

CS331 Project 3 Write Up – Ben Webb

The purpose of this project was to implement a web-based version of an IP address prefix calculator. The HTML file that the user was interfacing was provided and responded via a static HTML file using an AJAX request.

To implement this, I created a http server with a callback function for response and requests processing. When a request is made on the server, the path of the request is parsed. This was a relatively simple webpage, and so there were two different path cases.

The first path is if is responding to a GET request on the path /prefcalc. This is a scenario that can only be triggered by elements on the webpage, and thus the webpage must be loaded first. The request also includes the IP address and the prefix length as a part of the query. The prefix length is then parsed into its subnet mask. The block of code to handle this relied heavily on bitwise comparison.

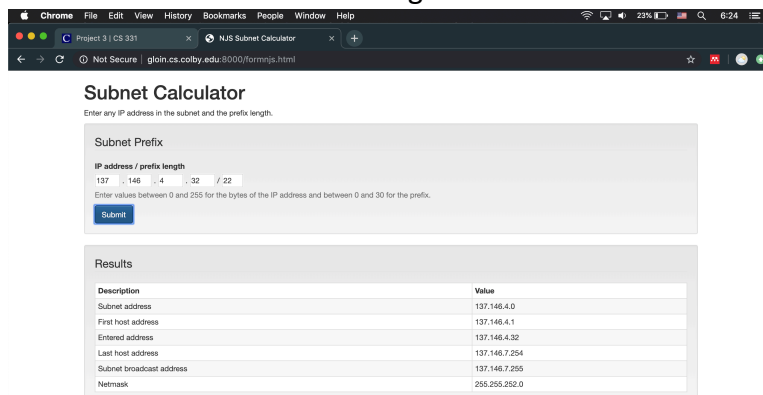
```
var prefix = (0xFFFFFFFF ^ (1 << 32 - query.prefixlen)) - 1);
var ma = (prefix >> 24) & 0xFF;
var mb = (prefix >> 16) & 0xFF;
var mc = (prefix >> 8) & 0xFF;
var md = (prefix) & 0xFF;
```

This was then added to a JSON structure to hold all of the results. The subnet is each section of the mask bitwise and with the provided IP address. The first host is one larger than the subnet. The broadcast address was found by doing a bitwise and with the mask and the IP address, the result of which was bitwise or'd with 0xFF that was bitwise not compared to the mask.

```
((query.b1 & ma) | (255^ma))
```

I wasn't sure how the process was supposed to go. I was unable to write anything to /prefcalc without it become a single use unix file while leaving it typeless. Because of that, I then just wrote a traditional response using the JSON structure as the data.

The other path reads a file that already exists in the server. This is how the HTML webpage was opened in the first place. They were separate processes because the HTML file and the processing of the IP are different processes. The HTML file was to remain unedited, and the IP processing was only ever responding directly to user input. Below is an example of the webpage and the associated table with the IP range result.

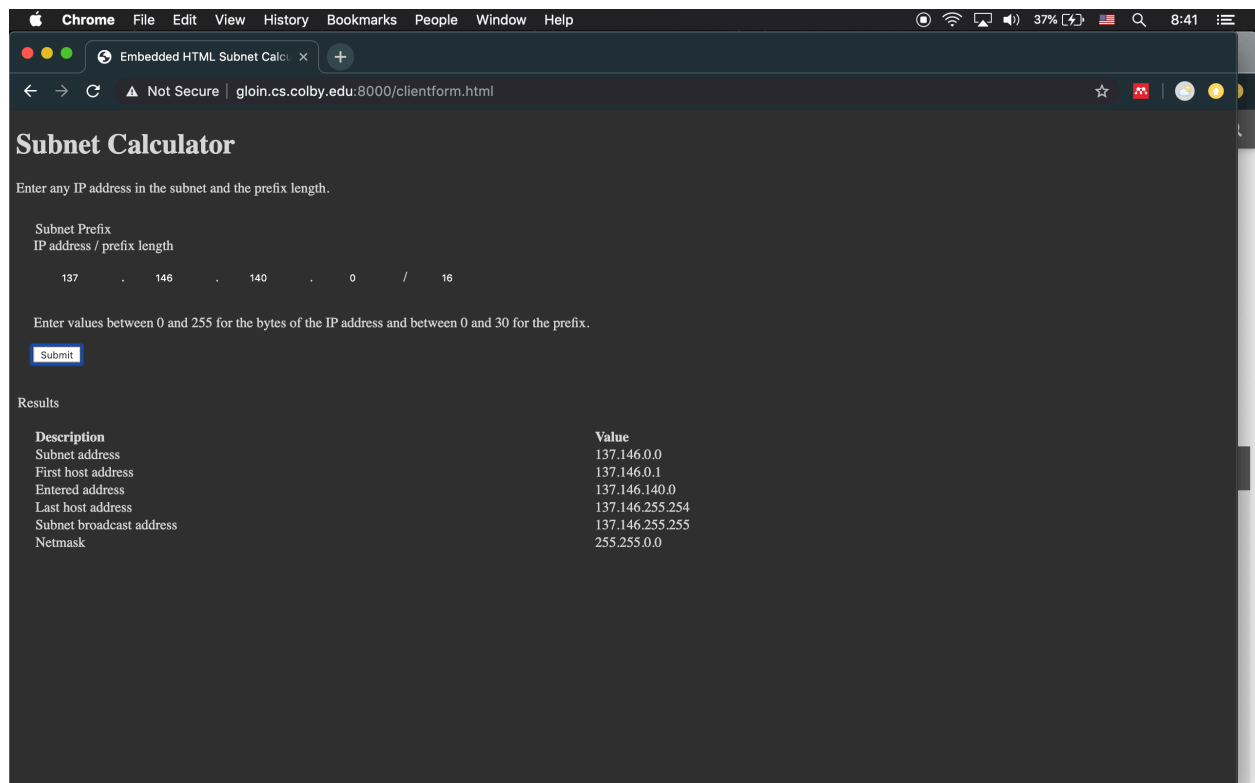


Description	Value
Subnet address	137.146.4.0
First host address	137.146.4.1
Entered address	137.146.4.32
Last host address	137.146.7.254
Subnet broadcast address	137.146.7.255
Netmask	255.255.252.0

Extensions

For my first extension, I wrote the code for the calculation in the body of the html file. This meant that the client page had the script to calculate the IP. This was still compatible with my node.js script that I made for the project because this only requested the html file and never actually sends a GET request to the server to do the calculation.

This was relatively easy to add. The `<script>` `</script>` section of an HTML file can be JavaScript. What took the most time and effort was making it so the entire page would not reload like it normally does when processing a script. The solution to this was to encapsulate the entire calculation inside of a function that was called upon the pressing of the button. This is similar to how the AJAX request was processed. The result of the calculation was then scripted directly to the HTML element and the innerHTML value was edited. Below is an example of this code implemented. As you can see in the URL, the HTML file was still requested from the server.



Subnet Calculator

Enter any IP address in the subnet and the prefix length.

Subnet Prefix
IP address / prefix length

137 . 146 . 140 . 0 / 16

Enter values between 0 and 255 for the bytes of the IP address and between 0 and 30 for the prefix.

Results

Description	Value
Subnet address	137.146.0.0
First host address	137.146.0.1
Entered address	137.146.140.0
Last host address	137.146.255.254
Subnet broadcast address	137.146.255.255
Netmask	255.255.0.0

Below is the console log. In this, you can see the requesting of the form, `clientform.html` and then the IP address is calculated without ever sending a request to the server. I then opened the `nodenjs.js` file we created for the project and calculated the IP address. Because this file still has the server process the request, there is a request for `/prefcalc`. This demonstrates that the HTML embedded script contains the calculation for IP address.

```
bmwebb20@glain:~/Senior Year/Fall 2019/CS331/p3/code$ node node.js
Server running at http://localhost:8000/
Request for /formnjs.html      Value
Request for /style.css         137.146.4.0
Request for /clientform.html   137.146.4.1
Request for /style.css         137.146.4.32
Request for /formnjs.html      137.146.7.254
Request for /prefcalc          137.146.7.255
```

I also worked to speed up how I calculated the IP address. In my first implementation I used a fair number of C++ packages to do the prefix math. For this one, I realized it could pretty easily be done just by using bitwise comparison. In this way I rewrote the lines of code that were needed to calculate the subnet mask. I also switched from writing in C to writing in JS to improve compatibility with the webpage itself. Below are screenshots first of subnet calculation in C and then in JavaScript.

```
// Find mask from prefix
ma = 256 - pow(2, 8 - std::min(prefix,8));
if (prefix > 8) {mb = 256 - pow(2, 8 - std::min(prefix-8,8)); } else {mb = 0;}
if (prefix > 16) {mc = 256 - pow(2, 8 - std::min(prefix-16,8)); } else {mc = 0;}
if (prefix > 24) {md = 256 - pow(2, 8 - std::min(prefix-24,8)); } else {md = 0;}
```

```
var prefix = (0xFFFFFFFF ^ (1 << (32 - query.preflen)) - 1);
var ma = (prefix >> 24) & 0xFF;
var mb = (prefix >> 16) & 0xFF;
var mc = (prefix >> 8) & 0xFF;
var md = (prefix) & 0xFF;
```

The first one is much slower because of the slow decimal math that I was doing. I also inverted how the bytes were oriented to make them more easily understood. My second implementation created the full mask for the IP address and then I shift each byte one at a time and do a bitwise comparison between the right byte and 0xFF which is much steadier and easier for a human to comprehend how the math was done.

I additionally wanted to play more around with user interfacing. To do this I decided that I wanted to add dark mode from iOS compatibility into the HTML content of my website. This was done by building a css style sheet. I found the relatively simple answer from [this](#) website. Essentially within a CSS file, the media content preference can now be requested. The media preference for dark more in my CSS stylesheet was created to look like such:

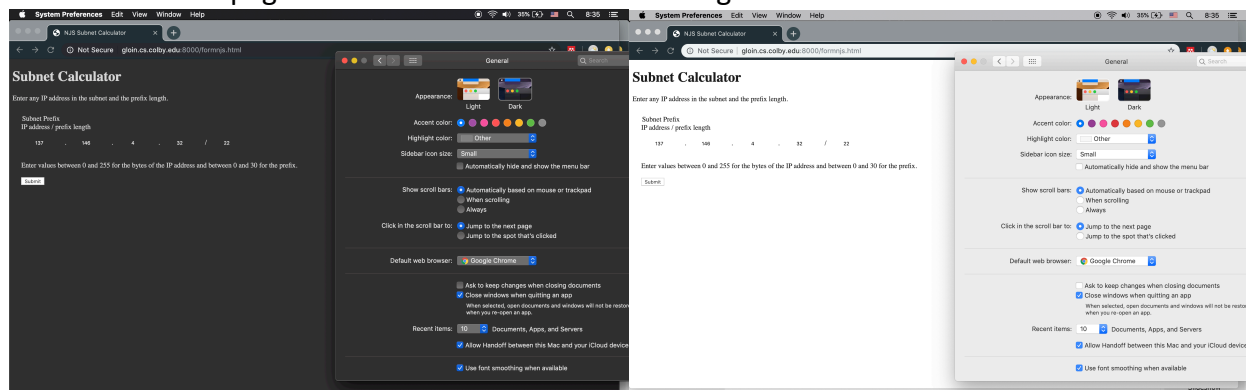
```
@media (prefers-color-scheme: dark) {
  body {
    background-color: rgb(52,51,52);
    background: rgb(52,51,52);
```

I also created a preference sheet for light mode in case I wanted to create some unifying changes among the two implementations. Doing this required updating how the server processes requests because now the only type of information was sent was not text/html. When I first wrote my CSS file, I noticed that nothing was changing but the developer console was giving me an error that my MIME formatting was not telling the client correctly what type of information was being sent by the server. To fix this, I used [this](#) solution that I found on StackOverflow allowing for the declaration of the MIME format to be automatically evaluated. The code do this that looks like this:

```
var dotoffset = req.url.lastIndexOf('.');
var mimetype = dotoffset == -1
  ? 'text/plain'
  : {
    '.html' : 'text/html',
    '.ico' : 'image/x-icon',
    '.jpg' : 'image/jpeg',
    '.png' : 'image/png',
    '.gif' : 'image/gif',
    '.css' : 'text/css',
    '.js' : 'text/javascript'
  }[ req.url.substr(dotoffset) ];
resp.setHeader('Content-type' , mimetype);
resp.write(data.toString());
```

This also means that if I use my node.js script in the future, the compatibility with more than just text/html MIME formats will be already built in and compatible.

Below is the webpage both while on Dark Mode and Light Mode.



I also created more style preferences within the CSS and HTML file like the removal of borders around a lot of the objects. While this work was primarily working with the aesthetics of the page, it was my understanding of MIME and text transfer protocols that made it very easy to debug where my code was failing, which was exciting!