import java.util.HashMap;

import java.util.Map;

// Game Manager

public class GameExecutor {

private Map<EdgeComputeNode, Integer> nodeScores;

public GameExecutor() {

this.nodeScores = new HashMap<>();

}

// Start the game process

public void startGame(List<EdgeComputeNode> participatingNodes) {

initializeScores(participatingNodes);

// Simulate the game process

for (int round = 1; round <= 3; round++) {

System.out.println("Round " + round + " of the game:");

for (EdgeComputeNode node : participatingNodes) {

// Simulate node behavior and task execution

simulateNodeBehavior(node);

simulateTaskExecution(node);

}

// Settle the result of this round of game

settleRoundScores(participatingNodes);

// Prints the score of the current node

printNodeScores();

}

// End the game and offer the final reward

rewardParticipants(participatingNodes);

}

// Initializes the node score

private void initializeScores(List<EdgeComputeNode> nodes) {

for (EdgeComputeNode node : nodes) {

nodeScores.put(node, 0);

}

}

// Simulate node behavior, such as actively participating in a task, rejecting a task, etc

private void simulateNodeBehavior(EdgeComputeNode node) {

// Logic that simulates node behavior and can be expanded according to actual needs

}

// Simulate task execution and update node score

private void simulateTaskExecution(EdgeComputeNode node) {

// Simulate the logic of task execution, such as updating node scores

int taskScore = simulateTaskScore();

nodeScores.put(node, nodeScores.get(node) + taskScore);

}

// Simulation task score

private int simulateTaskScore() {

// The logic of simulated task scoring can be expanded according to actual needs

return (int) (Math.random() \* 10);

}

// Settle the result of this round of game

private void settleRoundScores(List<EdgeComputeNode> nodes) {

// Settlement logic can be implemented according to actual needs, such as allocating rewards according to scores

}

// Prints the score of the current node

private void printNodeScores() {

System.out.println("Current Node Scores:");

for (Map.Entry<EdgeComputeNode, Integer> entry : nodeScores.entrySet()) {

System.out.println(entry.getKey().getName() + ": " + entry.getValue() + " points");

}

System.out.println();

}

// End the game and offer the final reward

private void rewardParticipants(List<EdgeComputeNode> nodes) {

System.out.println("Game Over! Final Node Scores:");

// Print the final score

printNodeScores();

// The final reward logic can be implemented according to the actual needs, such as allocating rewards according to the total score

}

// Simulate node behavior, such as actively participating in a task, rejecting a task, etc

private void simulateNodeBehavior(EdgeComputeNode node) {

// Simulation node actively participates in the task logic

if (shouldParticipateInTask()) {

System.out.println("Node '" + node.getName() + "' actively participates in the task.");

node.executeTaskLocally(createTask());

} else {

System.out.println("Node '" + node.getName() + "' decides not to participate in the task.");

}

// Simulate the logic of the node rejecting the task

if (shouldRejectTask()) {

System.out.println("Node '" + node.getName() + "' rejects the task.");

// You can perform the rejection task logic based on actual requirements, such as updating node scores

}

}

// Whether the simulation node actively participates in the decision of the task

private boolean shouldParticipateInTask() {

// The logic of simulated decisions can be expanded according to actual needs

return Math.random() < 0.8;

// 80% probability of active participation in the task

}

private boolean shouldRejectTask() {

return Math.random() < 0.2;

}

private Task createTask() {

String taskName = "Task\_" + System.currentTimeMillis();

return new Task(taskName);

}

private boolean shouldRejectTask(EdgeComputeNode node) {

if (isNodeOverloaded(node)) {

System.out.println("Node '" + node.getName() + "' is overloaded. Rejecting task.");

return true;

}

if (isTaskSuccessRateLow()) {

System.out.println("Task success rate is low. Rejecting task.");

return true;

}

if (shouldRejectBasedOnRewards(node)) {

System.out.println("Node '" + node.getName() + "' decides to reject task based on rewards.");

return true;

}

return false;

}

private boolean isNodeOverloaded(EdgeComputeNode node) {

return node.getCurrentTaskCount() > 5;

}

private boolean shouldRejectTask(EdgeComputeNode node) {

if (isNodeOverloaded(node)) {

System.out.println("Node '" + node.getName() + "' is overloaded. Rejecting task.");

return true;

}

if (isTaskSuccessRateLow()) {

System.out.println("Task success rate is low. Rejecting task.");

return true;

}

if (shouldRejectBasedOnRewards(node)) {

System.out.println("Node '" + node.getName() + "' decides to reject task based on rewards.");

return true;

}

return false;

}

private boolean isNodeOverloaded(EdgeComputeNode node) {

return node.getCurrentTaskCount() > 5;

}

private boolean isTaskSuccessRateLow() {

return Math.random() < 0.3;

}

private boolean shouldRejectBasedOnRewards(EdgeComputeNode node) {

int nodeScore = node.getScore();

return nodeScore < 50;

}

}