Design Notes
Assignment 3

Majid Ghaderi

Disclaimer

These notes are based on my own implementation. I do not claim that my implementation is the simplest or the best. Feel free to use or disregard any of these suggestions as you wish.

Guidelines

- Read the assignment description carefully. It contains a lot of useful information.
- Do not jump to coding. Spend some time to come up with a design for your program.

Testing the Web Server Multi-Threading

Start the server by simply running the ServerDriver class supplied with this assignment. To stop the server, type "quit" at the command prompt. Once the server is running, use Telnet/putty to open a connection to the server. While this telnet connection is open, send a request to the server, *e.g.*, using a web browser, to test for multi-threading functionality.

Non-Responsive Clients

To detect non-responsive clients, I used the method `Socket.setSoTimeout()` to set a timeout value for the client socket in my worker thread before attempting to read from the socket. If the socket timeout expires while the worker thread is still blocked on reading from the socket, a `SocketTimeoutException` will be thrown, which indicates non-responsive clients. In the catch block for this exception, I create the HTTP response with status code `408`, send it to the client, and then clean up and terminate the worker thread.

Terminating Web Server and `shutdown()` Method

While there are several techniques for terminating a thread, a simple approach is to define a flag variable in your WebServer class (*e.g.*, `boolean shutdown`). While the flag variable is true, the server is listening for incoming connections. You then set the flag to `false` from within the `shutdown()` method, to break the loop and terminate the main thread.

Since `ServerSocket.accept()` is a blocking call, you need to force the server thread to periodically time-out to return from the blocking method `accept()`, and check the status of the flag. The following pseudo-code shows you how I implemented this in my code. The method `ServerSocket.setSoTimeout()` takes a parameter to specify the timeout interval. See Java documentation for details. A timeout value of 100 milli-seconds is a reasonable choice.

```
public class WebServer extends Thread {
  private boolean shutdown = false;

  public void run(){
    - open the server socket
    - set socket timeout option using ServerSocket.setSoTimeout(100)

        while (!shutdown) {
            try {
                - accept incomming connection request
                - create a worker thread to handle the accepted connection
            } catch (SocketTimeoutException e) {
                // do nothing, this is OK
                // allows the process to check the shutdown flag
                // if not shutdown, it goes to listening mode again
            }

        }// while

    //
    // a good implementation will wait for all running worker threads to terminate
    // before terminating the server. You can keep track of your worker threads
    // using a list and then call join() on each of them. A better approach is
    // to use Java ExecutorService to schedule workers using a
    // FixedThreadPool executor.
    //

    - clean up (e.g., close the socket)
  }

  public void shutdown(){
        shutdown = true;
  }
}
```

Thread Management

It is possible to simply use a list to keep track of active worker threads and then manage them as needed. In my implementation, however, I used the Java Executor service to create a fixed thread pool:

```
ExecutorService executor = Executors.newFixedThreadPool(POOL_SIZE);
```

where, `POOL-SIZE` specifies how many threads are allowed to run in parallel. The best way to set this number is to find out how many CPU cores the server machine has, but for simplicity you can just set it to a reasonable number, say 8. Then, to schedule a new worker thread, simply call `executor.execute()` and pass a worker object as argument. Note that you need to define your worker class to implement the Runnable interface. Finally, to wait for the running workers to terminate before terminating the server thread:

```
    // shutdown the executor
    try {
        // do not accept any new tasks
        executor.shutdown();

        // wait 5 seconds for existing tasks to terminate
        if (!executor.awaitTermination(5, TimeUnit.SECONDS)) {
            executor.shutdownNow(); // cancel currently executing tasks
        }
    } catch (InterruptedException e) {
        // cancel currently executing tasks
        executor.shutdownNow();
    }
```

Also, notice that the `ServerDriver` class calls `System.exit(0)` once the server has shutdown to kill any lingering threads that may not have been stopped properly.

Server Port Number

The server port number should be greater than 1024 and less than 65536. Most port numbers less than 1024 are reserved for well-known applications.

Sending HTTP Response

I simply used the underlying socket output stream to send the headers and the object body. Format a String object using `String.format()` that holds the entire header part of the message. Then use `String.getBytes("US-ASCII")` to convert it to a byte array, which can be directly written to the socket output stream. To read the object from the local file, I used `FileInputStream` which is a low level byte stream for reading both text and binary data in byte format. The method `FileInputStream.read(byte[])` can be used to read a byte array from the file.