

CASE STUDY

Intelligent Floor Plan Management System for a Seamless Workspace Experience

— Mukul Jain (200001050)

Admin's Floor Plan Management

- **Develop a conflict resolution mechanism for simultaneous seat/room information uploads.**
- **Implement a version control system to track changes and merge floor plans seamlessly.**
- **Resolve conflicts intelligently, considering factors such as priority, timestamp, or user roles.**

The Floor Plan Management System described in the problem description suffers from conflicts due to simultaneous uploads and requires conflict resolution. This problem can be approached by observing it analogous to **Google Docs**.

Google Docs also suffers from simultaneous/collaborative editing and requires real-time conflict resolution. The solutions it adopts are precisely the same as those required here for the Floor Plans.

Real-Time Updates:

To get real-time updates in case the other users are editing/uploading the floor plans, one can either use Polling or Web Sockets. However, polling suffers from the problem of increasing unnecessary network traffic, which could slow down the system and cost the company dearly. One should go for **Web Sockets**. They allow for a full duplex connection between the client and server efficiently and in real-time.

Versioning:

One can use a Time Series DB to store all the floor plans and the authors' names to implement versioning in Floor Plans. InfluxDB is one such time series database that can be used for this purpose. However, this will unnecessarily increase the storage space required for saving the same plan with minor edits. Similarly, tracking changes would mean comparing the two plans entirely.

Alternatively, **Merkle Tree** can be used for version control. It creates a hash for a chunk of information and stores the hashes as the leaf nodes. Then, the hash of the combination of children nodes is calculated to get the parent node in **$O(n)$ time complexity**.

Further, changes can be tracked down the tree by comparing the hashes in **$O(\log n)$ time complexity**. Not only will this reduce the storage required, but will also be faster in tracking changes between the plans.

Conflict Resolution:

One can resolve conflicts by locking the editor for a user while the other user is editing, similar to semaphores/mutex locks. However, this would cause unnecessary delays for the other users who get locked out, nor will this system be called robust.

For conflicts to be resolved, it should take care of:

- **Commutativity:** The order of applied operations shouldn't affect the end result.
- **Idempotency:** Similar operations that have been repeated should apply only once.

Google Docs resolve conflicts using a lock-free, non-blocking approach called **Operation Transformation**. The server keeps the original copy as the document's current state. An operation consists of (what, where and whom) sent to the server for processing. If multiple users modify the same section of a document simultaneously, OT detects these conflicts by comparing the order in which the server received them. The operation is applied directly to the document if there's no conflict. When conflicts occur, OT resolves these conflicts by transforming the operations to maintain the document's integrity and ensure consistency to accommodate both changes without altering the intended meaning.



Offline Mechanism for Admins

- **Develop a local storage mechanism for admins to make changes offline.**
- **Implement synchronization to update the server when the internet or server connection is re-established.**
- **Ensure data integrity and consistency during offline and online transitions.**

Local Storage:

Local storage could be a browser storage mechanism to store local copies. When the user accesses them online, the browser caches them to enable offline access.


While online, any changes made to documents are synchronised with servers in real-time. These changes are also saved locally in the browser's storage, ensuring that the most recent version is available offline.

Service Workers:

Service workers allow for the caching of assets and documents, enabling offline access by serving locally cached content when the user is offline. Once the device reconnects to the internet, it syncs the locally saved changes with the server. The changes made offline are uploaded, and operation transformers could incorporate them.

Meeting Room Optimization

- **Develop a meeting room booking system considering the number of participants and other requirements.**
- **Implement a recommendation system to suggest meeting rooms based on capacity and proximity.**
- **Ensure dynamic updates to meeting room suggestions as bookings occur and capacities change.**
- **Show the preferred meeting room based on the last booking weightage.**



A meeting room can be defined by the number of participants. A list can be created that stores the room's capacities in sorted order. All the requirements will also be stored in another list in sorted order.

For each requirement, the room which has the equivalent or just bigger capacity will be allocated by using Binary Search with Time Complexity $O(\log n)$. As soon as a room gets unoccupied, the requirement which is equivalent or just smaller than the capacity will be allocated by using Binary Search with Time Complexity $O(\log n)$.

Other Points

Authentication:

Authentication and Authorisation can be done using the OAuth2 Framework by giving Role Based Access Control to the users.

System Failure:

One should use **distributed storage systems** like Google Cloud Storage or Amazon S3 to store documents and prevent single-point-of-failure. **Gossip protocol** can be used to share the floor plans and maintain consistency among the nodes.

Implementing **sharding or partitioning** techniques for **scalability**. Using **replication and backups** to ensure data **durability**. Storing document metadata in a database for efficient retrieval by implementing indexing.


Further, **encryption** should be employed before storing the data to protect it from attacks.

Object-Oriented Programming Language (OOPS):

Languages like Java and Python can be used to create the backend. These are Object-Oriented Languages with a broad user base, thus providing existing code bases for writing codes.

Scalability:

Distributing incoming traffic across multiple servers to prevent overload through **load balancing** will help achieve optimal performance. Separating Meeting Room Optimization



from the other two would help create a smaller, independent **microservice** for easy scalability and deployment. Utilising **Content Delivery Networks** and caching mechanisms to reduce latency and enhance performance.

System Monitoring:

Elastic Search can be used to log the data and query the logs in case of faults.