Step 1: Requirements

Functional requirements:
- Support one-on-one conversation between users
- Track online and offline statuses of users
- Support persistent storage of chat history

Non-Functional Requirements
- Real-time chat experience with minimal latency
- Highly consistent i.e., same chat history on all devices
- High availability is desirable but consistency more important

Step 2: Back of Envelope Calculations

Storage
- 100b/msg → 200B/day*100b/msg=2TB/day
- To store 5 years = 3.6PB
- User info, message metadata also need to be stored

Bandwidth
- 2TB/day → 25MB/s
- Same amount required for upload and download

Step 3: System Interface Definition

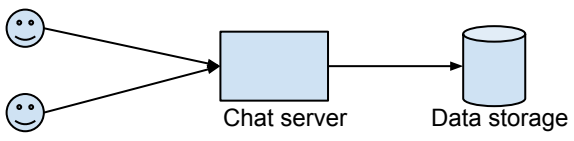
Skipped

Step 4: Define Data Model

Skipped

Step 5: High-Level Design

- Chat server orchestrating communication between users



Order of events
- User A sends message to user B through chat server
- Server receives message and sends ack to user A
- Server stores message in data storage and sends message to user B
- User B receives message and sends ack to server
- Server notifies user A that message delivered to user B

Step 6: Detailed Design

Message handling
- To get message from server:
  a. Pull model: users periodically ask server if there are new messages
     - Requires frequent checks with most checks having empty responses (waste of resources)
  b. Push model: users keep connection open with server and server notifies them when new messages appear
     - Server does not need to track pending messages and minimum latency as messages are delivered instantly
     - Requires HTTP long polling or websockets
- If 500M connections at any time, modern servers can handle 50K concurrent connections, need 10K servers
- Use software load balancer in front of chat server to map user ID to a server
- Chat server needs to find server for receiver and pass message to that server to send it to the receiver
- Store timestamp with each message

Storing and retrieving messages from database
- When chat server receives new message, need to store it in database. Either:
  - Start a separate thread to work with database
  - Send an asynchronous request to database
- Large number of small messages that need to be inserted and user mostly interested in sequential access
  - Cannot use RDBMS like MySQL or NoSQL like MongoDB because cannot afford to read/write a row every time
  - This will create high latency and create a huge load on databases
  - Requirement easily met with wide-column database e.g., HBase (runs on top of HDFS)
  - Groups data together to store new data in a memory buffer and once buffer is full dumps the data to disk
  - Allows storage of a lot of small data quickly but also fetches rows by key or scan ranges of rows

Data partitioning
- Partition based on user ID so that we can keep all messages of a user on same database
- If one shard is 4TB then we will have 3.6PB/4TB=900 shards for five years
- Shard number = hash(user ID)%900
- Start with small number (< 900) of servers, partitioning scheme should map multiple logical partitions to one physical server so that we maintain 900 over a small number of servers
- As storage demand increases we can just add more physical servers to distribute our logical partitions

Cache
- Cache a few recent messages e.g., 15in a few recent conversations that are visible in user's viewport

Load balancing
- Load balancer in front of chat servers that can map user ID to a server that holds the connection for the user
- Load balancer for cache servers

Fault tolerance and replication
- Clients should automatically connect if connection lost (difficult to failover TCP connections)
- Create database replicas