We chose to use a multi-threaded approach in our web server because we were already pretty familiar with threads and had seen them demonstrated in class. On our first meeting, we looked at all the server requirements then divvied up the parts of the server that we each were to implement. After that was done, we met up again to address code design and how we wanted to structure all the pieces. In our final meeting, we spent time debugging and prepping for a final meeting with Professor Sprenkle.

Some of the challenges that we faced included figuring out why our content wasn't being transmitted, returning response headers, especially with images, implementing a server timeout, and using telnet and organizing our code properly to allow for testing. We were initially unwittingly using two instances of an output stream, so our content wasn't being transmitted properly. Once we moved it over to one instance by refactoring some code, our server transmitted content as expected. The response header issue was eventually resolved once we realized that we weren't closing output stream after writing the response (inside the catch block of the try-catch statements). Additionally, we had to make a nested try-catch so that when exceptions were caught for the appropriate HTTP errors codes, we could print the response header to the output stream--this was not possible without a nested try-catch because the catch block did not have access to the output stream. We also had problems allowing for multiple requests in the same connection if the client was using HTTP/1.1. This problem also resulted in the response headers not being timely transmitted with images--there appeared to be a ~2minute delay between getting the content and getting the response headers when we testing from Firefox. Apparently we needed to flush the output stream for the responses and file contents to actually show up (using telnet). As far as the delay in response headers is concerned, we have concluded that it might be a Firefox thing, because the delay does not happen with Google Chrome.

In designing our server, we decided to give the ServerSocket a backlog of 15 connections. This number was fairly arbitrary, but we thought it was a good number for our small server that will not be handling very many connections. We do not reasonably expect there to be more than 15 connections to the web server at the same time. As far as the timeout is concerned, again we had to implement fairly arbitrary timeouts. We could not think of any “good” way to determine what the timeout should be, and upon discussion with Prof. Sprenkle, ended up deciding to go for a higher timeout (20 seconds) if the server has only active connection. This timeout value decreases as the number of connections increase--again, sort of arbitrarily. We understand that there have been studies on determining the load on a distributed system and then deciding the timeout value accordingly, so at this level, we don’t think there is a “correct” timeout value for the sockets.

We decided to test the web server through Firefox initially. We downloaded files with extensions that our server should support, and then tested from the browser to ensure that the server is able to retrieve the files and display them without any problems. In our test cases through the browser, we also attempted GET requests for files that did not exist, to check our error codes and appropriate response headers. We also used different kinds of .html files, just to make sure that they all show up as expected. Once we had concluded that the web server works fine from within Firefox, we decided to use telnet to test our server independently of the browser. This proved particularly useful because we were able to figure out problems with requests using HTTP/1.0--something which slipped unnoticed during our tests from Firefox, because Firefox uses the HTTP/1.1 protocol. We also implemented small JUnit test cases to test our getContentType() method, since it was the only method that could be tested without having to establish a proper connection.

In order to implement .htaccess files, we could have the main method in WebServer.java, before the ServerSocket is initialized, call a private static method in the same class that searches for all .htaccess files in our document root with the regular expression: “*.htaccess”, then reads each of them, adding each line of IP addresses from the files to an array of Strings that houses all the IP addresses to block, then returns that array. Then, in our while true loop where we accept connections, we would add an if statement that checks if socket.getRemoteSocketAddress() is in our array of IPs. If it isn’t, then we can go ahead and create and start the client thread. Otherwise, we won’t, and the loop will continue to run.

In comparing HTTP 1.0 and HTTP 1.1, it seems that it would be best to use HTTP 1.0 over HTTP 1.1 when retrieving enough objects, or objects with large file sizes, so that the time it would take to retrieve them in serial over HTTP 1.1 would be greater than the time to retrieve them all concurrently in HTTP 1.0 in addition to the overhead incurred from establishing and closing the TCP connection after the response is retrieved. It would probably be better to use HTTP 1.1 in situations with high bandwidth and low latency because the time to transmit objects in serial would be fast enough to warrant using it over HTTP 1.0, since you wouldn’t have to deal with the overhead of establishing and tearing down TCP connections with each response.