Detailed System Flow (CRDT Transmission)

1. **Client Authentication and Authorization:**
   - A Client (user) sends an authentication request (username/password) to the API Gateway.
   - API Gateway routes request to Load Balancer.
   - Load Balancer distributes the request to a Server in the Server Cluster.
   - Server verifies the credentials against the *user* entity in PostgreSQL.
   - Upon successful verification, the Server issues a JWT token and sends it to the Client via the Load Balancer.
   - Client includes the JWT token in subsequent authorized requests.
   - The Load bBalancer distributes authorized requests to the Server Cluster, where the Server verifies the token against the *permission* entity in PostgreSQL.

2. **New Document Creation:**
   - Client sends a *"create new document"* request (JWT, optional title) to the API Gateway.
   - API Gateway routes request to Load Balancer.
   - Load Balancer distributes the request to a Server in the Server Cluster.
   - Server verifies authorization against the *permission* entity in PostgreSQL (checking if the user has permission to create documents).
   - Server creates a new *documents* entity in the MongoDB Cluster, generating a new documentId.
   - Server creates the initial version in *document_versions* entity.
   - Server sends the new documentId to the Client.

3. **Initial Document Load and WebSocket Connection:**
   - Client sends a load document request (document ID, JWT) to the API Gateway.
   - Client establishes a WebSocket connection with the Load Balancer (sticky sessions enabled).
   - Load Balancer distributes the document request and WebSocket connection to a Server in the Server Cluster.
   - Server verifies authorization against the *permission* entity in PostgreSQL.
   - Server checks Redis for cached document snapshot.
   - If not in Redis, Server retrieves the latest *document_versions.snapshot* (Serialized Document Data) from the MongoDB Cluster, based on the documentId.
   - Server deserializes the snapshot data and creates an in-memory representation of the document.
   - Server sends the deserialized document data to the Client.

4. **CRDT Operation Processing:**
   - Client makes an edit to the document in the browser.
   - Client applies CRDT Operation to local in-memory document.
   - Client determines when to send the CRDT Operation based on a hybrid approach:
     - a. When the user pauses typing for a maximum delay (e.g., 200ms).
     - b. At a regular interval as a safety net (e.g., every 2 seconds).
     - c. Upon reaching a character threshold (e.g., 20 characters).
   - Client sends the CRDT Operation (including documentId, uniqueId, versionVector and event data) to the Server via websocket.
   - Server acknowledges receipt to the Client.
   - Server checks the documentId, uniqueId, versionVector against its own for idempotency.
   - If the operation is new, the Server merges the operationData into its in-memory Document state.
   - Server obtains a distributed lock on the CRDT Operation entity in the MongoDB Cluster.

- Server writes the raw *crdt_operations* entity to the MongoDB Cluster.
- Server stores CRDT Data to *documents* entity in the MongoDB Cluster.
- Server publishes the CRDT Operation (including documentId, uniqueId, versionVector) to Kafka.
- Server releases the distributed lock.
- Server sends an acknowledgement to the Client.
- Server broadcasts operation to all connected Clients for the same document.

5. **Kafka Consumer Processing:**
   - A Kafka consumer receives the CRDT Operation and details (documentId, uniqueId, versionVector) from Kafka.
   - The consumer checks the documentId, uniqueId, versionVector for idempotency.
   - If the operation is new, the consumer merges the operationData into its in-memory Document state.
   - The consumer stores the documentId, uniqueId, versionVector in the *last_processed_version* entity in MongoDB.
   - The consumer periodically creates a snapshot of the *Document* (from its in-memory representation) and stores it as a new Document_Version entity in the MongoDB Cluster.
   - The consumer caches the snapshot in Redis.

6. **Client-Side Operation Application:**
   - Server sends the CRDT Operation (including versionVector) to all connected Clients based on documentId via WebSocket.
   - Client checks the versionVector against its own for idempotency.
   - If the operation is new, the Client merges the operationData into its local in-memory Document state.
   - Client updates the rendered document in the browser.

7. **Document Versioning (History):**
   - Client sends a request for a specific document version to the Load Balancer.
   - Load Balancer distributes the request to a Server in the Server Cluster.
   - Server checks for the snapshot in Redis.
   - If the snapshot is in Redis, the Server sends it to the Client via the Load Balancer.
   - If the snapshot is not in Redis, the Server retrieves it from the Document_Version entity in the MongoDB Cluster, caches it in Redis, and sends it to the Client via the Load Balancer.

8. **Kafka Failure Recovery:**
   - If Kafka is unavailable, the Server retrieves pending CRDT Operation entities from the MongoDB Cluster.
   - Server applies the pending operationData to its in-memory Document state.
   - Server stores the latest versionVector from the applied operations in the *last_processed_version* entity in MongoDB.

9. **Server Failure Recovery:**
   - If a Server fails, a new Server retrieves the last processed versionVector from the *last_processed_version* entity in MongoDB.
   - Server retrieves any CRDT Operation entities from the MongoDB Cluster with a timestamp greater

than the timestamp associated with the last processed version vector.

- Server retrieves the latest *document_versions* entity from the MongoDB Cluster.
- Server applies the retrieved CRDT Operation entities to the snapshot from the *document_versions* entity, to reconstruct the in-memory Document state.
- Further explanation on how does the version vector works in here
  - A Server in the Cluster goes down.
  - A new Server comes online to replace it.
  - The new Server queries the *last_processed_version* entity in MongoDB to get the last processed version vector for each document it needs to handle.
  - Server then queries the *crdt_operations* entity in MongoDB to retrieve any operations that have occurred *after* the point indicated by the *last_processed_version*. This means it retrieves operations whose version vectors indicate they haven't been seen.
  - Server applies these missing operations, in causal order (determined by the version vectors), to its in-memory document state.
  - Server is now caught up and consistent with the rest of the system.