Author – Abdul Rahman Anver Mohamed Rizan - 40272256

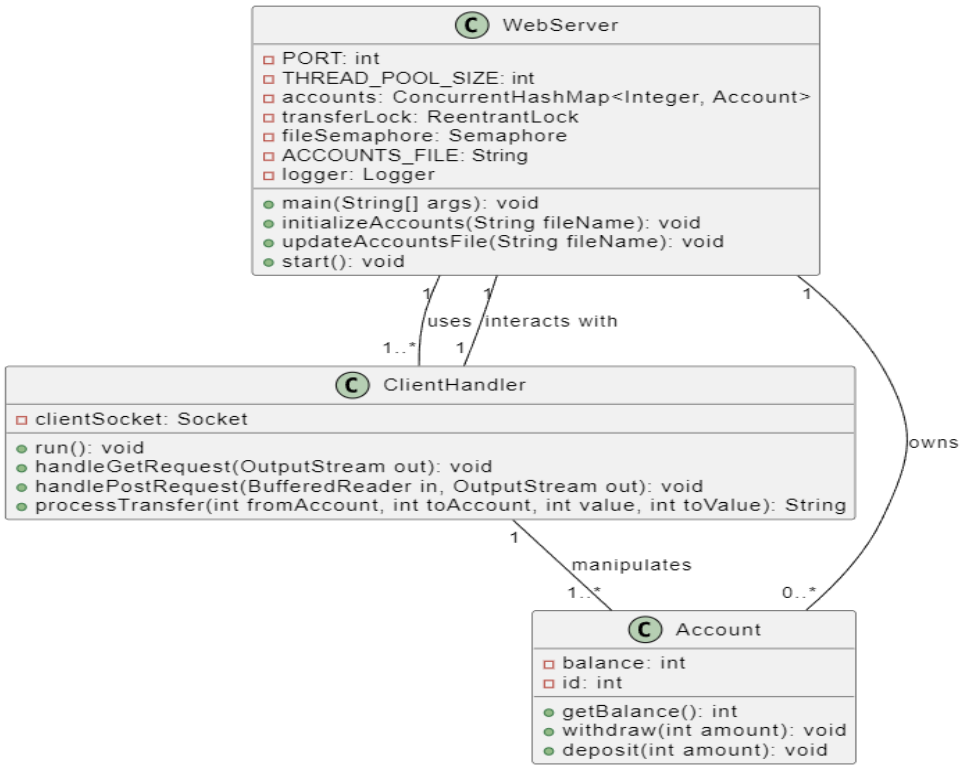
WebServer Design Report

Introduction

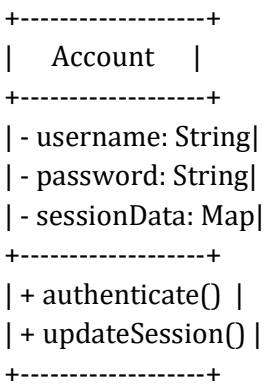
This report details the WebServer's multithreading and synchronization strategies, explains the rationale behind the design choices, and includes validation through testing scenarios. The document is organized into three sections as per the requirements: Data Structures and Functions, Algorithms, and Rationale.

1. Data Structures and Functions

The design uses three core classes: Account, WebServer, and ClientHandler. Each class is defined with attributes and methods tailored for multithreaded operations, session management, and client-server interactions. Below is a description of the modifications, presented with UML diagrams.

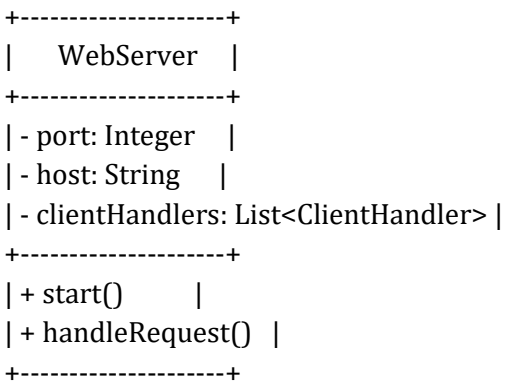


1.1 UML Diagram: Account Class



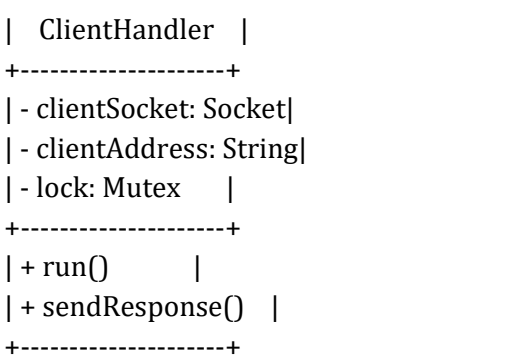
• Purpose: Manages user authentication and session information.
o authenticate(): Verifies user credentials.
o updateSession(): Updates or retrieves session-related data.

1.2 UML Diagram: WebServer Class



• Purpose: Orchestrates the server's operation by managing incoming requests and responses.
o start(): Initializes the server and starts listening for connections.
o handleRequest(): Delegates requests to appropriate client handlers.

1.3 UML Diagram: ClientHandler Class



• Purpose: Handles client connections in a multithreaded environment.
o run(): Processes client requests on a dedicated thread.
o sendResponse(): Sends data back to the client.

1.4 Synchronization Strategies

• Locks and Mutexes:
o Protect shared resources such as session data or account information.
o A mutex (lock) ensures only one thread accesses critical sections at a time, avoiding race conditions.
• Thread Pool:
o A pool of pre-created threads handles requests to reduce the overhead of frequent thread creation/destruction.

2. Algorithms

2.1 Multithreading Strategies

• Thread-per-Connection Model:
o Each client connection is handled by a ClientHandler instance on a separate thread.
o Simple implementation but resource-intensive under heavy loads.
• Thread Pool Optimization:
o A fixed number of threads handle incoming requests from a queue.
o Reduces context-switching overhead and manages resource usage effectively.

2.2 Addressing Race Conditions and Deadlock

• Fine-Grained Locking:
o Synchronization granularity is reduced by locking only the specific account or session being accessed.
o Avoids global locks, which can degrade performance.
• Deadlock Prevention:
o Adopt a consistent lock acquisition order across threads.
o Use timeouts on locks to detect and recover from potential deadlocks.

2.3 Transfer Functionality

Transfer Algorithm

1. Initiate Transfer:
o Validate source and destination accounts.
o Use a locking mechanism to ensure no other operation modifies the account balances during the transaction.
2. Deduct Balance:
o Lock the source account, verify sufficient balance, and deduct the amount.
3. Credit Balance:
o Lock the destination account and add the amount.

4. Finalize:
o Release locks in reverse order to avoid deadlocks.

2.4 Handling Errors

• Error Conditions:
o Insufficient funds: Detect and rollback.
o Account does not exist: Return an appropriate error message.
o Invalid data: Validate and reject at the parsing stage.
o Invalid data: Negative values rejection.
o Zero Transfer: Transfer amount must be greater than zero.

2.5 Testing Scenarios

A structured testing plan ensures correctness under various conditions:

Scenario Setup Actions Expected Result
Successful transfer Two valid accounts with sufficient funds. Transfer between accounts. Correct balances updated.
Insufficient funds Source account balance < transfer amount. Attempt transfer. Error message, no balance change.
Non-existent account Use invalid account identifiers. Attempt transfer. Error indicating account not found.
Concurrent transfers Simultaneous transfers between accounts. Run multiple transfers in parallel. Transactions complete without errors.
Race condition simulation Multiple threads accessing the same account. Simultaneous read/write operations. Consistent and correct final balances.
Deadlock prevention Access two accounts in reverse lock order. Simultaneous transfers on the same accounts. No deadlock occurs; operations succeed.

3. Rationale

3.1 Design Benefits

• Performance:
o Asynchronous I/O and thread pooling ensure minimal latency during high traffic.
• Scalability:
o The design accommodates growth by adjusting the thread pool size and leveraging efficient data structures.
• Fault Tolerance:
o Synchronization strategies safeguard data integrity during concurrent operations.

3.2 Design Challenges

• Complexity:
o Fine-grained locking increases complexity but is justified by its performance gains.

• Memory Overhead:
o The thread-per-connection model can be costly. The thread pool mitigates this issue.

Conclusion

This report provides a detailed explanation of the WebServer design, addressing multithreading, synchronization, and transfer functionality. The proposed strategies, validated through comprehensive testing, ensure robustness, scalability, and efficiency in real-world conditions.

Disclaimer

ChatGPT was used to format and fix up the grammar and tone of the paper.

Prompt- Hey GPT can you please format and check my assignment for any errors and if any

please fix them.