CSDS 293
Software Craftsmanship
2024 Fall Semester

Programming Assignment 5

Due at the beginning of your discussion session on

Sep 30 – Oct 4, 2024

Reading

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

- Chapter 14 (creating high-quality routines that work in distributed environments) in Code Complete.
- Items 48 and 50 in Effective Java.
- Section 19.1 (excluding "Forming Boolean Expression Positively", "Guidelines for Comparison to 0", "In C++, consider ...") in Code Complete.
- Section 19.4 and 19.6 in Code Complete.
- Java's BigInteger documentation (secure random number generation).
- Section 17.1 (high-quality routines) in Code Complete.

Programming

In this assignment, you will complete the Distributed Task Scheduling System by implementing distributed execution capabilities, finalizing error handling and monitoring features, and conducting comprehensive system testing. This assignment will tie together all the components you've built so far, creating a fully functional and robust distributed system.

System Architecture and Class Interactions

Building on the architecture established in the previous assignments, we'll now extend the system to support true distributed execution across multiple machines. This will involve creating a new `RemoteServer` class that extends the existing `Server` class, implementing network communication

capabilities, and modifying the `TaskScheduler` to manage these remote resources.

We'll also introduce an `AlertSystem` class to work alongside the `PerformanceMonitor`, enabling the system to proactively notify administrators of potential issues. The error-handling mechanisms will be refined to deal with network-related failures and other complexities introduced by distributed execution.

Finally, we'll implement comprehensive testing strategies to ensure the reliability and performance of the entire system under various conditions.

Distributed Execution

Modify the `TaskScheduler` and `Server` classes to support distributed execution across multiple physical or virtual machines:

- Implement a new `RemoteServer` class that extends `Server` and simulates network communication. You may use Java RMI (Remote Method Invocation) or implement a simple socket-based communication protocol for remote task execution.
- Update the `TaskScheduler` to manage both local and remote servers, handling the additional complexity of network communication and potential failures.
- Implement proper serialization and deserialization of `Task` objects for network transfer, following the guidelines in Effective Java for creating serializable classes.

Ensure that your distributed execution implementation can handle network latency and failures gracefully. This may involve implementing timeouts for remote operations and fallback strategies for when remote servers become unreachable.

Finalizing Error Handling

Refine the error handling mechanisms to create a comprehensive error recovery strategy for various failure scenarios:

- Implement the Circuit Breaker pattern to handle temporary failures in remote server communication, preventing cascading failures across the system.

- Implement a system-wide error reporting mechanism that aggregates errors from all components (local and

remote) and provides a coherent view of the system's health.

Ensure all network-related errors are properly caught and handled. This should include appropriate logging and potential retry mechanisms for transient network issues.

Monitoring and Alerts

Enhance the `PerformanceMonitor` class to support real-time monitoring of system health and performance across all servers (local and remote). Implement an `AlertSystem` class that works in conjunction with the `PerformanceMonitor` to trigger alerts based on predefined thresholds for various metrics (e.g., high failure rates, excessive task queues, or unresponsive servers).

Use streams and lambdas for efficient data processing in monitoring and alerting, as discussed in Item 48 of Effective Java. The `AlertSystem` should be capable of sending notifications through multiple channels (e.g., log files, email, or a simple console output for this assignment).

System Testing

Conduct extensive testing of the entire Distributed Task Scheduling System.

- Implement a comprehensive suite of unit tests for all major components using JUnit. These tests should cover both normal operation scenarios and edge cases, including error conditions and recovery processes.
- Create integration tests to verify the interaction between different parts of the system, particularly focusing on the newly implemented distributed execution capabilities. These tests should verify that tasks are correctly distributed, executed, and results are properly collected across multiple servers.
- Develop performance tests to evaluate system behavior under various load conditions. This should include scenarios with high concurrency, large numbers of tasks, and simulated network issues for remote servers.
- Use mocking frameworks (e.g., Mockito) to simulate various scenarios, especially for distributed execution and network communication. This will allow you to test error handling and recovery mechanisms without needing to set up actual distributed environments.

Optimization

Based on the results of your performance tests, optimize the system for improved efficiency and scalability. Use profiling tools to identify bottlenecks in your implementation. This may involve optimizing your task distribution algorithm, implementing caching mechanisms where appropriate, or fine-tuning your use of concurrent data structures.

Pay particular attention to the performance of your distributed execution implementation. Look for opportunities to minimize network communication overhead and optimize the serialization/deserialization process for tasks and results.

Optimize your use of streams and parallel processing for large-scale task management, applying the guidelines from Effective Java. However, be mindful of the potential pitfalls of excessive parallelism, especially in a distributed environment.

General Considerations

Ensure that your implementation is well-documented and thoroughly tested. Pay special attention to code quality, adhering to the software craftsmanship principles we've studied throughout the course. Your system should be robust, capable of handling various failure scenarios gracefully, and scalable to manage large numbers of tasks across multiple servers.

Consider the overall architecture of your system and be prepared to discuss how it could be further extended or integrated with other systems in a real-world scenario.

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:

- Challenges and solutions in implementing distributed execution
- Comprehensive error handling and system reliability
- Performance optimization techniques

- Overall system architecture and design decisions

Finally, we'll consider the system as a whole, discussing its strengths, potential areas for improvement, and how it might be adapted or extended for different use cases in real-world applications.

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.