

Web Server Report

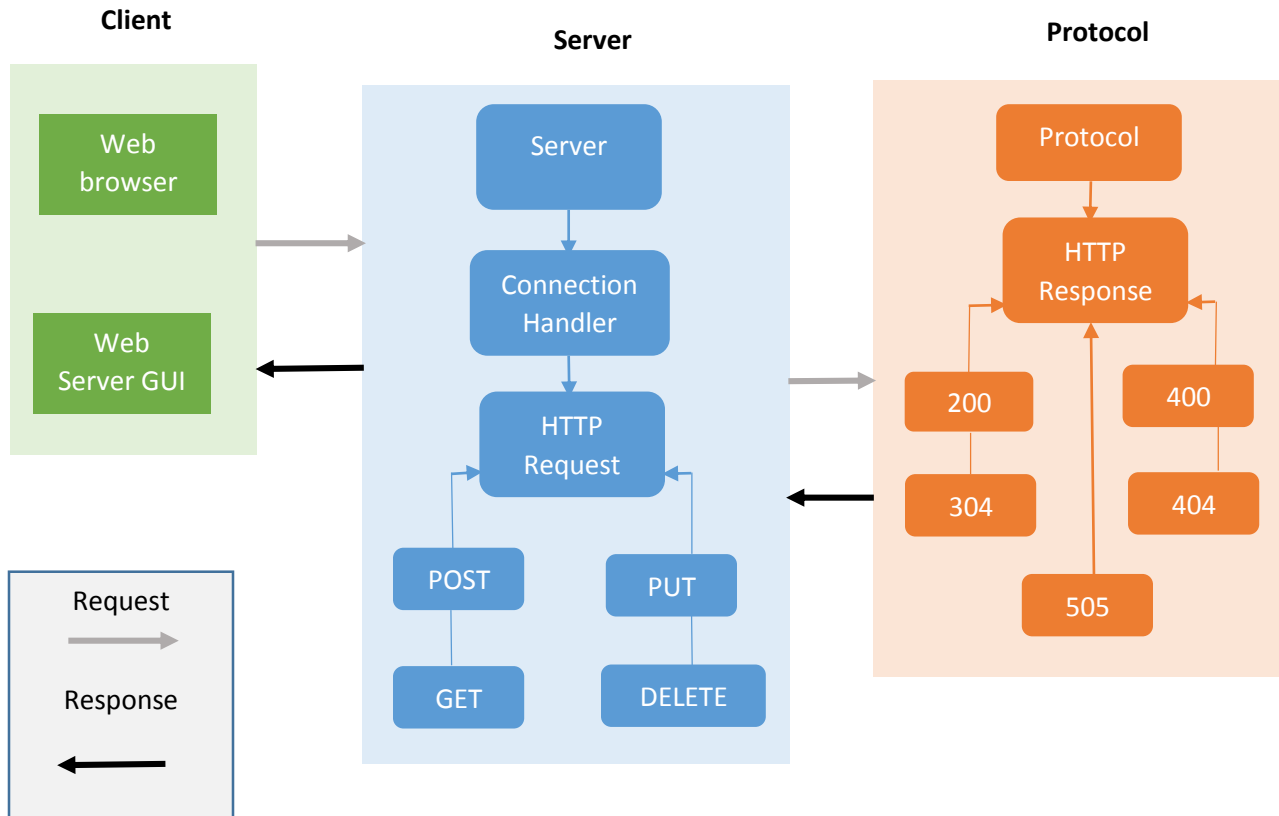
Team RAJ – Angelica Rodriguez, John Krasich CSSE 477

Table of Contents

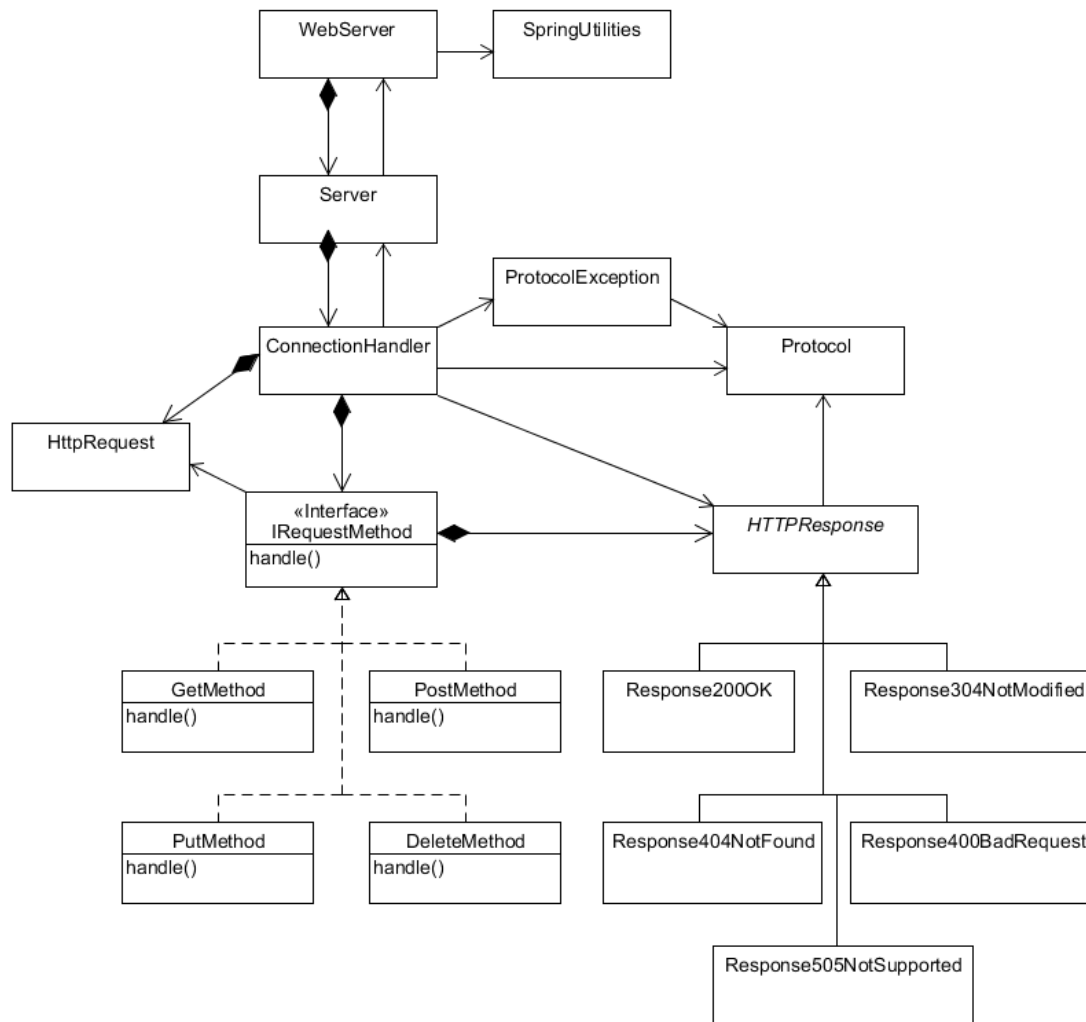
Milestone 1	3
Architecture Diagram	3
Detailed Design	4
Further Improvements	5
Test Report	6
Testing Utility	6
Web Browser	10
Change History – MS2	11
Updated Architecture Diagram	11
Updated Detailed Design	12
Brief Description	12
Feature Listing & Assignment	13
Test Report	14
Plugin Addition	14
POST	15
GET	16
PUT	17
DELETE	18
Future Improvements	19
Change History – MS3	20
Updated Architecture Diagram	20
Updated Detailed Design	21
Brief Description	21
Tactics/Feature Listing	22
Architectural Evaluation and Improvements	23
Availability	23
Performance	26
Security	28
Future Improvements	31

Milestone 1

Architecture Diagram



Detailed Design



Our refactoring of the web server utilized the following design patterns:

Strategy Pattern – The **IRequestMethod** interface allows for the various implementations of request handling to be completed in unique classes. This way, additional request handling can be implemented with minimal changes to the **ConnectionHandler** class – simply add the new request to the **ConnectionHandler**'s map of request methods.

Bridge Pattern – The abstract **HTTPResponse** class is used by the **ConnectionHandler** to write the generated response back to the client. However, the responses vary depending on the response code. Using a bridge pattern, each different response's implementation can be handled in separate classes without the **ConnectionHandler** needing to have any knowledge of how it is implemented.

Further Improvements

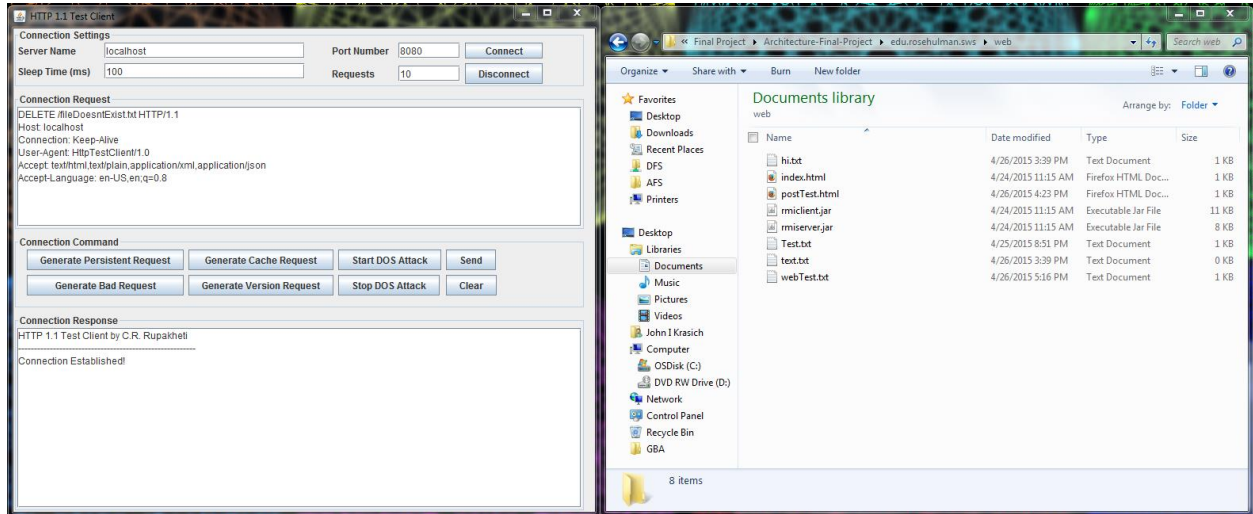
One area that can be further refactored would involve dividing the responsibilities of the `ConnectionHandler` class separately between requests and responses. The “run” method is rather long – breaking this up into different methods (or different classes) would make the code much more organized and easier to understand.

Test Report

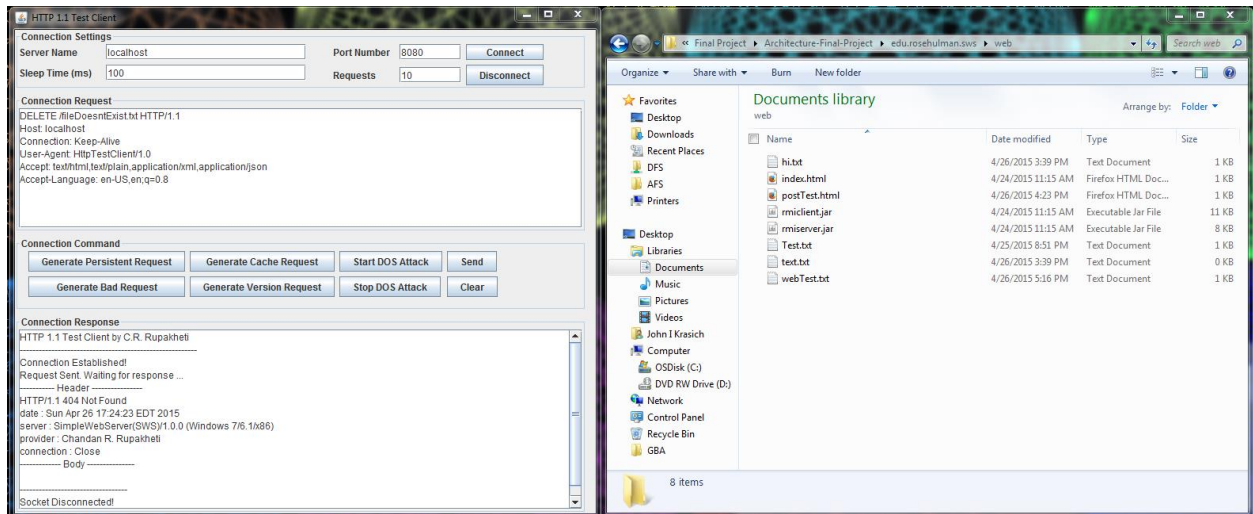
Testing Utility

DELETE of Non-Existent File

Before:

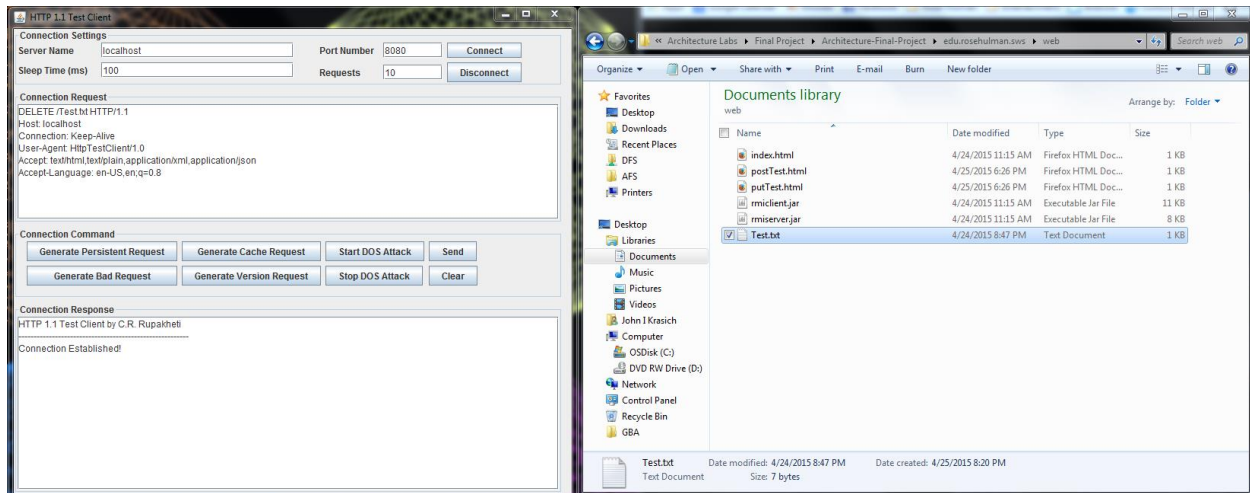


After: Response 404 Not Found

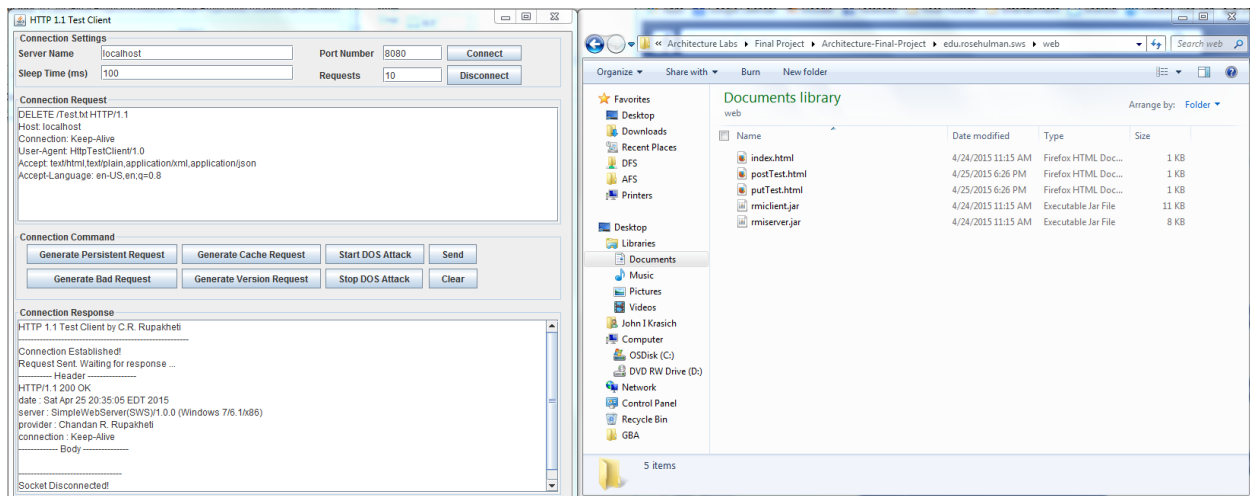


DELETE

Before:

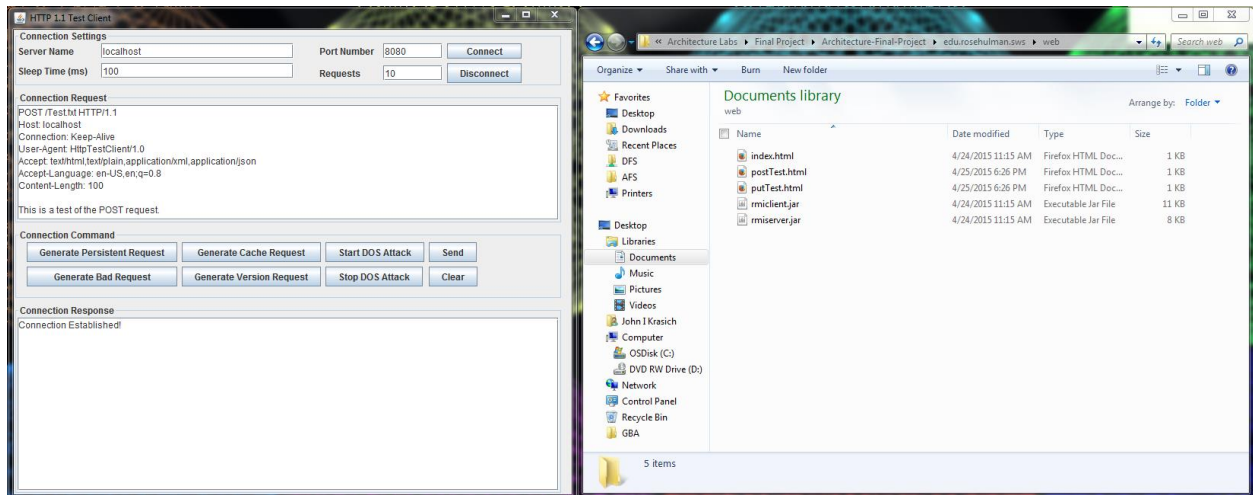


After: Response – 200 OK. File Test.txt has been deleted successfully

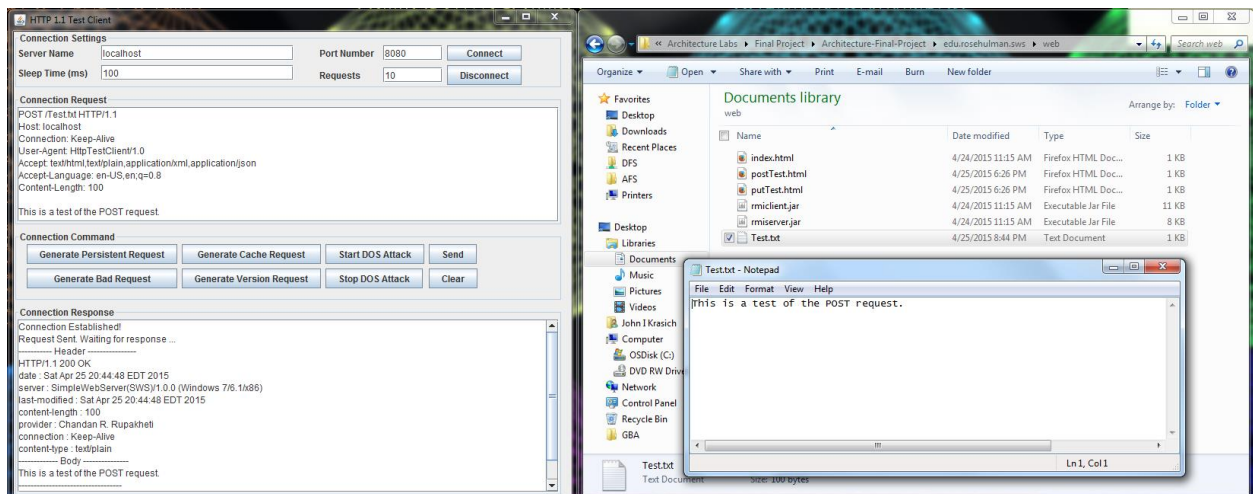


POST

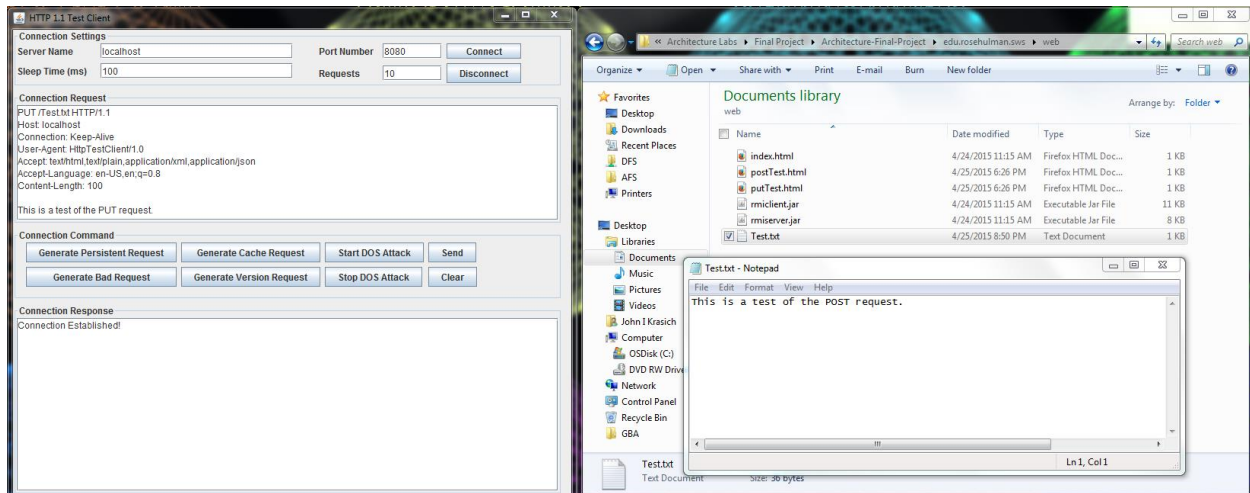
Before:



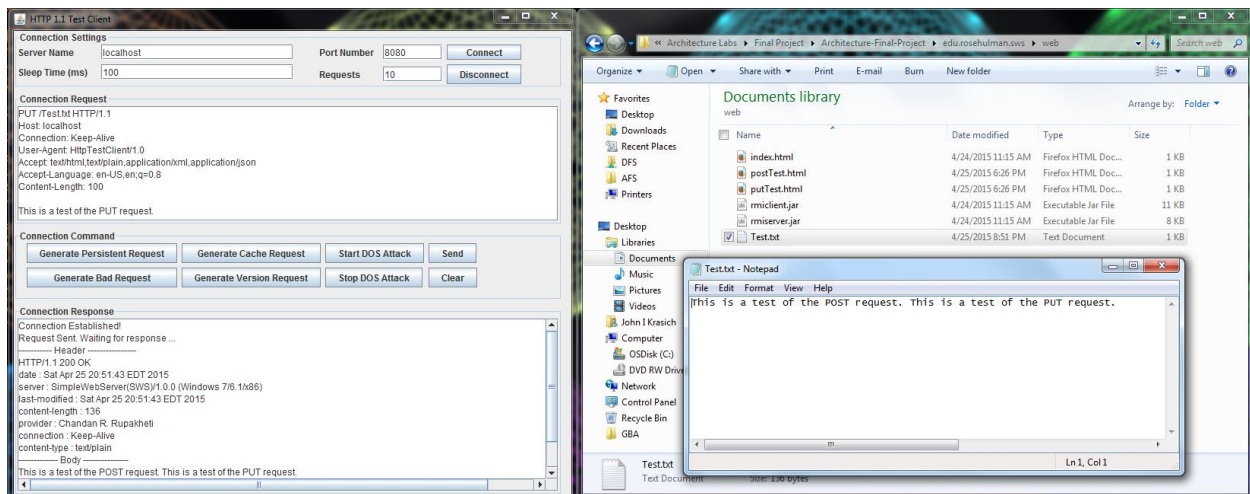
After: Response 200 OK. The file Test.txt has been created and filled with the body of the request.



PUT
Before:



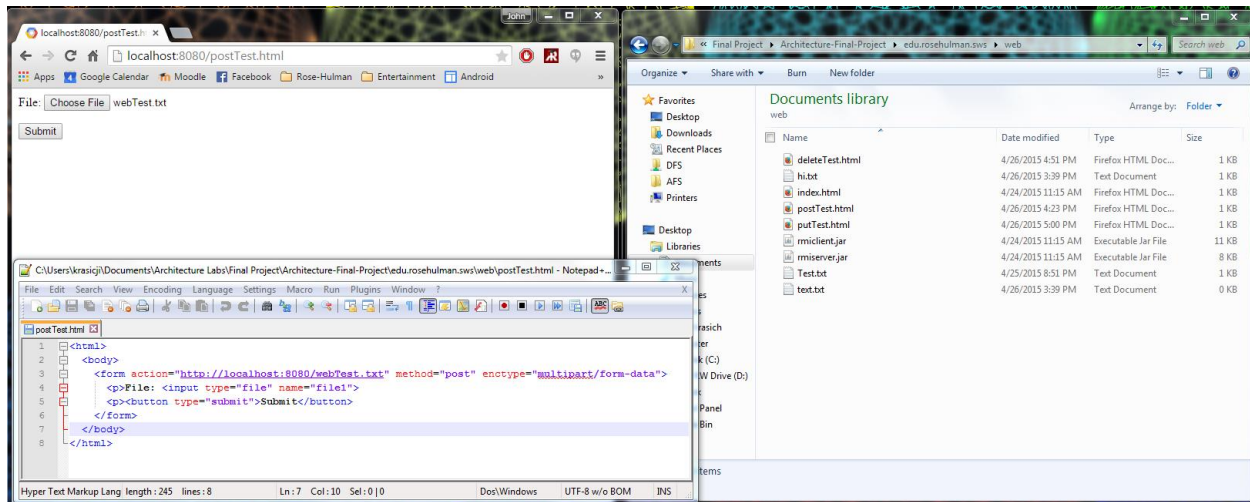
After: Response 200 OK. The body of the request was appended to the Test.txt file.



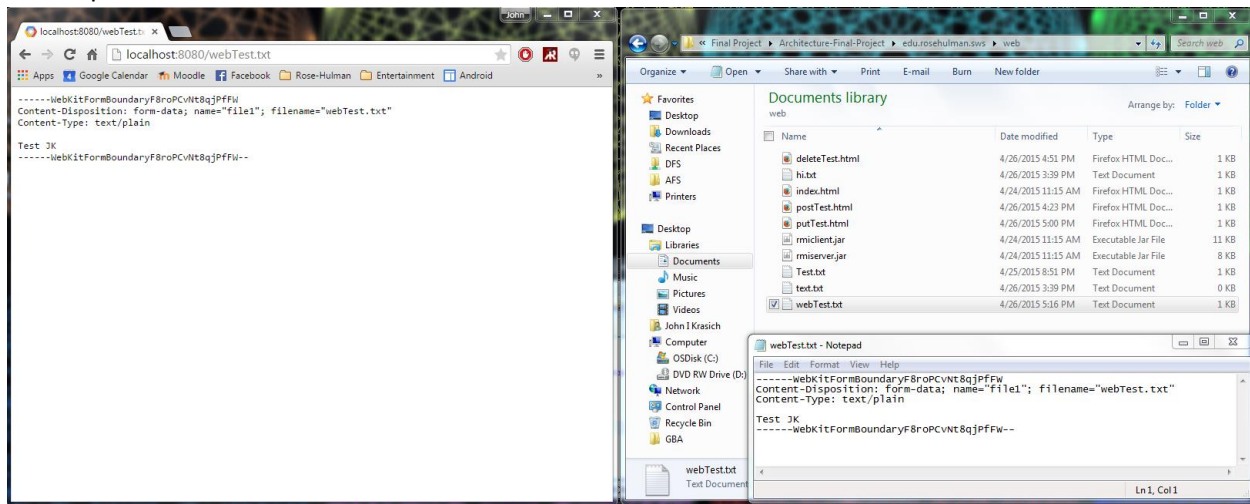
Web Browser

POST

Before:

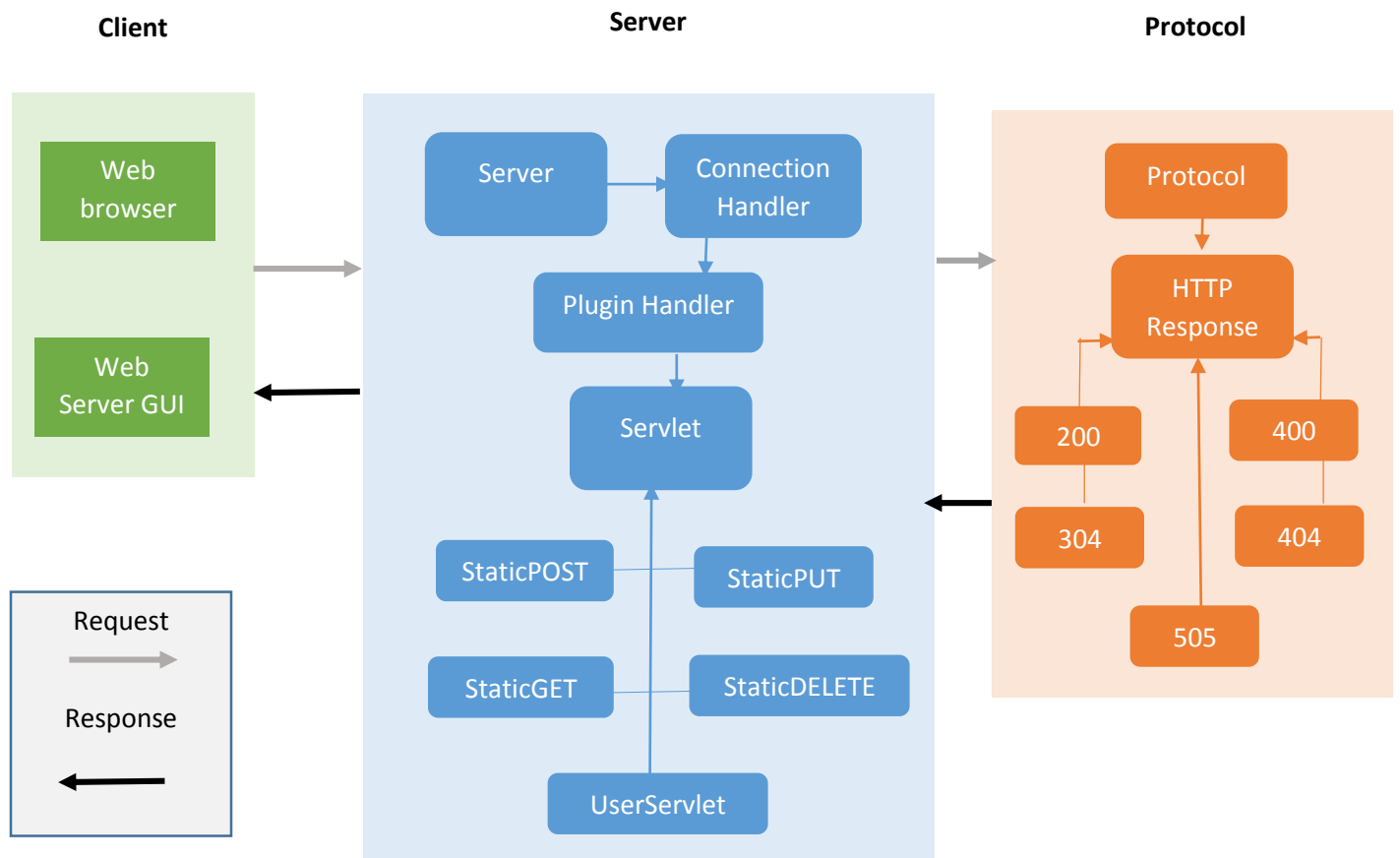


After: Response 200 OK – the text of the file was written into webTest.txt and returned as the body in the response.

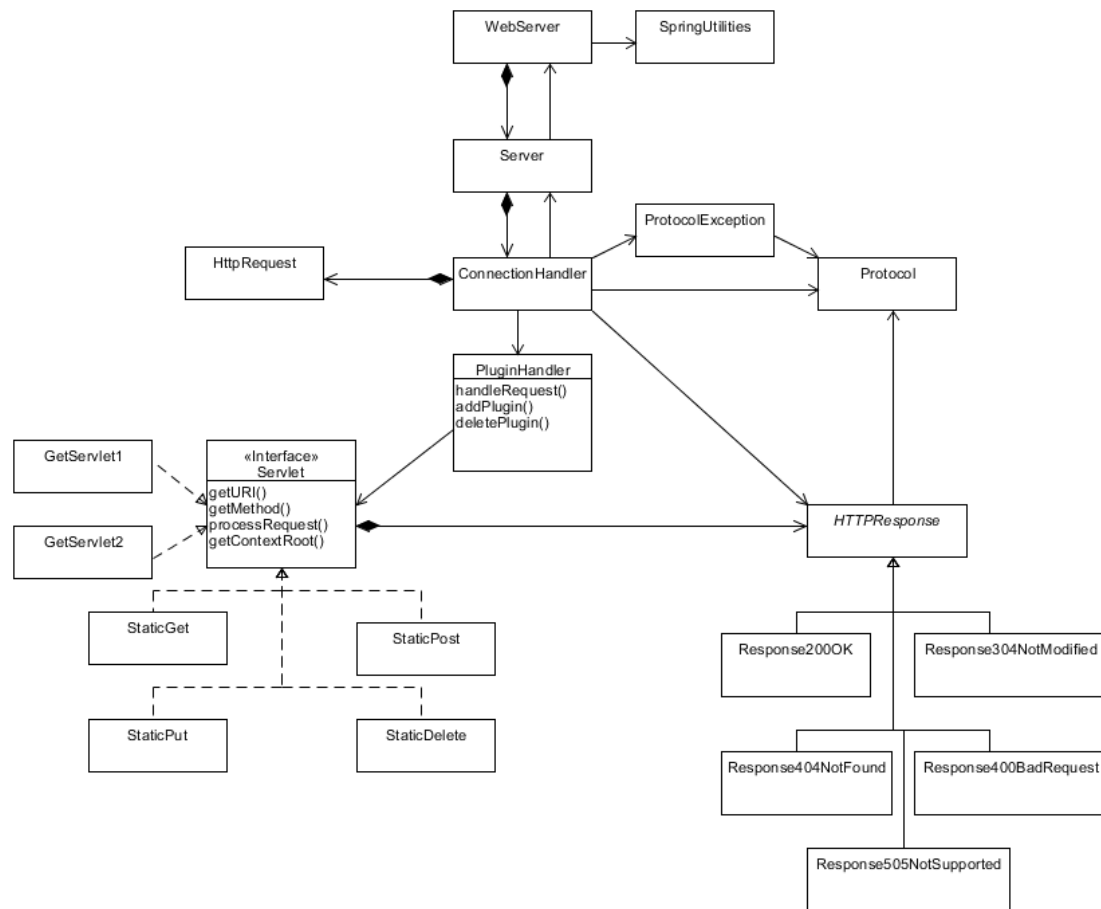


Change History – MS2

Updated Architecture Diagram



Updated Detailed Design



Brief Description

The most significant changes made for this milestone were the addition of the PluginHandler class and Servlet Interface. The PluginHandler watches a Plugins directory for the addition of Jar files from which new servlets would be dynamically included into the web server. The ConnectionHandler communicates with the PluginHandler, passing along the request for the PlugHandler to process correctly. This is done through a HashMap, which relates the context root to a second HashMap that stores the servlets and their respective URIs. Any servlet must implement the Servlet interface, which contains information necessary for the PluginHandler as well as its custom request processing method. The basic GET, POST, PUT, and DELETE methods from MS1 became “static servlets” that will be run if no plugin is found for that kind of request.

Feature Listing & Assignment

Angelica Rodriguez

- W-1: GET Requests
- W-2: POST Requests
- W-3: PUT Requests
- W-4: DELETE Requests

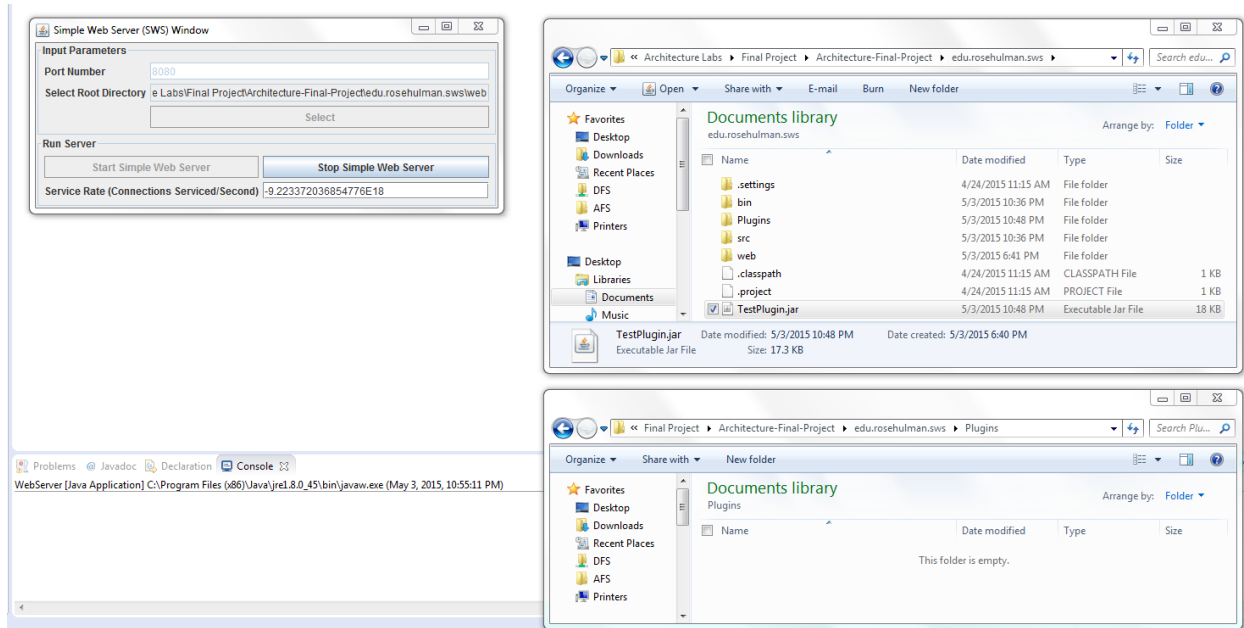
John Krasich

- P-1: Dynamic Loading
- E-1: Root Context and Configurable Route
- Test Report

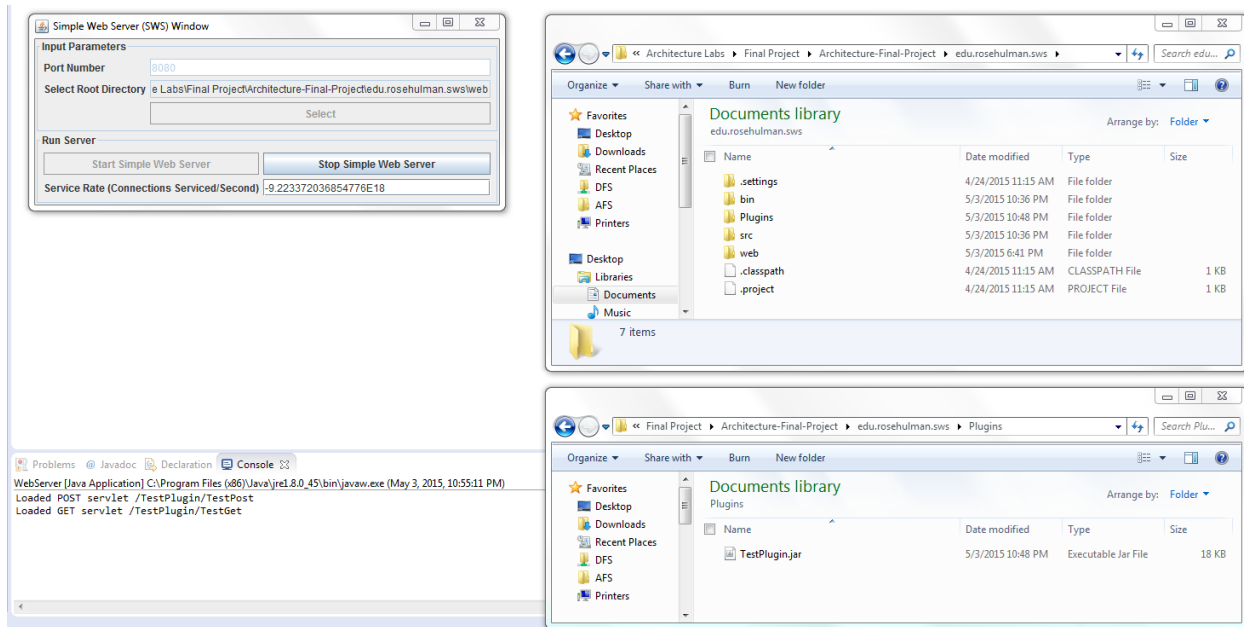
Test Report

Plugin Addition

Before:

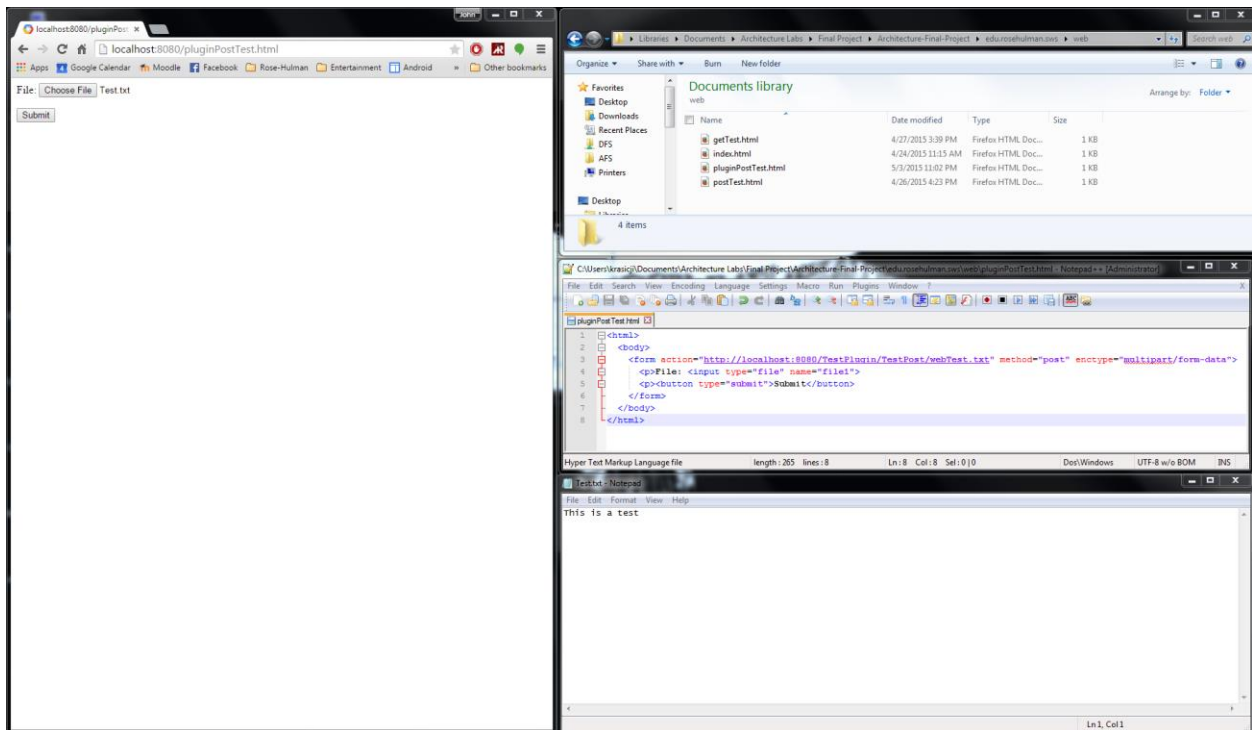


After: the plugin containing two servlets were dynamically loaded into the web server.

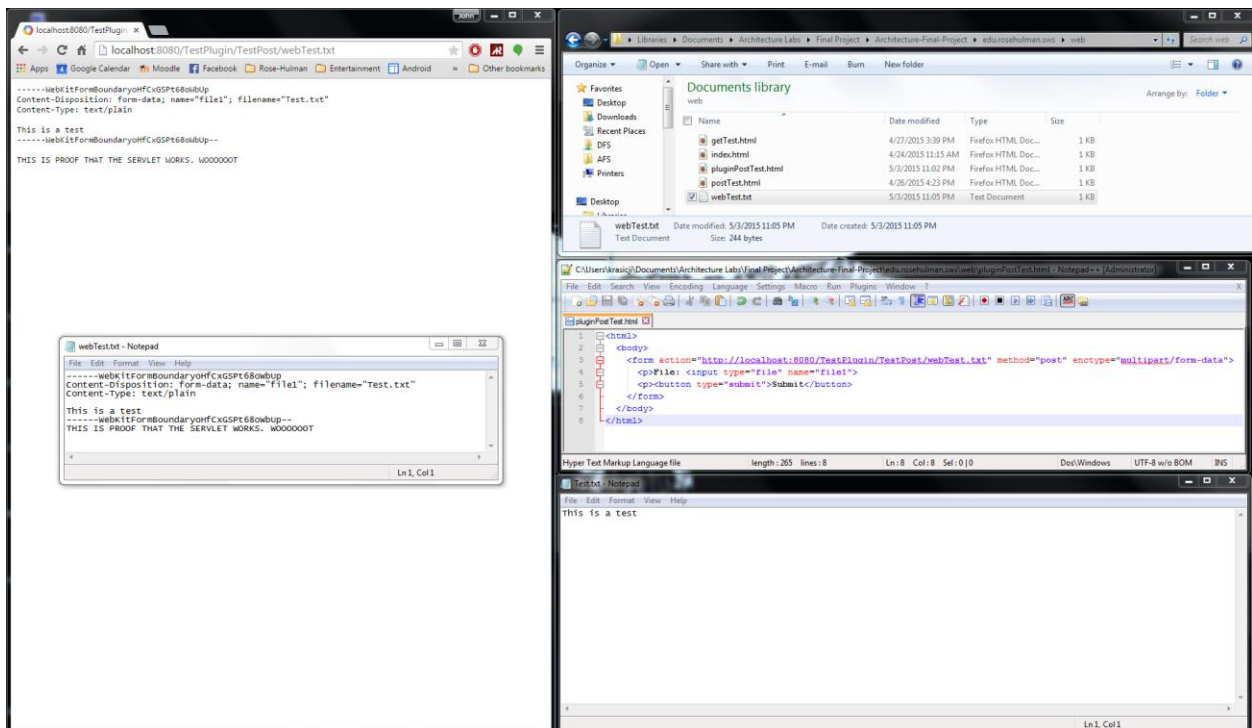


POST

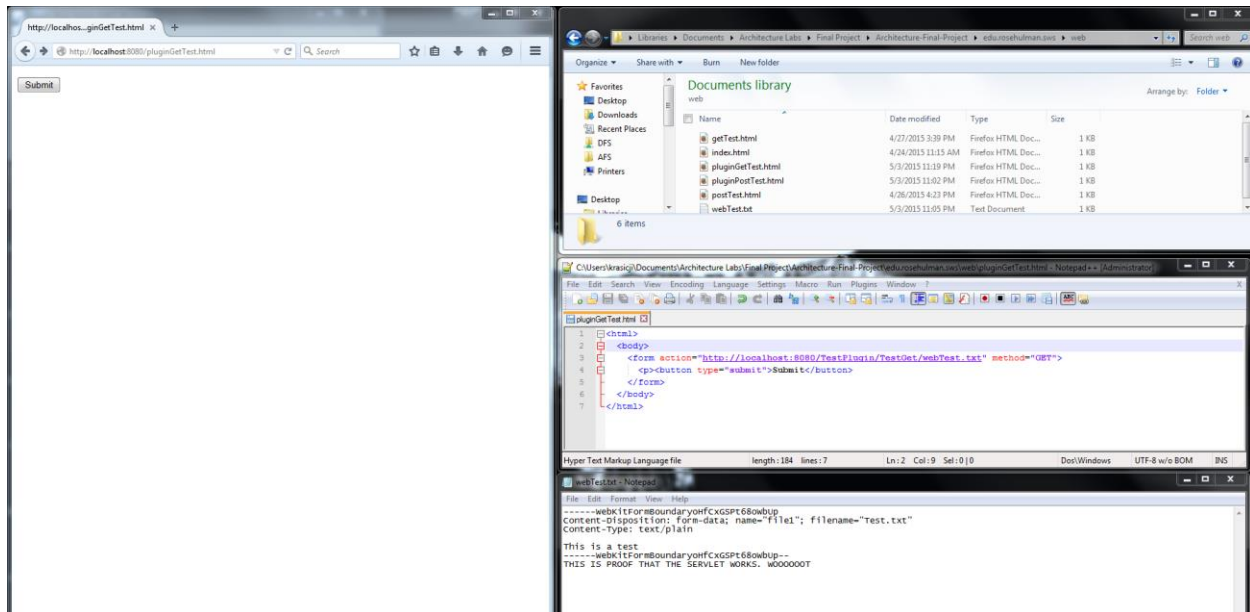
Before:



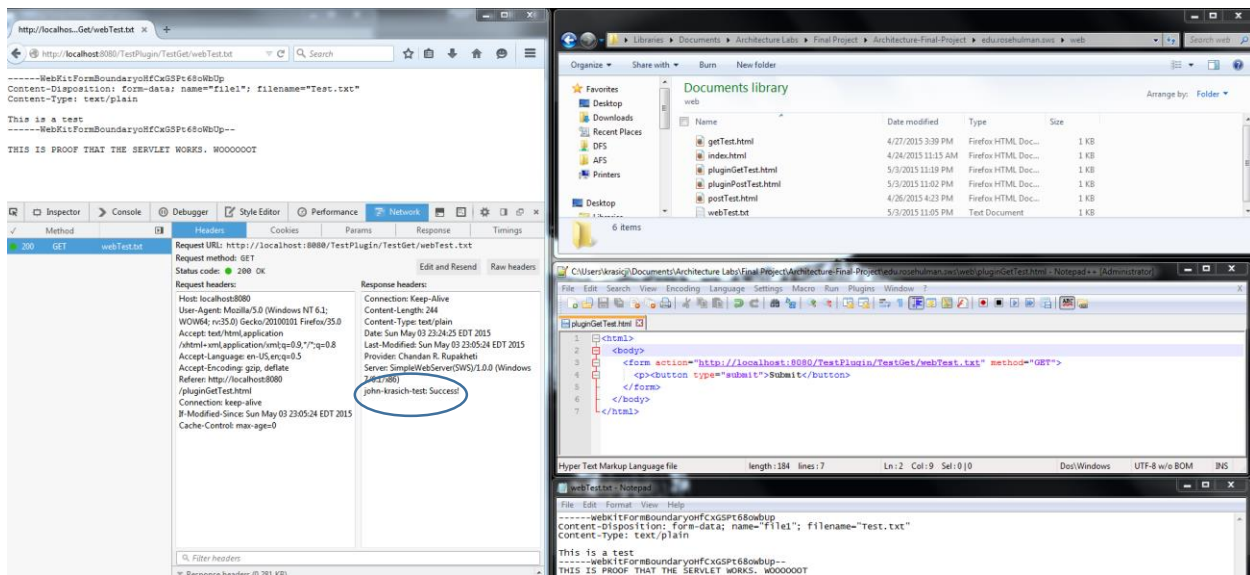
After: the servlet appended “THIS IS PROFF THAT THE SERVLET WORKS. WOOOOOT” to the file + body.



GET
Before:

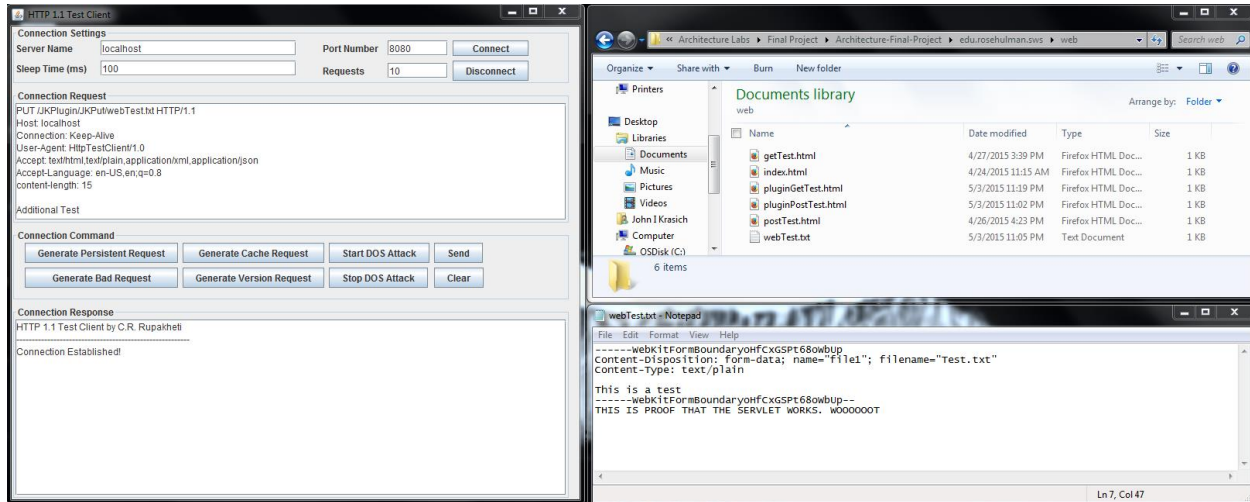


After: the servlet appended an additional header to the response.

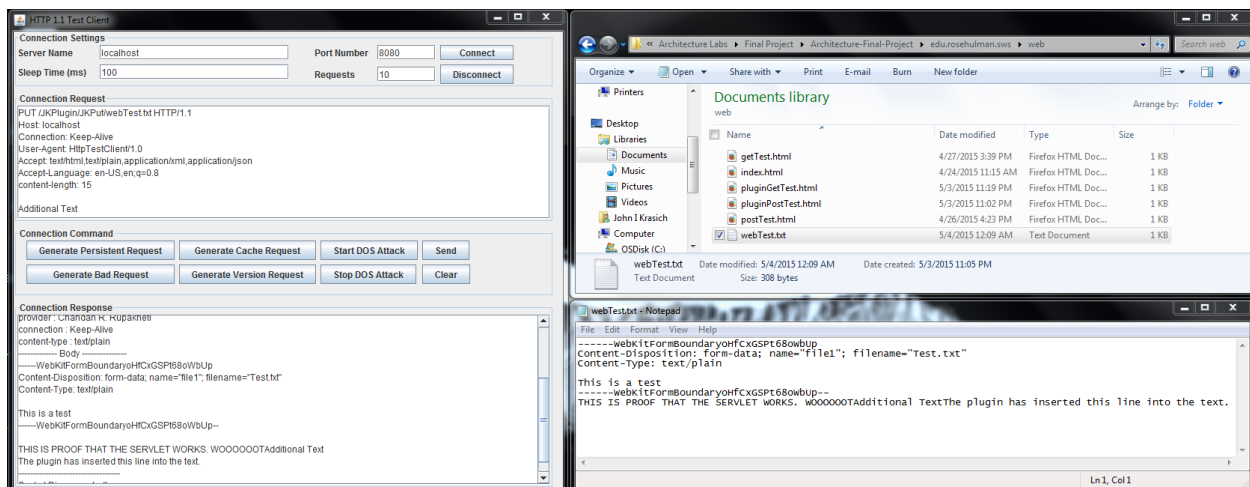


PUT

Before:

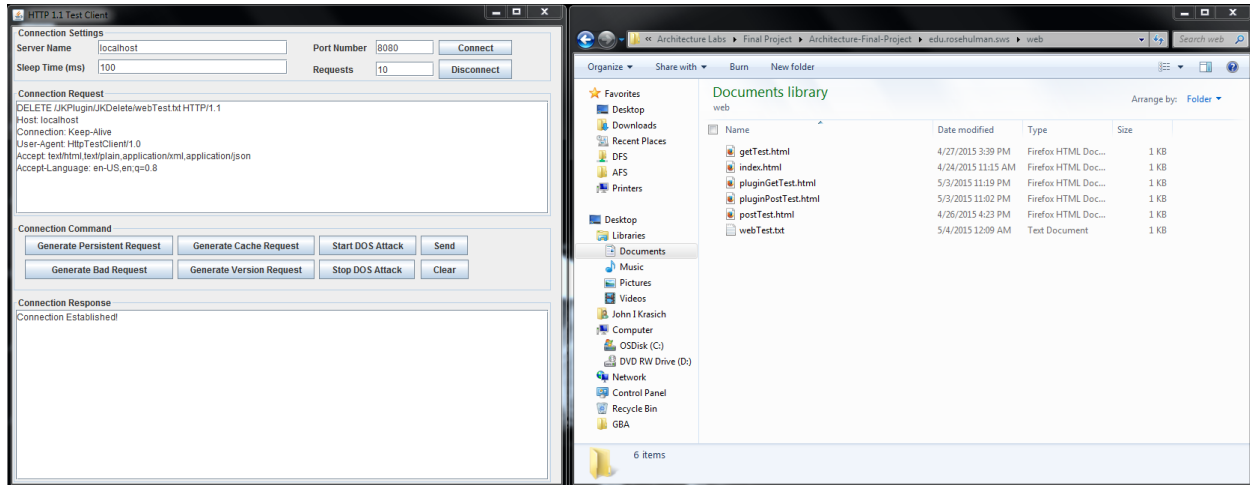


After: the servlet appended the extra text “The plugin has inserted this line into the text” into the file.

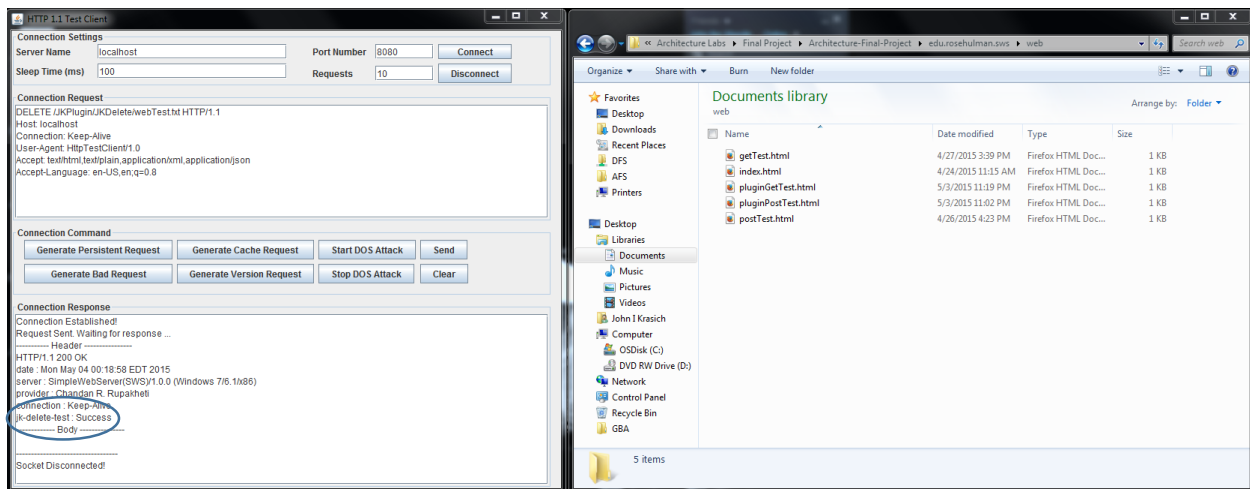


DELETE

Before:



After: the servlet appended an extra header into the delete response.

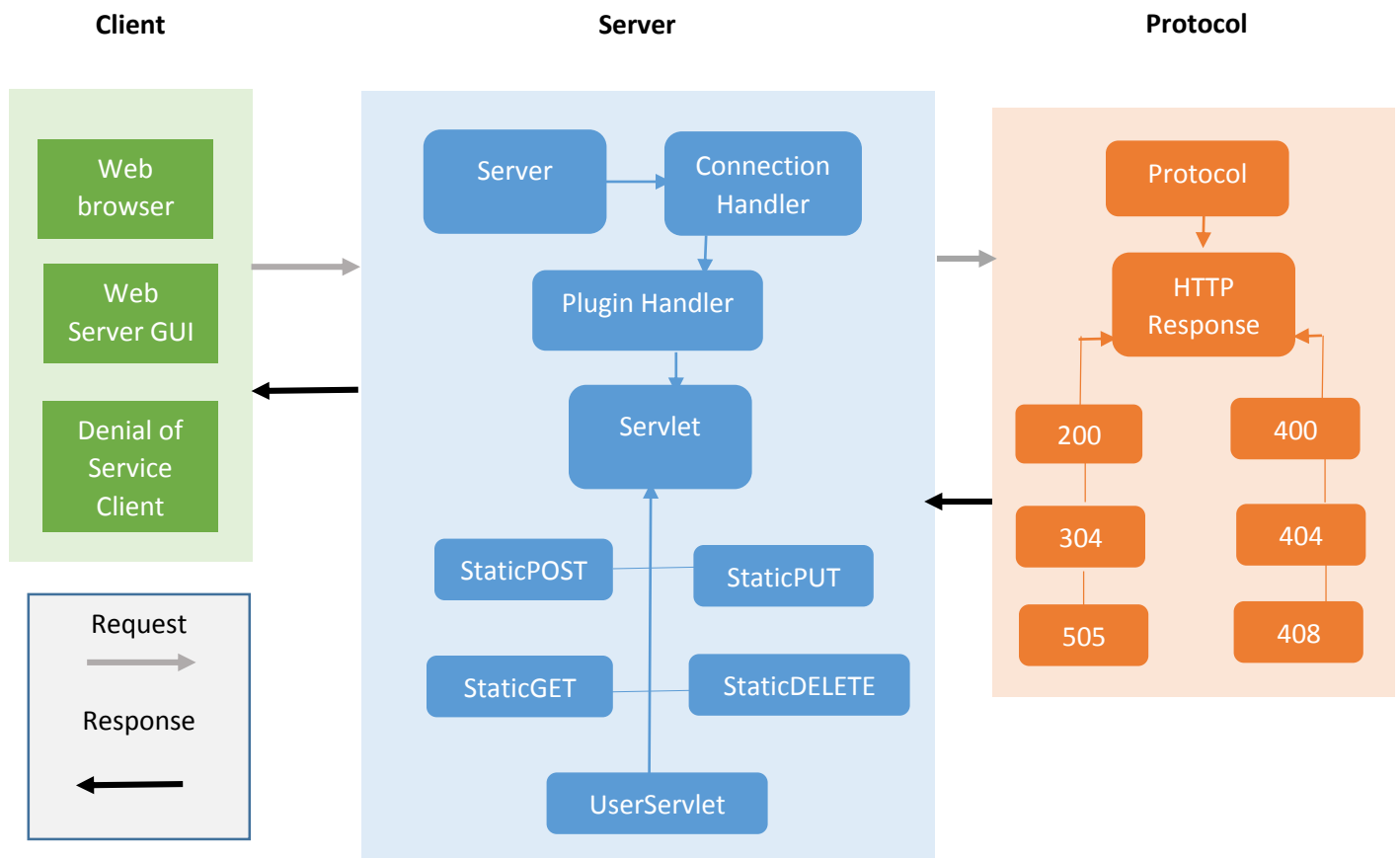


Future Improvements

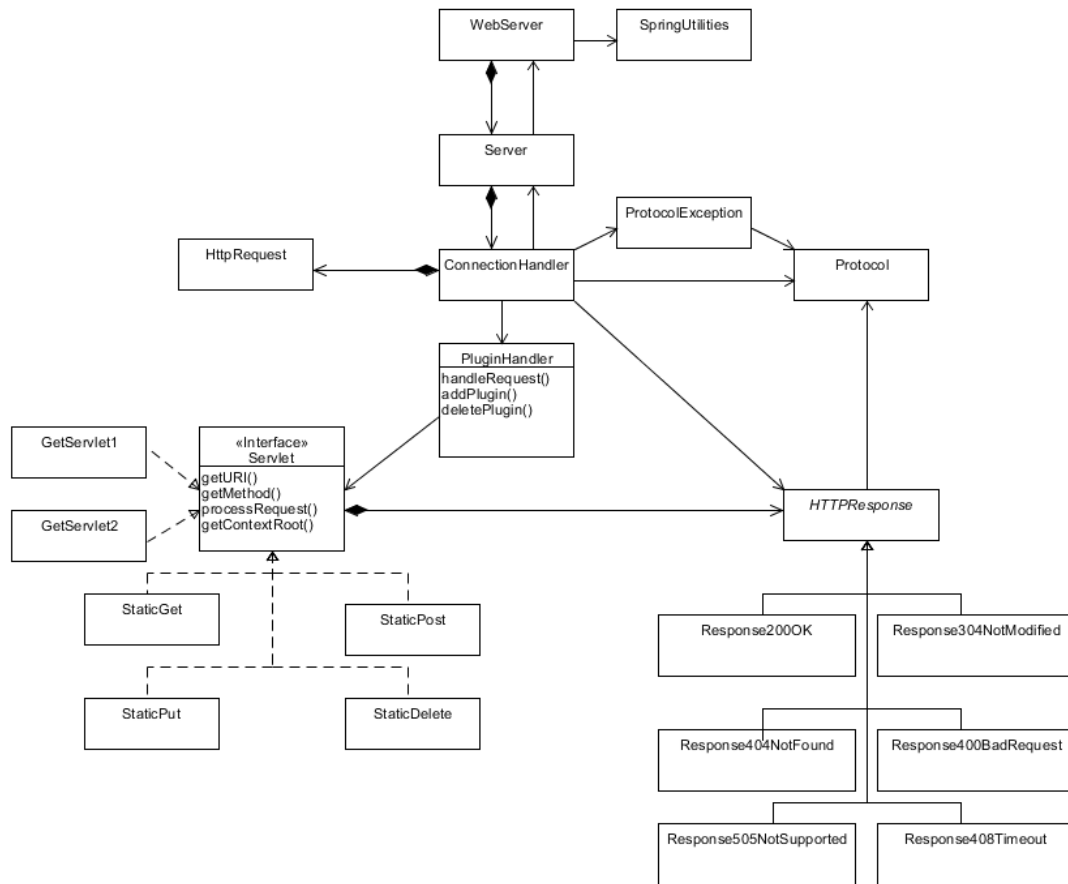
One idea we did not get to try but wanted to was to have the users supply a configuration file for the servlets with the information necessary, rather than have them hardcode the request code and create the JAR file. This would be a significant improvement because it would allow it to be easily modifiable, since they could make changes dynamically.

Change History – MS3

Updated Architecture Diagram



Updated Detailed Design



Brief Description

The only new addition to our class structure was the **Response408Timeout** class that would generate a response if a response is not generated within 10 seconds of attempting to read the request. All other changes were internal to the classes; the details of which can be found in our tactic implementation specifications.

Tactics/Feature Listing

John

- A1 – Request Timeout
- A2 – Incorrect Plugin Drop
- S1 – Handling DDoS Attacks

Jelly

- P1 – Handling Numerous Requests
- P2 – Scheduling Events
- S2 – Blacklisted IP Connection

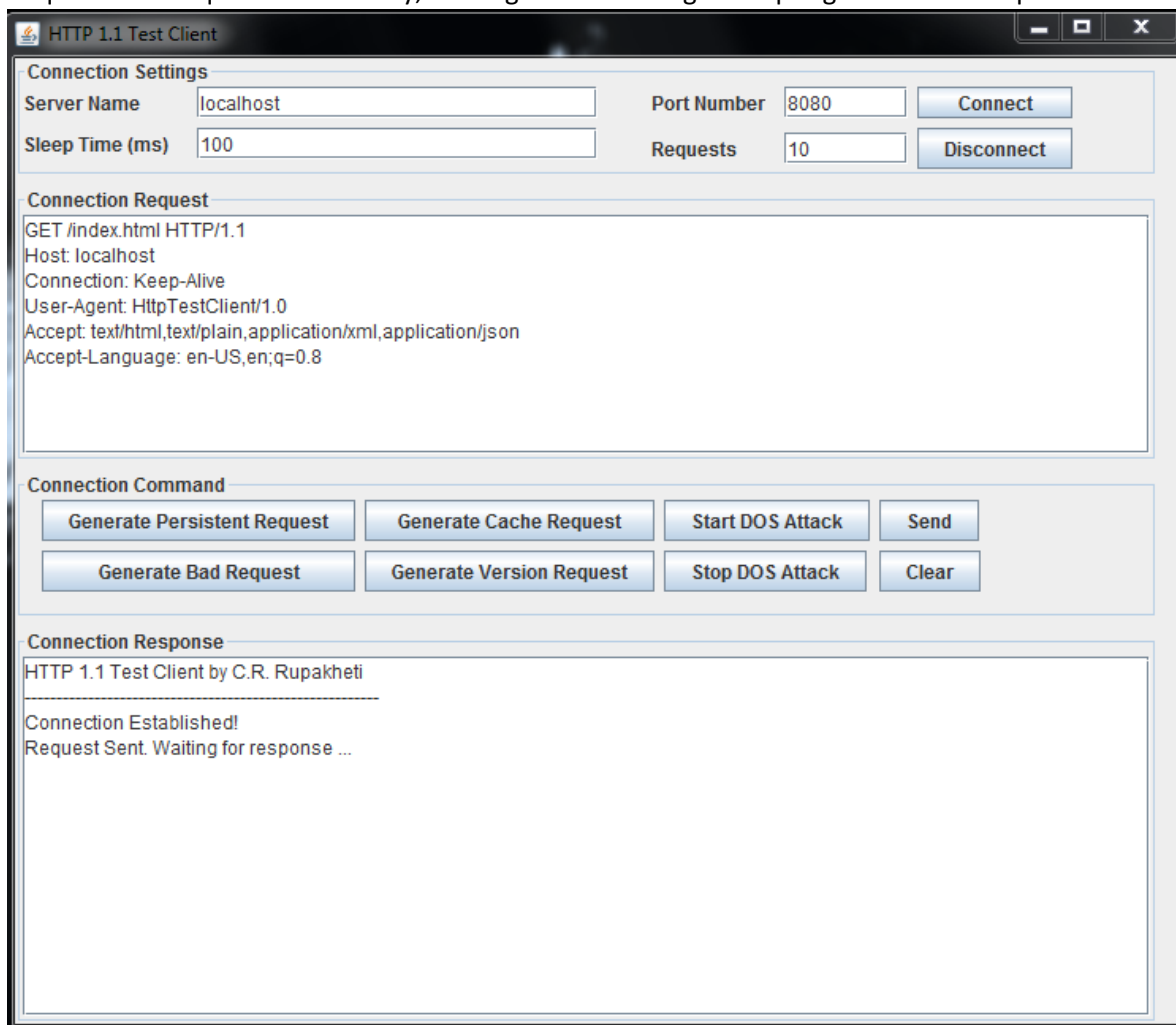
Architectural Evaluation and Improvements

Availability

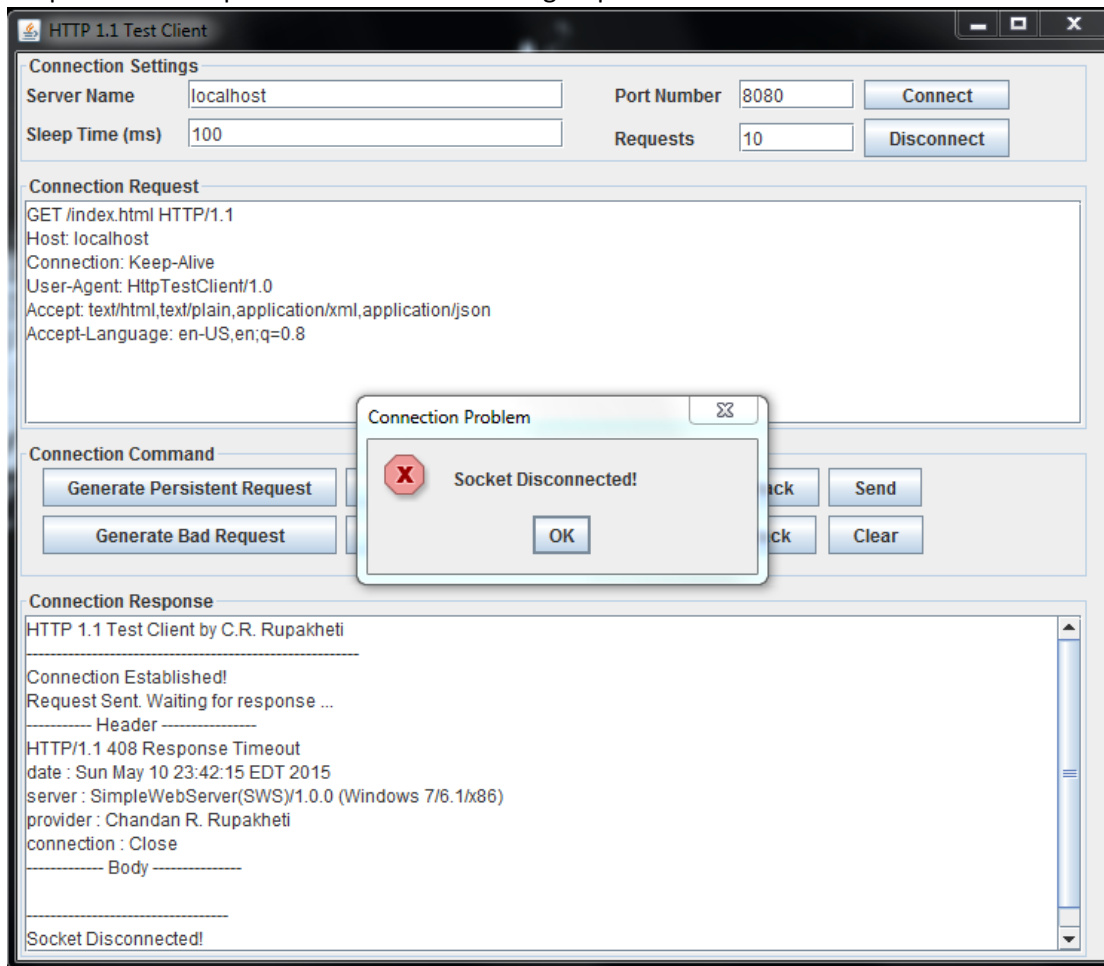
A1 – Request Timeout

- A1.1 Concrete Scenario
 - Source – User
 - Stimulus – User makes a malformed request
 - Artifact – Web Server
 - Environment – Normal operation
 - Response – 408 Response Timeout + disconnect socket
 - Response Measure – 10 seconds
- A1.2 Test Plan
 - We will use the test utility to send a GET request with no body (currently causes webserver to hang)
- A1.3 Baseline
 - 408 Response received one minute after malformed request

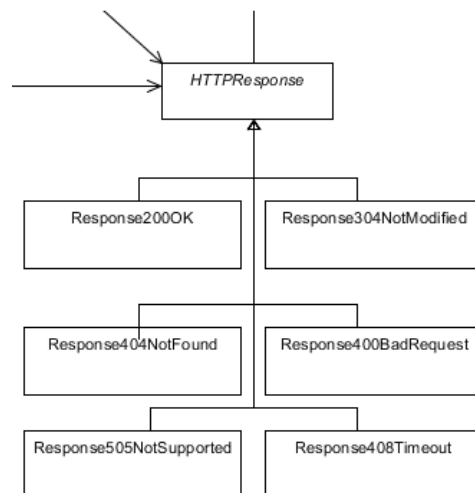
Request: GET request has no body, causing server to hang attempting to read the request.



Response: 408 response one minute following request.



- A1.4 Improvement Tactics
 - We will create a `Response408ResponseTimeout` class, which will be called from the `ConnectionHandler` class if a response is not generated within one minute of attempting to read the request.

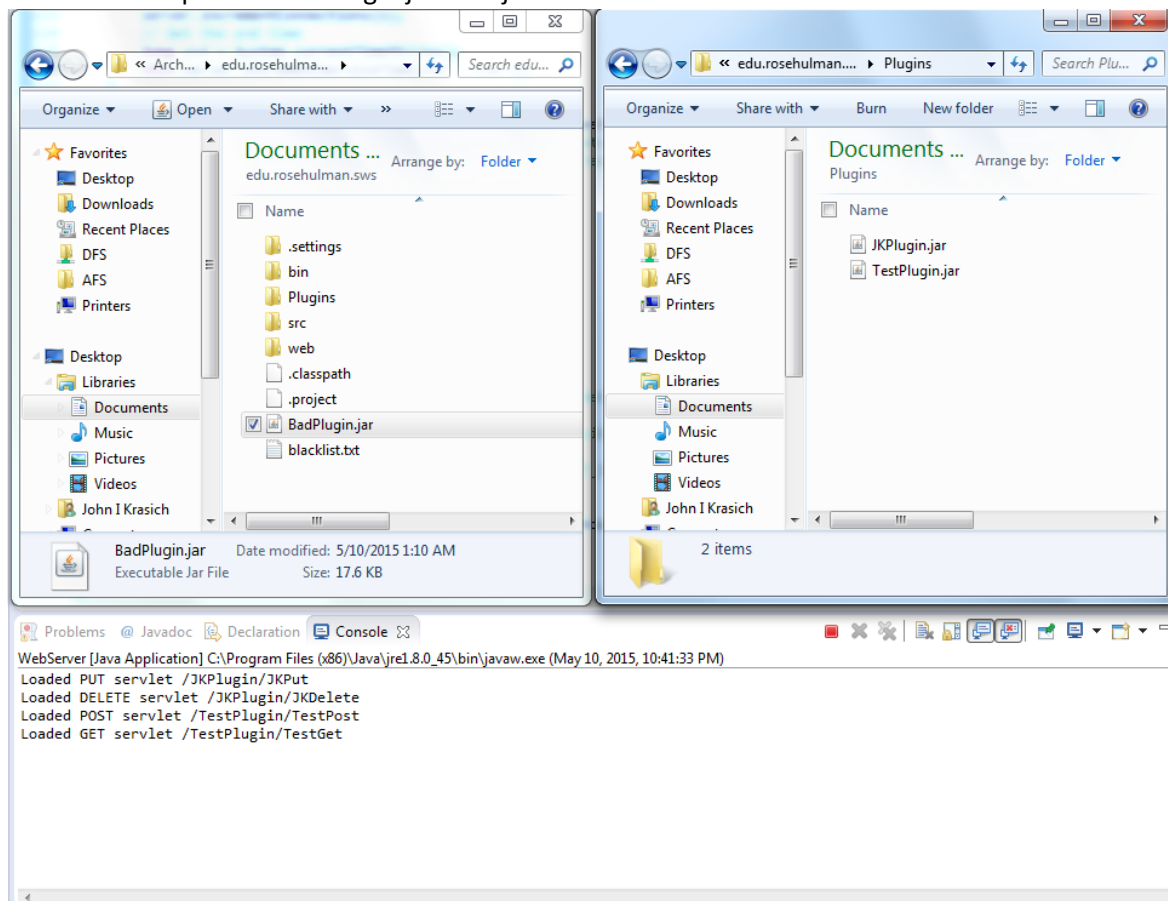


- A1.5 Conclusion
 - Implementing a timeout not only improves the availability of the webserver by preventing the server from freeze, but also improves performance by eliminating unnecessary stale connections and security by defending DDoS attacks.

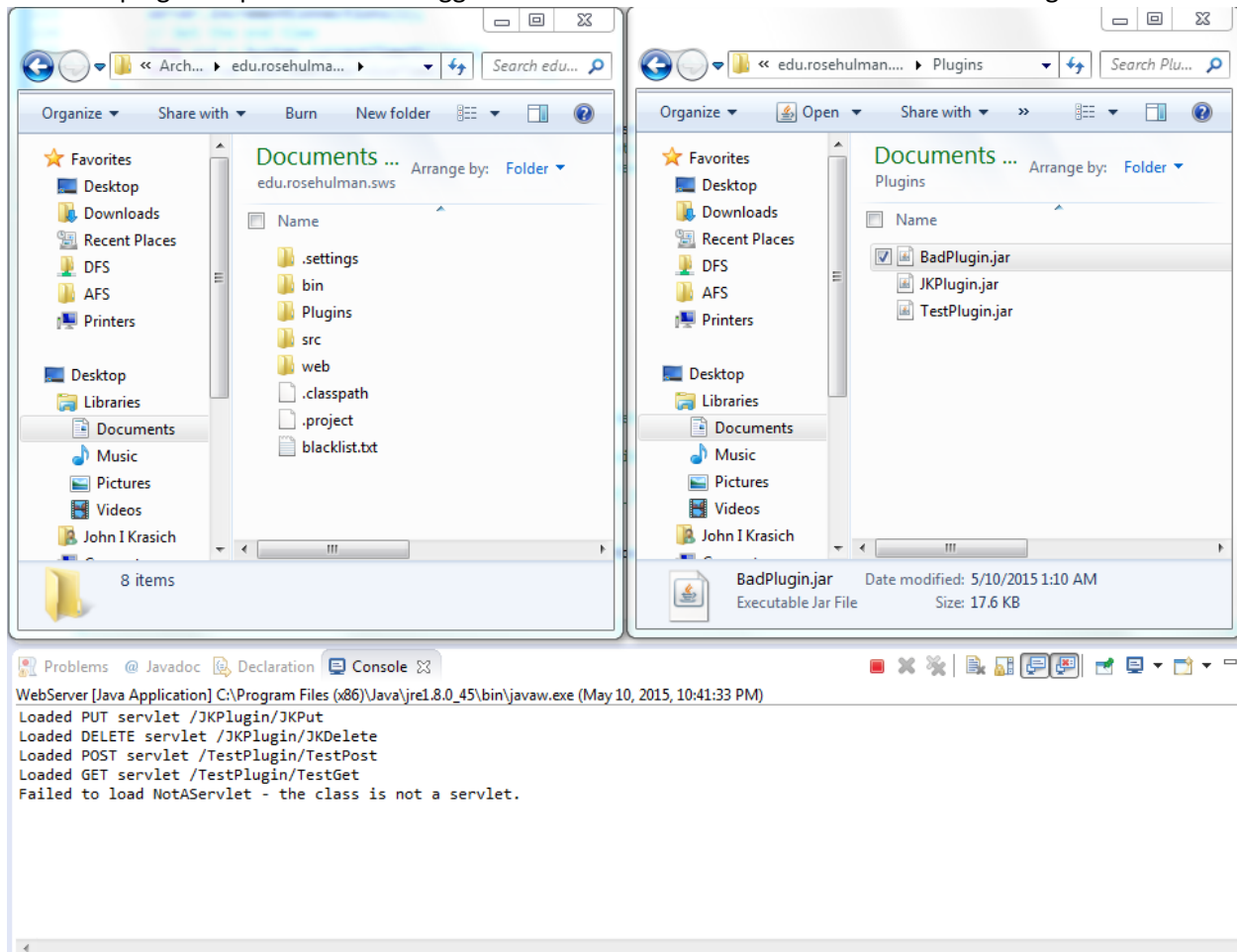
A2 – Incorrect Plugin Drop

- A2.1 Concrete Scenario
 - Source – Developer
 - Stimulus – Developer adds a new plugin to the webserver that has inconsistencies
 - Artifact – Web Server
 - Environment – Normal operation
 - Response – Developer should be notified and the plugin removed (ignored by server)
 - Response Measure – Web server does not crash, and does not implement the plugin's servlets.
- A2.2 Test Plan
 - We will create a plugin with non-servlet classes and drop the plugin into the plugins folder.
- A2.3 Baseline
 - Zero non-servlet plugins were loaded into the web server, with no exceptions thrown in the system when dropped into the plugin folder.

Before the drop of the BadPlugin jar. The jar contains a class NotAServlet.



After the plugin drop. The event is logged that the class is not a servlet and is therefore ignored.



- A2.4 Improvement Tactics
 - We will add a check that the loaded class is a Servlet with exception handling within our PluginHandler class to notify developers of the error.
- A2.5 Conclusion
 - Detecting faulty classes prevents the system from crashing, but also improves security by eliminating possibly malicious plugins that are not servlets.

Performance

P1 – Handling Numerous Requests

- P1.1 Concrete Scenario
 - Source – Multiple Users
 - Stimulus – A large number of requests are being sent to the server simultaneously
 - Artifact – Web Server
 - Environment – Normal operation
 - Response – Web server should maintain immediate responses even through a large number of simultaneous requests.

- Response Measure – Web server runs for a minute at 100 connections/second without errors
- P1.2 Test Plan
 - We will send numerous requests to the webserver such that its connection rate is always at its peak to see how long the server can maintain the connections while still immediately responding.
- P1.3 Baseline
 - See the results from **S1 – Handling DDoS Attacks**. The webserver responded to the request with many current connections.
- P1.4 Improvement Tactics
 - By scheduling events and removing stale/timed-out connections, the webserver will be able to handle a multitude of active requests so that each request is properly and immediately handled.
- P1.5 Conclusion
 - The increased performance of the web server also means that it is more available to other users, and can perform even while under attack

P2 – Scheduling Events

- P2.1 Concrete Scenario
 - Source – User(s)
 - Stimulus – Multiple requests of various sizes being sent simultaneously
 - Artifact – Web Server
 - Environment – Normal operation
 - Response – Web server should prioritize request based on size (with starvation prevention)
 - Response Measure – Speed at which requests are returned should be quicker than its original speed before scheduling.
- P2.2 Test Plan
 - We will bombard the server with many different requests of various sizes
- P2.3 Baseline
 - Through the use of a comparator class, the requests are ordered in a priority queue by content length

```

// Creates the comparator that our queue will use to compare requests by
// content-length
private class ContentLengthComparator implements Comparator<HttpRequest> {

    /*
     * (non-Javadoc)
     * @see java.util.Comparator#compare(java.lang.Object, java.lang.Object)
     */
    @Override
    public int compare(HttpRequest x, HttpRequest y) {

        int r1Size = Integer.parseInt(x.getHeader().get("content-length"));
        int r2Size = Integer.parseInt(y.getHeader().get("content-length"));

        if (r1Size < r2Size) {
            return -1;
        }
        if (r1Size > r2Size) {
            return 1;
        }
        return 0;
    }
}

public ConnectionHandler(Server server, Socket socket) {
    this.server = server;
    this.socket = socket;
    this.comparator = new ContentLengthComparator();
    // The 10 in the Queue constructor is not definitive, it will grow if it
    // needs to
    this.queue = new PriorityQueue<HttpRequest>(10, comparator);
}

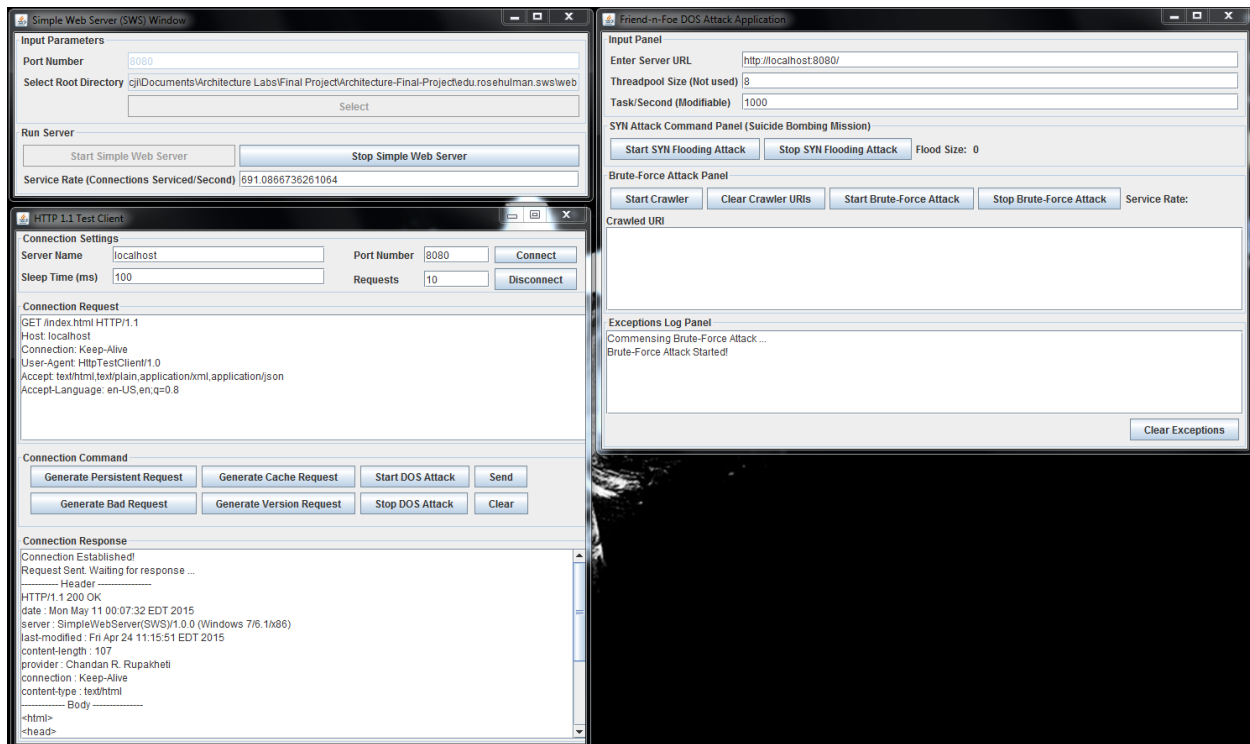
```

- P2.4 Improvement Tactics
 - We will create a queue of requests organized by content length and will process the requests in order when available.
- P2.5 Conclusion
 - Creating a request scheduler helps the system respond more quickly to small requests, which users would expect immediate results from.

Security

S1 – Handling DDoS Attacks

- S1.1 Concrete Scenario
 - Source – The Denial of Service Launcher
 - Stimulus – The Denial of Service Launcher attacks the web server
 - Artifact – Web Server
 - Environment – Normal operation
 - Response – Web server should properly remove stale connections
 - Response Measure – Web server still responds to request during attack.
- S1.2 Test Plan
 - We will run the denial of service launcher and “attack” the web server
- S1.3 Baseline
 - Web server is able to serve request during a brute force attack ran for 5 minutes. Peak service rate during attack was 815 connections / second.



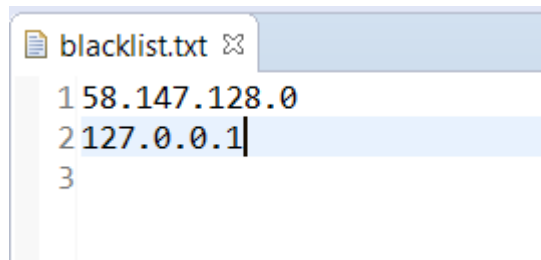
- S1.4 Improvement Tactics

- We will monitor the service rate to make sure the rate is in an acceptable range, as well as periodically (or as needed) remove stale connections
- S1.5 Conclusion
 - Being able to fend off DDoS attacks while continuing to service requests is an improvement to the web server's availability in addition to its increased security. Not only will the system not crash, but can still service a real user while under attack.

S2 – Blacklisted IP Connection

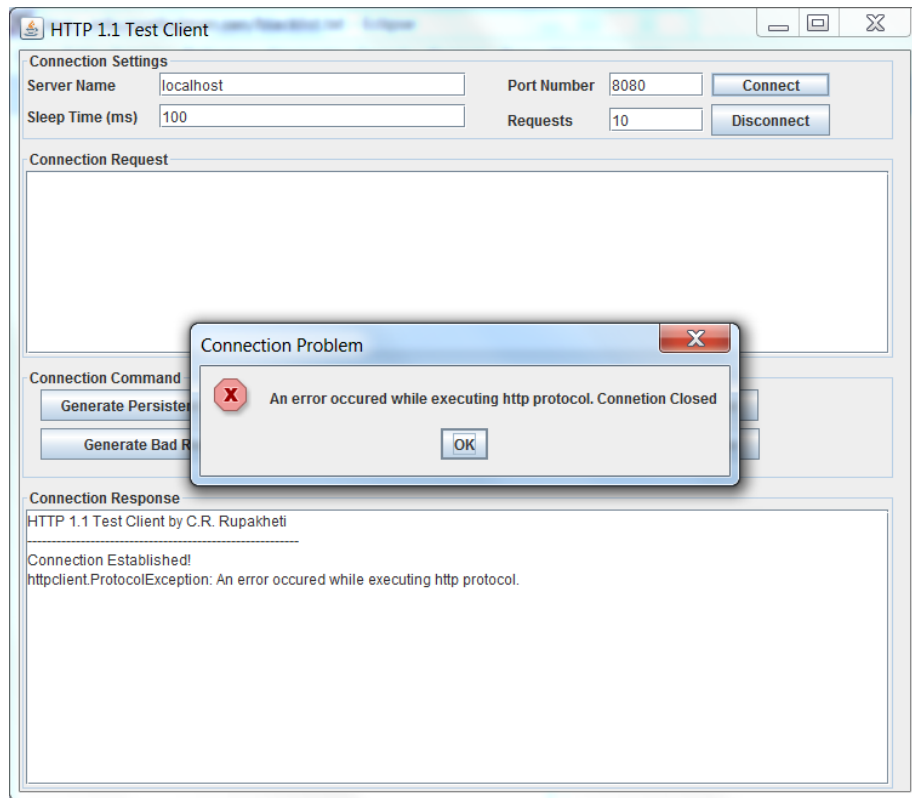
- S2.1 Concrete Scenario
 - Source – A blacklisted IP
 - Stimulus – A blacklisted IP connects to the web server
 - Artifact – Web Server
 - Environment – Normal operation
 - Response – Web server should deny the connection to the IP and log the event
 - Response Measure – The event is logged and no connection is made
- S2.2 Test Plan
 - We will add a known IP (such as Jelly's computer) to the blacklist and attempt to access the webserver.
- S2.3 Baseline
 - Zero connections were made to blacklisted IP's

Add Jelly's loopback IP to the blacklist:



```
blacklist.txt
1 158.147.128.0
2 127.0.0.1
3
```

Attempt to connect from Jelly's computer using the test utility:



As you can see the connection is rejected, and the event is logged:

```
Console
WebServer [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (May 10, 2015, 8:46:10 PM)
SEVERE: Blacklisted IP: /127.0.0.1 attempted to connect.
May 10, 2015 8:47:44 PM server.Server run
WARNING: Closing connection to /127.0.0.1
```

- S2.4 Improvement Tactics
 - We will create a blacklist to check against when connections are attempted on the server.
- S2.5 Conclusion
 - By blocking a blacklisted IP, the server is kept safer, which ultimately can prevent malicious attacks that threaten the availability and integrity of the server to normal users.

Future Improvements

Something we could do in the future to make our system more robust is add the ability to authorize and authenticate users to have different levels of access and ensure no one is able to see something they are not supposed to. Setting a limit on the length of our queue or revising our queueing policy might be necessary as well (especially if request patterns change in such a way that it would no longer be ideal to add them to the cue in order by content-length).