**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**

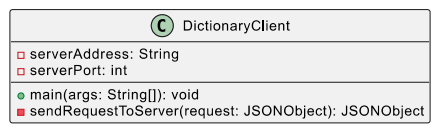
The DictionaryClient class is responsible for establishing a connection to the DictionaryServer, sending requests based on user input, and displaying the server's responses to the user. It provides a simple user interface for users to interact with the dictionary service.

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 System Interaction**

**The worker-pool architecture is a design pattern used in the DictionaryServer to efficiently manage the execution of tasks in a multi-threaded environment. It employs a set of worker threads to carry out tasks that are stored in a queue.**

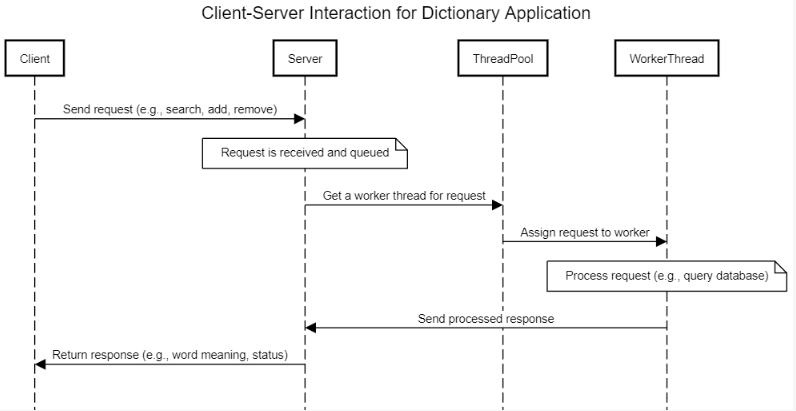
**In our implementation, we use a custom-built ThreadPool class that manages the worker threads. The ThreadPool class maintains a queue of tasks and a set of worker threads, represented by instances of the WorkerThread class. When a new client connection is established, the server places the task of handling that client into the task queue. Worker threads, which are already running in a loop, pick up these tasks from the queue and execute them.**

**Concurrent Request Handling**

**Here's a high-level overview of how the worker-pool architecture handles incoming requests:**

1. **Task Queuing: When a new client sends a request to the server, the server encapsulates the handling of that request as a 'task' and places it in the task queue.**
2. **Worker Availability: Worker threads in the ThreadPool are continuously monitoring the task queue for new tasks.**
3. **Task Pickup: As soon as a worker thread finds a task in the queue, it picks it up for execution. This allows multiple threads to work simultaneously, each handling a different client request.**
4. **Task Execution: The worker thread executes the task, which could involve querying, adding, removing, or updating a word in the dictionary. Since each worker thread operates independently of the others, multiple requests can be handled in parallel.**
5. **Response: After completing the task, the worker thread sends the appropriate response back to the client.**
6. **Awaiting Next Task: Once a task is completed, the worker thread goes back to monitoring the task queue, waiting to pick up the next available task.**

**By using this worker-pool architecture, we not only distribute the computational load across multiple threads but also keep the server responsive even under heavy loads. This setup is particularly useful when dealing with a large number of simultaneous client connections, as it allows the system to scale more efficiently. We also prepare an interaction diagram that visualizing how this system are interacted with each other.**



**6. Error Handling**

**6.1 Normal Error Handling**

**Multiple Clients**

During testing, we initialised 4 clients on different device and sent the request to the server, 4 clients can get the responses, and the result of each action are visible to all clients. It was verified that multiple clients could connect to the server simultaneously and perform various actions like adding, updating, and searching for words.

**6.2 Specific Test Scenarios**

The system was rigorously tested for all functionalities: Querying, Adding, Removing, and Updating words. Here are some highlights:

* **Query Word**: The system can process the ‘search’ action about the **special character**, while the **empty string** will respond to the error messages to clients.
* **Add a New Word**: The system prevented the addition of **duplicate** words and words **without meanings**.
* **Remove an Existing Word**: The server responded with a "not found" message when trying to remove a word that does not exist.
* **Update Meaning**: Similar to "Remove", the system efficiently handled the "not found" scenario.

**6.3 Abnormal Scenarios**

For testing the scenarios in which the client sends an invalid or malformed, we simulate a situation in which the request is not sent from GUI. In this situation, the system responds with suitable error messages like "Invalid JSON format", "Missing 'word' field in search request", or “Unknown Action”.

**7. Excellence Elements**

**7.1 Notification of Error in Edge Cases**

1. Database Errors:  
   If the SQLite database encounters an issue, such as being deleted or renamed while the server is running, the server responds with appropriate error messages for "add", "remove", and "update" actions. However, "search" actions continue to work as expected because the server also maintains a HashMap of the dictionary.
2. Abnormal Scenarios:  
   For testing the scenarios in which the client sends an invalid or malformed request, we simulate a situation where the request is not sent via the GUI. In this situation, the system responds with suitable error messages like "Invalid JSON format," "Missing 'word' field in search request," or "Unknown Action."
3. Server's GUI Worker Limit:  
   The server's GUI will inform the client that the worker count should be greater than 1 when the user tries to decrease the worker count to 0.
4. Address Binding Error:  
   If the client wants to run the server on a port that is already in use, the log area will display an associated error message to the user.

**7.2 Notification of Dangerous Actions**

In the server's GUI, we provide buttons for increasing and decreasing the number of worker threads. If a user attempts to increase the worker count higher than the available processors on the current device, the log area will display a warning message.

**7.3 Critical Analysis of Worker-Pool Architecture**

To test how the system behaves in a more realistic situation, we conduct a high concurrency test for the DictionaryServer by using the JMeter. Here is the experiment setup:

1. **400 concurrency, 4 workers**:
   * **Errors**: 2 errors were encountered out of 400 requests. This is a 0.5% error rate.
   * **Average Response Time**: 7ms
   * **Analysis**: With 400 concurrent requests and only 4 workers, it's notable that the system was able to handle the load with a very low error rate and a reasonable response time.
2. **350 concurrency, 4 workers**:
   * **Errors**: 0 errors
   * **Average Response Time**: 6ms
   * **Analysis**: Reducing the concurrency by 50 requests resulted in no errors and slightly improved response time.
3. **350 concurrency, 3 workers**:
   * **Errors**: 0 errors
   * **Average Response Time**: 6ms
   * **Analysis**: Despite reducing the number of workers by one, the system was able to handle the same load (350 requests) without any errors and with the same average response time as with 4 workers. This suggests that the system is efficiently distributing tasks among the available workers.
4. **350 concurrency, 2 workers**:
   * **Errors**: 10 out of 350 requests, approximately a 2.86% error rate.
   * **Average Response Time**: 4ms
   * **Analysis**: Reducing the worker count to half while maintaining the same level of concurrency resulted in a higher error rate. Interestingly, the average response time improved, which might suggest that while the system can process requests faster with fewer workers, it becomes more prone to errors due to overloading.

Based on the experiments conducted, here are some advantages and disadvantages of the worker-pool architecture:

Advantages:

* Efficiency: The system can handle a high number of concurrent requests with a relatively low error rate, as evidenced by the 0.5% error rate with 400 concurrent requests and 4 workers.
* Scalability: The system can maintain low average response times even when the number of concurrent requests is high.
* Resource Utilization: The experiment with 350 requests and 3 workers showed no errors and consistent response times, indicating that the worker-pool architecture efficiently utilizes resources.

Disadvantages:

* Error Sensitivity: Reducing the number of workers while maintaining high concurrency levels can increase the error rate, as seen with 350 requests and 2 workers.
* Manually Intervention: Although we set up a recommended number of workers for users, it still needs some intervention. In some cases, using up all the available processors of the device might not be the expectation of the user. So, it will require the user to have some basic understanding of the device and the work for choosing a suitable number of workers.
* Optimal Configuration Needed: The results suggest that there is a trade-off between the number of workers and the level of concurrency. Striking the right balance is crucial for optimal performance, but this will require many times to do the experiments.

Conclusion:The system appears to meet expectations for handling concurrent requests efficiently using a worker-pool architecture. However, further exploration into the causes of the errors, especially when using fewer workers, could provide avenues for optimizing the system further.