**📝 Deep Dive: Google Docs System Design (Collaborative Document Editing)**

**✅ 1. Functional Requirements**

* Create/Edit rich-text documents
* Real-time collaborative editing
* User presence (cursors, names)
* Commenting and suggestions
* Version history
* Document sharing and permissions (view/edit)
* Autosave and offline support

**🔒 Non-Functional Requirements**

* High availability
* Low latency real-time updates
* Conflict-free collaboration
* Scalability to millions of users
* Secure storage and sharing

**🌟 Story: A Multi-User Google Docs Editing Session**

**🎬 Chapter 1: Alex Starts the Document**

**Alex** opens Google Docs and creates a **new document**. As soon as Alex opens the document:

1. The **client** (browser or app) connects to the **backend server** using **WebSockets**.
2. The backend server establishes a **real-time connection** with Alex’s document in the database, and Alex’s cursor appears at the top of the page.
3. Alex starts typing the title: “**My Awesome Project**.”

**Key Concept**:

* **WebSocket** is a protocol that allows **bidirectional communication** between the client and server, so changes can be sent back and forth instantly.

**🎬 Chapter 2: Jordan Joins the Document**

**Jordan**, a collaborator, clicks on the shared link Alex sent. Now, here’s where the real-time magic happens:

* **Jordan's client** also connects to the **backend via WebSocket**.
* Jordan’s cursor shows up **at the same time** as Alex's. If Alex is typing, Jordan can see those changes in real-time.
* Both of them can see each other’s **cursor positions** in the document.

**🔄 Chapter 3: Real-Time Collaboration Begins – The Power of CRDT or OT**

Now, let’s get into how **real-time editing** works and how Google Docs keeps everyone’s document synchronized:

1. **Alex** types a sentence: **“This is a great project.”**
2. **Jordan**, in a different part of the document, types: **“Let’s start working on it.”**

The edits are sent from both clients to the **backend server**, which needs to figure out how to merge these two changes, ensuring that both users see the same thing despite typing in different places. This is done using two primary methods:

* **Operational Transformation (OT)**
* **Conflict-Free Replicated Data Types (CRDT)**

**🧠 Operational Transformation (OT)**

* **OT** works by **transforming** operations (like insert, delete) to ensure that operations are applied in the correct order, even when the users make changes simultaneously.
* **How OT works**:
  1. Alex's insert operation (typing “This is a great project”) is sent to the server.
  2. Jordan’s insert operation (typing “Let’s start working on it”) is also sent.
  3. The server **transforms** the operations to maintain the correct order, applying them in such a way that the final document appears consistent for both users.
  4. The transformed operations are sent back to **both clients**, ensuring their local versions match.

**OT ensures no conflict** occurs even if two people are editing the same part of the document at the same time.

**🧠 Conflict-Free Replicated Data Types (CRDT)**

* **CRDT** is an alternative approach that avoids needing a central server for coordination. It allows all users to **edit independently** and ensures the final document **converges** to the same state.
* Each client stores a **local copy** of the document. When changes happen:
  1. The client **generates an update** (like inserting a character).
  2. It broadcasts the update to other clients.
  3. Clients **merge updates locally** without needing to ask the server for the final version.

CRDT can be more **distributed** because it doesn't rely as much on a central server for conflict resolution, but it’s more complex to implement.

**🧮 Chapter 4: Maintaining Real-Time Synchronization**

Now that both users are editing, how does the system ensure everyone stays in sync?

1. **Event-driven architecture** is key here:
   * Every time Alex or Jordan makes an edit, it generates an **event** (e.g., insert, delete, move cursor).
   * These events are sent to a **message queue** (like **Kafka**), and the system **broadcasts them** to all clients that are viewing the document.
2. **Versioning**: The server tracks the **version number** of the document. If a client loses connection and then reconnects, it can **sync to the latest version** by fetching any events that were missed.
3. **Latency Handling**: When there’s network lag, the system queues events and applies them when the connection is stable. This is why changes appear "instantly" even if the network connection is slow.

**🧳 Chapter 5: Autosave & Version History**

* **Autosave** is one of the core features of any online document editor. As both Alex and Jordan type, their changes are **automatically saved** in the background.
* Every time a change is made, a **snapshot** of the document (or the **delta** of changes) is saved.
* The **server** periodically saves versions of the document into a **database** (like **Google Bigtable** or **Cassandra**), ensuring that even if the app crashes, no work is lost.

**🧠 Versioning:**

* Google Docs stores each **version** of the document, allowing users to **restore** to a previous version.
* The **version history** is stored as a **series of deltas** (changes). For example:
  1. Version 1: "Hello, world!"
  2. Version 2: "Hello, world! This is a test."
  3. Version 3: "Hello, world! This is a test. Let’s edit it together."

This allows Alex and Jordan to easily **rollback** or **track changes**.

**🔒 Chapter 6: Permissions and Access Control**

* **Permissions** ensure that only authorized users can edit or view the document.
* Google Docs uses a **role-based access control (RBAC)** system where users can have different levels of access:
  + **Viewer** (can only read)
  + **Commenter** (can comment but not edit)
  + **Editor** (can edit the document)

This is handled via **OAuth** for authentication and **JWT tokens** for secure sessions.

**The backend** uses these roles to control access, ensuring that unauthorized users cannot make changes.

**🕒 Chapter 7: Offline Support and Syncing**

Sometimes, the internet connection drops. What happens then?

1. **Local Caching**: Both Alex and Jordan’s clients store **local copies** of the document on their devices.
2. **Local Changes**: When offline, the clients **track changes** (in-memory) as they type.
3. When the connection returns, the **local changes are synced** with the backend. The system resolves conflicts if both users made different changes in the same part of the document during the offline period.

This ensures a smooth user experience, where you can keep working even without an internet connection.

**🧑‍🤝‍🧑 Chapter 8: Real-Time Presence & Cursors**

* As both users work on the document, their **cursors** are visible to each other in real-time.
* The backend **tracks cursor positions** using an **in-memory store** (like Redis), sending updates to clients whenever someone moves or clicks their cursor.
* Each user’s **cursor is color-coded** and appears in the document, allowing everyone to see where others are working.

**🎯 Chapter 9: Wrapping It All Together**

Let’s recap how the system works:

1. **Real-time collaboration**: WebSocket ensures that all users see each other's changes in real-time.
2. **Conflict resolution**: CRDT or OT ensures no conflicts happen when two users make changes at the same time.
3. **Versioning & Autosave**: Changes are automatically saved and versioned, with deltas stored for rollback.
4. **Offline support**: Local caching and syncing ensure you can keep working, even without an internet connection.
5. **Permissions & Security**: OAuth and RBAC handle authentication and authorization.

**📊 Key Concepts & Algorithms**

| **Concept/Algorithm** | **Explanation** |
| --- | --- |
| **WebSockets** | Real-time communication between client and server (low-latency). |
| **OT (Operational Transformation)** | Resolves conflicts by transforming operations applied to the document. |
| **CRDT (Conflict-Free Replicated Data Types)** | Allows concurrent changes without needing a central authority, ensuring eventual consistency. |
| **Versioning** | Keeps track of document changes over time, allowing rollbacks and viewing change history. |
| **Caching & Syncing** | Ensures offline work is preserved and synced when the internet connection returns. |
| **Permissions (RBAC)** | Controls user access and actions (view, comment, edit) based on roles. |
| **Cursor Tracking** | Shows real-time positions of each user’s cursor for smooth collaboration. |

**🎯 Conclusion**

Through this story, we saw how **Google Docs** (or a similar system) provides seamless collaboration by:

* Using **real-time synchronization** techniques (OT or CRDT)
* Offering **offline support** and **autosaving** to prevent data loss
* Implementing **version control** and **permissions** to track document history and ensure secure access
* Using **cursors and presence** to make the collaborative experience more engaging.

**🧱 2. High-Level Architecture**

+------------+ +----------------+ +------------------+

| Frontend | <-> | WebSocket/GW | <-> | Collab Service |

+------------+ +----------------+ +------------------+

| | |

v | v

+---------------+ +-----------+ +--------------------+

| Document Store| <--> | Auth Svc | | Operational Log |

| (S3, Spanner)| +-----------+ +--------------------+

|

v

+-----------------+

| Versioning Svc |

+-----------------+

**📑 3. Real-Time Collaboration**

**CRDT (Conflict-free Replicated Data Type)**

* Each user sends operations (insert/delete/format) instead of full doc.
* All changes are timestamped and mergeable.
* CRDT ensures convergence regardless of order of delivery.

**Alternatives:**

* **Operational Transform (OT)**: Used in early Google Docs.
* CRDT preferred for distributed peer-based editing.

**Key Algorithms:**

* RGA (Replicated Growable Array)
* LSEQ for position tracking

**🔄 4. Document Change Flow**

1. User types character "A"
2. Operation: Insert(A, pos=5, user\_id, timestamp)
3. Operation sent via WebSocket to Collab Service
4. Collab Service rebroadcasts to other users
5. Operation applied to local doc + persisted to log
6. Periodic autosave to durable storage (e.g., S3)

**💾 5. Storage & Persistence**

* **Document Store**: Finalized versions in Google Cloud Storage / Spanner
* **Operation Log**: Append-only log of changes (for replay/version recovery)
* **Versioning Service**: Snapshots at intervals, used to reconstruct versions

**👥 6. User Presence and Cursors**

* Frontend sends cursor position on change or interval
* Collab Service maps user\_id to cursor + metadata (color, name)
* Cursor states are broadcast to other users in the doc

**🔐 7. Permissions & Sharing**

* ACL-based sharing (view, comment, edit)
* Auth Service with JWT/OAuth tokens
* Docs organized per user/organization

**Permission Scopes:**

* read\_only, comment\_only, edit, owner

**🛠️ 8. Offline Support**

* Local buffer with CRDT queue
* On reconnect, replay operations to sync
* IndexedDB/cache stores doc snapshot locally

**🧠 9. Interview Insights**

|  |  |
| --- | --- |
| **Topic** | **Concept/Tool** |
| Real-time sync | WebSockets, CRDT |
| Conflict resolution | CRDT / OT |
| Persistence | Append-only logs |
| Versioning | Snapshot diffs |
| Permissions | Role-based ACL |
| Offline support | Local storage + sync |

**📌 Common Questions**

* How do you handle network splits?
* How do you resolve simultaneous edits?
* CRDT vs OT trade-offs?
* What if two users delete the same word?
* How would you support media or comments?
* How do you rollback to an earlier version?