Design Notes
Assignment 2

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

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, you can test it using Telnet, a web browser, or your Assignment 1 code. With Telnet, connect to the host running your web server at the specified port number. Alternatively, you can test your web server using a browser by sending the port number in the URL. For example, if the server is running on your local machine on port 2225, you can enter the URL `http://localhost:2225/obj.html` to retrieve the html file `obj.html`. Yet another approach is to modify the Tester program from Assignment 1 to programmatically send HTTP requests to the server using the `getObject()` method in your `QuickUrl` class.

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 1000 milli-seconds is a reasonable choice.

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

  public void run(){
    - open the server socket
    - set socket timeout option using ServerSocket.setSoTimeout(1000)

        while (!shutdown) {
            try {
                - accept new connection
        - create a worker thread to handle the new 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;
  }
}
```

In my implementation, 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:

```
    // shutduwn 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 class method 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 in Web Serve Mode

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.

Processign HTTP Request in Proxy Mode

I reconstruct the client request and add Connection: close if not already included in the client's request and send the request to the remote server. In my implementation, header lines are kept in a HashMap as (key, value) pairs, where key is the header field and value is its string value in the corresponding header line. So, it is easy to check if key `Connection` already exists in the request header or not. To send the request to the remote server, I open a new socket. As soon as the request is submitted, I start reading from the socket (while not seeing the end of file) and write to the client socket. I use a byte array of size 32 packets (each packet is 1024 bytes) to read from remote server and write to the client. The method `InputStream.read(byte[])` returns the number of bytes that were actually read. I use this number when writing to the client socket using `OutputStream.write(b, 0, actualBytesRead).`

Utils Methods

Several useful methods are defined in class `Utils`:

- `isLocalHost`: This method checks if a given host refers to the local host. It will be actually educational for you to look at how I implemented this method, which enumerates all network interfaces on the local machine when the class `Utils` is loaded.
- `getCurrentDate`: To format the current date/time as a string consistent with the formatting requirement in HTTP headers.
- `getContentType`: To query the content type of a file for inclusion in the web server's response.
- `getLastModified`: This method obtains the last date/time the file was modified and converts it to an HTTP compliant string format, which can be directly included in the web server's response.