**1. Introduction**

In the age of digitization, efficient access to information is paramount. The Distributed Dictionary System developed here offers a solution that allows users to seamlessly query, add, update, or remove words and their meanings in a digital dictionary. The system is designed keeping scalability, concurrency, and error handling in mind, leveraging the power of TCP sockets for communication and a worker-pool architecture for efficient request processing.

This report provides a comprehensive overview of the system, detailing the design choices made during its development, the components involved, and their interactions. It also delves into the critical analysis of the system's capabilities and the challenges encountered during its implementation.

**2. Problem Context**

The Distributed Dictionary System was developed in the context of increasing demand for distributed applications that allow multiple users to access and modify shared resources in real-time. With the proliferation of digital content and the need for efficient information retrieval, a system that can handle concurrent requests for word definitions becomes indispensable.

The challenges presented in this context include:

1. **Concurrent Access:** Multiple users might try to access or modify the same word simultaneously. The system must ensure that these operations are handled without conflicts, maintaining data consistency.
2. **Scalability:** As the number of users grows, the system should be able to handle increased load without significant degradation in performance.
3. **Error Handling:** In a distributed environment, various types of errors can occur, from network issues to database inconsistencies. The system must be robust enough to handle these errors gracefully, providing meaningful feedback to the user.
4. **Real-time Updates:** When a word is added, modified, or removed by one client, the changes should be immediately visible to all other connected clients.
5. **Efficiency:** Given the potentially vast size of the dictionary, the system should be able to retrieve or modify entries swiftly, ensuring a seamless user experience.

To address these challenges, the Distributed Dictionary System was designed with a clear focus on concurrency management, efficient data retrieval, and robust error handling mechanisms. Through the use of TCP sockets and a worker-pool architecture, the system ensures that multiple clients can interact with the dictionary concurrently, with minimal wait times and consistent data views.

**3. Components of the System**

The Distributed Dictionary System is structured into two primary components: the **DictionaryServer** and the **DictionaryClient**. These components interact with each other to facilitate the various operations available in the system. Below is a detailed description of each component aligned with the provided code:

3.1 DictionaryServer (**DictionaryServer.java**)

The **DictionaryServer** acts as the central authority that manages the shared dictionary resource and handles incoming client requests. It comprises several key elements:

* **ServerSocket**: This is the main listening socket that waits for incoming client connections. Once a client connects, it spawns a new task to handle the client's request, ensuring that the main server remains available to accept new connections.
* **Connection to Database**: The server establishes a connection to an SQLite database, which stores the dictionary's word-meaning pairs. This database connection facilitates operations like adding, removing, updating, and querying words.
* **Worker-Pool Architecture**: The server employs a thread pool to manage incoming client requests. This ensures that a fixed number of worker threads are available to handle client operations concurrently, enhancing the system's scalability and responsiveness.
* **Dictionary HashMap**: An in-memory representation of the dictionary, which allows for swift lookups and modifications. It gets populated at server startup by reading from the database.
* **Server GUI**: Although not elaborated in the provided snippets, it can be inferred that there's a server GUI component. This might be used for displaying server statistics, logs, or other relevant information.

3.2 DictionaryClient (**DictionaryClient.java**)

The **DictionaryClient** serves as the user's interface to interact with the dictionary. It encapsulates the following functionalities:

* **TCP Client Socket**: Establishes a connection to the server, enabling the client to send requests and receive responses.
* **JSON Request-Response Protocol**: The client and server communicate using a JSON-based protocol. This ensures structured and consistent data exchange, making it easier to handle different types of requests and responses.
* **Operations**: The client can perform various operations such as querying a word's meaning, adding a new word, removing an existing word, and updating a word's meaning. Each operation corresponds to a specific action in the JSON request sent to the server.
* **Error Handling**: The client is equipped to handle various types of errors, from network issues to data-related inconsistencies. Meaningful error messages are displayed to the user in case of any issues.

**4. Class Design**

**4.1 DictionaryClient**

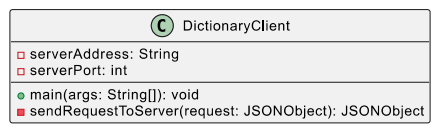
The DictionaryClient class is responsible for establishing a connection to the DictionaryServer, sending requests based on user input, and displaying the server's responses to the user. It provides a simple user interface for users to interact with the dictionary service.

Attributes:

* serverAddress: The IP address or hostname of the DictionaryServer.
* port: The port on which the server is listening.
* clientSocket: The client socket used for communication with the server.
* Various GUI components to facilitate user interaction, such as text fields, buttons, and labels.

Key Methods:

* connectToServer(): Establishes a connection to the DictionaryServer.
* sendRequestToServer(JSONObject request): Sends a JSON-formatted request to the server and receives a response.
* Various helper methods to handle specific user actions, such as searching for a word, adding a new word, updating a word's meaning, and removing a word.



**4.2 DictionaryServer**

The DictionaryServer class serves as the backbone of the dictionary service. This component is responsible for establishing the server-side socket connection, listening for incoming client requests, and handling those requests by delegating them to a pool of worker threads. It also manages the dictionary data by interfacing with an SQLite database.

The following are the key attributes and methods associated with the DictionaryServer class:

Attributes:

* numberOfWorkers: Represents the number of worker threads in the thread pool.
* threadPool: A thread pool to efficiently manage and reuse worker threads.
* port: The port on which the server listens for incoming connections.
* dictionaryFilePath: Path to the SQLite database file.
* dictionary: A HashMap used as a cache to store the dictionary data for faster access.
* serverSocket: The server socket for accepting client connections.
* lblCurrentWorkers: A GUI label to display the number of active worker threads.
* connection: A database connection object for interfacing with the SQLite database.

Key Methods:

* connectToDatabase(): Establishes a connection to the SQLite database and ensures the necessary tables exist.
* loadDictionary(): Loads the dictionary data from the SQLite database into the HashMap for caching purposes.
* startServer(): Initializes the server socket and listens for incoming client connections.
* handleClientRequest(Socket clientSocket): Handles individual client requests, including searching for words, adding new words, updating word meanings, and removing words.

日程表

描述已自动生成

**4.3 Interaction Diagram**

The interaction diagram will showcase the sequence of operations and interactions between the **DictionaryServer** and **DictionaryClient** during typical use cases, such as searching for a word or adding a new word to the dictionary.

