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.

ThreadManager

The `ThreadManager` is the most important piece of this project. It ties together nearly everything. The thread manager has `POLL_FREQUENCY` variable which is how long it will sleep in its event loop. The event loop condition is whether the server `KILL` command has come in, or if the `ThreadManager` itself has been notified by the main thread to stop (`kill()` has been called).

In the event loop the number of jobs is obtained from the queue. If the number of jobs has grown since the last iteration of the loop *and* any of the following conditions are met, then the number of `Worker` threads will double. Conditions:

- Number of jobs is > `Threshold1` and previous threshold was not `1`, workers was not 1.
- Number of jobs is > `Threshold2` and number of jobs has changed since previous iteration (growth).

If the number of jobs since the last iteration has decreased then the jobs are halved if any of the conditions are met:

- Number of jobs is > `Threshold1` and previous threshold was not `1`

In any case if the number of jobs is `< Threshold1`, then the number of workers are set to `minimumWorkers`.

Once the loop is exited, the manager notifies the `ThreadPool` to `stop()`, which tells all `Worker` threads to stop, then the main thread is interrupted and the manager exits.

Challenges

1. My `SharedQueue` turned out to have a race condition somewhere when either adding or removing items, or both, would cause the size of the queue to be off by 1. I *think* the condition was that if one thread was removing an item while another thread was adding, then a node would be skipped in the removal process. I cleaned up that code and moved the synchronized block for incrementing/decrementing and the issue seemed to go away. Still not sure exactly how it was happening since my sequence of locking was correct and no one thread could update the size while any other thread was viewing or modifying the size...
2. The `KILL` message didn't work at one point because of refactoring the jobs to only accept one command then close the connection. The code was previously set to have the jobs kill the socket only if the `KILL` command was received, but that changed once it was understood that only one command would ever be read from the client.