Assignment 1-Distributed System

Name: Ninghao Zhu(Bruce)

Student ID: 1446180

School of Computing and Information Systems, The University of Melbourne

COMP90015: Distributed System

April 13, 2025

Problem Context

This report aims to illustrate the design and implementation of a dictionary system called NetDictionary, which adopts a client-server architecture. Built upon two fundamental technologies, Sockets and Threads, NetDictionary was developed step by step to support a multi-threaded server. It allows multiple clients to concurrently search for word meanings, add new words, remove existing words, add additional meanings, and update word meanings through a graphical user interface (GUI). Additionally, the server GUI can display active client connections and their operations in real time.

System Overview

NetDictionary uses a client-server architecture, allowing multiple clients to connect to a multi-threaded server and perform operations concurrently. Initially, the server was implemented using a thread-per-request architecture, but it was later modified to adopt a worker pool architecture for improved efficiency and scalability.

Client

The client side consists of two main components: Request and View.

Request

This contains classes that define a standard and unified message exchange protocol between the client and the server.

View

This is responsible for launching the Client GUI and establishing a TCP connection to the server. Once connected, it registers event listeners to handle various dictionary operations. The client and server communicate using the standardized classes from the Request component to maintain consistent message formatting.

Server

The server side includes the following components: Data, Request, and Threads folders, as well as the MultiController, Server, and ServerGUI classes.

Data

This folder contains classes responsible for saving and reading data from a JSON file, as well as providing methods to perform dictionary operations on this standardized data.

Request

This folder mirrors the Request folder in the client and includes a RequestHandler, which processes dictionary operation requests sent from the client.

Threads

This component implements a worker pool architecture to enqueue client requests and process them using a fixed number of threads, ensuring efficient task management.

Server, MultiController, and ServerGUI

These components launch the Server GUI, open sockets for client connections, and instantiate the worker pool to execute incoming client requests. The ServerGUI also displays real-time messages and client interactions.

Class Design

Message.java

The Message class includes a type property used to distinguish between different request types. The word, meanings, existMeaning, and newMeaning properties are used for data exchange between the client and server.

RequestType.java

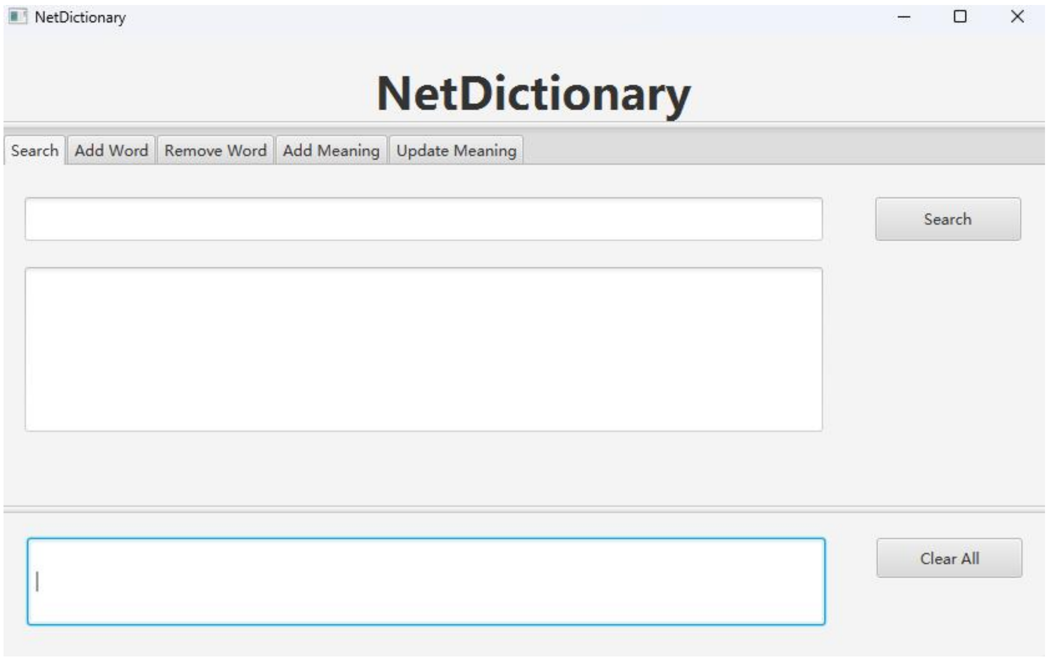
The RequestType class defines five types of dictionary operations: query, add, remove, addMeanings, and update.

Response.java

The success property indicates whether a request was successfully processed by the server. The message property carries notifications or response messages returned from the server after each operation.

NetDictionaryClient.java

This class receives the server's address and port number from the command line. It establishes a TCP connection to the server, opens input and output streams for communication, and launches the Client GUI.



ViewController.java

This class manages all buttons and text fields in the Client GUI. It registers event handlers to perform different dictionary operations based on user interaction.

Dictionary.java

This class uses a ConcurrentHashMap to store and retrieve words along with their associated meanings (in the form of a String and a List<String>) concurrently. A synchronized lock is applied to ensure that only one dictionary operation is processed at a time, maintaining thread safety.

DictionaryFile.java

This class handles saving and reading dictionary data from a local JSON file. If the original dictionary JSON file is not located in the same directory as the DictionaryServer.jar file, the system will automatically create an empty temp.json file for storing and retrieving dictionary data.

RequestHandler.java

This class implements the Runnable interface. In the run() method, it identifies the type of request such as query, add, remove, addMeanings, and update, and processes it accordingly. It interacts with both the Dictionary and DictionaryFile classes to handle the data, and then sends an appropriate response back to the client.

ThreadsWorkerPool.java

This class enqueues operations from RequestHandler into a LinkedBlockingQueue. It uses a WorkerFactory to create a fixed number of worker threads, which continuously execute operations from the queue.

MultiController.java

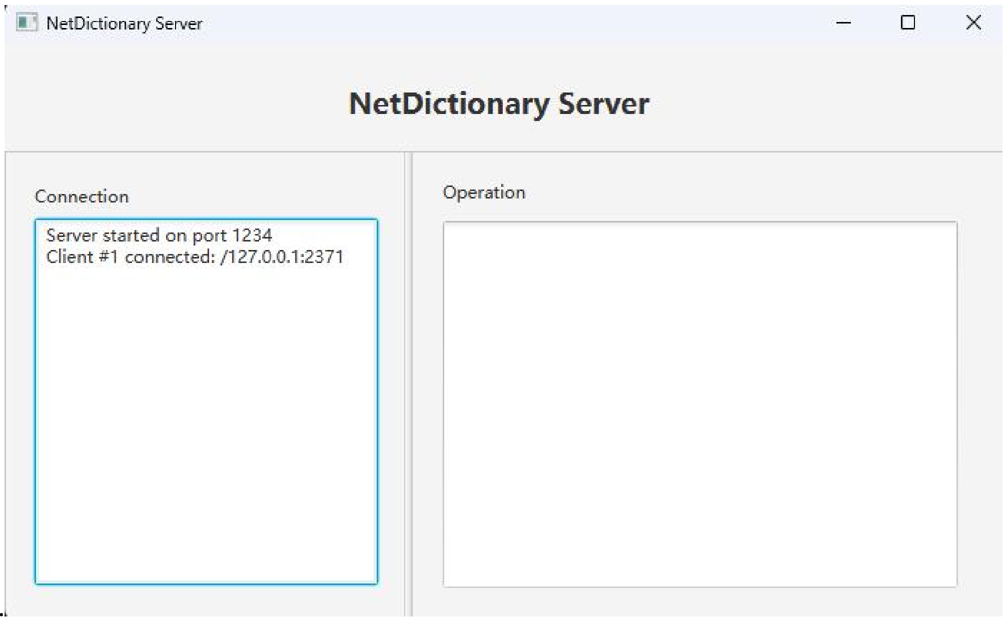
This class displays TCP connection messages from clients as well as logs of their interactions with the dictionary.

Server.java

This class opens sockets and ports to establish a TCP connection and initializes the worker pool to handle client requests.

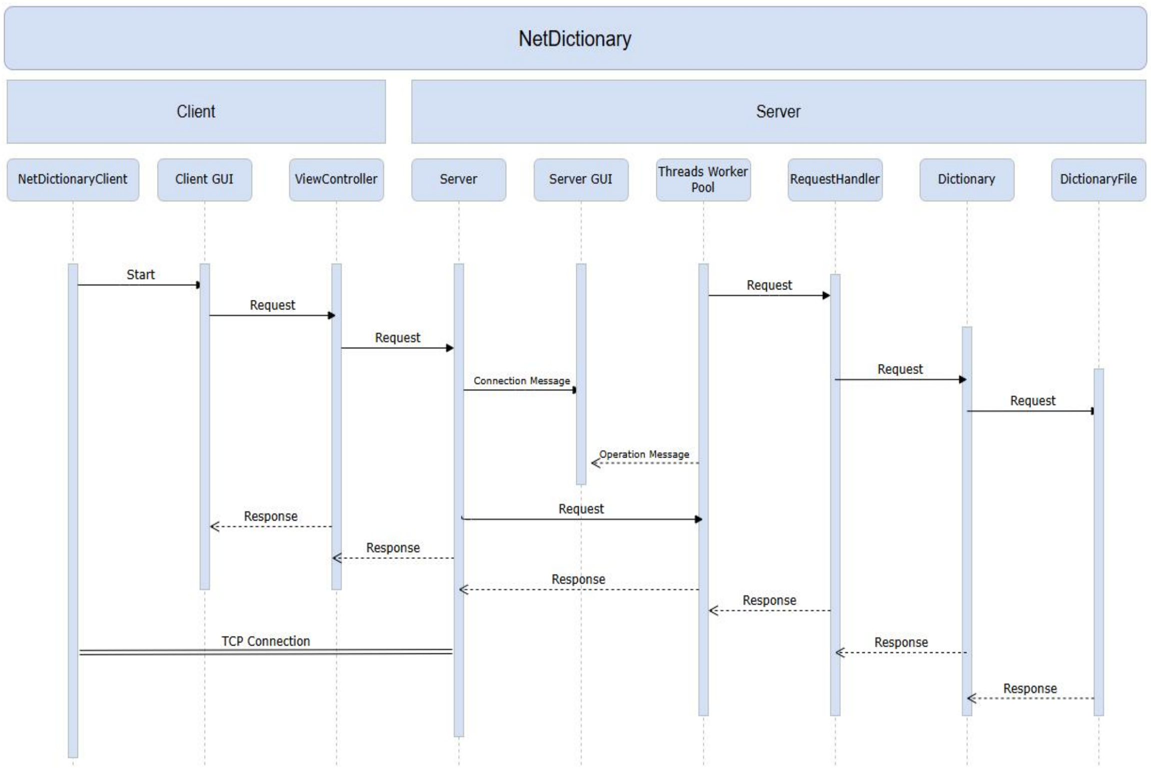
ServerGUI.java

This class initializes instances of DictionaryFile and Dictionary, and launches the server's GUI



```
Server started on port 1234
Client #1 connected: /127.0.0.1:2371
```

Interaction Diagram



Excellence

Data exchange protocol

At the beginning, NetDictionary used plain strings as the data exchange protocol. As a result, different dictionary operations required transferring multiple individual strings, such as the request type, word, and meanings. Both the server and client had to send and receive multiple strings through the input and output streams for each operation. Currently, the system uses the Message and Response classes as the standardized data exchange protocol, with data transferred in JSON format between the client and server. This approach unifies and standardizes the communication. Each dictionary operation now only needs to send a single Message object to the server, and the client receives a corresponding Response object. This change improves the readability and maintainability of the code, making the operation methods easier to understand and modify.

Exception and error handling

The NetDictionary system handles exceptions and errors related to network issues, I/O operations, dictionary file access, and dictionary operations. This enhances the reliability of the system across the server, client, dictionary, and dictionary file components.

On the server side, the system provides clear prompts to guide users in starting and running the service properly, including how to input various parameters. If the provided port number has format issues, the system will automatically fall back to the default port 1230 for socket connections. It notifies the user of the port used to start the socket and whether the connection was successfully established.

On the client side, the system also provides guidance on how to launch the client and prompts users to input the server address and port. If the parameters for the TCP connection are incorrect, the system displays appropriate prompts to help users input valid values. For each dictionary operation request, the console provides feedback messages indicating whether the operation was successful or failed.

Within the dictionary component, each operation returns a response to inform the client of the result. For example, when searching for a word, the system informs the user whether the word was found. Similar messages are returned for add and remove operations. When adding additional meanings to a word, the user is notified whether the new meanings were successfully added or if they already exist. The same applies to update operations.

Regarding the dictionary file, if the original local dictionary file does not exist, the system notifies the user and uses a temporary file instead. In case of a failure during file saving, the system also provides the path where it attempted to save the file.

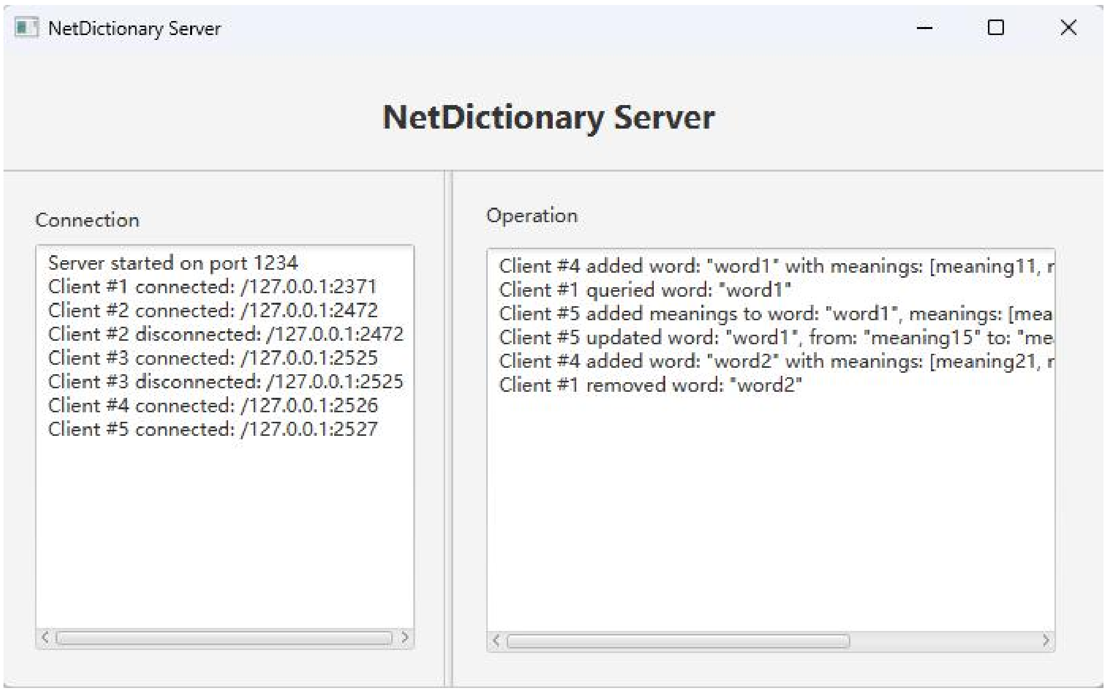
Concurrency

To ensure that the server can handle concurrent client requests reliably, NetDictionary uses locking mechanisms to protect shared resources such as the dictionary and the dictionary file. The dictionary and file operations are synchronized to prevent race conditions, ensuring that only one thread can access or modify these resources at a time. This synchronization guarantees data consistency and thread-safe execution during concurrent operations.

Creativity

Server GUI

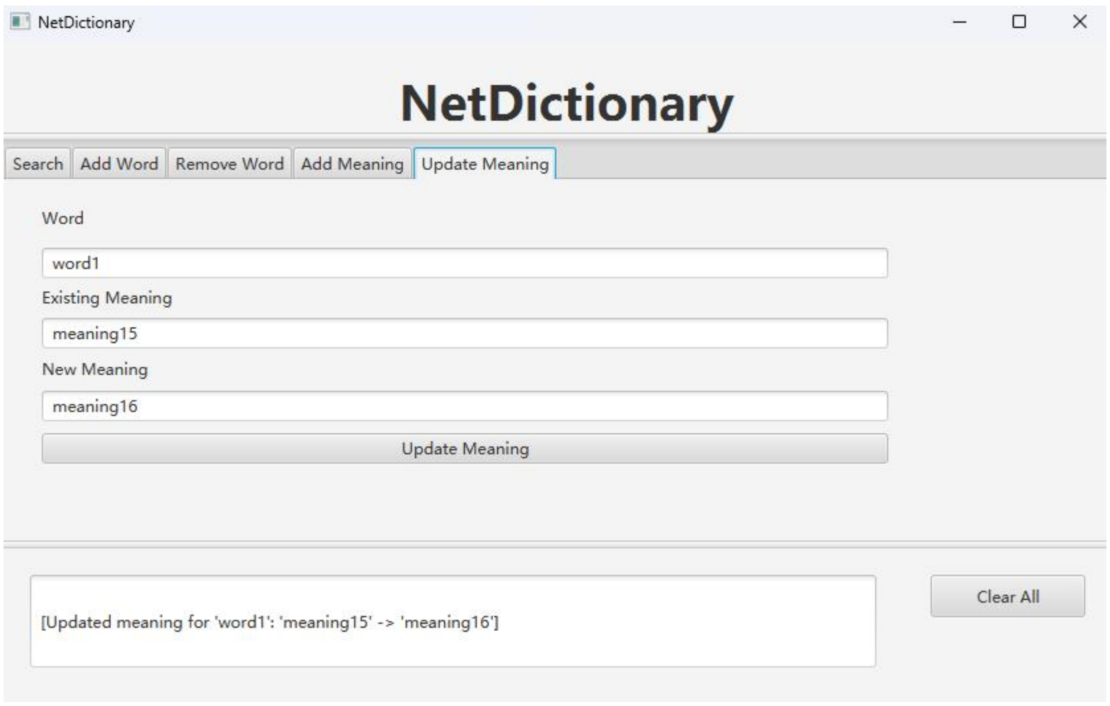
In the Server GUI, the system displays the port on which the server is running. As multiple clients operate concurrently in the Client GUI, the Server GUI presents information about their connections and operations. It shows connection messages for multiple clients, including when they connect and disconnect. Additionally, it displays which client is performing operations on the dictionary.



```
Server started on port 1234
Client #1 connected: /127.0.0.1:2371
Client #2 connected: /127.0.0.1:2472
Client #2 disconnected: /127.0.0.1:2472
Client #3 connected: /127.0.0.1:2525
Client #3 disconnected: /127.0.0.1:2525
Client #4 connected: /127.0.0.1:2526
Client #5 connected: /127.0.0.1:2527
```

```
Client #4 added word: "word1" with meanings: [meaning11, r
Client #1 queried word: "word1"
Client #5 added meanings to word: "word1", meanings: [mea
Client #5 updated word: "word1", from: "meaning15" to: "me
Client #4 added word: "word2" with meanings: [meaning21, r
Client #1 removed word: "word2"
```

Full Name: Ninghao Zhu(Bruce) Student ID: 1446180 NetDictionary

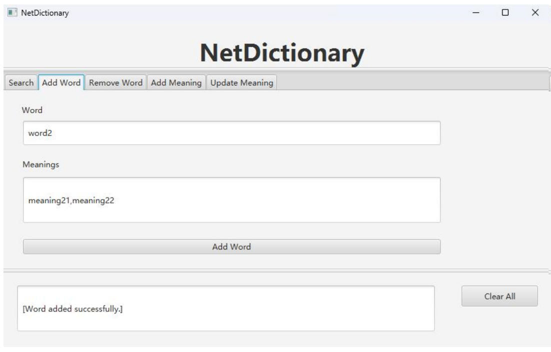


Word: word1

Existing Meaning: meaning15

New Meaning: meaning16

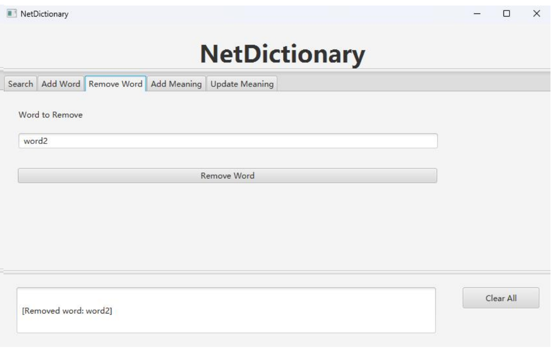
[Updated meaning for 'word1': 'meaning15' -> 'meaning16']



Word: word2

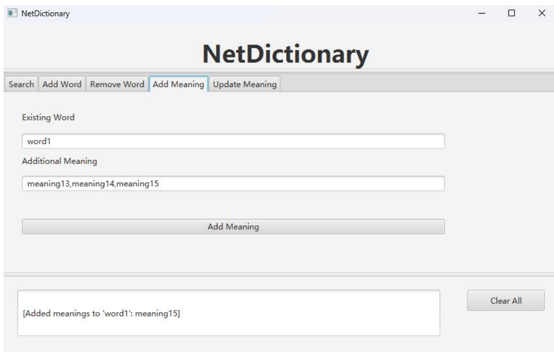
Meanings: meaning21,meaning22

[Word added successfully.]



Word to Remove: word2

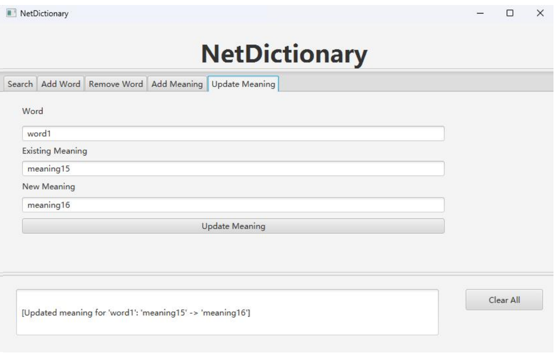
[Removed word: word2]



Existing Word: word1

Additional Meaning: meaning13,meaning14,meaning15

[Added meanings to 'word1': meaning15]



Word: word1

Existing Meaning: meaning15

New Meaning: meaning16

[Updated meaning for 'word1': 'meaning15' -> 'meaning16']

Worker Pool Architecture

The Worker Pool Architecture in NetDictionary improves performance by reusing a fixed number of threads, reducing the overhead of thread creation and destruction. It enhances resource management and scalability by efficiently handling concurrent requests through a synchronized queue. However, it has limitations, such as the fixed thread pool size, which may cause delays if the load exceeds the pool's capacity, and idle threads when demand is low. Additionally, managing the pool size for optimal performance can be complex.

Conclusion

In conclusion, the NetDictionary system demonstrates an effective implementation of a client-server architecture designed to handle concurrent dictionary operations through a multi-threaded server and a worker pool. By utilizing standardized message protocols and JSON data formats for communication, the system ensures efficient and consistent data exchange between the client and server. The worker pool architecture optimizes resource usage and enhances performance by reusing threads for concurrent requests. Although there are some limitations, such as managing thread pool size and potential idle threads, these challenges can be addressed through careful configuration. Overall, NetDictionary provides a reliable, scalable, and maintainable solution for handling multiple dictionary operations in real-time.