the API

Design Constraints

Collect functional constraints during interview

- Numbers - how many and how much
- Needed when scaling the system
- Do your research: Interviewer can request that you come up with these constraints
- Can be collected at a later stage

Messaging/Chat Service

Example

Functional requirements and design constraints

Interview Kickstart

Functional Requirements

1. Direct messaging (one-to-one chat)
2. Group messaging
3. Message status (sent, received, read)
4. User status (online, last seen)
5. Send pictures/videos

Design Constraints

- **2 billion** MAU (monthly active users)
- **1 billion** groups
- **100 billion** messages sent per day
- All messages must be persisted
- Groups can have max 200 members
- Messages can have up to **700 characters**
- Message can be at most **16 mb**
- Assume **10%** of all messages have attachments (pictures, videos)

Messaging/Chat Service

Step 2

List Microservices

Interview Kickstart

List Microservices

Propose high-level microservices for the design

- Just list (in point form) the microservices involved
- This is quite subjective: There is no correct answer, but you must be able to explain your decisions.
- One approach: data-models and apis differ for two requirements, put the in separate microservices
- https://microservices.io/

Messaging/Chat Service

Example: List microservices

Interview Kickstart

Microservices

- **Messaging** - responsible for receiving and sending messages from/to users.
- **Group** - sends messages to users who are part of a group
- **Profile** - information store about a user
- **Blob** - for storing/sending pictures/videos

Messaging Service

The messaging service should be fully duplex and operate in realtime. How can this be achieved?

- **HTTP long polling** - use HTTP protocol by continually pole the server. Half duplex solution.
- **Websockets** - full duplex, which supports both push and pull.

Messaging/Chat Service

Step 3

Logical diagrams

Interview Kickstart

Logical Diagrams

Draw logical diagrams → aim is to show how the microservices interact

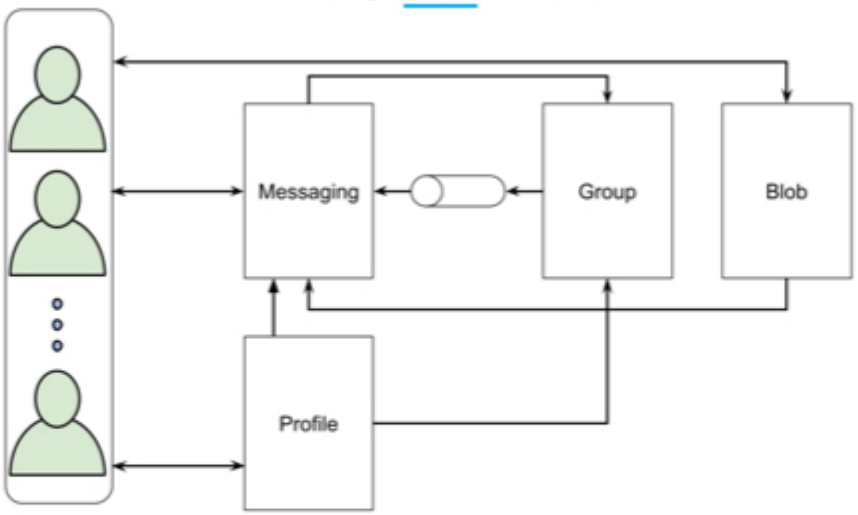
- Create a **block diagram** for each service
- Show the data flow and logic between services:
  - If data is high volume and needs to be pushed in near real-time → **use pub/sub**
  - If data needs to be pulled from server to client → **use REST**
  - If data needs be computed offline → **use batched ETL (extract, transform and load) jobs**

Messaging/Chat Service

Example: Logical diagrams

Interview Kickstart

Messaging/Chat Service



- **Messaging** **(send/receive message to/from user)**
  - Group message request → group
- **Group**
  - Messages to each individual → messaging
- **Profile** **(create/update and read profile for users)**
  - Message author → messaging
  - Message author → group
- **Blob** **(send/receive picture/video to/from user)**
  - Send http URL for blob → messaging

Messaging/Chat Service

Step 4

Deep Dive Into Microservices

a) Logic and tiers
b) Justify the need to scale
c) Architect for scale

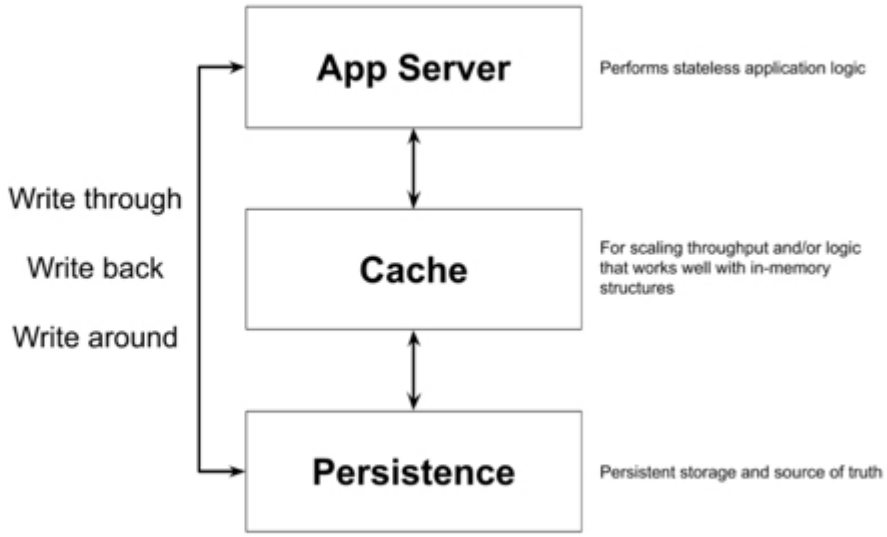
Interview Kickstart

Deep Dive

Deep dive into each microservice. If there are too many microservices, negotiate with interviewer which microservice to focus on

- Remember: you don't have infinite time, so budget your time per microservice
- Each microservice consists of one or more tiers
  - **App server** → handles application logic, bound to cpu
  - **Cache** → needed for high throughput data access and in-memory compute
  - **Storage** → for data persistence

Tiers


App Server
Performs stateless application logic
Write through
Write back
Write around
Cache
For scaling throughput and/or logic that works well with in-memory structures
Persistence
Persistent storage and source of truth

Messaging/Chat Service

Microservices to focus on

Interview Kickstart

Focus on the following microservices

1. **Profile**
2. **Messaging**
3. **Group**
4. **Blob**

Messaging/Chat Service

Step 4a Logic And Tiers

Deep Dive Into Microservices

Interview Kickstart

Logic And Tiers

For each microservice

- Solve the tier logically
- Ignore scale (for now)
- Propose APIs to match functional requirements
- Create a data model that matches the functional requirements
- Discuss cache and storage (if applicable)
- Propose workflow/algorithm for each tier
- Discuss flow amongst tiers in microservice

Logic And Tiers (contd)

The logic/tiers section is the most difficult aspect of a systems design interview

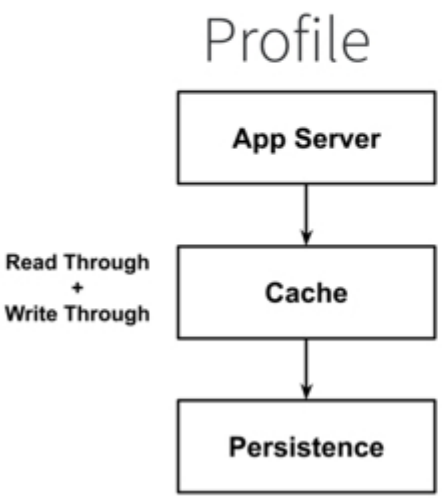
- Subjective → Every candidate can propose a different solution
- Every problem is unique, and therefore no pattern can be used
- Most thinking will be done here
- Highest probability of failing the interview unless candidate is prepared

Messaging/Chat Service

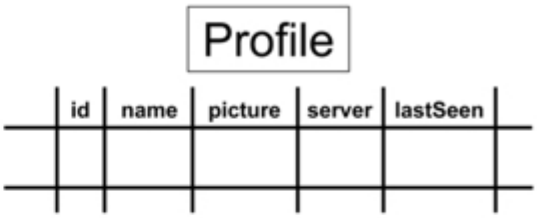
Example

Logic And Tiers

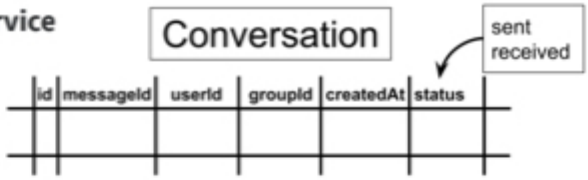
Interview Kickstart



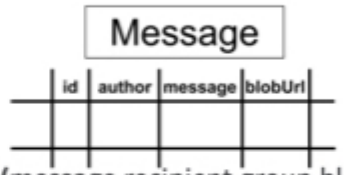
- Data Model



- API:
  - **get(**userId**)** → returns info, such as server
  - **create/update(**name, picture**)**
  - **updateServer(**userId, serverId**)**
  - **updateLastSeen(**userid**)**
- How to organize data
  - row oriented key-value store
- Cache
  - 100 billion messages sent daily - prevent read hotspot, increase throughput ⇒ use cache



Messenger



- API:

  sendMessage(message,recipient,group,blob**)**

  Will send message to either a single recipient or a group

  sendMessagesWhenUserComesOnline(userId)

  Will send messages to a recipient that was sent when they were not online

  updateConversationStatus(conversationId)

- How to organize data
  - row oriented key-value store. Index on status
- Crux:
  - Use websockets for full duplex realtime communication

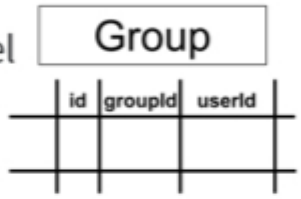
Messenger
(continued)

sendMessage

1. Persist the message to the data-store
2. If group conversation, use **fanout** from Group service (see below)... else proceed
3. Obtain recipient server
4. If server available, then put message on servers internal queue
5. Update status to sent if successfully sent

Group

- Data Model



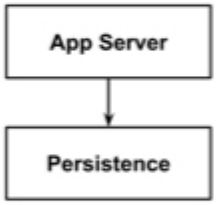
- API:

  fanout(groupId, messageId)

  Will create a conversation for every and send a message for each

- How to organize data
  - row oriented key-value store

Blob



- API:
  create(blob) → will return URL
  get() → http get or URL returned by create
- How to organize data
  - file system, blob store

Messaging/Chat Service

Step 4b Justify the need to scale

Deep Dive Into Microservices

Interview Kickstart

For each microservice

Justify the need to scale

- Check whether the tier needs to scale
- There are only 6 reasons to scale:
  a. Storage
  b. Throughput (large number of requests a second)
  c. High response time (bulky processing)
  d. High latency
  e. Hotspots
  f. Availability, geolocation, spof (single point of failure)
- Constraints and numbers change for every problem
- Solve algebraically
- If phase 1 takes time, do not spend lots of time on calculations/estimates unless asked to by interviewer

Storage

Justify the need to scale

Storage formulas

- **a** = number of rows
  b = size of row → (comes from your data model proposal)
  c = number of *insertions/updates* a second
- Formula is one of these:
  1) **a * b**
  2) **c * b * number of seconds in years**
  3) **c * b * ttl in seconds**
- With replication:
  num of servers = num above * replication factor
- With cache, assume 20-30% of storage:
  num above * 0.2 OR **num above * 0.3**

Numbers You Should Know For Scaling

Powers Of 10	
10^1 = ten	10^10 = ten billion
10^2 = hundred	10^11 = hundred billion
10^3 = thousand	10^12 = trillion
10^4 = ten thousand	10^13 = ten trillion
10^5 = hundred thousand	10^14 = hundred trillion
10^6 = million	10^15 = quadrillion
10^7 = ten million	10^16 = ten quadrillion
10^8 = hundred million	10^17 = hundred quadrillion
10^9 = billion	10^18 = quintillion

Data Powers	
8 bits = 1 byte	1 byte = 8 bits = 2^3 bits
-----	-----
10^3 bytes = 1 kilobyte	1 kilobyte = 2^10 bytes = 2^13 bits
10^6 bytes = 1 megabyte	1 megabyte = 2^20 bytes = 2^23 bits
10^9 bytes = 1 gigabyte	1 gigabyte = 2^30 bytes = 2^33 bits
10^12 bytes = 1 terabyte	1 terabyte = 2^40 bytes = 2^43 bits
10^15 bytes = 1 petabyte	1 petabyte = 2^50 bytes = 2^53bits

Messaging/Chat Service

Example: Justify need to scale

Interview Kickstart

Numbers

- Number of seconds in a day: 90,000
- 100 billion messages a day is (100 X 10^9)/(90 X 10^3) about 1 million messages a second
- In general, the following is assumed:
  Database read: 10 ms
  Cache read: 2 ms
  App: 1 ms
- Assuming a single thread can handle 1000/x ops a sec, where x is one of the number above:
  Database read: 1 thread can do 100 ops/sec
  Cache read: 1 thread can do 500 ops/sec
  App: 1 thread can do 1000 ops/sec

Numbers

Justify the need to scale

Profile

Justify the need to scale

Storage

- Size of row = 164 bytes
  id: **8 bytes**
  name: **100 bytes** (100 characters)
  pictureUrl: **40 bytes**
  serverId: **8 bytes**
  heartbeat: **8 bytes**
  Total is 164 bytes
- 2 billion users:
  2 billion X 164 bytes = 328 gigabytes ⇒ don't need to scale storage except for SPOF (so replicate)
- In fact, all data can be stored in in-memory cache, with data-store backing (Redis)

Profile

Justify the need to scale

Number of machines

- 1 million messages a second → 1 million qps
- A machine can have up to 65,000 socket connections (limited by the availability of ports) - but let's round the number down to 50,000 ⇒ 50,000 reads per machine
- A commodity machine has about 100 threads → assume we devote 40% to websockets, that's 40 threads per machine. 50,000 X 0.4 = 20,000 reads per machine
- 50 machines are therefore needed for 1 million qps (20000 X 50 = 1 million)

Messaging (message)

Justify the need to scale

Storage

- Each message is 800 bytes

  id: **8 bytes**
  author: **8 bytes**
  blobUrl: **50 bytes** (allows for 1 quadrillion urls)
  message: **700 bytes** (max 700 characters)
  Total is 764 bytes ⇒ round up to 800 bytes

- Storing data for 5 years (1500 days):

  1500 X 100 * 10^11 X 800 bytes (1500 days X 100 billion messages X 800 bytes)
  800 X 1.5 * 10^3 X 10^11 = 12000 X 10^14 bytes
  = 1.2 X 10^18 bytes = 1.2 exabytes!!!!!

- Largest commercial SSD is 100 terabytes (as of 1 Jan 2021). Ten thousand 100 terabytes will be 1 exabyte. Multiple by 3 for SPOF and availability - that is 30,000 SSD, 7500 machines (assuming each machine has 4 SSD)

Messaging (conversation)

Justify the need to scale

Storage

- Each conversation is 50 bytes
  id: **8 bytes**
  messageId: **8 bytes**
  userId: **8 bytes**
  groupId: **8 bytes**
  createdAt: **8 bytes** (timestamp)
  status: **4 bytes** (enum)
  Total is 44 bytes ⇒ round up to 50 bytes
- This is 1/16'th of message storage, so about 74 petabytes
- Only about 500 machines (1/16 number of machines for message)
- **Grand total for storage (for entire microservice)**: 8000 machines

Messaging

Justify the need to scale

Number of machines app

- 2 billion MAU - means that 2 billion websockets would be needed.
- A machine can have up to 65,000 socket connections (limited by the availability of ports) - but let's round the number down to 50,000 (also, should leave some ports free)
- A commodity machine has about 100 threads → assume we devote 50% to websockets, that's 50 threads per machine.

Messaging

Justify the need to scale

Number of machines

- If each thread handles 1000 websockets, each machine will handle 50,000 ws
- 40,000 machines to handle 2 billion users, each with a websocket
- Redundancy means we should probably have 50,000 machines ⇒ no SPOF

Group

Justify the need to scale

Storage

- Each group is 30 bytes
  id: **8 bytes**
  groupId: **8 bytes**
  userId: **8 bytes**
  Total is 24 bytes ⇒ round up to 30 bytes
- **1 billion** groups, with 200 members:
  1 billion X 30 bytes = 30 gigabytes X 200
  6 terabytes → will every group have 200 users?
- Replication factor of 3 ⇒ 18 terabytes

Blob

Justify the need to scale

Storage

- Messages can be a total of 16mb
- 10% of messages with attachments is 10 billion messages a day
- Assuming all messages have 16mb attachments, then 5 years of data (1500 days) is:
  16000 bytes X 10^10 X 1500 =
  16 X 15 X 10^15 = 240 petabytes
- Not every message will have max attachments. So assume only 40% of messages do: 100 petabytes
- Replication: 300 petabytes

Messaging/Chat Service

Step 4c Architect For Scale

Deep Dive Into Microservices

Interview Kickstart

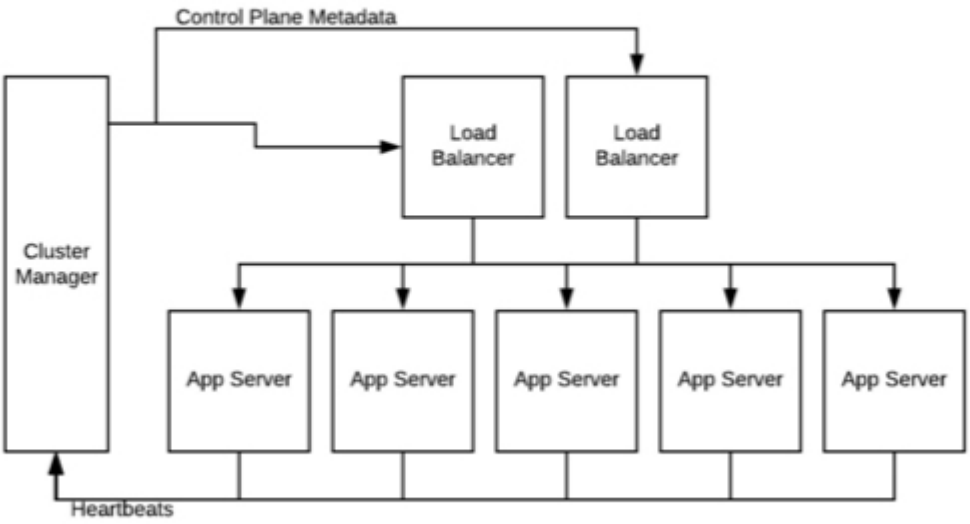
Architect For Scale

For each microservice

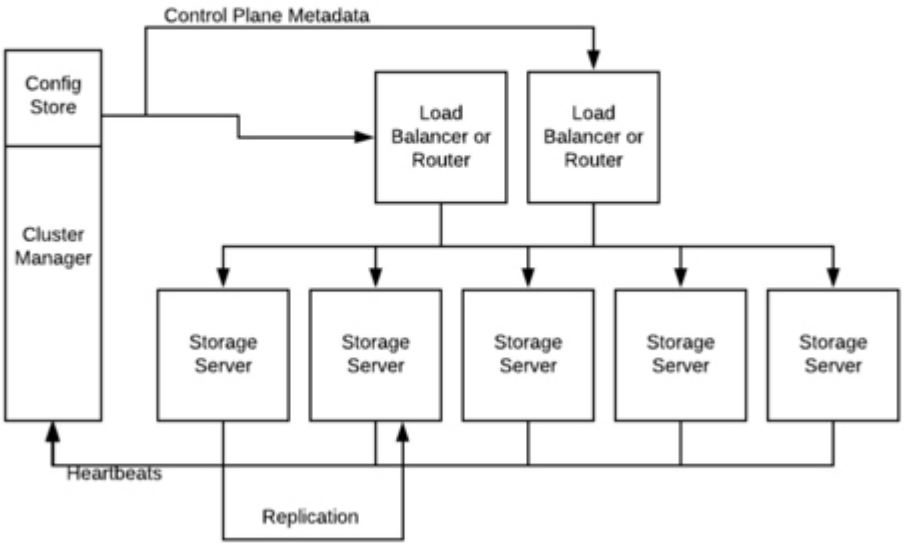
- If necessary, scale the tier and propose the distributed system
- This is the most “deterministic” part of the solution:
- Draw a generic distributed architecture per tier (see next page) → rinse and repeat
  - App tier (stateless) → just round robin requests
  - Cache or storage:
    - Partition data into shards or buckets to suite requirements of scale
    - Algorithm to place shards on server → consistent hashing
    - Propose replication
    - Propose CP or AP (algorithms for CAP theorem do not change from problem to problem)
- This is how each microservice looks within a single data center, either on-premise or VPC in cloud

Messaging/Chat Service

Standard Distributed Architecture For Stateless Round Robin Servers



Standard Distributed Architecture For Stateful Storage



Messaging/Chat Service

Example: Control Plane

Interview Kickstart

Profile/Messaging/Group

control plane

App server: Round robin workers

- Just use the diagram!
- If need be, more workers can easily be added

Storage:

- Sharding → use consistent hashing to find the shard
- Replication (more for SPOF than for throughput)