CSDS 293
Software Craftsmanship
2024 Fall Semester

Programming Assignment 4

Due at the beginning of your discussion session on

Sep 23 – 27, 2024

Reading

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

- Chapter 14 (creating routines that handle dependencies and complex interactions) in Code Complete.
- Items 50 and 62 in Effective Java.
- Section 19.4 (exception handling) in Code Complete.
- Section 19.6 (debugging techniques) in Code Complete.
- Java's BigInteger documentation (methods for comparison and arithmetic operations).
- Section 17.1 (high-quality routines) in Code Complete.
- Section 15.1 ("Plain if-then Statements" only) in Code Complete.

Programming

In this assignment, you will further enhance the Distributed Task Scheduling System by implementing task dependencies, advanced error handling, and logging. You'll also introduce performance monitoring capabilities and implement a task timeout mechanism. These additions will significantly increase the sophistication and reliability of your system.

System Architecture and Class Interactions

Building on the architecture established in the previous assignments, we'll now introduce the concept of task dependencies. This will require modifications to the Task interface and the implementation of a new `DependentTask` class. The `TaskScheduler` will need to be updated to handle these dependencies when scheduling tasks.

We'll also introduce a new `RetryPolicy` class to manage task retries, a logging system for comprehensive system monitoring, and a `PerformanceMonitor` class to track and analyze system performance metrics.

The existing `Server` class will be modified to support task timeouts, adding another layer of control over task execution.

Task Dependencies

Extend the `Task` interface to support dependencies between tasks:

- Add a `Set<String> getDependencies()` method to return task IDs this task depends on.
- Implement a `DependentTask` class that extends `PriorityTask` and includes dependency information.

Ensure that `DependentTask` maintains immutability and follows the design principles from Chapter 14 of Code Complete. The `TaskScheduler` will need to be updated to check these dependencies before scheduling a task, ensuring that all dependent tasks have been completed first.

Advanced Error Handling

Implement a retry mechanism for failed tasks:

- Add a `RetryPolicy` class with configurable retry attempts and delay.
- Modify `TaskScheduler` to retry failed tasks based on the retry policy.

Implement proper exception handling for retry attempts, following the guidelines in Section 19.4 of Code Complete. This should include appropriate logging of retry attempts and final failure states.

The `RetryPolicy` should define how the system attempts to re-execute failed tasks. Here are some examples to illustrate its expected behavior:

- A task is scheduled that fails on its first execution due to a temporary network issue. The `RetryPolicy` is configured to attempt up to 3 retries with a 5-second

delay between attempts. The expected outputs are the following: Task execution failed (Attempt 1); Waiting 5 seconds before retry; Task execution successful (Attempt 2).
- A task consistently fails due to a persistent error. The `RetryPolicy` is set to a maximum of 3 retries with an exponential backoff starting at 2 seconds (doubling each time). The expected outputs are the following: Task execution failed (Attempt 1); Waiting 2 seconds before retry; Task execution failed (Attempt 2); Waiting 4 seconds before retry; Task execution failed (Attempt 3); Waiting 8 seconds before retry; Task execution failed (Attempt 4); Maximum retries reached, task marked as permanently failed.
- Two tasks are scheduled: Task A and Task B. Task B depends on the successful completion of Task A. Task A fails on its first attempt but succeeds on the second try. The `RetryPolicy` is set to 2 retries with a 3-second delay. The expected outputs are the following: Task A fails on its first execution; The system waits for 3 second; Task A is retried and succeeds; Task B is then executed and succeeds.

You should consider how to design your `RetryPolicy` to be flexible and configurable. Think about how to integrate it with the existing `TaskScheduler` and how it will interact with other features like task dependencies and performance monitoring. Remember to log appropriate information at each step to aid in debugging and system analysis.

Logging

Integrate a logging framework (e.g., `java.util.logging`):
- Add comprehensive logging to `TaskScheduler` and `Server` classes.
- Log all major events: task scheduling, execution, completion, failures, and retry attempts.
- Implement different log levels for various types of events, allowing for easy filtering of logs based on severity or type of operation.

Follow the principles of effective logging as discussed in Code Complete, ensuring that your logs provide valuable information for debugging and system monitoring without becoming overly verbose.

Task Timeout

Implement a timeout mechanism for tasks:

- Add a `timeout` field to the `Task` interface.
- Modify `Server` to interrupt tasks that exceed their timeout.
- Use `java.util.concurrent.Future` to manage task execution with timeouts. This will allow you to cancel tasks that are taking too long to complete, preventing them from consuming resources indefinitely.

The Task Timeout feature should ensure that tasks do not run indefinitely, potentially consuming system resources or blocking other tasks. Here are some examples:

- A task is scheduled with a timeout of 5 seconds. The task's actual execution takes 7 seconds. The expected behaviors are: The task starts executing; After 5 seconds, the system interrupts the task; The task is marked as failed due to timeout.
- Three tasks (A, B, and C) are scheduled in a dependency chain (C depends on B, which depends on A). Each task has a timeout of 5 seconds. Task B exceeds its timeout. The expected behaviors are: Task A starts and completes successfully within 5 seconds; Task B starts but doesn't complete within 5 seconds; The system interrupts Task B and marks it as failed; Task C is not executed due to the failure of its dependency (Task B).

The system should ensure that the interrupted task can perform necessary cleanup (e.g., rolling back the database transaction). Make sure your timeout mechanism is robust and integrated with other system features like the retry policy and task dependencies. Think about how to handle resource cleanup for interrupted long-running tasks. Remember to log appropriate information at each step to aid in debugging and system analysis.

Performance Monitoring

Create a `PerformanceMonitor` class to track and analyze system performance. This class should track metrics such as average task execution time, success rate, and server utilization. Use streams to calculate and update these performance metrics efficiently, applying the principles from Item 48 in Effective Java.

The `PerformanceMonitor` should provide methods to retrieve current performance statistics and potentially trigger alerts if certain thresholds are exceeded. This will give you insights into the system's behavior and help identify potential bottlenecks or issues.

General Considerations

As you implement these advanced features, continue to adhere to the principles of good software design discussed in our readings. Your classes may contain additional auxiliary methods as needed, but remember to keep each class focused on its primary responsibility.

Consider the potential performance implications of the new features you're adding, particularly the logging and performance monitoring capabilities. Strive to implement these in a way that provides valuable insights without significantly impacting the system's overall performance.

As the system grows in complexity, it becomes increasingly important to maintain a clean and modular architecture. Ensure that your new components integrate smoothly with the existing system without introducing unnecessary coupling..

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:

- Implementation of task dependencies and their impact on scheduling
- Advanced error handling and retry mechanisms
- Effective use of logging for system monitoring and debugging

  Application of streams and lambdas in performance monitoring

Be prepared to explain your design decisions, particularly regarding the trade-offs involved in your retry policy and performance monitoring implementations. We'll also consider how these new features integrate with the existing components of the system and how they contribute to the overall robustness and reliability of the Distributed Task Scheduling System.

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.