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.

Creating a Secure Socket

As discussed in class, the TCP/IP protocol stack does not provide data security for applications. Instead, such functionality must be implemented at the application layer. Specifically, the application layer protocols such as *Secure Socket Layer (SSL)* and *Transport Layer Security (TLS)* can be used to provide security for applications that use TCP for network communication. In Java, class `SSLSocket` is designed by extending the plain text `Socket` class and adding SSL/TLS functionality to the `Socket` class. The result is a convenient way for working with secure TCP connections using the same familiar `Socket` API.

While a `Socket` object can be directly created using `new Socket()`, to create an `SSLSocket` instance, you should use a socket factory, as shown below:

```
SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket)factory.createSocket(host, port);
```

The rest of the program is exactly the same as if you were working with a regular `Socket` object created as:

```
Socket socket = new Socket(host, port);
```

You can even create a regular socket using the factory approach (check out Java docs for the `SocketFactory` class), which can be utilized to consolidate your code for different socket types. In my implementation, I utilize polymorphism in Java to simplify my code as `SSLSocket` and `SSLSocketFactory` are subclasses of `Socket` and `SocketFactory`, respectively.

Reading Binary and Text form Socket

My suggestion is to read everything from the socket as a sequence of bytes using the low level input stream associated with the socket. That is, call the method `Socket.getInputStream()` to gain access to the byte input stream associate with the socket and then just use method `InputStream.read()`. It is very easy to convert an array of bytes to a string using one of class String's constructors. You can also write a method to read the header part of the response line-by-line. Just keep reading bytes from the input stream until you see the sequence `"\r\n"`. Once you have read the entire header, you can use input stream `read(byte[])` method to read from the socket chunk-by-chunk instead of byte-by-byte (for improved performance).

Even for writing text data to a socket, *e.g.*, HTTP headers, you can create a string object and then call `String.getBytes("US-ASCII")` to convert the string to a sequence of bytes that can be written to the low level socket stream, which can be obtained using `Socket.getOutputStream()`. Make sure to **flush** the output stream so that the data is actually written to the socket.

Parsing URL

The class String in Java is very powerful. Use method `String.split()` to breakdown the URL to its various components. You can split a string using different delimiters.