David Xiang

1083691

Distributed Systems (COMP90015)

**Problem Context**

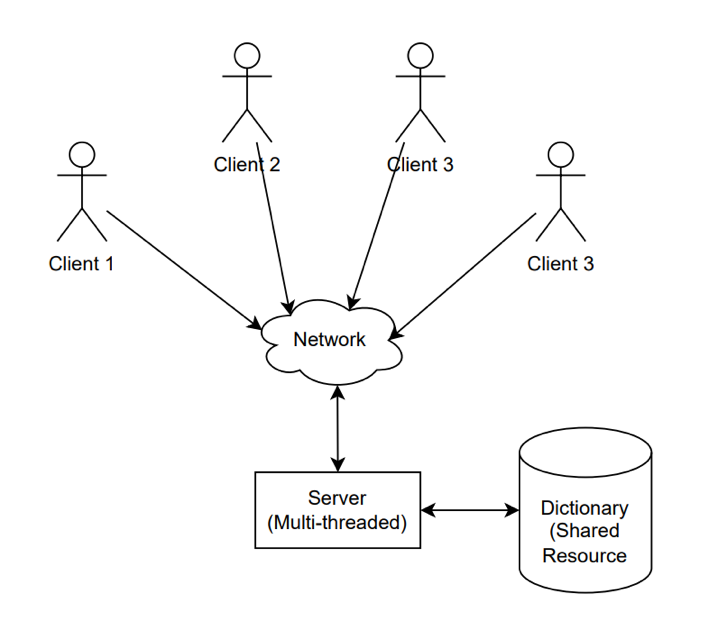
The objective of this project is to design and implement a multi-threaded dictionary server that supports concurrent access by multiple clients over a distributed network. The system is built following the client-server architecture, where a single, centralised server manages a shared dictionary resource, and multiple clients interact with it concurrently. The objectives of this project is outlined below:

* Develop a server that can handle simultaneous client connections using multi-threading.
* Implement core dictionary operations with proper synchronization to prevent data inconsistency and corruption.
* Ensure clear and informative responses for both successful and failed operations.
* Model the system on a client-server architecture, enabling remote access via a network.

The system follows a **client-server** architecture where:

* A single server is responsible for maintaining the dictionary (shared resource).
* Multiple clients can connect to the server concurrently and request operations on the dictionary.
* The server handles each client in a separate thread, allowing for parallel execution of operations.

A simplified diagram of the architecture is illustrated below:

****

The functional requirements specify that a client can make the following operations on the dictionary:

* **Query Word**
  + Output: Returns the meaning(s) of the word.
  + Error Handling: If the word is not found or an error occurs, the client receives a clear and descriptive error message.
* **Add Word**
  + Output: Indicates the status of the operation (e.g., success, duplicate).
  + Error Handling: Any errors during the addition are reported to the client.
* **Remove Word** 
  + Output: Indicates the status of the operation (e.g., success, not found).
  + Error Handling: Errors encountered during removal are communicated clearly.
* **Append Meaning**
  + Output: Indicates the status of the operation (e.g., success, not found).
  + Error Handling: If the operation fails, the user is notified with an appropriate message.
* **Update Meaning** 
  + Output: Indicates the status of the operation (e.g., success, not found).
  + Error Handling: Errors encountered during the update are reported to the client.
* **Exit** 
  + Ends the session and disconnects the client from the server gracefully.

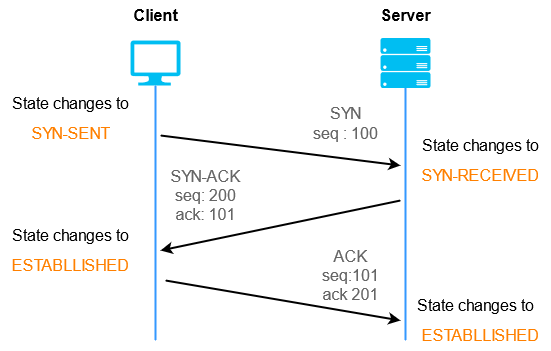
This project emphasizes **network communication**, **multi-threaded programming**, and **shared data access** in a concurrent environment. It aims to simulate a realistic server-client setup that requires careful attention to **synchronization**, **robustness**, and **user feedback** to ensure a functional and reliable system.

**System Components**

**Network Communication**

Sockets and Threads serve as the lowest level of abstraction for handling communication and concurrency. The system uses TCP sockets via Java’s ServerSocket and Socket classes to manage reliable two-way communication between client and server. TCP was chosen due to its built-in reliability guarantees:

* Ensures ordered, lossless, and duplicate-free transmission of data.
* Establishes a connection-oriented session before any data is exchanged, making it ideal for client-server applications.
* A simplified illustration of TCP’s reliability (three-way handshake, ACKs, retransmission) may be included for clarity.



**Dictionary Data Storage**

The dictionary is initially loaded from a CSV (Comma-Separated Values) file.

This format was chosen due to its simplicity and ease of parsing in Java. Users are instructed on how to format their entries to comply with the CSV structure. Basic error handling ensures malformed entries are rejected or handled gracefully (to be confirmed).

When the server starts: It loads the dictionary from the CSV file into memory. A thread-safe in-memory data structure (ConcurrentHashMap) is used to maintain fast, concurrent access to dictionary entries. All updates (additions, deletions, updates) are reflected in both memory and the CSV file to ensure persistence across sessions.

Why ConcurrentHashMap was chosen:

* Implements the ConcurrentMap interface, providing a highly concurrent and scalable map implementation.
* Utilizes segment-based locking to allow concurrent reads and writes on different parts of the map.
* Ensures thread safety without needing explicit synchronization for most operations.
* Does not allow null keys or values, which aligns with dictionary use cases.

**Server-Side Concurrency Model**

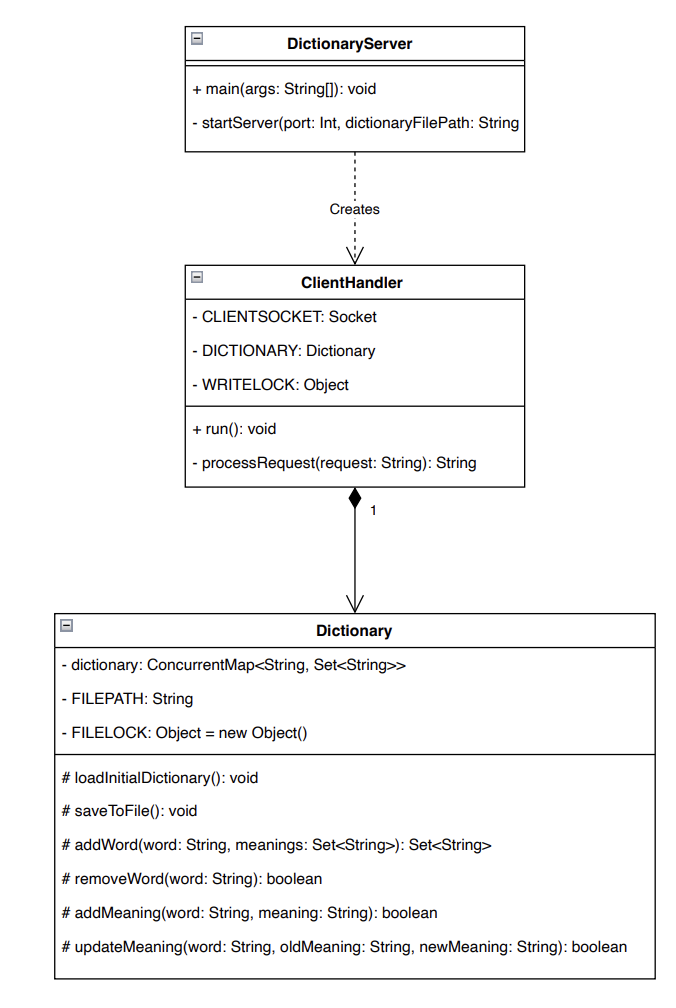
The server follows a **thread-per-connection** model.

1. The main method reads the port number and dictionary file path from the command line.
2. It calls the StartServer method which:
   1. Loads the dictionary from the CSV.
   2. Sets up a ServerSocket listening on the specified port.
   3. Accepts client connections in a loop.
   4. For each client, a new thread is created using ClientHandler.

This design ensures that one connection is accepted at a time, but each is delegated to its own thread. The server remains responsive to new clients and that each client is handled independently without blocking others.

The ClientHandler responsibilities are:

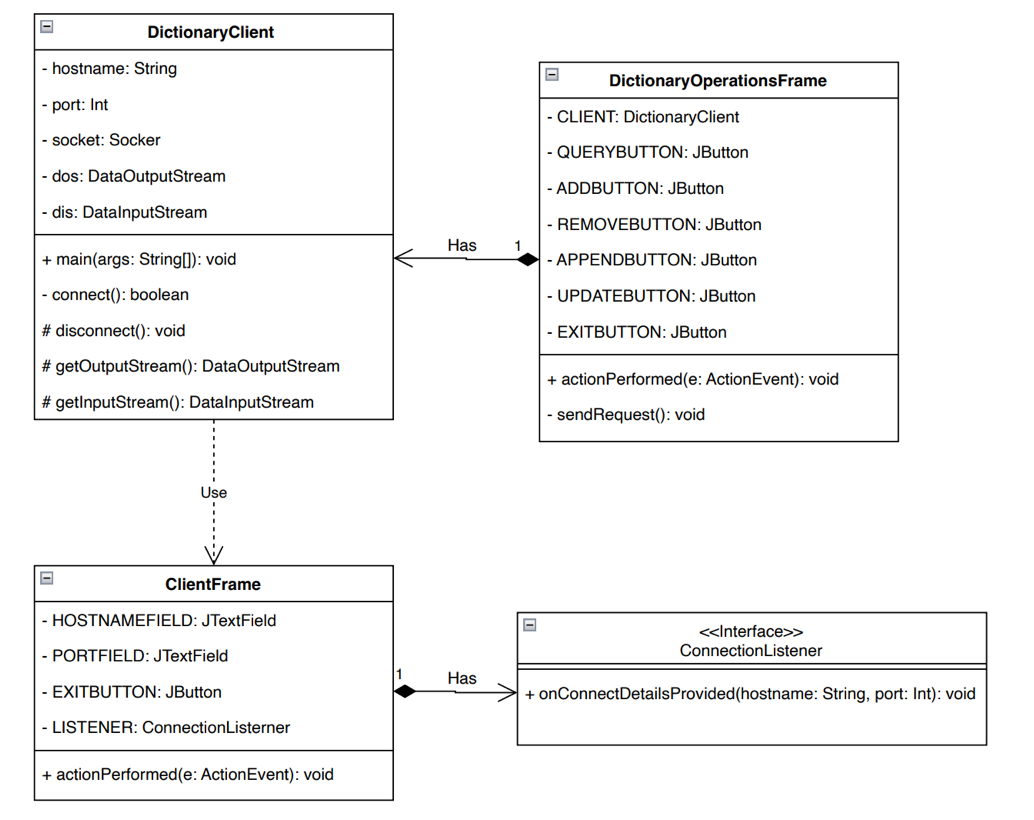
* Implements Runnable.
* Parses and responds to client commands (query, add, remove, etc.).
* Maintains a persistent session with the client over a single TCP connection.
* Processes sequential requests from the client—concurrent operations within a single client are not supported (and not needed).



**Client-Side Model and GUI**

A graphical interface is provided using Java Swing for enhanced user interaction. The components for the GUI include:

* **ClientFrame**
  + Displayed upon launching the DictionaryClient program.
  + Contains input fields for the hostname and port number.
  + Replaces command-line argument entry to mimic real-world usability.
* **DictionaryOperationsFrame**
  + Appears after a successful connection to the server.
  + Displays a homepage with buttons for supported dictionary operations.
  + Each button links to a dedicated interface prompting the user for relevant input and displaying the results.



**Analysis of Design Choices and Implementation**

The design of the multi-threaded dictionary server reflects careful consideration of concurrency, reliability, usability, and maintainability. Below is an analysis of the major design decisions and their justifications.

**TCP Sockets for Network Communication**

TCP was selected for its reliable, ordered, and connection-oriented communication model which is essential for consistent interaction between clients and the server. The advantages of using TCP include:

* Built-in error checking, packet reordering, and retransmission.
* Ensures data integrity — crucial for command and response handling in a stateful system.
* Supports persistent connections, allowing clients to perform multiple operations without reconnecting.

Other alternatives were considered such UDP which is lightweight and connectionless. However, it does not guarantee delivery or order, making it unsuitable for this use case.

**Thread-per-Client Model**

A straightforward and intuitive model where each client is assigned a dedicated thread. It simplifies client management and state tracking. In addition, this model makes it easy to implement persistent connections, where clients stay connected and perform multiple operations.

Advantages:

* Decouples client handling from connection acceptance.
* Naturally aligns with the object-oriented structure (e.g., one ClientHandler per client).
* Well-suited for systems with a moderate number of clients.

Trade-offs:

* Threads are relatively heavy, not optimal for thousands of concurrent clients.
* Requires careful resource management and exception handling to prevent thread leaks.

Scalability Consideration: Could be extended in the future using a thread pool or non-blocking I/O (NIO) model if scalability becomes a concern.

**CSV File Format**

CSV provides a lightweight and human-readable format for storing the dictionary. It is relatively easy to load and parse using standard Java I/O libraries. It is also sufficient for storing simple key-value pairs like words and their meanings. The advantages of CSV include:

* Easy to inspect and edit manually if needed.
* Simple integration into the server’s load/save logic.

The disadvantages of using CSV may include:

* Not suitable for complex data relationships or very large datasets.
* File I/O operations must be synchronized to avoid corruption in a concurrent environment.

An alternative that was considered was JSON however in consideration of usability and simplicity, it was decided against.

**Use of ConcurrentHashMap for the Dictionary**

The dictionary represents a shared resource accessed by multiple threads (clients) concurrently. To ensure thread-safe operations without incurring the performance penalties of synchronizing the entire data structure, ConcurrentHashMap was used. This data structure allows:

* Safe concurrent reads without locking.
* Safe writes using fine-grained locking (segment-level).
* High throughput in multi-threaded environments.

Advantages: •

* Eliminates the need for explicit synchronization on most operations.
* Scales well with the number of threads.
* Standard and well-tested in the Java concurrency library.

Alternatives Considered:

* HashMap (not thread-safe) — would require manual synchronization.

**Concurrency Control at the Dictionary Level**

Managing concurrent access was implemented at the dictionary level and not within each ClientHandler. This choice was made largely because each ClientHandler runs in its own thread and handles one client at a time. Since client operations are sequential, there is no need for additional concurrency control within each handler. The only shared resource across handlers is the dictionary, so concurrency concerns are centralized there.

Advantages:

* Simplifies the ClientHandler logic since there is no need for additional locks or thread coordination.
* Reduces the risk of deadlocks or race conditions at the handler level.
* Ensures clean separation of concerns.

Therefore, by implementing concurrency control at the Dictionary level, each client has a dedicated thread that serves them throughout their session. Shared access to dictionary methods is managed internally using thread-safe data structures and minimal synchronized blocks where necessary.

**Use of Java Swing for the GUI**

Java Swing is a built-in GUI framework that integrates well with Java networking and is sufficient for a basic client interface. The project’s implementation allows users to connect to the server, input commands, and receive responses in an intuitive and visual way.

Advantages:

* Provides a more realistic and user-friendly experience compared to command-line arguments.
* Organizes user flow with separate screens (frames) for connection setup and operations.
* Makes the application accessible to non-technical users.

Trade-offs:

* Swing is somewhat dated and less responsive compared to modern GUI frameworks.
* Limited styling/customization capabilities without additional libraries.

Alternatives Considered:

* JavaFX – More modern but may add unnecessary complexity.
* Console-only UI – Simpler but less user-friendly and realistic.

**Error Handling and Feedback Design**

Each client operation provides clear status messages, helping users understand whether:

* The operation succeeded (e.g., word added).
* The operation failed (e.g., word not found, duplicate entry).
* An unexpected error occurred (e.g., malformed command).

Error handling is crucial as it enhances user experience and facilitates debugging and error recovery. It also helps promote robustness and reliability.

**Conclusion**

This project successfully demonstrates the design and implementation of a robust, multi-threaded dictionary server using the client-server architecture. By leveraging Java’s concurrency features and networking capabilities, the system enables multiple clients to access and manipulate a shared dictionary resource concurrently and reliably. Key design decisions—such as using ConcurrentHashMap for thread-safe dictionary access, TCP sockets for reliable communication, and CSV for lightweight persistence—were made with a focus on simplicity, performance, and maintainability. The adoption of a thread-per-client model and centralized concurrency control further ensures responsiveness and correctness under concurrent usage. The addition of a user-friendly GUI using Java Swing enhances accessibility and simulates a more realistic application scenario, while the modular structure of the system allows for future scalability and extension. Overall, the system meets its functional and non-functional requirements, providing a strong foundation for further development or adaptation in more complex distributed applications.