**Chapter 12**

Design a Chat System

*Questions*

* What type of chat app? Should it focus on 1-on-1 communications or group-based?
* Is it a mobile, web or hybrid application?
* What scale is the application? How many daily users should it support?
* What is the max capacity of users for group chats?
* What can a user post or message in the chat? Can it support images, gifs, videos, audio, external links, etc.?

*Basic requirements*

* The chat service must support the following functions
  + Receives messages from other clients
  + Find the right recipients and relay the message
  + Hold the message on the server until a recipient is back online

***High-level overview***

* The choice of network protocols is important
* Requests are initiated by the client for most client/server applications
* HTTP – most common web protocol
  + The keep-alive is efficient to maintain a persistent connection with the chat and reduces the number of TCP handshakes
* Techniques for a server-initiated connection
  + Polling
  + Long polling
  + WebSocket

*Polling* – client periodically asks the server if there are messages available

* Costly and inefficient depending on the polling frequency

*Long polling* – client holds the connection open until there are new messages available, or a timeout threshold has been reached

* Once the client receives new messages, it immediately sends another request to the server, restarting the process
* Drawbacks
  + Sender and receiver may not connect to the same chat server
  + A server has no good way to tell if a client is disconnected
  + Inefficient if the chat is not active

*WebSocket*

* Most common solution for sending asynchronous updates from server to client
* Connection is initiated by the client
* Bi-directional and persistent
* Works even if a firewall is in place because it uses port 80 or 443 which are also used by HTTP/HTTPS connections
* Can be used for sending and receiving messages from both sides
  + Simplifies the design and makes implementation on both client and server more straightforward
  + Since WebSocket connections are persistent, efficient connection management is critical on the server-side
* Overall chat system can be broken down into 3 categories
  + Stateless services
  + Stateful services
  + Third-party services

A diagram of a computer system

Description automatically generated

*Stateless Service*

* Traditional public-facing request/response services
* Used to manage login, signup, user profile, etc.
* Sits behind a load balancer that routes request to the correct services based on the request paths
* Can be monolithic or individual microservices
* Many third-party services in the market available
* E.g. service discovery – give the client a list of DNS host names that client could connect to

*Stateful Service*

* Only stateful service is the chat service
* “Stateful” due to maintaining a persistent network connection to a chat server
* Service discovery coordinates closely with the chat service to avoid server overloading

*Third-Party Integration*

* For a chat app, push notification is the most important third-party integration
* Scalability
  + Small scale
    - All services can fit in one server
    - Number of concurrent connections a server can handle will most likely be the limiting factor
    - Single point of failure
  + Large scale – servers are split to manage different tasks
    - Chat servers – facilitates message sending/receiving
    - Prescence servers – manage online/offline status
    - API servers – handle requests like login, signup, change profile
    - Notification servers – sends push notifications
    - Key-value store – used to store chat history
* Storage
  + Generic data – user profile, user friends list
    - Stored in robust and relational databases
  + Chat history data
    - Enormous quantity of data
    - Only recent chats are accessed frequently
    - Features like random access of data is needed for searching, view mentions, or jump to specifics
    - The read to write ratio is about 1:1 for 1 on 1 conversations
  + Key-value stores to store chat history data:
    - Offers easy horizontal scaling
    - Low latency to access
    - Relational databases do not handle long tail of data well
    - Adopted by other proven reliable chat apps – Facebook uses HBase, and Discord uses Cassandra

*Message ID*

* Ensures the order of messages
* IDs must be unique and sortable by time
* Strategies to designing an ID generator (CH 7)
  + Use local sequence number generator – local means IDs are only unique within a group

***Design deep dive***

*Service Discovery*

* Recommend the best chat server for a client based on the criteria like
  + Geographic location
  + Server capacity
* Apache Zookeeper – a popular open-source solution
  + Registers all available chat servers
* How a service discover works

1. User attempts to login to an app
2. Load balancer sends requests to API servers
3. After authentication, service discovery finds the best chat server for user
4. User connects to chat server through WebSocket

*Message flows*

*1 on 1 chat flows*

* If the recipient is online – a message pushed directly to the server they are connected through WebSocket
* If the recipient is offline – a push notification is sent

*Message synchronization across multiple devices*

* Each device maintains a *cur\_max\_message\_id* to keep track of the latest message ID on the device and identifies which messages are new to the device

*Small group chat flows*

* A message is copied to each group members’ message sync queue
* For small groups, storing copies for each member is not too expensive

*Online presence*

* Presence servers are responsible for managing online status and communicating with clients through WebSocket

*User disconnection*

* Realtime method – make user as offline when disconnected and online when user connection is re-established
  + It is common for users to disconnect and reconnect in short time frames
  + Making presence indicator flicker on and off results in poor user experience
* Heartbeat mechanism – periodically an online client sends a heartbeat event to presence events
  + If the presence servers received a heartbeat event within a certain time, a user is considered online

A diagram of a patient

Description automatically generated

*Online status fanout*

* For small groups
  + Use a channel to fanout updates is good enough
  + When a user’s online status change, it publishes the event to every friends’ channel
* For large groups
  + The above approach is expensive and will be a performance bottleneck
  + Possible solution – only fetch online status when a user enters a group or manually refreshes the friend list

*Extra topics*

* Extend chat app to support media files
  + Compression, cloud storage and thumbnails are interesting topics
* End-to-end encryption
  + Only the send and recipient can read messages
* Caching messages on the client-side
  + Efficient to reduce data transfer between client and server
* Improve load data and error handling
  + Server discovery can provide a new chat server if one goes offline
  + Message resent mechanisms