CSDS 293

Software Craftsmanship

2024 Fall Semester

Programming Assignment 2

Due at the beginning of your discussion session on

Sep 9 – 13, 2024

Reading

In addition to the following topics, the quiz syllabus includes any material covered in the lectures:

- Chapter 14 in Code Complete
- Items 48 and 50 in Effective Java
- Java's BigInteger documentation (introduction only)
- Section 19.1 in Code Complete (excluding "Forming Boolean Expression Positively", "Guidelines for Comparison to 0", "In C++, consider...")
- Section 15.1 ("Plain if-then Statements" only) in Code Complete
- Sections 17.1 and 19.4 in Code Complete
- Section 17.3 ("Rewrite with a status variable" only) in Code Complete

Programming

In this assignment, you will begin developing a Distributed Task Scheduling System that manages the execution of computational tasks across multiple servers. The goal is to ensure that the system is scalable, efficient, and robust. This first assignment focuses on setting up the basic framework, including task definition, server management, and basic scheduling.

Package

You should organize your project in a package. The package name is up to you: it could range from the simple ('taskscheduler') to the detailed ('edu.cwru.<cwruid>.taskscheduler').

System Architecture and Classes

The Distributed Task Scheduling System consists of several key components working together to manage and execute tasks across multiple servers. The main classes in this system are `Task` (interface), `SimpleTask`, `Server`, `TaskScheduler`, and `Duration`.

The `Task` interface defines the contract for all tasks in the system, with `SimpleTask` providing a concrete implementation. `Tasks` are the fundamental units of work in the system. The `Server` class represents a computational node capable of executing tasks. It maintains a queue of tasks and interacts directly with `Task` objects, executing them and managing their state. At the heart of the system is the `TaskScheduler` class. This central component manages the distribution of tasks across multiple servers. It maintains a list of available `Server` instances and interacts with both `Task` and `Server` objects to efficiently distribute the workload. The `Duration` class serves as a utility, used by `Task` to represent and manipulate time durations. It's crucial for estimating task execution times and managing timeouts.

In terms of interaction, the `TaskScheduler` receives `Task` objects to be scheduled. It then decides which `Server` should handle each task based on its scheduling algorithm. The `TaskScheduler` adds tasks to the chosen `Server` instances. When execution is triggered, each `Server` processes its queue of `Task` objects. Finally, `Server` instances report back to the `TaskScheduler` with the results of task execution.

Task Interface and SimpleTask Class

Define a `Task` interface that models a computational task. This interface should include methods for getting the task's ID, executing the task, checking its completion status, and retrieving its estimated duration:

- `String getId()`: Returns the unique identifier of the task.
- `void execute() throws TaskException`: Executes the task.
- `boolean isCompleted()`: Returns the completion status of the task.
- `Duration getEstimatedDuration()`: Returns the estimated duration of the task.

Implement a concrete `SimpleTask` class that implements the `Task` interface. Ensure that this class is immutable, following the principles discussed in Effective Java. This class represents a basic unit of work in the system, providing implementations for all the methods defined in the `Task` interface.

Server Class

Implement a `Server` class that simulates a computational node. This class should have methods for adding tasks to the server's queue, executing all tasks in the queue, and retrieving lists of completed and failed tasks:

- `void addTask(Task task)`: Adds a task to the server's queue.
- `List<Task> executeTasks() throws ServerException`: Executes all tasks in the queue and returns completed tasks.
- `List<Task> getFailedTasks()`: Returns a list of failed tasks.

Use streams to process tasks in the `executeTasks` method, as discussed in Item 48 of Effective Java.

TaskScheduler Class

Develop a `TaskScheduler` class that manages task distribution across multiple servers. This class should include methods for adding servers to the scheduler, scheduling individual tasks, and executing all tasks across all servers:

- `void addServer(Server server)`: Adds a server to the scheduler.
- `void scheduleTask(Task task)`: Schedules a task to an appropriate server.
- `Map<Server, List<Task>> executeAll() throws SchedulerException`: Executes tasks on all servers.

Implement defensive copying in the `addServer` and `scheduleTask` methods, as recommended in Item 50 of Effective Java.

Duration Class

Create a `Duration` class to represent task durations. This class should use **`BigInteger`** to store duration in milliseconds, providing high precision for time measurements. Include methods for adding durations, subtracting durations, and comparing durations:

- Use `BigInteger` to store duration in milliseconds.
- Provide methods for addition, subtraction, and comparison.

- Implement a static factory method `ofMillis(long millis)`.

Ensure this class is immutable and follows the principles of good class design from Chapter 14 of Code Complete. The `Duration` class will be used throughout the system for precise time-related calculations and for specifying estimated task execution times.

Error Handling

Implement custom exceptions for your system: `TaskException`, `ServerException`, and `SchedulerException`. Use these for appropriate error scenarios in your classes, following the guidelines in Section 19.4 of Code Complete. Proper error handling and reporting will be crucial for maintaining the robustness and reliability of your distributed system.

General Considerations

These classes may contain as many auxiliary private methods as you see fit, and additional helper classes may be defined.

You should write JUnit tests to make sure that your primary methods work as intended. However, we will revisit testing later on in the course, so extensive testing is not yet recommended. Similarly, your code should have a reasonable number of comments, but documentation is going to be the topic of a future assignment. As a general guideline at this stage of the course, comments and tests should be similar to those accepted in CSDS 132.

Discussion Guidelines

The class discussion will focus on:

- High-level code organization
- The design and documentation of the implementation
- Straight-line code, conditional code

Your grade will be affected by the topics covered in this and in previous weeks. Your discussion leader may introduce material that has not been covered yet in the lecture. Although future topics will not contribute to your current grade, you are expected to follow your leader's advice in revising the assignment.

Submission

Bring a copy to the discussion to display on a projector. Additionally, submit an electronic copy of your program to Canvas. In addition to your code, include a README file explaining how to compile and run the code. The code should be handed in a zip, tar.bz2, or tar.gz archive. Archives in 7z cannot be accepted.

Note on Academic Integrity

It is crucial to maintain academic integrity throughout this course. Copying another group's work or providing your code for others to copy, including posting it on public platforms or social media, is a violation of academic integrity. Ensure that the work you submit is your own, reflecting your understanding and application of the course materials.