Adaptive Weighted Multi-Tenant Scheduling: System Design and Simulation Report

Mou Yangyi-1306078

Taoying Jia-1306194

Wenzhou Kean University

Dr. Hamza Djigal

CPS 3250 W05&06

November 30 2025

1.1Introduction

In modern cloud computing platforms, multiple tenants usually share physical resources. But if we continue to apply the traditional scheduling policies, we will find that certain tenants will consume more resources while other tenants are starved, resulting in poor fairness and utilization.

A case of a noisy neighbor happened in Netflix where containers were consuming too much CPU/network resources and this adversely affected the performance of other co-located tenants proving the importance of adaptive scheduling in multi-tenant settings. Therefore, a method that ensure fair resource allocation and task scheduling is vary necessary, which have become critical challenges in multi-tenant cloud services [2].

To meet the diverse service-quality requirements of different tenants, adaptive scheduling mechanisms that can dynamically adjust resource distribution based on real-time load is necessary.

1.2 System Objectives

In this project, we try to design an adaptive cloud scheduler aimed at addressing inter-tenant resource contention and ensuring fair scheduling across tenants. The main aims:

1.Ensure long-term intended weighted fairness across multiple tenants with changing workloads.

2.Adapt dynamically to workload imbalance by adjusting scheduling weights.

3.Prevent starvation, even when some tenants temporarily generate heavy workloads.

4.Maintain good resource utilization while improving fairness metrics.

2. System Architecture

The system consists of three primary components: the **Task module**, the **Tenant module**, and the **Scheduler module**.

2.1 Task Module

Each task has:

- A unique ID,
- An associated tenant ID,
- A constant CPU demand (equal to 1 in this model).

```
static class Task {
    int id;
    int tenantId;
    int cpuDemand;
    public Task(int id, int tenantId, int cpuDemand) {
        this.id = id;
        this.tenantId = tenantId;
        this.cpuDemand = cpuDemand;
    }
}
```

This abstraction allows the scheduler to focus on fairness rather than variable resource costs.

2.2 Tenant Module

Each tenant holds:

- A unique integer ID
- A scheduling weight
- A FIFO queue of pending activities

```
static class Tenant {
    int id;
    double weight;
    Queue<Task> queue = new LinkedList<>();
    public Tenant(int id, double weight) {
        this.id = id;
        this.weight = weight;
    }
}
```

2.3. Scheduler Module

The scheduler selects one task per time slice using **weighted random scheduling**.

2.3.1 Weighted Random Selection Algorithm

The core idea: tenants with larger weights have proportionally higher chances of being selected.

```
static class Scheduler {
    List<Tenant> tenants;
    Random rng;
    public Scheduler(List<Tenant> tenants, Random rng) {
        this.tenants = tenants;
        this.rng = rng;
    }
    public Task selectTask() {
        List<Tenant> active = new ArrayList<>();
        for (Tenant t : tenants) {
            if (!t.queue.isEmpty()) active.add(t);
        }
        if (active.isEmpty()) return null;
        if (active.size() == 1) {
            return active.get(0).queue.poll();
        }
        double sumW = 0.0;
        for (Tenant t : active) sumW += t.weight;
        double r = rng.nextDouble() * sumW;
        double acc = 0.0;
        for (Tenant t : active) {
            acc += t.weight;
            if (r <= acc) {
                return t.queue.poll();
            }
        }
        return active.get(active.size() - 1).queue.poll();
    }
```

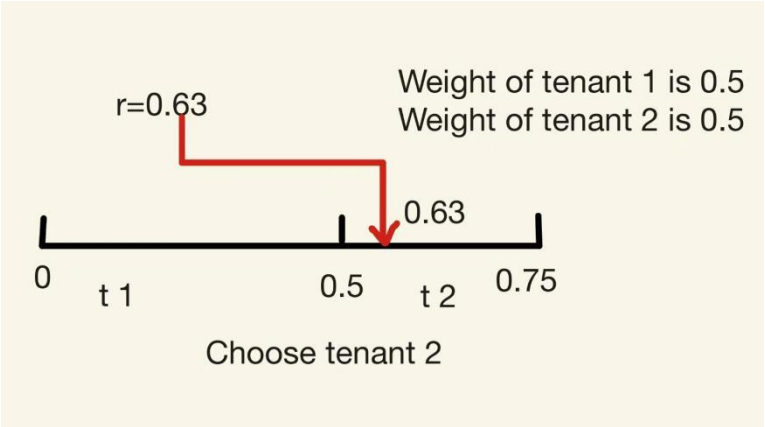
2.3.2 Main Process flow

SelectTask( ) : Firstly read the tenant information from List<Tenant>, extract tasks of different tenants, then add them to a new List<Tenant> called active. If active is empty, return null. If active only have one task, then select it(return active.get(0).queue.poll();). If there are many tasks, calculate the total task weight in active and use weighted random sampling:

Firstly, use rng to create a random number that is within 0-sum of weight called r.

Then traverse active and accumulate acc += weight_i. When r <= acc, select that tenant.

Example:



After choose the tenant, we just poll a task from its queue(`return t.queue.poll()`).

3. Adaptive Weight Adjustment Algorithm (Core Algorithm)

3.1 Method Parameters

1. tenants (List<Tenant>)

Type: List of Tenant objects

Meaning: Represents all tenants in the system.

Role: Their weight fields will be adjusted based on recent scheduling fairness.

2. usageWindow (List<Integer>)

Type: List of integers

Meaning: usageWindow.get(i) is the number of tasks executed for tenant i in the current sliding window(50 slides as a window).

Role: Used to compute each tenant's actual scheduling ratio to decide weight adjustments.

3. alpha (double)

Type: Double

Meaning: Smoothing factor between 0 and 1.

Role: Controls how strongly new weights replace old weights.

Larger alpha → stronger adjustment

Smallr alpha → slower/ smoother response

4. initialWeight

Type: double[]

Meaning: The initial weight of the tenants.

Role: Participate in adjusting Tenant's weight.

3.2 Some important Local Variables

1. total

Type: int

Meaning: Sum of all usageWindow values.

Role: Represents total tasks executed in the current fairness window.

2. target

Type: double

Meaning: The ideal fair share each tenant should have = 1.0 / n.

Role: Used to see whether a tenant got more or fewer tasks than ideal.

3. candidate

Type: double[]

Meaning: Temporary array storing unnormalized new weights.

Role:

Step 1: Calculate candidate[i] based on old weight and fairness ratio

Step 2: Normalize it

Step 3: Use it to smooth-update real weight

4. actual

Type: double

Meaning: Actual scheduling ratio of tenant i:

actual = usageWindow[i] / total

Role:

Compare with target to determine if weight should increase or decrease.

3.3 Code Snippet: Weight Adjustment Logic

```
    public static void adjustWeightsSmoothed(List<Tenant> tenants,
            List<Integer> windowUsage,
            double alpha, double[] initialWeight) {

            double eps = 1e-9;
            int n = tenants.size();

            double[] oldW = new double[n];
            double[] actualUsage = new double[n];
            for (int i = 0; i < n; i++) {

                 oldW[i] = tenants.get(i).weight;
                 actualUsage[i] = windowUsage.get(i);  // Actual number of executions within the window
           }

//  Compute candidate[i]
            double[] candidate = new double[n];
            for (int i = 0; i < n; i++) {
               candidate[i] = oldW[i] / (actualUsage[i]+eps);
             }

// Normalize candidate so that the total sum becomes 1
           double sum = 0.0;
           for (double c : candidate) sum += c;
           for (int i = 0; i < n; i++) candidate[i] /= sum;

//  Smoothed update: newW = (1 - alpha) * initialWeight + alpha * candidate
              for (int i = 0; i < n; i++) {
                   double newW = (1.0 - alpha) * initialWeight[i] + alpha * candidate[i];
                   tenants.get(i).weight = newW;
              }
//Normalize final weights again
          double total = 0.0;
          for (Tenant t : tenants) total += t.weight;
          for (Tenant t : tenants) t.weight /= total;
}
```

Key points:

- If a tenant performs more than its share → reduce weight
- If less → increase weight
- If zero executions → keep weight non-zero
- alpha smooths oscillations (like low-pass filtering)

3.4 Main Process flow

Firstly we calculate the actual weight of finished tasks of these tenants in the last window(actual =usageWindow.get(i) / total).

Then we will compare tenant i's actualUsage with its current weight and give this tenant a adjusted weight which is stored in candidate[i]. Formula is(A very small epsilon (eps = 1e-9) is added to avoid division by zero when actual usage is zero.):

$$candidate[i] = \frac{oldW[i]}{actualUsage[\mathrm{i}] + \varepsilon}$$

So if a tenant's actual weight is larger than target its weight will decrease, if actual weight is smaller than target, its weight will increase.

But if we change the weight directly, which may cause excessive weight fluctuations. So we choose **Smoothing Fusion.** Formula is:

$$newW[i]=(1-\alpha)\cdot initialWeight[i]+\alpha\cdot candidate[i]$$

In our configuration, we choose $\alpha = 0.3$, which makes the weight adjustments more gradual and stable. And in the long run, the adjusted weights will still largely adhere to the initial weight.

4. Simulation Process

The main simulation controls time slices, stochastic task generation, scheduling, and periodic weight updates.

4.1 Code

```
    public static void main(String[] args) {

        // Configurable parameters (can be passed via command line)
        int n = 5;                      // Number of tenants (default: 5)
        int totalTimeSlices = 1000;     // Total number of simulated time slices
        int maxNewTasksPerSlice = 3;    // Max tasks a tenant may generate per time slice (0..max)
        int adjustInterval = 50;        // Interval (in time slices) for weight adjustment
        double smoothAlpha = 0.2;       // Smoothing factor alpha, 0.0-1.0
        long seed = System.currentTimeMillis();  // Random seed (default: current time)

        Random rng = new Random(seed);
        System.out.printf(
            "Start simulating: n=%d, timeSlices=%d, maxNewTasksPerSlice=%d, adjustInterval=%d, alpha=%.3f, seed=%d%n",
            n, totalTimeSlices, maxNewTasksPerSlice, adjustInterval, smoothAlpha, seed
        );

        // Initialize tenants (weights initially set manually here)
        List<Tenant> tenants = new ArrayList<>();
        tenants.add(new Tenant(1, 0.2));
        tenants.add(new Tenant(2, 0.1));
        tenants.add(new Tenant(3, 0.4));
        tenants.add(new Tenant(4, 0.1));
        tenants.add(new Tenant(5, 0.2));
        double[] initialWeight = new double[n];
        for (int i = 0; i < n; i++) {
            initialWeight[i] = tenants.get(i).weight;}
        Scheduler scheduler = new Scheduler(tenants, rng);

        // Global execution count (how many tasks each tenant executed)
        List<Integer> usage = new ArrayList<>(Collections.nCopies(n, 0));
        int executedTasks = 0;
        int taskIdCounter = 0;

        // Record usage in the most recent adjustment window
        List<Integer> windowUsage = new ArrayList<>(Collections.nCopies(n, 0));
        int windowCount = 0;
        // Main simulation loop: for each time slice
        for (int t = 1; t <= totalTimeSlices; t++) {
            // Step 1: each tenant randomly generates new tasks (model D)
```

```
      for (int t = 1; t <= totalTimeSlices; t++) {
          // Step 1: each tenant randomly generates new tasks (model D)
          for (int i = 0; i < n; i++) {
              int newTasks = rng.nextInt(maxNewTasksPerSlice + 1); // 0..maxNewTasksPerSlice
              for (int k = 0; k < newTasks; k++) {
                  tenants.get(i).queue.add(new Task(taskIdCounter++, tenants.get(i).id, 1));
              }
          }
          // Step 2: scheduling (execute at most 1 task per time slice)
          Task task = scheduler.selectTask();
          if (task != null) {
              int idx = task.tenantId - 1;
              usage.set(idx, usage.get(idx) + 1);
              windowUsage.set(idx, windowUsage.get(idx) + 1);
              executedTasks++;
          }
          windowCount++;
          // Step 3: adjust weights at the specified interval
          if (t % adjustInterval == 0) {
              adjustWeightsSmoothed(tenants, windowUsage, smoothAlpha, initialWeight);
              // Print a snapshot of updated weights (optional)
              double[] curW = scheduler.getWeights();
              System.out.printf("t=%d adjusted weight: ", t);
              for (int i = 0; i < curW.length; i++) {
                  System.out.printf("T%d=%s ",
                          tenants.get(i).id,
                          new DecimalFormat("0.000").format(curW[i]));
              }
              System.out.println();
              // Reset the window usage
              for (int i = 0; i < n; i++) windowUsage.set(i, 0);
              windowCount = 0;
          }
      }
      // End of simulation — print final summary
      printSummary(tenants, usage, executedTasks, totalTimeSlices);
      System.out.printf("Jain Fairness: %.4f%n", computeFairness(usage));
  }
}
```

4.2 Main Process flow

The main code here firstly set the parameters (such as tenant count, total time slice, smoothing factor, etc.) to initialize the scheduling simulation environment. After that, the program will let the tenants to create tasks randomly in every time slides and use selectTask() to choose the task that will be executed. Then two tracking metrics are updated: usage (the global execution count for overall analysis) and windowUsage (the recent-window execution count used for the next weight adjustment). After every 50 time slides(window), the program calls adjustWeightsSmoothed to update tenant weights based on the recent-window usage, ensuring the scheduler dynamically adapts to short-term workload changes. Then the program will initialize the windowCount() and continues until all time slices are completed and finally outputs summary statistics and scheduling results.

5. Fairness Evaluation (Jain's Index)

Fairness is measured using Jain's Fairness Index:

$$J = \frac{(\sum_i x_i)^2}{n \cdot \sum_i x_i^2}$$

```
static double computeFairness(List<Integer> shares) {
    int n = shares.size();
    double sum = 0.0, sumSq = 0.0;
    for (int x : shares) { sum += x; sumSq += (double)x * x; }
    if (sum == 0) return 0.0;
    return (sum * sum) / (n * sumSq);
}
```

6.Normalize weight

After updating the weights, the system performs a normalization step to ensure that the total weight across all tenants remains equal to 1. This prevents numerical drift caused by floating-point operations and guarantees that the weighted scheduling mechanism remains consistent and mathematically valid.

```java
static void normalizeWeights(List<Tenant> tenants) {
    double s = 0.0;
    for (Tenant t : tenants) s += t.weight;
    if (s == 0) {
        double equal = 1.0 / tenants.size();
    for (Tenant t : tenants) t.weight = equal;
    return;
    }
    else
    for (Tenant t : tenants) t.weight /= s;
    }
```

7. Printer

Print the current information.

```java
static void printSummary(List<Tenant> tenants, List<Integer> usage, int executed, int totalTimeSlices) {
    DecimalFormat df = new DecimalFormat("0.000");
    System.out.println("=== result ===");
    for (int i = 0; i < tenants.size(); i++) {
        System.out.printf("Tenant %d: executed=%d, finalWeight=%s%n",
                          tenants.get(i).id, usage.get(i), df.format(tenants.get(i).weight));
    }
    double utilization = (double) executed / totalTimeSlices;
    System.out.printf("Time slices: %d, Executed tasks: %d, Idle slices: %d, CPU utilization: %s%n",
            totalTimeSlices, executed, totalTimeSlices - executed, df.format(utilization));
    System.out.printf("Jain Fairness: %.4f%n", computeFairness(usage));
}
```

8. Result and Analysis

We test 5 tenants with the data in the photo:

```java
List<Tenant> tenants = new ArrayList<>();
tenants.add(new Tenant(1, 0.2));
tenants.add(new Tenant(2, 0.1));
tenants.add(new Tenant(3, 0.4));
tenants.add(new Tenant(4, 0.1));
tenants.add(new Tenant(5, 0.2));
```

The result is:

```
Start simulating: n=5, timeSlices=1000, maxNewTasksPerSlice=3, adjustInterval=50, alpha=0.200, seed=1764144446750
t=50 adjusted weight: T1=0.223 T2=0.112 T3=0.344 T4=0.133 T5=0.189
t=100 adjusted weight: T1=0.190 T2=0.199 T3=0.345 T4=0.092 T5=0.174
t=150 adjusted weight: T1=0.235 T2=0.120 T3=0.344 T4=0.116 T5=0.185
t=200 adjusted weight: T1=0.188 T2=0.141 T3=0.348 T4=0.140 T5=0.184
t=250 adjusted weight: T1=0.198 T2=0.145 T3=0.363 T4=0.103 T5=0.191
t=300 adjusted weight: T1=0.201 T2=0.117 T3=0.367 T4=0.116 T5=0.199
t=350 adjusted weight: T1=0.181 T2=0.098 T3=0.338 T4=0.202 T5=0.181
t=400 adjusted weight: T1=0.197 T2=0.109 T3=0.374 T4=0.126 T5=0.194
t=450 adjusted weight: T1=0.201 T2=0.114 T3=0.352 T4=0.120 T5=0.212
t=500 adjusted weight: T1=0.233 T2=0.103 T3=0.378 T4=0.102 T5=0.185
t=550 adjusted weight: T1=0.180 T2=0.194 T3=0.343 T4=0.108 T5=0.175
t=600 adjusted weight: T1=0.197 T2=0.116 T3=0.379 T4=0.112 T5=0.196
t=650 adjusted weight: T1=0.209 T2=0.106 T3=0.362 T4=0.124 T5=0.199
t=700 adjusted weight: T1=0.210 T2=0.121 T3=0.353 T4=0.114 T5=0.202
t=750 adjusted weight: T1=0.196 T2=0.127 T3=0.342 T4=0.139 T5=0.195
t=800 adjusted weight: T1=0.195 T2=0.107 T3=0.362 T4=0.134 T5=0.202
t=850 adjusted weight: T1=0.208 T2=0.122 T3=0.359 T4=0.118 T5=0.193
t=900 adjusted weight: T1=0.200 T2=0.138 T3=0.366 T4=0.108 T5=0.188
t=950 adjusted weight: T1=0.180 T2=0.108 T3=0.343 T4=0.156 T5=0.213
t=1000 adjusted weight: T1=0.208 T2=0.105 T3=0.363 T4=0.138 T5=0.186
=== result ===
Tenant 1: executed=181, finalWeight=0.208
Tenant 2: executed=121, finalWeight=0.105
Tenant 3: executed=362, finalWeight=0.363
Tenant 4: executed=118, finalWeight=0.138
Tenant 5: executed=218, finalWeight=0.186
Time slices: 1000, Executed tasks: 1000, Idle slices: 0, CPU utilization: 1.000
Jain Fairness: 0.8337
```

8.1 Result analysis

The results show that throughout the 1000 time slices, the weights of the five tenants were continuously adjusted every 50 slices according to their actual execution counts. This dynamic adjustment mechanism ensures that resources were distributed to the tenants fairly. Because the algorithm that will decreases a tenant's weight when its actual weight exceeds its current weight, and increases it when the actual weight is below its current weight, which effectively prevents starvation and ensures that no tenant monopolizes resources even under fluctuating workloads.

Besides, after many times of adjustments, the final weights of the five tenants remain close to their initial weights. This demonstrated that the smoothing mechanism successfully limited long-term drift and preserved the intended weighted fairness. Resource utilization remained optimal (100%), and the achieved Jain fairness index (0.8337) showed a reasonable balance between fairness and responsiveness.

So in general, this algorithm achieved our aims:

1. It ensures long-term intended weighted fairness across multiple tenants with changing workloads.

2. It adapts dynamically to workload imbalance by adjusting scheduling weights.

3.It prevents starvation, even when some tenants temporarily generate heavy workloads.

4. It maintains good resource utilization while improving fairness metrics.

8.2 Problem faced when designing

First version:

```
    double[] candidate = new double[n];
    for (int i = 0; i < n; i++) {
        double actual;
        if (total == 0) {
            actual = 0.0;
        } else {
            actual = (double) usageWindow.get(i) / total;
        }
        double w = tenants.get(i).weight;
        if (actual > 0) {
            candidate[i] = w * (target / actual);
        } else {
                        candidate[i] = w * 1.0; )
        }
    }
    double sumC = 0.0;
    for (double c : candidate) sumC += c;
    if (sumC == 0) {

        for (Tenant t : tenants) t.weight = 1.0 / n;
        return;
    }
    for (int i = 0; i < n; i++) {
        candidate[i] = candidate[i] / sumC;
    }
    for (int i = 0; i < n; i++) {
        double oldW = tenants.get(i).weight;
        double newW = (1.0 - alpha) * oldW + alpha * candidate[i];
        tenants.get(i).weight = newW;
    }
    normalizeWeights(tenants);
}
```

In our first version of adjustWeightSmoothed(), we simply let the tenants' weight change towards average weight, which caused the system to drift away from the intended initial weighted fairness configuration. Besides, the first version did not address the starvation problem. It ignored the tenants that got 0 actualUsage and did not increase their weights.

8.3 Aspects need to improve

First, the response speed of the smoothing mechanism depends on α; a single global α may not optimally adapt across different load regimes.

Second, the model assumes uniform task size (cpuDemand = 1), which simplifies the simulation but does not fully capture real-world workload diversity.

References

[1] H. Li and R. Cruz, "Investigation of a Cross-regional Network Performance Issue," Netflix Technology Blog, Apr. 24 2024. [Online].Available: https://netflixtechblog.com/investigation-of-a-cross-regional-network-performance-issue-422d6218fdf1.

[2] M. N. Alatawi, "Optimizing Multitenancy: Adaptive Resource Allocation in Serverless Cloud Environments Using Reinforcement Learning," Electronics, vol. 14, no. 15, p. 3004, Jul. 2025, doi: 10.3390/electronics14153004.