Multi-threaded Dictionary Server

Problem Context

The goal of this project is to develop a multi-threaded dictionary application that allows multiple users to access the dictionary server concurrently. The project aims to showcase the effective utilisation of two fundamental technologies in distributed systems: threads and sockets.
Threads are employed as lightweight processes to handle client requests on the server side. As the application needs to process multiple users' requests concurrently, using threads enables the server to delegate tasks to different threads, allowing for concurrent execution of the logic without delays or interruptions.

Sockets, on the other hand, are used to establish communication channels between the client and the server programs. In the dictionary application, both the server and clients rely on sockets to create and maintain appropriate communication channels for seamless interaction.

Components of the System

The multi-threaded dictionary application consists of several key components that work in synergy to ensure smooth operation and a seamless user experience.

Client

The Client class is purposefully designed to enable users to interact with the server and access the dictionary service. This class facilitates users in sending search, add, delete, and update requests to the server, streamlining their experience with the application.

Server

The Server class is responsible for actively listening to incoming connection requests and assigning them to threads using a work pool design. The main logic is not performed in this class, as the Handler class efficiently handles the processing of requests.

Handler

The Handler class is a lightweight thread class that manages the server's logic. Each Handler instance can process a single user's request and communicates with the Dictionary class to perform the actual actions for searching, adding, deleting, and updating entries in the local dictionary.

Dictionary

The Dictionary class is meticulously developed to handle the main operations of the dictionary application. It performs actions such as searching, adding, deleting, and updating dictionary entries, ensuring that the data is properly managed and maintained.

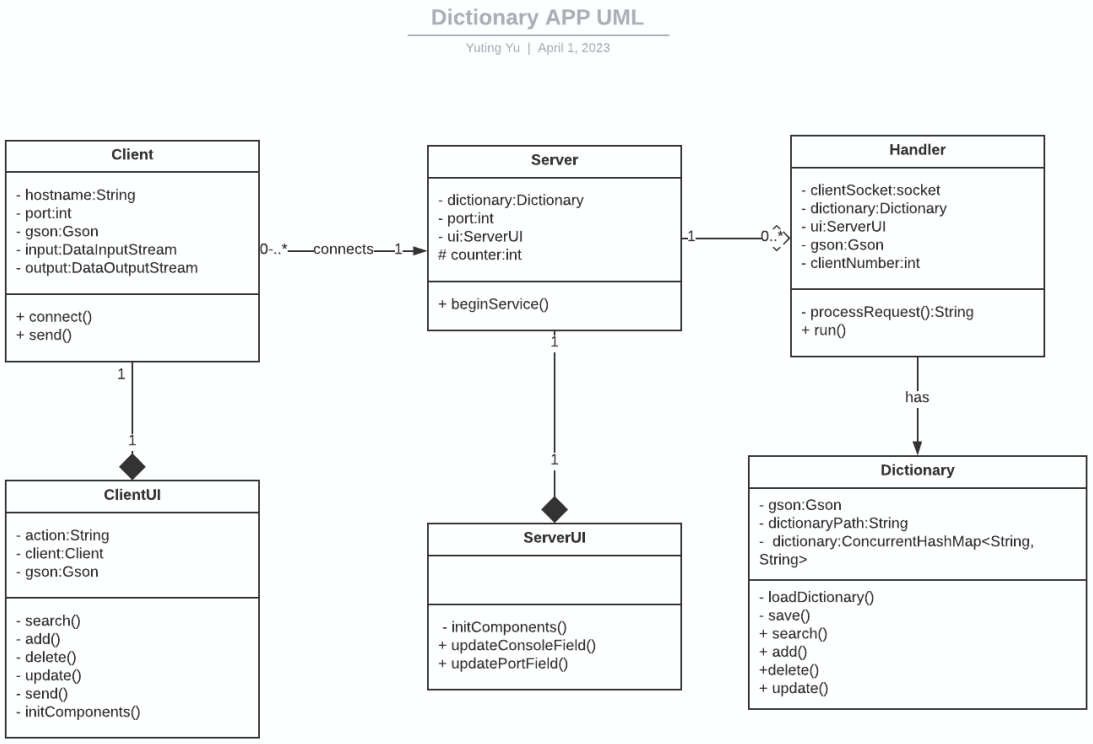
ClientUI

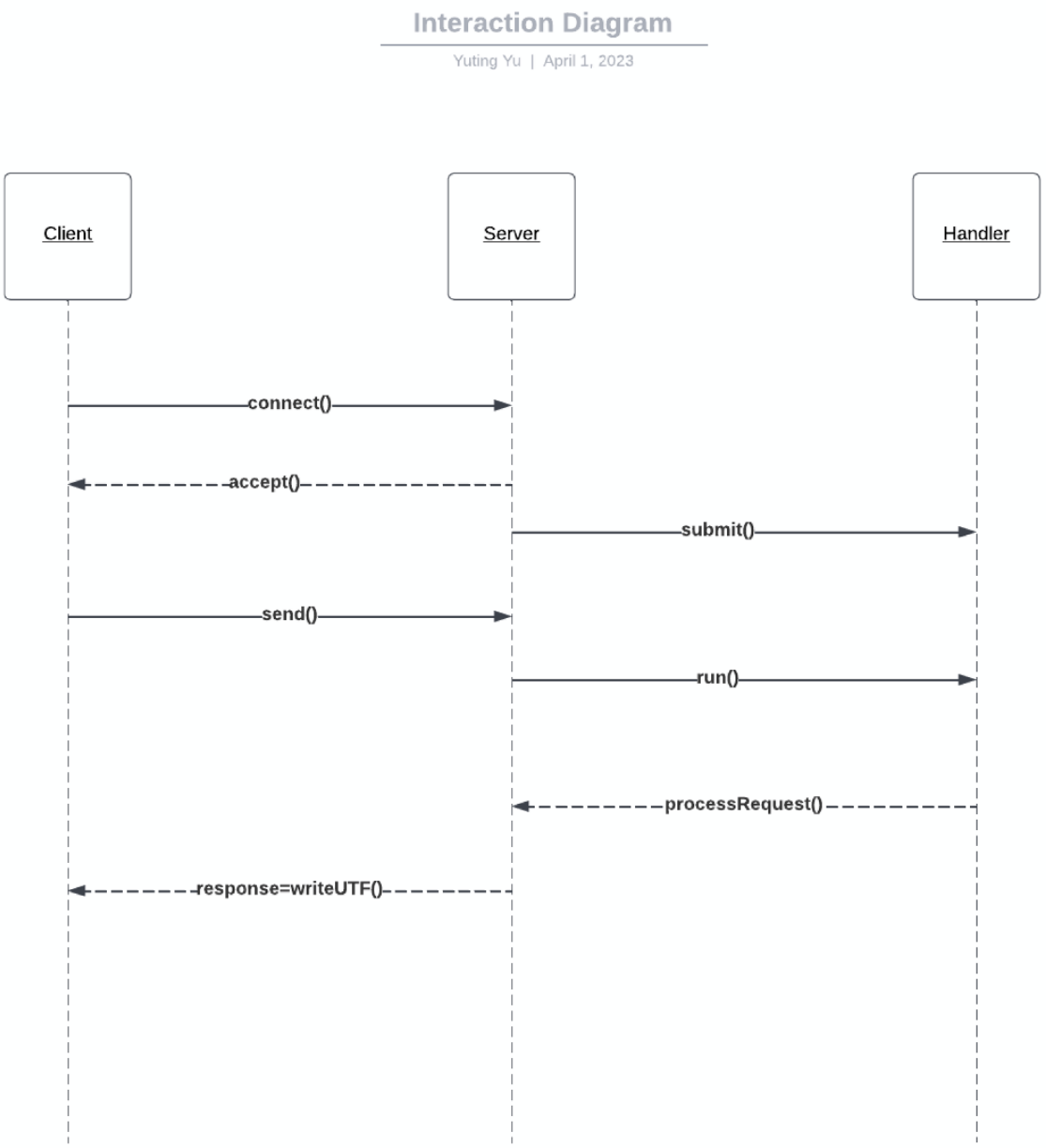
The ClientUI class, implemented using the Swing framework, provides a graphical user interface for user input. The underlying logic is identical to the Client class, but the user-friendly interface streamlines user interaction with the application.

ServerUI

The ServerUI class is used to furnish a graphical user interface for the server application, enabling administrators to actively monitor server console logs. It tracks user connection and disconnection operations and displays the current port on which the server is running. This UI makes it convenient for administrators to manage the server and ensure its smooth operation, as it simplifies tracking the application status.

Class Diagram and Interaction Diagram





Analysis

Architecture

The system employs a cached thread pool architecture, which is particularly well-suited for short-lived tasks such as those required by the dictionary application. Cached thread pools dynamically create new threads as needed to handle incoming requests, and automatically terminate threads that have been idle for sixty seconds. This approach significantly reduces resource consumption on the server side, as unused threads are efficiently managed and removed.

By leveraging the capabilities of cached thread pools, the architecture is optimised for handling a large number of simultaneous connections without sacrificing performance or responsiveness. This ensures that the dictionary application remains efficient and scalable, even as the number of concurrent users increases.

Concurrency

As outlined in the architecture section, the system employs a cached thread pool to dynamically allocate threads for handling user requests. This design choice enables concurrent processing of multiple users' requests simultaneously, ensuring that the dictionary remains up-to-date and maintains the correct state.

Interaction

Communication Protocol

The communication protocol employed in this project is TCP (Transmission Control Protocol). TCP was chosen over UDP (User Datagram Protocol) due to its inherent reliability and stateful nature. In the context of a dictionary application, it is crucial for both clients and servers to ensure that messages are delivered in order and securely. Loss of messages or misordering could result in unpredictable behaviour in both client and server applications. While UDP can also implement message ordering and mitigate message loss with some additional implementation, its strengths are not fully utilised in the context of the dictionary application. This is because the application does not require continuous packet delivery, as would be necessary for a video streaming app.

Message Exchange Protocol

The message exchange protocol utilised in this project is JSON (JavaScript Object Notation). JSON was chosen over other types of protocols due to its widespread adoption in the web industry. By employing JSON, the server application can communicate seamlessly with any modern frontend framework, regardless of the device or operating system.

For instance, if the server application needs to connect with an iOS device developed using Swift, no additional work would be required to facilitate message exchange between the Java server and the iOS client application. In addition to its extensibility, JSON is a human-readable format that can be easily understood by both technical and non-technical individuals, making it an ideal choice for this project.

In summary, the use of TCP as the communication protocol and JSON as the message exchange protocol ensures a reliable, secure, and flexible foundation for the multi-threaded dictionary application.

Failure model

In the context of the multi-threaded dictionary application, the failure model is primarily concerned with addressing potential issues that may arise from user input errors and socket errors. These issues are managed by implementing error-handling mechanisms in the application, specifically using try-catch blocks to catch and handle exceptions effectively. For user input errors, the system incorporates try-catch blocks to capture any IO exceptions that may occur during the input process. This helps ensure that any invalid or unexpected user input is identified and addressed appropriately, preventing the application from entering an unstable state or crashing.

In the case of socket errors, the application employs try-catch blocks to handle SocketException instances. By capturing and managing these exceptions, the system can gracefully handle communication issues, such as connection loss or unresponsive servers, without affecting the overall stability of the application. This approach allows for the detection and resolution of connectivity issues, ensuring that the system remains robust and functional despite potential communication challenges.

By incorporating these error-handling mechanisms into the failure model, the multi-threaded dictionary application can maintain a high level of reliability, security, and performance, even in the face of user input errors or socket-related issues. This robustness further solidifies the system's ability to handle a large number of concurrent users and deliver an excellent user experience.

Creativity

Server-side Graphical User Interface

The server-side graphical user interface (GUI) represents a creative approach to monitoring the server application. This GUI allows administrators to easily track server logs and gain a clearer understanding of the server application's current status. The interface also conveniently displays the current port number, which is particularly useful when the server application runs for extended periods. By incorporating a server-side GUI, the system enhances the manageability and user experience for administrators, making it simpler to oversee the server application's performance and health.

Conclusion

In conclusion, the multi-threaded dictionary application has been carefully designed and developed to provide a robust, scalable, and user-friendly solution for dictionary services. By utilising key technologies such as threads, sockets, and a cached thread pool architecture, the system achieves efficient concurrent processing of user requests and maintains an up-to-date and accurate dictionary state. The choice of TCP as the communication protocol and JSON as the message exchange protocol ensures reliability, security, and flexibility in communication between clients and the server.

The application's components, such as Client, Server, Handler, Dictionary, ClientUI, and ServerUI, work cohesively to provide a seamless user experience, while also simplifying the management process for administrators. Creative solutions like the server-side GUI further improve manageability and monitoring capabilities, allowing administrators to oversee the server's performance and health with ease.

Overall, the multi-threaded dictionary application demonstrates a successful implementation of distributed systems concepts, effectively addressing the challenges of concurrency, reliability, and usability. The system's design and implementation can serve as a strong foundation for future enhancements and adaptations, ensuring the application's longevity and continued success in providing exceptional dictionary services.