COMP90015 – Assignment 1

James Barnes, 820946 – 2019, Semester 1

Problem Context

The problem posed for this assignment was to design and implement a dictionary, based upon a distributed system. The distributed system must be multi-threaded (at least on the server side, though I have implemented this behaviour for the client also) and rely on threads as the lowest form of abstraction for communication. Various factors must be handled, such as concurrency and failure tolerance. Clients must be able to add, delete, or get definitions of words that they or other users may have entered, displaying information indicating, for example, if a word has multiple definitions in the dictionary, or if the word to delete was not in the dictionary.

Additional features are detailed at the end of this document.

System Components

DictionaryServer

The DictionaryServer class is the main class of the server component of this system. This class handles all requests to connect to the server and delegates the connection to a new RqequestHandler in it's WorkerThreadPool.

Dictionary

This class's function is to maintain a JSONObject that is used by the server as a dictionary. This JSONObject has the following informal shema:

```
{
  "type": "object"
  "<word>": {
    "type": "array",
    "description":
      "Array of <word>'s definitions. <word> can be any string"
    "items": {
      "type": "object"
      "definition": {
        "type": "string"
      },
      "author": {
        "type": "string"
      }
    },
  }
}
```

RequestHandler

This class handles all incoming and outgoing communication with its given socket. The socket also updates or queries a Dictionary based on the client's requests.

ShutdownThread

This Thread class handle the shutdown of the server gracefully. On shutdown (Ctrl-C or otherwise) this thread saves the current state of the Dictionary to the same file it was read from.

WorkerThreadPool

This class implements a simple worker pool. The class spawns a limited number of WorkerThreads, which are then tasked with running tasks added to a queue of potentially unbounded size.

WorkerThread

This class implements a simple worker thread. This class's only purposed is to fetch tasks from its WorkerThreadPool's queue and run them. Once a task is finished, it awaits its next task from the queue.

DictionaryClient

This class acts as the primary class for a client. This class manages the state of the client, as well as handling the UI and creating a RequestThread for when the user wants to communicate with the server.

ClientUI

The purpose of this component is to render the user interface of the client program, as well as send the user's input to and receives input from the connected server from a RequestThread.

RequestThread

This component connects to a server, as defined by its constructor's parameters, sends a message to the server, then receives a response or an error and updates the UI accordingly.

JSONConsts

This class simply contains various constants used by all classes that communicate in my implementation. This class simply acts as a standardised reference for these field names and their possible values.

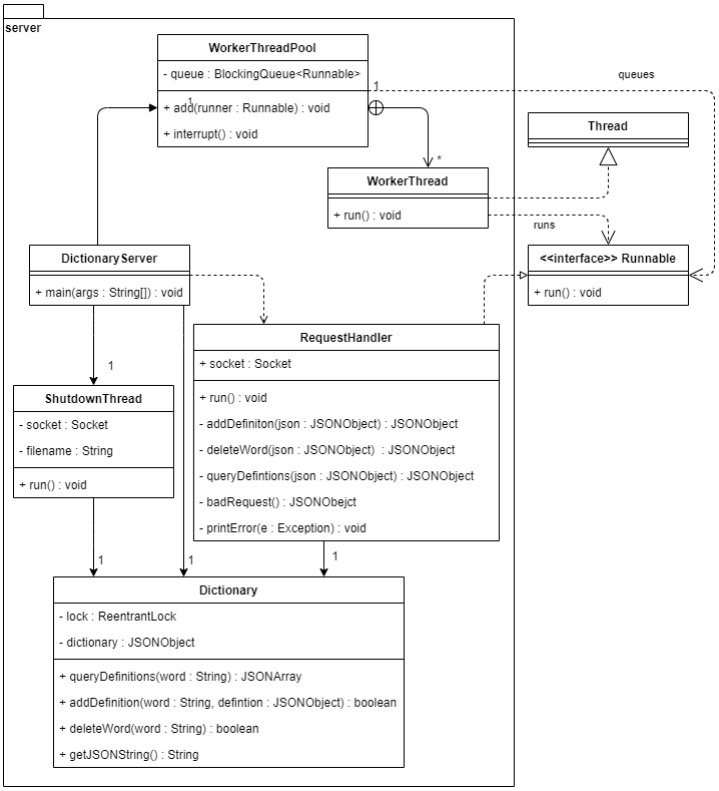
System Diagrams

Class Design Diagram

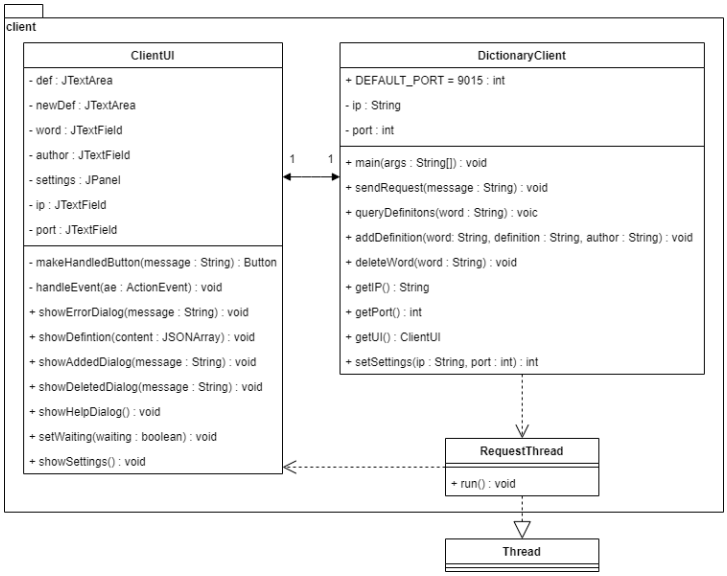
For sake of clarity, the JSONConsts class is missing from the diagram, as it is used by many classes and makes the diagram needlessly cluttered.

Various final static properties have also been omitted, as they add little useful information to the diagram.

The server components fall into four main categories. The server (DictionaryServer, ShutdownThread), the dictionary (Dictionary), the request handlers (RequestHandler) and the thread pool (WorkerThreadPool, WorkerThread). These component work together to form a cohesive unit which servers as the server of the distributed dictionary.



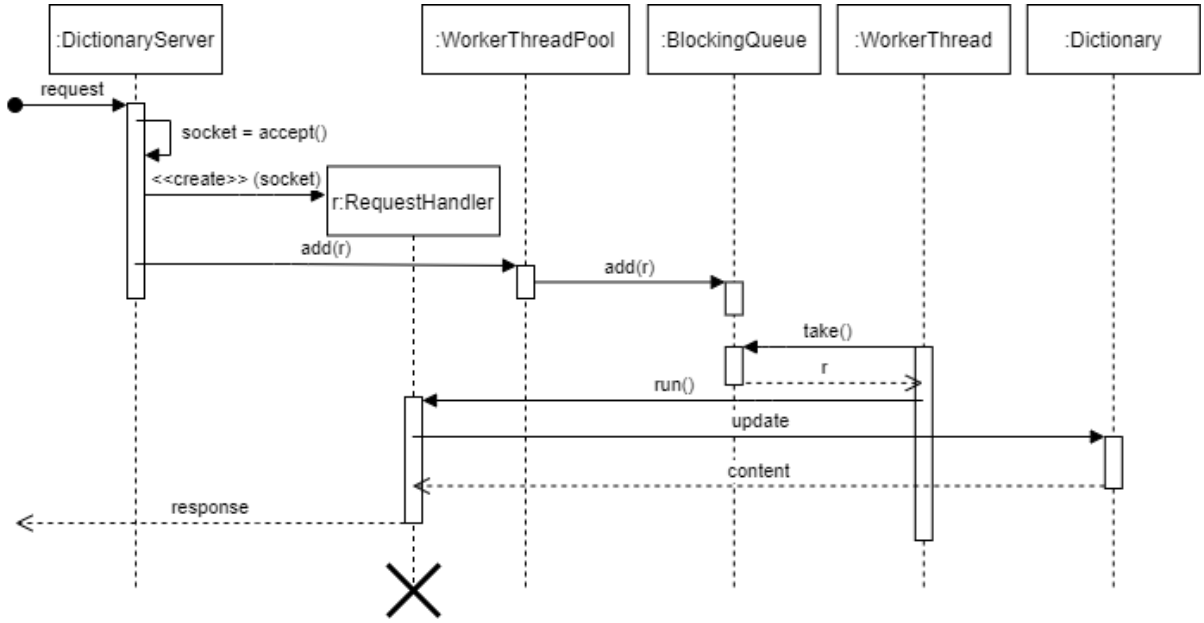
The server components handle the overall state of the server, as well as accepting new requests and the shutdown behaviour. The dictionary maintains the state of the dictionary itself, while the request handlers handle communication with client requests and communicate with the dictionary to update accordingly. The thread pool is responsible for managing and running the request handlers.



The client component is composed of three parts, the DictionaryClient (Client), the ClientUI (UI), and the RequestThread. The UI component handles all input from the user, and displays any request's responses accordingly, while the client handles creating the request threads, as well as managing the client's state (settings, etc.). The request thread's purpose is to manage an outgoing message to a server and tells the client's UI how to update with either a response from the server or an error.

Interaction Diagram

This sequence diagram demonstrates the behaviour of the server when a request from a client comes in. The client component of this diagram has been abstracted away as simply a request, as the client side is not the important part of this interaction.



Firstly, upon receiving a request from a client the server creates a new RequestHandler, `r`, to handle the request from the client, using the socket created from the server socket's `accept` call. `r` is then added to the server's WorkerThreadPool's BlockingQueue before it is taken by an idle WorkerThread of the WorkerThreadPool. The thread then runs the RequestHandler, with this communicating with the server's Dictionary to update or query the state of the Dictionary, before sending the response back to the Client.

The details of the interaction are left vague, as "update" and "content" as this varies greatly depending on the request sent by a client.

Critical Analysis

I have chosen to use a thread-pool architecture, with a custom WorkerThreadPool detailed below in the Additional Features section.

Advantages

I believe that users will not frequently be sending many requests to the server. This means that, if I Were to utilise a thread-per-client approach, there would be a lot of idle time in a thread's lifetime. This is a waste of the server's resources, as a thread could be performing something else instead of waiting idly for a request from the client.

Alternatively, one could use a thread-per-request architecture. The main issue with architecture is that threads will need to be spawned and destroyed each time a request is made. This will utilise a decent amount of CPU time of the server, as well as potentially eating up the server's memory, as many concurrent requests will cause the server to spawn many threads.

The thread-pool architecture alleviates both of these issues. It has the benefit that idle threads can only exist when there are no unserviced requests for the server, as well as amortising the costs of spawning and destroyed threads as they are all spawned and destroyed once.

Disadvantages

A disadvantage of the thread-pool architecture is that the pool may become swamped with requests if too many arrive and backup the queue, while workers are busy servicing other requests. This can cause a feedback loop and have huge memory usage. I have attempted to alleviate this by minimising the processing time required by each thread to service requests and allowing a client to send a single request at a time. Alternatively, the server could be tuned to allow the thread-pool to use more threads if this bottleneck is an issue.

Further, there is no security or access restriction (via the use of a username-password system) for the server, leaving the server open and vulnerable to potential predatory clients or malicious agents. This means that anyone who knows the server's IP and port could potentially cause all kinds of issues, such as performing a DDoS attack.

Additional Features

Custom Worker Thread Pool

I decided to implement my own threaded worker pool for this project. The pool uses a defined amount of threads which then wait for tasks to be queued in a BlockingQueue. This queue can be added to via the add method in the WorkerThreadPool class and is potentially unbounded is size.

To use a pool, invoke the pool's `add` method with a Runnable instance. When a thread becomes available Runnable is taken from the queue and the thread runs it, before returning to wait to take another Runnable from the queue.

Further Dictionary Information

The dictionary stores more information than just a list of a word's definitions, but also maintains an author field. This field can be entered when adding a definition and can be seen when the definitions of a word is fetched.

Further, the system allows users to add definitions to words that already exist by maintaining arrays of definitions per word. This allows for better cooperation to maintain a more complete, total, dictionary among users. The user is still notified that the word is a duplicate if the word already exists in the dictionary. Users are also restricted by allowing entry of a single definition at a time, however can still enter multiple if sent through multiple requests.

Dictionary Saves to File on Server Shutdown

When the server shuts down, the server ends itself by attempting to save the dictionary to file, allowing for any client updates to the dictionary that may have occurred to persist over multiple runs of the server. This file is the same as the dictionary was read from and is passed as the second command line argument when booting up the server.

Client Settings

There is an additional button on the UI called "Settings". This button opens a dialog box allowing the user to change settings of the client. These settings are the IP address and port of the server the client attempts to connect to. These settings are cleaned and then used in the future requests to the server.