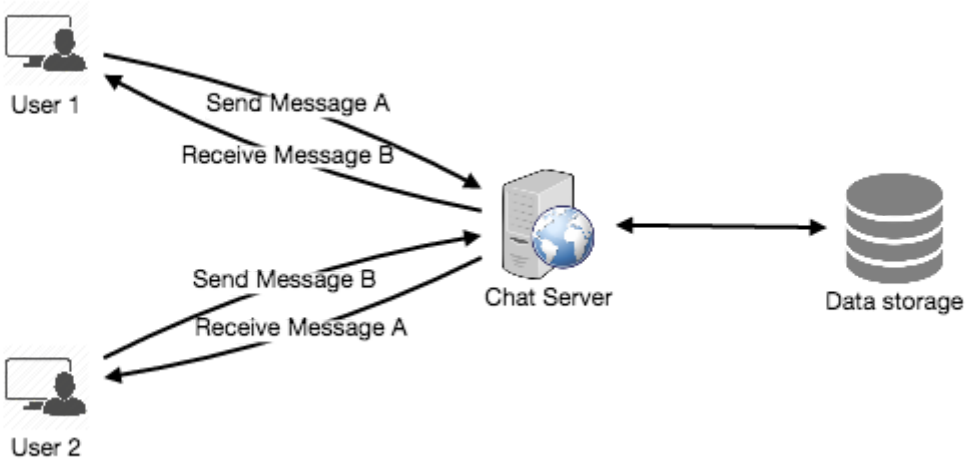
Designing Facebook Messenger (수정완료)

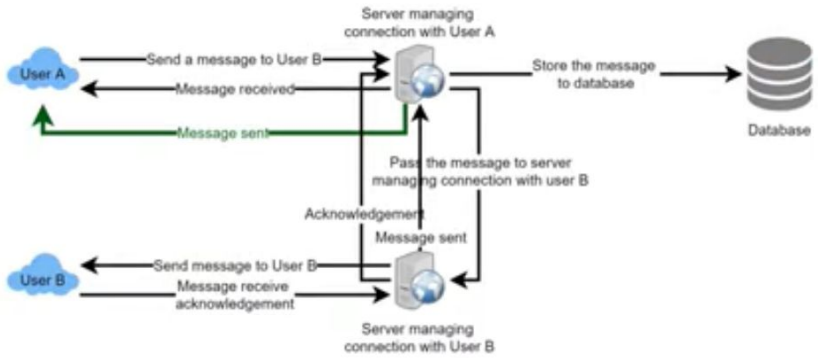
1. facebook messenger
    a. Text based instant messaging service
    b. Website and Mobile App Integration
2. requirements and goal of the system
    a. functional requirements
        i. one-on-one conversation
        ii. user status (online/offline) notification
        iii. persistent storage of chat history
    b. non-functional requirements
        i. real-time chat with minimum latency
        Ii. high consistent
        (ex) All users' chat history should be the same
        ii. high availability <high consistent
    c. extended requirements
        i. group chat
        ii. push notification: send message to offline user
3. capacity estimation and constraints

[assume]			
DAU (daily active user)	500,000,000	user	
average message	20,000,000,000	message/day	
average message per user	40	message/day	
average message size	100	byte	
[estimate]			
storage	2,000,000,000,000	byte/day	
	3,650,000,000,000,000	byte/5years	
bandwidth	23,148,148	byte/sec	incomming data
	23,148,148	byte/sec	outgoing data

4. high level design (15.jpg)



a. workflow (16_x.jpg)
   i. user1 sends a message to the chat server
   ii. The chat server is approved by user1 after receiving the message.
   iii. The chat server sends the message to user2 while saving the message in DB.
   iv. user2 receives the message and sends an acknowledgment to the char server
   v. chat server notifies user1 that the message has been delivered



Request flow for sending a message

5. component design
    a. message handling
        i. model
            - pull (server perspective)
                ✓ Client periodically requests the server
                ✓ Client mostly receives an empty response
                ✓ server needs a pending message processing function
                ✓ Heavy server resource waste (frequent requests from clients, empty responses)
            - push (server perspective)
                ✓ Client keeps only connection with server (http long polling, WebSocket)
                ✓ The server does not need to process the pending message-because a new message is sent to the client immediately.
                ✓ workflow: client request-> keep request open (wait for new message)-> send a new message to the client as soon as it arrives at the server-> when long polling request timeout, client sends connection request again
        ii. How does the server find connections to specific clients?
            - Using hash table
            - key : UserID
            - value: connection object
            - After the server receives the message, it finds the user in the hash table and finds the connection object and sends the message

iii. What happens when the server sends a message to offline users?
    - If the receiver disconnects, the transmission fails and the caller is notified
    - If the recipient's long polling request is timed out, request the sender to try sending the message again
    - Prevent re-entry messages from being re-entered

iv. How many chat servers do you need?
    - 500M connections, 500,000,000 / 50,000 = 10,000 servers assuming one server handles 50K connections

v. Can you tell which server is connected to which user?
    - LB can be installed in front of the chat server
    - Map UserID to connected server

vi. How does the server handle message requests?
    - Receive new message-> Save message to DB-> Send message to recipient-> Send approval to sender
    - Find the server that has the recipient connection (using the hash table)-> Forward the message to the server and send it to the recipient-> When the recipient receives the message, the recipient sends an acknowledgment to the server-> The server notifies the sender that it was successful

vii. How does messenger maintain message order?
    - When the server receives the message, it stores the message and time stamp + serial number
    - Eage Case: Order is not guaranteed when two users send a message at the same time

- Although the order of messages seems different for both clients, they appear the same on devices owned by each user-consistency can be maintained

b. Store and retrieve messages in the database
   i. How to save the message in DB
      - Start a separate thread that works with DB
      - Send asynchronous request to DB and save message
   ii. DB design considerations
      - How can I use DB connection pool efficiently?
      - How to retry a failed request
      - Where to log the request if it fails even after retries
      - How to retry a recorded request after resolving all issues (failure after retry)
   iii. Which storage should I use?
      - Terms of use
        - ✓ Must be capable of small updates at very high speed
        - ✓ Need to be able to fetch various records quickly
      - HBase
        - ✓ Suitable for message service
        - ✓ Store multiple values in one key (several values in multiple columns)
        - ✓ Column-oriented key-value NoSQL DB
        - ✓ HBase groups data, storing new data in a memory buffer and dumping it to disk when the buffer is full-helping to quickly store a lot of small data.
        - ✓ Also good for storing HBase variable size data

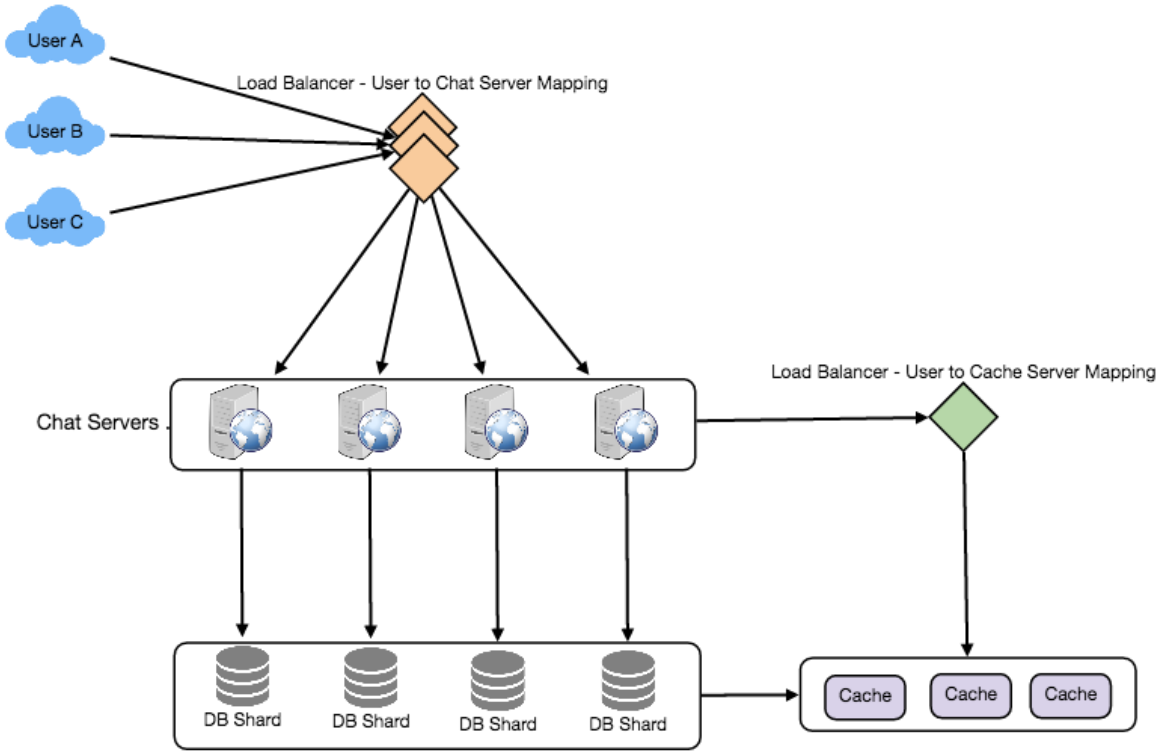
✓ HBase: Google's BigTable as a model and run on HDFS (Hadoop Distributed File System)

- RDBMS
  - ✓ Not suitable for message service
  - ✓ When receiving or sending a message, the number of rows read or stored in the DB table is large (the waiting time is long and the DB has a heavy load)

iv. How can I fetch data efficiently from the server?

- Because the screen of the mobile phone is small, fewer messages / conversations are required than the PC
- Page selection when fetching data from server

c. Managing user's status (17.jpg)



i. Track users' online / offline status and notify all relevant users
ii. The server can easily check the user status-because the server maintains the connection objects of all users.
iii. Assuming you have a 500M DAU, it is expected to be resource intensive if you need to broadcast status changes to related online users. How to reduce resource consumption?
    - Only when the client app starts, friends get their current status
    - If the recipient is offline after sending the message, an error is sent to the sender and the user's status is taken offline
    - Since the user may be online / offline for a short time, the operation should not be performed frequently and broadcast delayed for a few seconds (prevents online and offline).
    - Real-time update only for users displayed on the screen
    - When you start a new chat with another user, get your friends' presence
iv. summary
    - The client connects to the chat server and sends a message.
    - The chat server delivers the message to the recipient
    - online users receive messages while maintaining a connection to the server
    - The chat server pushes the message to the recipient according to a long polling request whenever a new message arrives
    - Messages are stored in HBase (storing small data quickly and supporting range-based searches)

- The chat server should broadcast the user's status to other related users.
- The client only gets the status update of the visible user

6. Data partitioning (5 years 3.7 PB) only
    a. UserID based partitioning
        i. Partitioning based on Hash of UserID (ex) When one shard is 4TB: 3.7 PB / 4 TB ~ = 900 shard

        ii. hash function : UserID % 900 (number of server)
        iii. Pros: Chat history can be fetched quickly (because all messages from one user are stored in the same DB)
        iv. Expansion method
            - Start with multiple logical partitions on a few physical servers
            - As storage demand increases, more physical servers can be added to distribute logical partitions.
    b. MessageID partitioning
        i. Can't use
        ii. When distributing and storing one user's message among all shards, it is too slow to fetch messages (because all shards have to be searched)
7. cache
    a. Can cache recent conversations you see on your screen
    b. Since all messages of the user are stored in one shard, the cache should also be in one server (shard)
8. load balancing
    a. Located in front of the chat server
    b. Map each UserID to a server that has a user connection, then send a request to that server
    c. LB for cache server is also required
9. fault tolerance and replication

a. If the chat server goes down, do I need to send the connection to another server?
    i. Failover of TCP connection to another server is very difficult
    ii. When the connection is lost, the client must be reconnected
b. Do I need to save multiple copies of a message?
    i. Yes, multiple copies of data are stored on different servers

10. extended requirements
    a. group chat
        i. Create a separate group chat object that can be stored on the chat server
        ii. group chat object
            - GroupChatID
            - List of people belonging to chat
        iii. LB can send each group chat message based on GroupChatID
        iv. DB stores group chat in a separate table divided based on GroupChatID
    b. push notification
        i. Currently only sending messages to online users
        ii. Offline user sends a fail to the caller
        iii. Push notification also allows messages to be sent to offline users-send the message to the manufacturer push notification server for notification