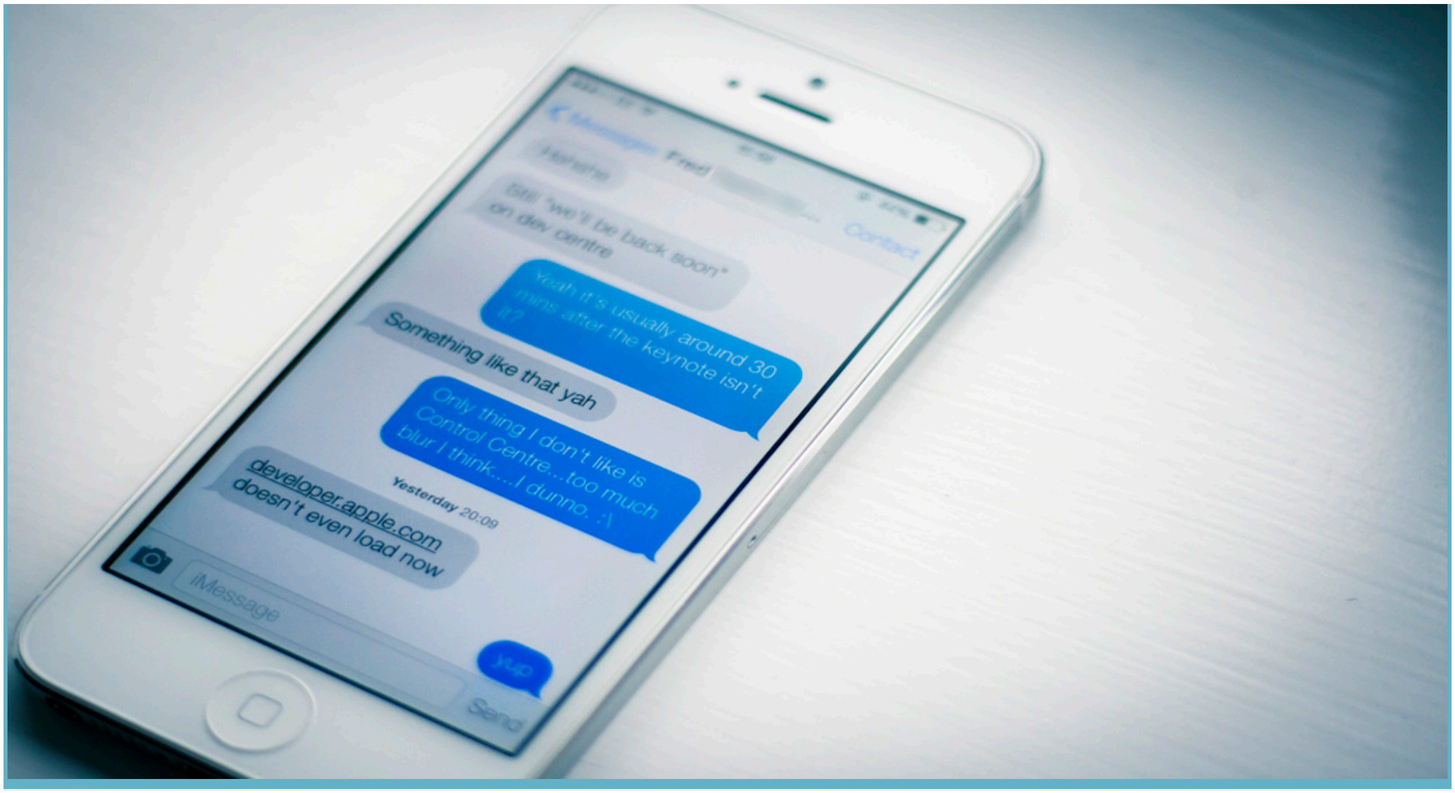
Design a messaging service, like Facebook Messenger.


Messages
Fred
Contact
Stil "we'll be back soon" on dev centre
Yeah it's usually around 30 mins after the keynote isn't it?
Something like that yah
Only thing I don't like is Control Centre...too much blur I think...I dunno. :\
Yesterday 20:09
developer.apple.com doesn't even load now
yup
iMessage
Send

Features:

This is the first part of any system design interview, coming up with the features which the system should support. As an interviewee, you should try to list down all the features you can think of which our system should support. Try to spend around 2 minutes for this section in the interview. You can use the notes section alongside to remember what you wrote.

- **Q:** What is the scale that we are looking at?
  A: Let's assume the scale of Facebook Messages. Let's say we need to hand around 10B message sends a day and around 300M users.
- **Q:** Do we only need to support 1:1 conversations or group conversations as well?
  A: Let's assume we are building things just for 1:1 conversations. We will extend it to group conversations if need be.
- **Q:** Do we need to support attachments?
  A: For now, let's assume that we don't. We will only look at building plain-text messaging system.
- **Q:** What is a reasonable limit to the size of a message?
  A: Let's assume that we are building a chat messaging system. As such, we would expect every message to be shorter in length. We can impose a limit here on the maximum size of such a message. Let's say we will only handle messages less than 64Kb in size and reject the others.
- **Q:** What about the notification system for new messages received?
  A: Considering the size of the discussion here (45 mins in the interview), we will not delve into the notification system for messages.

Estimation:

This is usually the second part of a design interview, coming up with the estimated numbers of how scalable our system should be. Important parameters to remember for this section is the number of queries per second and the data which the system will be required to handle.
Try to spend around 5 minutes for this section in the interview.

Let's estimate the volume of each. Assume that our system would be the one of the most popular messaging service.

Q: Given the number of messages being sent, what is the amount of message sent data size we are generating everyday?
A: Number of message sends : 10B
Assuming each message on average has 160 characters , that results in 10B * 160 = 1.6TB assuming no message metadata.
Q: What is the expected storage size?
A: From the previous section, we know that we generate 1.6TB data everyday if we only store one copy of the message. If we were to provision for 10 years, we are looking at 1.6 * 365 * 10 TB which is approximately 6 Petabytes.

Design Goals:

- **Latency** - Is this problem very latency sensitive (Or in other words, Are requests with high latency and a failing request, equally bad?). For example, search typeahead suggestions are useless if they take more than a second.
- **Consistency** - Does this problem require tight consistency? Or is it okay if things are eventually consistent?
- **Availability** - Does this problem require 100% availability?

There could be more goals depending on the problem. It's possible that all parameters might be important, and some of them might conflict. In that case, you'd need to prioritize one over the other.

Q: Is Latency a very important metric for us?
A: Yes. Chat is supposed to be realtime, and hence the end to end time actually matters.
Q: How important is Consistency for us?
A: Definitely, yes. Its not okay if someone sends me a sequence of message and I don’t see some of them. That could lead to huge confusion. Think of cases when you miss an emergency message or missed messages cause misunderstanding between individuals.
Q: How important is Availability for us?

A: Availability is good to have. If we had to choose between consistency and availability, consistency wins.

Skeleton of the design:

The next step in most cases is to come up with the barebone design of your system, both in terms of API and the overall workflow of a read and write request. Workflow of read/write request here refers to specifying the important components and how they interact. Try to spend around 5 minutes for this section in the interview.
Important *: Try to gather feedback from the interviewer here to indicate if you are headed in the right direction.*

Q: What are the operations that we need to support?
A:

- Send a message to another person
- For a user, fetch the most recent conversations
- For every conversation, fetch the most recent messages

Q: What would the API look like for the client?
Q: How would the *sendMessage* API look like?
A: Send Message : Things to take care of in this API

- sendMessage should be idempotent. If a client retries the message, the message should not be added twice. We can resolve this by generating a random timestamp based ID on the client which can be used to de-duplicate the same message being sent repeatedly.
- Ordering of messages should be maintained. If I send message A, and then send message B, then A should always appear before B. However, it is possible that due to delays, if two messages are sent quickly one after another, then the requests reach the DB out of order. How do we solve such a case? Obviously, we need to resolve based on the timestamp when they were sent at.
  Timestamp on the client is always unreliable. So, we would need to record the timestamp the first time the request hits the servers (Need not be a part of the API to the client)

```
sendMessage(senderId, recepientId, messageContent, clientMessageId)
```

Q: How would the API for fetching user's latest conversation look like?
A: This API would be called if I need to show a page of conversations/threads (Think of the view you see when you open the Whatsapp / Messenger app).
At a time, only a certain number of conversations will be in the viewport (let's say 20). Let's call it a page of conversations. For a user, we would only want to fetch a page of conversations at a time.
Gotcha: Would the page size remain constant across different situations?
Probably not. The page size would be different across clients based on screen size and resolution. For example, a mobile's page size might be lower than that of a web browser's.

- **Delta fetch**: In most cases, our API calls will be made by users who are active on the site. As such, they already have a view of conversations till a certain timestamp and are only looking for updates after the timestamp (which would typically be 0-2 more conversations). For clients which are data sensitive (like mobile), fetching the whole page every time even when I have all of the conversations can be draining. So, we need to support a way of fetching only the updates when the lastFetchedTimestamp is closer to currentTimestamp.

Keeping the above 2 facts in mind, following is how a hybrid API might look like :

```
ConversationResult fetchConversation(userId, pageNumber, pageSize,
lastUpdatedTimestamp)
```

where ConversationResult has the following fields :

```
ConversationResult {
List(Conversation) conversations,
boolean isDeltaUpdate
}
Conversation {
conversationId,
participants,
snippet,
lastUpdatedTimestamp
}
```

Q: How would the API for fetching most recent messages in a conversation look like?

A: Fetch most recent message in a conversation :

This API is almost identical to the fetchConversation API.

```
MessageResult fetchMessages(userId, pageNumber, pageSize, lastUpdatedTimestamp)
```

where MessageResult has the following fields :

```
MessageResult {
List(Message) messages,
boolean isDeltaUpdate
}
Message {
messageId,
senderId,
participants,
messageContent,
sentTimestamp
}
```

A: The first and last operation ends up doing a write to the database. The other operations are purely read operations. Following is how API's may look like:

- Send Message:

```
sendMessage(senderId, recepientId, messageContent, clientMessageId)
```

- Conversations of a user :

```
ConversationResult fetchConversation(userId, pageNumber, pageSize,
lastUpdatedTimestamp)
```

where ConversationResult has the following fields :

```
ConversationResult {
List(Conversation) conversations,
boolean isDeltaUpdate
}
Conversation {
conversationId,
participants,
snippet,
lastUpdatedTimestamp
}
```

- Fetch most recent message in a conversation : This API is almost identical to the fetchConversation API.

```
MessageResult fetchMessages(userId, pageNumber, pageSize, lastUpdatedTimestamp)
```

where MessageResult has the following fields :

```
MessageResult {
List(Message) messages,
boolean isDeltaUpdate
}
Message {
messageId,
senderId,
participants,
messageContent,
sentTimestamp
}
```

Q: How would a typical write query look like?

A: Components:

- Client (Mobile app / Browser, etc) which calls sendMessage(senderId, recepientId, messageContent, clientMessageId)
- Application server which interprets the API call and calls DB to do the following:
- Puts in the serverTimestamp
- Figures out the conversation to which the message should be appended based on the other participant
- Figures out if a recent message exists with the clientMessageId
- Store the message
- Database server which stores the message.

Q: How would a typical read query look like?

A: Components:

- Client (Mobile app/Browser, etc) which calls fetchConversation
- Application server which interprets the API call and queries the database for the top conversation.
- Database server which looks up the user’s conversations.

Deep Dive:

> *Lets dig deeper into every component one by one. Discussion for this section will take majority of the interview time(20-30 minutes).*

Let's dig deeper into every component one by one.

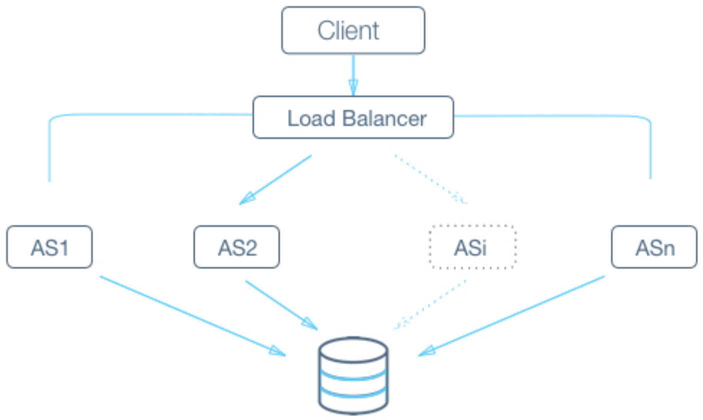
Application layer:

Think about all details/gotchas yourself before beginning.

Q: How would you take care of application layer fault tolerance?
Q: How do we handle the case where our application server dies?
A: The simplest thing that could be done here is to have multiple application server. They do not store any data (stateless) and all of them behave the exact same way when up. So, if one of them goes down, we still have other application servers who would keep the site running.

Q: How does our client know which application servers to talk to. How does it know which application servers have gone down and which ones are still working?
A: We introduce load balancers. Load balancers are a set of machines (an order of magnitude lower in number) which track the set of application servers which are active (not gone down). Client can send request to any of the load balancers who then forward the request to one of the working application servers
randomly.

HIGH LEVEL DESIGN



A: If we have only one application server machine, our whole service would become unavailable. Machines will fail and so will network. So, we need to plan for those events. Multiple application server machines along with load balancer is the way to go.

Database layer

This is the heart of the question. In the skeleton design, we assumed that the database is a black box which can magically store or retrieve anything efficiently. Let's dig into how we will build that magic black box.

Q: RDBMS or NoSQL?
Q: Are joins required?
A: NoSQL databases are inefficient for joins or handling relations. As such, NoSQL databases store everything in a denormalized fashion. In this case, we do have relations like

- user -> messages
- user -> conversations
- conversation -> messages

SQL seems to win on this parameter on ease of use.

Q: How much data would we have to store?
A: If the size of the data is so small that it fits on a single machine's main memory, SQL is a clear winner. SQL on a single machine has next to zero maintenance overhead and has great performance with right index built. If your index can fit into RAM, its best to go with a SQL solution. In our earlier estimations, we had already established that we will need to provision petabytes of data which most definitely does not fit on a single machine.
So, a SQL solution will have a sharding overhead. Most NoSQL solutions however are built with the assumption that the data does not fit on a single machine and hence have sharding builtin. NoSQL wins on this parameter.

Q: What is the read-write pattern?
A: Messaging is going to be very write heavy. Unlike photos or tweets or posts which are written once and then consumed a lot of times by a lot of people, messaging is written once and consumed by the other participant once.
For a write heavy system with a lot of data, RDBMS usually don't perform well. Every write is not just an append to a table but also an update to multiple index which might require locking and hence might interfere with reads and other writes.
However, there are NoSQL DBs like HBase where writes are cheaper. NoSQL seems to win here.

A: Things to consider :

- Are joins required?
- Size of the DB
- Technology Maturity

From the looks of it, NoSQL seems like a better fit. Let's proceed with NoSQL for now.
Q: How would we store the data? Discuss schema
A: As discussed before, with NoSQL, we need to store the data in denormalized form.
First thing first, this means that every user would have his/her own copy of the mailbox. That means that we will store 2 copies of the message, one for each participant for every message send.
Let's delve into how the schema would look. We'll assume that we are using HBase for this problem.
If this is your first time designing schema, we strongly recommend you go through a primer here .
For schema design, its good to recognize our access patterns. To achieve that, let's look at our major operations :

- For a user, append a message to a conversation
- Fetch timestamp ordered conversations for a user (Most recent)
- Fetch most recent messages in a conversation for a user (Most recent)

As you can see, the first lookup for all three operations is for the user. In NoSQL context, it hence makes sense to have userId as the row ID (Data is sharded based on users).
Now, within the user, we will need to lookup conversations, recent conversations and recent

messages.
One naive approach is to fetch the whole list of conversations or all the messages in a conversation and then filter the data we need. This however is really slow (Remember that one of our design goals was to have low latency). Even more so, in cases when some popular users get a lot of message and have a huge mailbox (in GBs).

Let's see, how we would solve each read request one by one.

- Recent conversations : We can separately store conversationId : timestamp mapping in the same row (In case of HBase, in a separate column family). This will not be a lot of data and its okay to read it completely (ofcourse, with caching).
- Recent messages in a conversation : Loading all the messages or loading all the messages in a conversation would make this really expensive. Doing the same thing with messageIds is still an improvement in terms of the amount of data that we have to load.
  As an improvement, if key in the same index is conversationID_timestamp, we can use a prefix search of conversationID and use the most recent messages based on timestamp in the key (assuming, the data is stored sorted with the key).

Q: How would we do sharding?
A: HBase inherently use consistent hashing(in this case on user_id). We explain it in detail at https://www.interviewbit.com/problems/sharding-a-database/
Q: How would we handle a DB machine going down?
A: We explain in detail how to build a reliable and consistent database system in https://www.interviewbit.com/problems/highly-consistent-database/
Q: What are some other things we can do to increase efficiency of the system?
A: Caching (Its the answer in most cases, isn't it? :)).
This one however is not that easy. If you remember, one of our design goals was to ensure tight consistency. Most distributed caching system are good with availability and they become eventually consistent. But they are not tightly consistent.
For example, let's say I have a distributed cache. If a user's messages or conversations are spread across machines, then it starts causing trouble for us because :

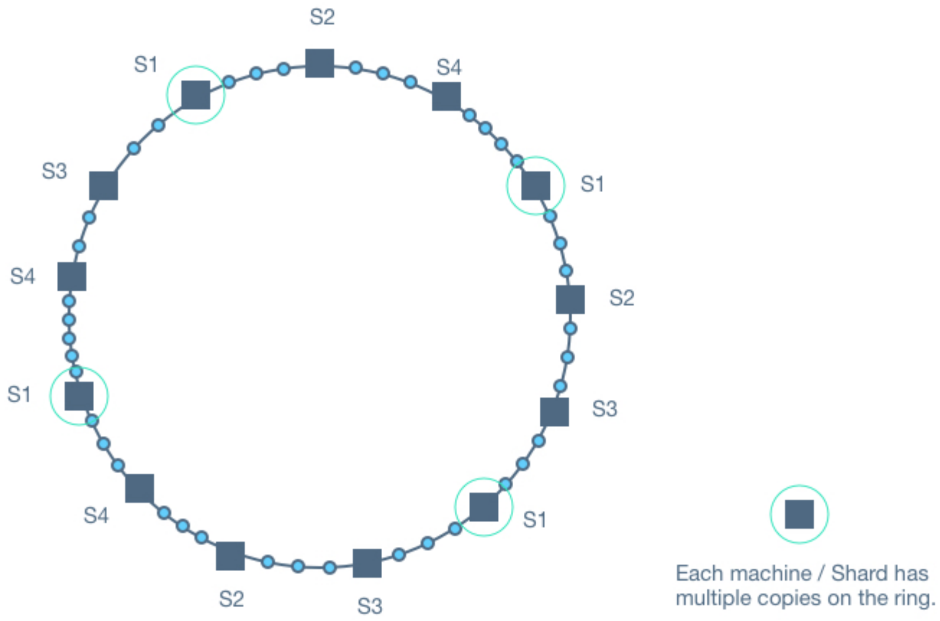
- The changes are no more atomic. Consider the case when messages for a user are one machine and conversations on another. When a message is added, the update request is sent to the server with messages and server with conversations for this user. There could potentially be a period when one server has processed the update and the other has not.
  - If changes are not atomic, the system is not tightly consistent anymore. I might see different views based on when I query the system.

One way to resolve this is to make sure that caching for a user completely resides on one server. The same server can also have other users as well, but users are assigned to exactly one server for caching. To further ensure consistency, all the writes for that user should be directed through this server and this server updates its cache when the write is successful on DB before confirming success.

There are some issues with this system :

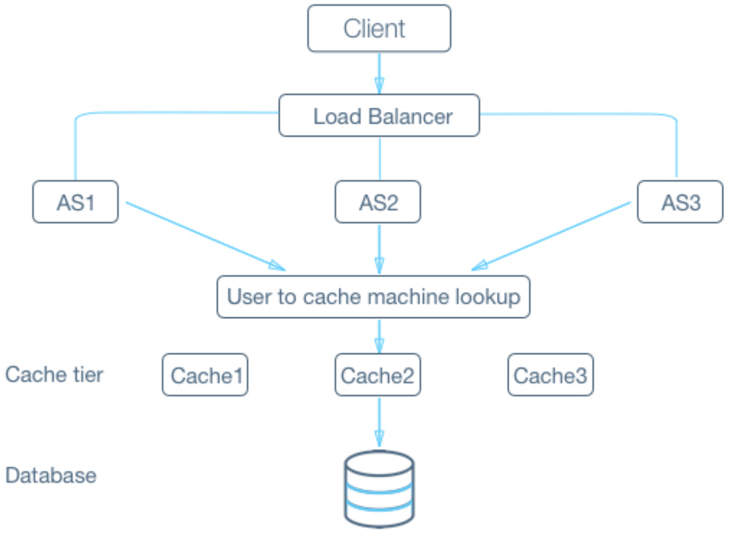
- A single point of failure for the user : My reads and writes are routed through this caching server. If this caching server suddenly dies, then my reads and writes suddenly start failing.
  To resolve this, we should be able to quickly detect the failed machine, mark it as dead and start from scratch (as cache, reading from DB) on a separate machine. If we have servers for backup, we can just have a heartbeat mechanism to detect the server going down and we can activate the backup server.

- Need for a reliable routing service : Obviously, if a user is assigned to a server, then there has to be a service which should be able to track that. All request would be routed through this service (kind of like a load balancer with a lookup based on user_id). The machine would need to track the IP of the active server for the user (If the server goes down, then the routing service should know instantly to route to the new server which starts with a cold cache).
  Distribution of user into different servers is another problem in itself. If we do static allotment, how do we handle the following cases :
- More servers are added to the caching pool. How do we re-distribute the users without causing a cold cache for the whole userbase?
- More users would keep registering. How would they be assigned ensuring uniform load?
-
  What if we did not have a backup server and I had to re-distribute this user's load into remaining servers?
  A classic solution to this problem is consistent hashing with multiple tokens for each server (See diagram attached). For more details, read https://www.interviewbit.com/problems/sharding-a-database/



Each machine / Shard has multiple copies on the ring.

- A minor problem - Multiple concurrent writes : The caching server will also multiple indices corresponding to the mailbox (the ones for recent conversations / recent messages). A single write would affect multiple columns. While a NoSQL DB might guarantee atomicity on a row level, in the caching layer, we will have to guarantee it artificially. One simple way of solving it would be to have a user level lock in the caching server for the user which allows only one write operation to go through at a time.



* AS - Application Server