Control Document

SAMSKRUTI COLLEGE OF ENGINEERING & TECHNOLOGY
Approved by AICTE & Affiliated to JNTU Hyderabad

Accreditedby NAAC With'A'Grade | AnAutonomous Institution | Accreditedby NBA

Department of Computer Science Engineering(AIML)

SAMSKRUTI COLLEGE OF ENGINEERING AND TECHNOLOGY

(Approved by AICTE, Autonomous under JNTUH)

Gatkesar Muncipality,Medchal Dist.Hyderabad-501301

2025-26

A Mini project

on

Resource Allocation System

Submitted in partial fulfillment of the

SOFTWARE ENGINEERING LAB

SMSK LAB AS PROJECT(SE-Lab)

By

MANNE NISHANTH (23U11A6666)

Under the esteemed guidance of

Mrs. G. SWATHI REDDY

Assistant professor

SAMSKRUTI COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF Computer Science Engineering (AIML)

CERTIFICATE

This is to certify that the mini project titled "**Resource Allocation System**" is a bonafide work done by ***MANNE NISHANTH*** ***(23U11A6666)*** under Software Engineering Lab-SMSK Lab As Project (**SE-LAB**) practice of our institute and that this work has not been submitted for the award of any other Degree/Diploma of any Institution/University.

Project Guide

Mrs. G. Swathi Reddy

Table of Contents

Resource Allocation System

Resource Allocation System

1. Introduction

1.1 Need for the Project

In multitasking environments, efficient resource allocation is crucial for ensuring system reliability, avoiding resource contention, and enhancing performance. This project aims to develop a system that simulates resource allocation to multiple processes, ensuring fairness and preventing issues like deadlocks.

1.2 Project Description

The project implements a Resource Allocation System that dynamically assigns resources to processes while maintaining a system state. It supports features like process resource requests, releases, and detecting potential deadlocks using algorithms such as the Banker's Algorithm.

1.3 Components of the Project

1. Resource Management Module: Simulates allocation, deallocation, and availability of resources.
2. Request Handling Module: Processes dynamic resource requests.
3. Deadlock Prevention Module: Ensures safe states using the Banker's Algorithm.
4. User Interface: Displays the system state and processes.

---

2. Requirement Analysis

- Hardware:
  - Processor: 2 GHz or higher
  - RAM: 4 GB or more
  - Storage: 5 GB free space
- Software:
  - Python 3.x
  - Libraries: NumPy, Matplotlib (optional for visualization)

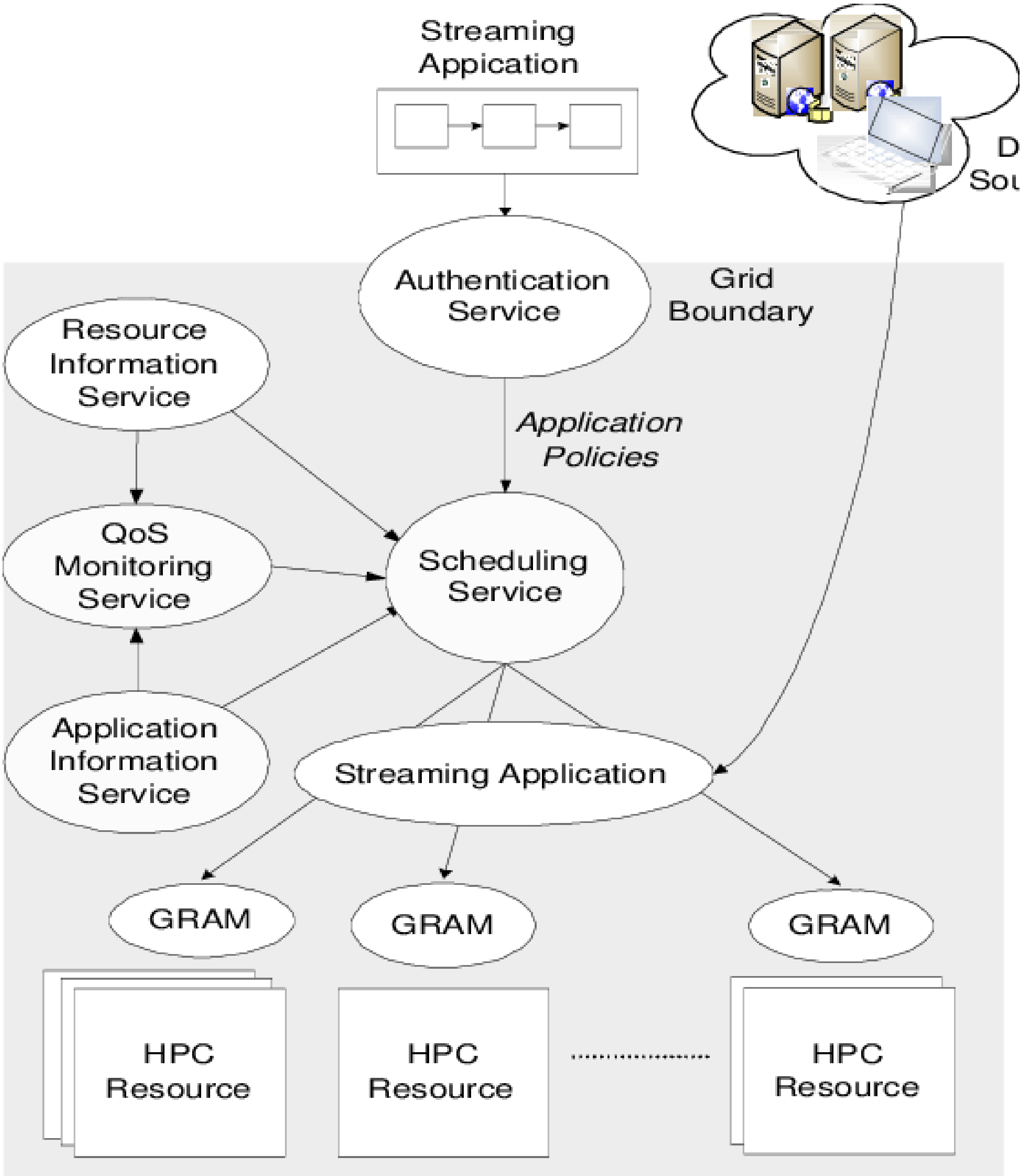
---

3. System Design

The Resource Allocation System (RAS) is designed to efficiently allocate resources (like personnel, equipment, or finances) to various projects or tasks within an organization. The goal is to ensure optimal utilization of resources while minimizing waste and downtime.

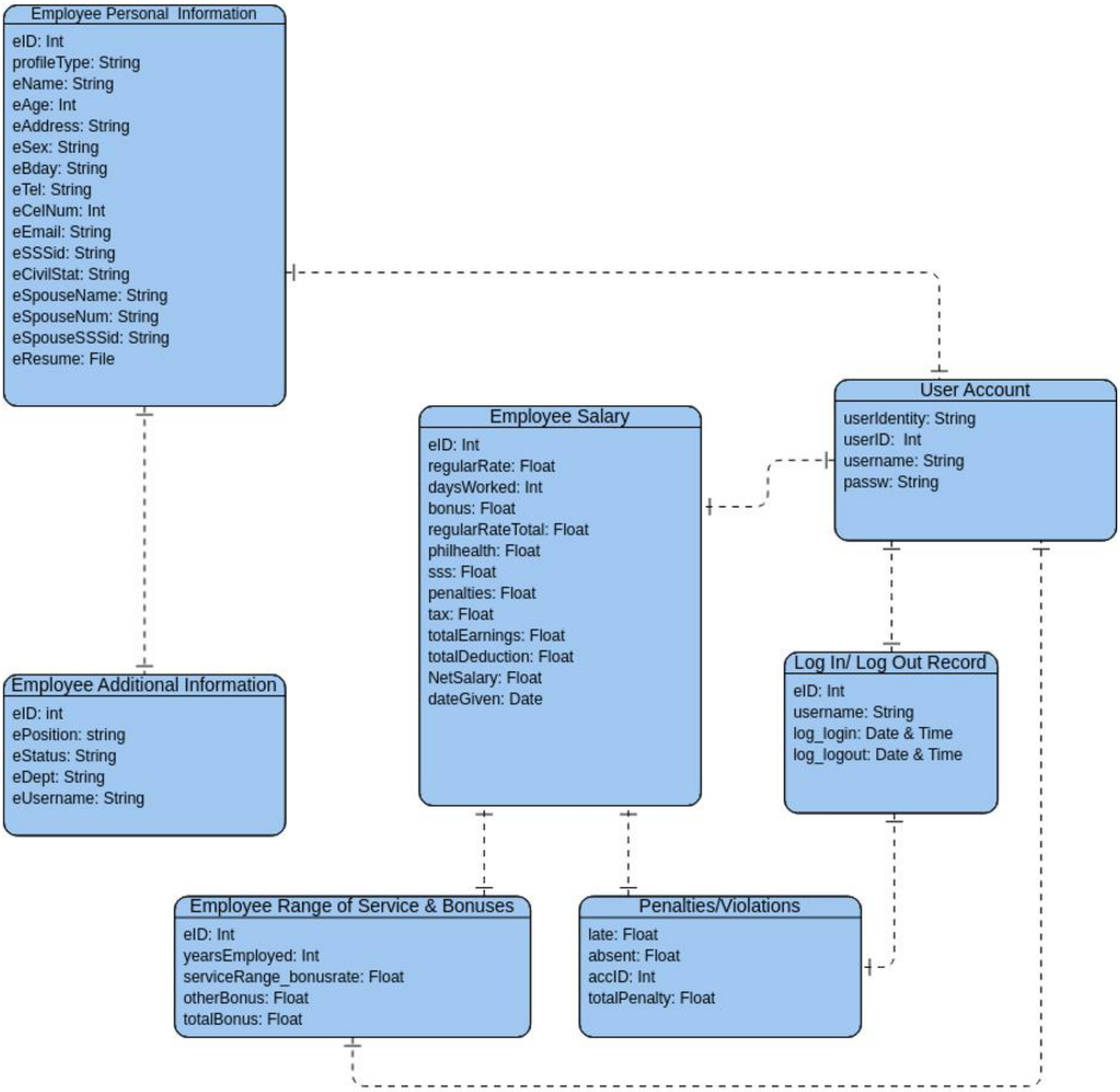
3.1 Architecture Diagram

The Architecture Diagram provides a high-level view of the system, showcasing the main components and how they interact with each other. Below is a textual representation of what an architecture diagram for the Resource Allocation System might include:



3.2 Flow Diagram

The Flow Diagram illustrates the sequence of actions and the workflow within the Resource Allocation System. Here's a simple flow for allocating resources:



Flow Explanation:

1. Start: User actions begin with logging into the system.
2. Request Resource: The user requests specific resources.
3. Check Resource Availability: The system checks if the requested resources are available.

4. Allocate Resource: If available, resources are allocated to the user.
5. Update Resource Database: The database is updated to reflect the new allocation.
6. Notify User: The user is informed of the successful allocation.
7. Handle Errors: If resources are unavailable, the system logs the error and notifies the manager for further actions.

---

4. Implementation (Python Code Example)

Below is a Python implementation of the Banker's Algorithm for resource allocation:

```
import numpy as np

class ResourceAllocationSystem:
    def __init__(self, total_resources, allocation, max_demand):

        self.total_resources = np.array(total_resources)
        self.allocation = np.array(allocation)
        self.max_demand = np.array(max_demand)
        self.num_processes = len(allocation)
        self.num_resources = len(total_resources)
        self.calculate_available()

    def calculate_available(self):
        """Calculate the available resources."""
        self.available = self.total_resources - self.allocation.sum(axis=0)

    def is_safe_state(self):
        """Check if the system is in a safe state."""
        work = self.available.copy()
        finish = [False] * self.num_processes
        safe_sequence = []

        while len(safe_sequence) < self.num_processes:
            allocated = False
            for i in range(self.num_processes):
                if not finish[i] and all(self.max_demand[i] - self.allocation[i] <= work):
                    work += self.allocation[i]
                    finish[i] = True
                    safe_sequence.append(i)
                    allocated = True
                    break
            if not allocated:
                break
```

```
        if len(safe_sequence) == self.num_processes:
            return True, safe_sequence
        else:
            return False, []

    def request_resources(self, process_id, request):
        """Handle a resource request."""
        request = np.array(request)
        if all(request <= self.max_demand[process_id] - self.allocation[process_id]) and
all(request <= self.available):
            # Tentatively allocate resources
            self.available -= request
            self.allocation[process_id] += request

            # Check if the system remains in a safe state
            safe, sequence = self.is_safe_state()
            if safe:
                return True, sequence
            else:
                # Rollback allocation
                self.available += request
                self.allocation[process_id] -= request
                return False, []
        else:
            return False, []

# Example input
total_resources = [10, 5, 7]
allocation = [
    [0, 1, 0],
    [2, 0, 0],
    [3, 0, 2],
    [2, 1, 1],
    [0, 0, 2]
]
max_demand = [
    [7, 5, 3],
    [3, 2, 2],
    [9, 0, 2],
    [2, 2, 2],
    [4, 3, 3]
]
```

```
# Initialize the system
ras = ResourceAllocationSystem(total_resources, allocation, max_demand)

# Example resource request
process_id = 1
request = [1, 0, 2]

success, sequence = ras.request_resources(process_id, request)
if success:
    print(f"Request granted. Safe sequence: {sequence}")
else:
    print("Request denied. System would enter an unsafe state.")
```

---

5. Results and Discussion

- Results: The system successfully allocates resources while maintaining a safe state. Unsafe requests are denied to avoid potential deadlocks.
- Discussion: The Banker's Algorithm ensures safety but requires accurate maximum demands from processes. Future improvements may include dynamic adjustments to resource allocation.

---

6. Conclusion

This project provides a comprehensive simulation of resource allocation in multitasking systems. It ensures fair and safe resource management using the Banker's Algorithm. Further enhancements could include graphical user interfaces and dynamic deadlock resolution strategies.

---

7. References

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2020). *Operating System Concepts*. Wiley.
2. Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems*. Pearson.
3. Python NumPy Documentation: https://numpy.org/doc/stable/enter an unsafe state.")

***