MoveinSync Technology

Intelligent Floor Plan Management System

1st December 2023

Introduction

An Intelligent Floor Management System is a sophisticated solution designed to optimize and streamline various aspects of floor operations within a physical space, such as a building or facility. Leveraging advanced technologies like sensors, data analytics, and automation, this system aims to enhance efficiency, resource utilization, and overall productivity. Key features often include real-time monitoring, data-driven decision-making, and the integration of smart technologies for better control over space utilization, security, and energy management. Such systems find applications in diverse environments, ranging from smart offices and commercial buildings to healthcare facilities and manufacturing plants. Through intelligent data analysis and automation, these systems contribute to creating smarter, more responsive, and resource-efficient spaces.

OBJECTIVES

1. Develop features for administrators to upload floor plans
2. Implement an offline mechanism for admins to update the floor plan in scenarios of internet connectivity loss or server downtime
3. Enhance the system to optimize meeting room bookings, suggesting the best meeting room based on capacity and availability

OBJECTIVE :DEVELOP FEATURES FOR ADMINISTRATORS TO UPLOAD FLOOR PLANS

Mechanism:

Let us assume that we have k number of admins and n number of floors in our building. Let us define the version for each floor i(1<=i<=n) version[i] consists of a version number, the current floor plan of that floor and the admin that made that version. Initial version number (just after the building is built and the admins are elected) is version 0.

Update-Before making any update, the admin is required to log in to the system using the admin id and email. After that the admin is asked to get the current version of that floor. The admin then makes changes to the floorplan of that floor. If there are no conflicts then changes are added and the version number of the floor is incremented by 1. The pseudocode for that(assuming no conflicts) is as given below

```
class FloorManagementSystem:

    def __init__(self, num_floors, num_admins):

        self.num_floors = num_floors

        self.num_admins = num_admins

        self.versions = {i: {'version_number': 0, 'floor_plan': None, 'admin_id': None} for i in
range(1, num_floors + 1)}


    def login_admin(self, admin_id, admin_email):

        # Implement admin login logic (authentication)


    def get_current_version(self, floor):

        return self.versions[floor]


    def make_floor_changes(self, floor, admin_id, new_floor_plan):
        current_version = self.versions[floor]

        # Verify admin credentials and get the current version
        if not self.login_admin(admin_id, admin_email):
            print("Invalid admin credentials. Unable to make changes.")
            return

        # Check for conflicts (pseudo logic, actual conflict detection depends on your application)
        if self.check_for_conflicts(current_version['floor_plan'], new_floor_plan):
            print("Conflict detected. Unable to make changes.")
            return

        # Update the floor plan and increment the version number
        self.versions[floor] = {
            'version_number': current_version['version_number'] + 1,
            'floor_plan': new_floor_plan,
            'admin_id': admin_id
        }

        print("Changes applied successfully.")

    def check_for_conflicts(self, current_floor_plan, new_floor_plan):
        # Implement conflict detection logic (depends on your application)
    return False
```

Conflict Resolution: We assign a priority number to each admin. Admin with higher priority would be more powerful than the admin with low number. The conflicts can be of the following types:

1) An employee is assigned 2 different seats on the same floor.
2) An employee is assigned 2 different seats on different floors.

So when the admin makes changes in the floorplan of the particular floor and before the changes are committed and a new version is made, the changed floorplan is compared with the current versions of every single floor. Let us say that our current floor is i and the floor being compared is j and there is a conflict.

If the admin i has lower priority, then the admin i has to change the floorplan again so as to incorporate the changes made by admin j.

If admin i has higher priority, then admin i can straightaway override the changes made by admin j. If admin j wants his/her changes, he/she needs to contact admin i.

```
class FloorManagementSystem:

  def __init__(self, num_floors, num_admins):

    self.num_floors = num_floors

    self.num_admins = num_admins

    self.versions = {i: {'version_number': 0, 'floor_plan': None, 'admin_id': None} for i in
range(1, num_floors + 1)}

    self.admin_priorities = {i: None for i in range(1, num_admins + 1)}
```

```
def login_admin(self, admin_id, admin_email, priority):

    # Implement admin login logic (authentication)

    self.admin_priorities[admin_id] = priority


def get_current_version(self, floor):

    return self.versions[floor]


def make_floor_changes(self, floor, admin_id, new_floor_plan):

    current_version = self.versions[floor]


    # Verify admin credentials and get the current version

    priority_i = self.admin_priorities[admin_id]


    # Check for conflicts with other floors

    for j in range(1, self.num_floors + 1):

        if j != floor:

            conflict = self.check_for_conflicts(current_version['floor_plan'],
self.versions[j]['floor_plan'])

            if conflict:

                priority_j = self.admin_priorities[self.versions[j]['admin_id']]

                if priority_i > priority_j:

                    # Admin i has higher priority, override changes made by admin j

                    self.versions[j] = {

                        'version_number': current_version['version_number'] + 1,

                        'floor_plan': new_floor_plan,
                    'admin_id': admin_id
                }
            else:
                # Admin i has lower priority, update floorplan for admin i
                # Notify admin j about the conflict
                print(f"Conflict detected with admin {self.versions[j]['admin_id']} on floor {j}. "
                      f"Admin {admin_id} needs to resolve the conflict.")

        # Update the floor plan for the current floor
        self.versions[floor] = {
            'version_number': current_version['version_number'] + 1,
            'floor_plan': new_floor_plan,
            'admin_id': admin_id
        }

        print("Changes applied successfully.")

    def check_for_conflicts(self, floor_plan_i, floor_plan_j):
        # Implement conflict detection logic (depends on your application)
        # This is a placeholder and might involve comparing the floor plans.
        # You need to define your conflict detection logic based on the structure of your floor
plans.

        return False
```

OBJECTIVE :DEVELOP FEATURES FOR ADMINISTRATORS TO UPLOAD FLOOR PLAN

Mechanism:

Each admin will have files of the current version of all the floors they are making changes in their local storage until the changes are committed. Once the changes are committed, the file is removed from the local storage.

Updates after re-establishment of the Internet: Once the Internet is back, the information in the cache will be transferred and the conflict handling will be the same as above.

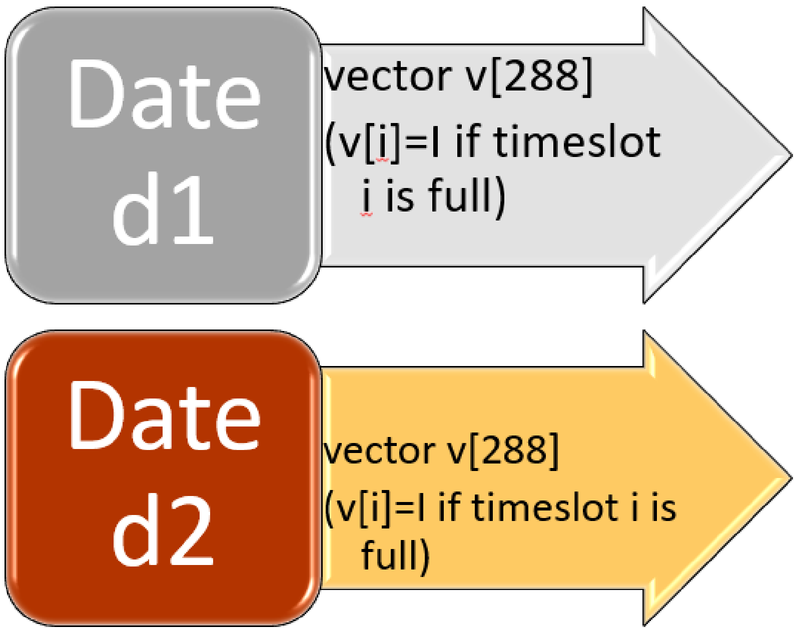
OBJECTIVE :MEETING ROOM OPTIMIZATION

Mechanism:

There will be a room class containing the following information about the room.

Room Number
Floor Number
Max Capacity
Bookings

The bookings would be a map consisting of key,value pairs as shown below.



Date d1
vector v[288]
(v[i]=I if timeslot i is full)
Date d2
vector v[288]
(v[i]=I if timeslot i is full)

Key would be the date and value would be a one hot encoding vector of size 288.Here we assume that the meeting will last for at least 5 minutes. So we divide the entire day into slots of 5 minutes, like slot 0 means time from 12 to 12:05, slot 1 means time from 12:05 to 12:10 and so on. Let us suppose that the vector is v and the time slot is t. Then the meeting room is free at time t, then v[t]=1 else the room is booked.

Booking Process: The user who wants to book a room will be asked the floor number preferred, the start time(in 24 hr clock format), the end time(in 24 hr clock format) and the number of attendees. We will iterate through every room in the building and find out whether that room is available in the time slot on that date we want.The process will be as follows.

The start time and the end time would be in hr,min format with minute being multiple of 5. Let us say that the hour is h and the minute is m, then the corresponding index of this slot in the encoding array would be h+m/5.

Thus in this way we find the indices of the start and the end time of the required time slots. Now we can use the Segment Tree with Lazy propagation to find whether that slot is available or not. If the indices of the time slots are i1,i2 then if sum(i1)==sum(i2) then the slot is free. Here sum(i) denotes the prefix sum up to i index.

Now if the slot is free, then we can update the range[i1,i2] to 1 using lazy propagation in segment Tree. Thus in this way the search time can be reduced to O(Nlog(288)) where N=number of rooms. The code for Segment Tree with Lazy propagation is as given below.

```
class LazySegmentTree:

    def __init__(self, arr):

        self.n = len(arr)

        self.tree = [0] * (4 * self.n)

        self.lazy = [0] * (4 * self.n)

        self.construct_segment_tree(arr, 0, self.n - 1, 0)


    def construct_segment_tree(self, arr, start, end, index):

        if start == end:

            self.tree[index] = arr[start]

            return


        mid = (start + end) // 2

        self.construct_segment_tree(arr, start, mid, 2 * index + 1)

        self.construct_segment_tree(arr, mid + 1, end, 2 * index + 2)

        self.tree[index] = self.tree[2 * index + 1] + self.tree[2 * index + 2]


    def update_range(self, start_range, end_range, delta):

        self.update_range_util(0, self.n - 1, start_range, end_range, delta, 0)
```

```
def update_range_util(self, start, end, start_range, end_range, delta, index):

    if self.lazy[index] != 0:

        self.tree[index] += (end - start + 1) * self.lazy[index]


        if start != end:

            self.lazy[2 * index + 1] += self.lazy[index]

            self.lazy[2 * index + 2] += self.lazy[index]


        self.lazy[index] = 0


    if start > end or start > end_range or end < start_range:

        return


    if start_range <= start and end_range >= end:

        self.tree[index] += (end - start + 1) * delta


        if start != end:

            self.lazy[2 * index + 1] += delta

            self.lazy[2 * index + 2] += delta


        return


    mid = (start + end) // 2

    self.update_range_util(start, mid, start_range, end_range, delta, 2 * index + 1)
```

```
        self.update_range_util(mid + 1, end, start_range, end_range, delta, 2 * index + 2)

        self.tree[index] = self.tree[2 * index + 1] + self.tree[2 * index + 2]


    def get_sum(self, start_range, end_range):

        return self.get_sum_util(0, self.n - 1, start_range, end_range, 0)


    def get_sum_util(self, start, end, start_range, end_range, index):

        if self.lazy[index] != 0:

            self.tree[index] += (end - start + 1) * self.lazy[index]


            if start != end:

                self.lazy[2 * index + 1] += self.lazy[index]

                self.lazy[2 * index + 2] += self.lazy[index]


            self.lazy[index] = 0


        if start_range <= start and end_range >= end:

            return self.tree[index]


        if start > end or start > end_range or end < start_range:

            return 0


        mid = (start + end) // 2

        left_sum = self.get_sum_util(start, mid, start_range, end_range, 2 * index + 1)
```

```
right_sum = self.get_sum_util(mid + 1, end, start_range, end_range, 2 * index + 2)

return left_sum + right_sum
```

Recommendation: We now get all the available rooms in that time slot. Now we will only recommend those rooms to the user whose capacity<=1.5*number of attendees. The recommendations will be sorted in the order of proximity from the user's preferred floor. Now it is for the user to decide which rooms he/she wants to book.

Conflict handling and Updation: We can introduce versioning for time slots in each date to prevent lost update queries. Each date will have a version. Thus a user can only book a time slot if the version the user read for booking is still the same. Otherwise the user will get conflict message.