Collaborative Document Editor

Variants

- Design Google docs.
- Design a collaborative code editor like Coderpad/Codepile.
- Design a collaborative markdown editor.

Requirements Gathering

- What is the intended platform?
    - Web
- What features are required?
    - Creating a document
    - Editing a document
    - Sharing a document
- Bonus features
    - Document revisions and reverting
    - Searching
    - Commenting
    - Chatting
    - Executing code (in the case of code editor)
- What is in a document?
    - Text
    - Images
- Which metrics should we optimize for?
    - Loading time
    - Synchronization
    - Throughput

Core Components

- Front end
    - WebSockets/long polling for real-time communication between front end and back end.
- Back end services behind a reverse proxy.
    - Reverse proxy will proxy the requests to the right server.
    - Split into a few services for different purposes.
    - The benefit of this is that each service can use different languages that best suits its purpose.

- API servers for non-collaborative features and endpoints.
    - Ruby/Rails/Django for the server that deals with CRUD operations on data models where performance is not that crucial.
- WebSocket servers for handling document edits and publishing updates to listeners.
    - Possibly Node/Golang for WebSocket server which will need high performance as updates are frequent.
- Task queue to persist document updates to the database.
- ELB in front of back end servers.
- MySQL database.
- S3 and CDN for images.

Data Modeling

- What kind of database to use?
    - Data is quite structured. Would go with SQL.
- Design the necessary tables, its columns and its relations.
    - `users`
        - `id`
        - `name`
    - `document`
        - `id`
        - `owner_id`
    - `permissions`
        - `id`
        - `name`
    - `document_permissions`
        - `id`
        - `document_id`
        - `user_id`

Collaborative Editing - Client

- Upon loading of the page and document, the client should connect to the WebSocket server over the WebSocket protocol `ws://`.
- Upon connection, perform a time sync with the server, possibly via Network Time Protocol (NTP).
- The most straightforward way is to send the whole updated document content to the back end, and all users currently viewing the document will receive the updated document. However, there are a few problems with this approach:
    - Race condition. If two users editing the document at the same time, the last

    one to edit will overwrite the changes by the previous user. One workaround is to lock the document when a user is currently editing it, but that will not make it real-time collaborative.
  - A large payload (the whole document) is being sent to servers and published to users on each change, and the user is likely to already have most of the content. A lot of redundant data being sent.

- A feasible approach would be to use operational transforms and send just the action deltas to the back end. The back end publishes the action deltas to the listeners. What is considered an action delta?
  - (a) Changing a character/word, (b) inserting a character/word/image, (c) deleting a character/word.
  - With this approach, the payload will contain only small amount of data, such as (a) type of change, (b) character/word, (c) position in document: line/column, (d) timestamp. Why is the timestamp needed? Read on to find out.

- Updates can also be throttled and batched, to avoid flooding the web server with requests. For example, if a user inserts a

Back End

The back end is split into a few portions: WebSocket server for receiving and broadcasting document updates, CRUD server for reading and writing non-document-related data, and a task queue for persistence of the document.

WebSocket Server

- Languages and frameworks that support async requests and non-blocking I/O will be suitable for the collaborative editor server. Node and Golang comes to my mind.
- However, the WebSocket server is not stateless, so is it not that straightforward to scale horizontally. One approach would be for a Load Balancer to use Redis to maintain a map of the client to the WebSocket server instance IP, such that subsequent requests from the same client will be routed to the same server.
- Each document corresponds to a room (more of namespace). Users can subscribe to the events happening within a room.
- When a action delta is being received, blast it out to the listeners within the room and add it to the task queue.

CRUD Server

- Provides APIs for reading and writing non-document-related data, such as users, permissions.

Task Queue + Worker Service

- Worker service retrieves messages from the task queue and writes the updated documents to the database in an async fashion.
- Batch the actions together and perform one larger write that consists of multiple actions. For example, instead of persisting to the database once per addition of a word, combine these additions and write them into the database at once.

- Publish the save completion event to the WebSocket server to be published to the listeners, informing that the latest version of the document is being saved.
- Benefit of using a task queue is that as the amount of tasks in the queue goes up, we can scale up the number of worker services to clear the backlog of work faster.

References

- http://blog.gainlo.co/index.php/2016/03/22/system-design-interview-question-how-to-design-google-docs/ (http://blog.gainlo.co/index.php/2016/03/22/system-design-interview-question-how-to-design-google-docs/)