**Implementing Multithreading in the Parking System Server** for

Master of Science

Information Technology

Bria Wright

University of Denver University College

June 1, 2025

Faculty: Nathan Braun MBA, BBA

Dean: Michael J. McGuire, MLS

Table of Contents

## Introduction

As modern applications increasingly emphasize responsiveness and scalability, the capability to manage multiple client requests concurrently is essential, particularly in real-time systems, such as those used for parking management. This paper confidently explores the implementation of a multithreaded Java server crafted for a University Parking System. The system empowers clients to register users, manage vehicles, and engage with a central parking office through efficient JSON and Java object-based communication protocols. To facilitate multiple simultaneous connections, the server effectively employs multithreading, processing each client in its own thread, thus enabling parallel execution without interrupting the user experience for others. This paper presents a comprehensive overview of the server's design, addresses the challenges encountered during development, offers a comparative analysis of threading models, and reflects on the profound lessons learned in the process of building a robust, concurrent socket-based Java application.

## Design Overview

The project's objective was to create a multithreaded Java server for the University Parking System using a JSON-based communication protocol. The server accepts client connections via sockets and responds to serialized command objects with structured ParkingResponse objects. Each client is handled in its own thread, ensuring concurrent processing and eliminating bottlenecks. The server uses a ServerSocket to listen on port 8080. When a client connects, a new thread is created with a lambda expression and passed to a ClientHandler that implements Runnable. This setup keeps the main thread unblocked. The ClientHandler processes serialized Java objects (commands implementing the Command

interface), executes them through a ParkingService, and returns a response via ObjectOutputStream. To enhance flexibility and scalability, I implemented a separate RequestHandler class for JSON-based clients that interprets ParkingRequest objects and maps them to appropriate ParkingResponse messages (Martin 2017). This design adheres to the Open/Closed Principle, enabling easy addition of new commands with minimal changes.

## Challenges and Solutions

### *Challenge 1: Thread Safety and Shared Resources*

The primary challenge in implementing multithreading within the application centered around the safe management of shared resources. Each ClientHandler operates in its thread, allowing simultaneous interactions with shared objects, such as the ParkingService, which is responsible for registering customers and processing vehicles (W3Schools 2024). Initial testing revealed the presence of race conditions, particularly when multiple threads attempted to access or modify shared resources concurrently. For instance, when two clients attempted to register customers simultaneously, it resulted in inconsistent states and occasional duplication of customer IDs. To address these issues, I implemented thread-safety measures by synchronizing critical operations within the shared service methods. While synchronization effectively ensured data integrity and consistency, it led to slight performance degradation due to potential thread contention (“Convert Java Object to Json String Using GSON” 2019). This trade-off highlighted the delicate balance required in multithreaded programming between performance and data safety.

*Challenge 2: Debugging Concurrency Issues*

Debugging concurrency-related bugs proved significantly more challenging than in single-threaded applications due to the complex interactions between threads. The non-deterministic interleaving of thread execution often rendered print statements misleading, potentially obscuring the underlying issues (Baeldung 2018). To improve diagnostics, I incorporated thread names and timestamps into the logging mechanism to establish a clearer execution order. Additionally, I utilized advanced debugging tools, including IntelliJ's integrated debugger and the Java VisualVM profiler, which were invaluable in identifying deadlocks, race conditions, and performance bottlenecks throughout the application.

*Challenge 3: JSON vs. Object Streams*

The requirement to support both JSON-based clients and Java object-based clients introduced a layer of complexity concerning serialization. The use of object streams (via ObjectInputStream and ObjectOutputStream) necessitated consistent class versions between the client and server. Any version mismatch could cause ClassNotFoundException,leading to runtime errors that could interrupt service. Conversely, while the JSON-based approach utilizing Gson offered greater flexibility and robustness in terms of versioning, it required additional effort for parsing and validation to ensure data integrity and consistency (GeeksforGeeks 2016).

## Comparison of Threading Models

*Manual Thread Creation*

In this project, I opted for manually creating threads using the syntax new Thread(() -> handleClient(socket)).start(). This method granted me complete control over the thread's

execution behavior and was relatively straightforward to implement for a small-scale server setup.

Advantages (GeeksforGeeks 2016):

- The model is easy to understand and debug, making it accessible for quick iterations.
- It is suitable for handling a limited number of concurrent users, typically fewer than 100.

Disadvantages (GeeksforGeeks 2016):

- Each client request results in a new thread, leading to significant overhead in larger-scale environments due to constant thread creation and destruction.
- Managing the lifecycle of threads became increasingly complex, raising concerns about resource leaks and the challenges of imposing limits on the number of active threads.

*Thread Pools / Executors*

As a more scalable alternative, employing the Java Executor framework (using Executors.newFixedThreadPool()) presents significant advantages. This approach enables thread reuse, improved resource management, and the inclusion of built-in queuing mechanisms for new tasks.

Advantages (Loy, Leuck, and Niemeyer 2023)):

- Improved performance through effective reuse of threads, reducing the overhead associated with thread creation.
- Scalability to accommodate higher levels of concurrency, thus enhancing the application's overall throughput.
- Simplified management of active thread count, allowing for more robust handling of system resources.

Disadvantages (Loy, Leuck, and Niemeyer 2023):

- The initial implementation and configuration are slightly more complex compared to manual thread creation.
- It may require tuning of pool sizes and queue types to optimize performance according to the specific workload.

Looking ahead, I am planning to migrate to a thread pool model in future iterations, targeting enhanced efficiency and responsiveness, especially as the number of concurrent clients continues to increase (Loy, Leuck, and Niemeyer 2023) This shift is likely to enhance the application's ability to handle a larger volume of requests while maintaining high levels of performance and reliability.

## Reflection

Multithreading has significantly enhanced the responsiveness and throughput of the parking server. Without it, the server would block client requests, resulting in poor scalability and a suboptimal user experience. With threads, multiple users can register or park simultaneously without delays. This project deepened my understanding of Java’s concurrency tools and the importance of designing loosely coupled components, such as separating request parsing from execution. It highlighted the need to strike a balance between safety and performance when managing shared data. From a software engineering perspective, multithreading creates a more realistic architecture, similar to how web servers and database engines handle many concurrent users (Fesenko 2019). This experience reinforced the value of testability and logging in addressing non-deterministic behaviors. Although multithreading adds complexity, it is essential for building scalable, efficient applications. By managing shared

resources and following best practices, I developed a more robust and maintainable parking system server.

**References**

Baeldung. 2018. “Baeldung.” Baeldung. May 5, 2018. https://www.baeldung.com/java-concurrency.

“Convert Java Object to Json String Using GSON.” 2019. GeeksforGeeks. April 4, 2019. https://www.geeksforgeeks.org/convert-java-object-to-json-string-using-gson/.

Fesenko, Tatiana . 2019. “Java Concurrency and Multithreading in Practice.” O’Reilly Online Learning. 2019. https://learning.oreilly.com/course/java-concurrency-and/9781789806410/.

GeeksforGeeks. 2016. “Multithreading in Java.” GeeksforGeeks. January 9, 2016. https://www.geeksforgeeks.org/multithreading-in-java/#.

Loy, Marc, Daniel Leuck, and Patrick Niemeyer. 2023. “Learning Java, 6th Edition [Book].” Www.oreilly.com. August 2023. https://learning.oreilly.com/library/view/learning-java-6th/9781098145521/.

Martin, Robert . 2017. *Clean Architecture*. Pearson.

W3Schools. n.d. “Java Threads.” Www.w3schools.com. https://www.w3schools.com/java/java_threads.asp.