Project design - Lander Brandt

The project is designed so that the `ThreadManager` class is really the core of the application. The main class looks at the `ThreadManager` to see if it should exit between handling requests, and the `ThreadManager` will also interrupt the main thread if a kill command has been sent to the server, this way if the main thread is waiting for incoming connections then it will be interrupted and can perform exit logic.

Main

The main class instantiates the `ServerSocket`, `ThreadPool`, `SharedQueue`, and `ThreadManager`, then waits for incoming connection. Each incoming connection is passed to a new `ConnectionHandler` thread, then continues the event loop. The event loop condition simply checks to see if the `ThreadManager` has its kill flag set to `true`, and if so then it performs cleanup (stops the `ThreadPool`, joins all `Worker` threads in the `ThreadPool`, and closes the `ServerSocket`)

ConnectionHandler

The `ConnectionHandler` handles all incoming connections on a separate thread. When a new connection comes in, the command is read, a `Job` is created, and an attempt is made to add it to the `SharedQueue`. If the `SharedQueue` is full, the add will fail and the `ConnectionHandler` will return a busy message then terminate the connection. If the add succeeds then the `ConnectionHandler` exits.

ThreadPool

The `ThreadPool` has a `capacity` and a number of active `Worker` threads. It has one main method for shrinking/growing the number of `Worker` threads, called `setNumActiveWorkers()` which takes a `count`. This method determines whether the number of threads should shrink or grow. If `count` exceeds `capacity`, then `capacity` is used instead.

Growing

When growing the number of `Worker` threads, free indices (indicated by `null`) are replaced with a new `Worker` given the name `Worker + index`, and started immediately.

Shrinking

When shrinking, the method tries to kill the `Worker` count delta (we'll call this `delta`) number of threads in the `WAITING` state. These threads are then replaced with `null` in the `Worker` array to indicate free slots and the `activeWorkerAcount` is also decremented by 1. If there are less `Worker`s waiting than `delta`, then the remaining number of `Worker`s to remove are simply `kill()`'d starting from the beginning of the `Worker` array.

Worker

The Worker threads have a main loop where they call the `take()` method on the `SharedQueue`. The `take()` method will make the calling thread `wait()` until an item is added to the `SharedQueue`, then return the item at the front of the queue.

Once a `Job` is removed from the `SharedQueue`, the `Worker` calls the `run()` method on the `Job` which runs synchronously.

The loop continues until the `Worker` is notified that it should be killed.

Job

The `Job` class implements the `Runnable` interface. In the `run()` method the command which received from the client is parsed. If the command is `KILL`, then the static `killServer` flag on the `ThreadManager` is set to `true`, and the `Job` notifies the client that the server is being killed.

If any other command is received then it is executed, and the result is sent back to the client.

At the end of the `run()` method the socket the client is connected to is closed.

SharedQueue

This `SharedQueue` implementation is one which was created for a previous assignment. It is fully thread-safe for add/remove/size operations and does not lock the entire structure when an add/remove is performed.