CLI Messaging Application

Nir Boneh

University of Colorado Boulder
nir.boneh@colorado.edu

Abstract

This project discusses the creation and design of a decentarlized CLI unix-based messaging application written in C. Examining closley on how to elimnate security risks and interospecting on the usefullness of the application in various situations. The advantages of using this application over other similiar application in areas such as security, practicality, and speed is also discussed. Charts showing speed comparisons of sending messages over lan or wifi are also shown. The source code for the project can be found at https://github.com/nboneh/CLIMessageApp and the repository can be used for downloading and installing the application.

I. Introduction

Instant messaging has become one of the most practical ways we communicate with one another. In this day and age, instead of having to send mail across the physical plane and wait one to four business weeks to hear back from family relatives or friends, we can simply pull out our mobile device, type a message and send it in nearly an instant. While such power as reduced our face to face communication [1], one can't deny the advantages this tool could bring working within a unix terminal. For us developers and UNIX enthusiasts, spending most of our time on a terminal managing files and building application; A CLI tool that allows messanging quickly from the terminal, would be nice to have added to our vast arsenel of gadgets that are within our efficient terminal fingertips. Not to mention such an application would be extremely practical in a situation where we are limited to just a terminal, such as with special types of machines or on remote ssh. While a great messaging CLI application does exist, called telegram [2], it does not serve the purpose of being a decenteralized application. That means that big brother could be watching your messages not to mention that security attacks could happen to the the telegram server which is stated to be vulnerable to MITM attacks according to a tech.eu article [4]. The other risk is that the server could go offline rendering the application useless. There is also a great chat application that is decentralized called chat terminal [3]. While it works great it's missing the key advantage of messaging which is being able to send a message while the other user is offline which could come in handy. Therefore, a messaging application that is decentralized is missing from the terminal tool arsenel which is the main focus of the CLI messaging application.

II. Design

The CLI messaging application will be programmed in C. There will be three main components to designing the CLI messaging application.

1. The CLI features, frontend
2. Communication between nodes
3. Avoiding security risks

II.1 The CLI features, frontend

A user will interact with an interactive CLI, it will provide a set of commands and features. The help command will let the user see all available commands and a description of how to use them. The user will be notified about this command when the user types in a wrong command. The add command that will take in two paramaters one being the IP and the other being the nickname for the user. This will add a new friend that you can quickly setup communication with. The message command which will take either an ip address or a nickname and establish a communication messaging channel with that IP address. Once a messaging channel has been established one can see all previous messages with that IP address. The delete command will let one delete all previous messages with the user that is currently being messaged with. The quit command will also be avialable which will exit the application either by calling quit or pressing CTRL+D. The CLI will also notify the user when one of his added friends goes online or offline and be notified when a new message is received.

II.2 Communication between nodes

A communication between two nodes will be done via TCP/IP connection using socket programming. Currently the application is limited to being able to establish a messaging channel between you and one other user, but might be expanded to group chat in future releases. Sicne a TCP connection cannot be established if another node is offline, the messaging will work as follows. You can send a message to an offline user, but he/she will only receive it later when both of you are online. This also makes the delete command more powerful, if the other user is offline you can delete the messages that you have sent.

```
ogr2ogr -f GeoJSON -t_srs EPSG:4326
counties.geojson cb_2013_us_counties.shp
```

What is important to note here is the use of EPSG:4326 as the spatial reference which is used for open-street map projection. Initially, there was also the idea of zip-level subdivision. This was because both the hate-group data and the USCB data are provided in zip-level, but the geojson file ended up being around 5 gb (too large for a web-app). Two solutions were proposed, but denied. The first was simplifying the data, but that ended up making the zip-codes sort of an odd improper shape and the second was using a database, but due to time and money constraints this idea was also denied.

II.3 Hate Data

Getting the hate-data from the SPLC was relatively easy, but it was set-up in a csv format which is incredibly inefficient to use when it comes to JavaScript.

```
CHAPTER NAME,CITY,STATE,MOVEMENT CLASS,ZIP
Aryan Nations 88,Ashland,AL,Neo-Nazi,36251-0663
Aryan Nations Knights of the Ku Klux Klan*,
Ashland,AL,Ku Klux Klan,36251
```

Java code was hacked up for conversion:

```
JSONArray json = new JSONArray();
while ((line = br.readLine()) != null) {
    String[] split = line.split(",");
    JSONObject entry = new JSONObject();
    entry.put("zipCode", split[0]);
    entry.put("class", split[1]);
    .
    .
    .
    json.put(entry);
retunr json;
```

The java code took the csv file split each line by commas and put the results into a JSONArray which looked something like this:

```
[
    {
        "chapterName": "Aryan Nations 88",
        "city": "Ashland",
        "state": "AL",
        "class": "Neo-Nazi",
        "zipCode": "36251-0663",
    },
    {
        "chapterName": "Aryan Nations
        Knights
        of the Ku Klux Klan*",
        "city": "Ashland",
        "state": "AL",
        "class": "Ku Klux Klan",
        "zipCode": "36251",
    },
    {
        "chapterName": "Brotherhood
        of Klans Knights of
        the Ku Klux Klan",
        "city": "incomplete",
        "state": "AL",
        "class": "Ku Klux Klan",
        "zipCode": "36104",
    },
    {
        "chapterName": "Confederate
        Hammerskins",
        "city": "Huntsville",
        "state": "AL",
        "class": "Racist Skinhead",
        "zipCode": "35801",
    },..
]
```

This looks good, but crucially what is missing here for serving data on a map is the latitude and longitude information. Luckily, GoogleâĂŹs map API for JavaScript provided a geocoder for turning an address, even zipcodes, into latitude and longitude. Running a small JavaScript code with the geocoder, latitude and longitude values were added to each element in the array.

```
[
    {
        "chapterName": "Aryan Nations 88",
        "city": "Ashland",
        "state": "AL",
        "class": "Neo-Nazi",
        "zipCode": "36251-0663",
        "lat": 33.243964,
        "lng": -85.84503
    },..
]
```

This format makes it incredibly easy with a single for loop to add all the markers needed to the map. It should also be noted that during this step, icons from the SPLC were added to the project in order to display a distinct icon image for each hate-group class. Next, a dataset for the hate-groups for the state-level of the map was needed to be created, because displaying 1002 hate-groups at once is too expensive.

```
[
 {
        "state": "WA",
        "count": 10,
        "lat": 47.7510741,
        "lng": -120.74013860000002
    },
    {
        "state": "OR",
        "count": 9,
        "lat": 44,
        "lng": -120.5
    },
    {
        "state": "CA",
        "count": 57,
        "lat": 36.778261,
        "lng": -119.41793239999998
    }...
]
```

Each object represents a state and the number of hate groups inhabiting it. A small script was ran in JavaScript to convert the zip codes into states using GoogleâĂŹs geocoder and grouping the hate-groups to each state, latitude and longitude was also found in the process.

II.4 United States Census Bureau Data

This was by far the most difficult data to format. The java code to format the data looked very similar to the hate-group java code except this time there were three different csv data to format and they were quite large. Using metadata it was possible to figure out which columns were needed to be added into the JSON data. However, there was one key difference between the hate-group data json format and census data format and that is the address to object format:

```
{
   "00601":{
      "hawaiian":0.0,
      "female":51.1,
      "nativeAmerican":0.4,
      "male":48.9,
      "white":93.1,
      "black":3.1,
      "asian":0.0,
      "population":18570.0
   },
   "00602":{
      "hawaiian":0.0,
      "female":50.9,
      "nativeAmerican":0.3,
      "male":49.1,
      "white":86.7,
      "black":5.3,
      "asian":0.1,
      "population":41520.0
   },...
}
```

The reason for this is that the jsonobject are now setup like a hash map, making look up times when a user clicks on a section of the map constant, instead of linear which is very efficient. The last thing that was needed to be done was to convert the data into state and county level instead of zip and so again using JavaScript code with google geocoder, that process was complete.

```
{
    "10001": {
        "population": 116522,
        "male": 56040,
        "female": 60482,
        "nativeAmerican": 852,
        "black": 29782,
        "white": 76371,
        "hawaiian": 53,
        "asian": 2745,
        "other": 1796
    },
    "10003": {
        "population": 564828,
        "male": 273390,
        "female": 291438,
        "nativeAmerican": 1594,
        "black": 133501,
        "white": 371967,
        "hawaiian": 147,
        "asian": 23720,
        "other": 206
    },..
]
```

All the necessary JSON files were created for making an efficient interactive web-map, but a server to host and serve data is still in order.

III. Server-side coding and hosting

There are two main reasons for creating a web-app rather than serving static HTML web-page.

1. Being able to serve some of the heavier JSON files in a http get request rather than including them in an HTML tag. Reason for this being is that the client side JavaScript can send an Ajax call 'asynchronously' saving initial loading time of the webpage.
2. Easier deployment to a hosting server. By having web-app itâĂŹs easier to configure large number of files to a serviceable public webpage.

Ultimately, the web-app framework that was decided on was nodeJS. NodeJS is easy, clean, simple, written in JavaScript with minimal configuration that needed to be done. The server can easily be started on a localhost for testing by first running the 'npm install' command and then calling on 'node index.js'. The main functions that were needed to be added to the server side code were the http get methods.

```
app.get('/counties.json'
, function(req, res) {
  fs.readFile('public/data/counties.json'
  , function(err, data) {
    res.setHeader('Content-Type'
    , 'application/json');
    res.send(data);
  });
});
```

So now calling on the serverurl/counties.json through a browser, will serve up the large json file. The reason this is useful again, is for 'asynchronously' loading the data from the client side which saves initial loading time of the webpage. The next step was deploying the web-app to a server, so the map can be viewed from a public url. A simple server hosting service which was used was heroku. After installing heroku and with the use of a git repository running heroku create followed by git push heroku master deploys the web-app. The web-app can now be accessed publicly through `https://hate-map.herokuapp.com/`. The last step and most crucial step to getting the interactive-map working is setting up the HTML webpage with the necessary functionality of client-side JavaScript.

IV. Client-Side Javascript and HTML

After setting-up all the necessary components as the basis for building the map, it was time to actually configure and display the map.

IV.1 HTML and CSS

HTML and CSS are the view components of the map and are responsible for design rather than logic. The html only has two major components, the map which takes up the entire screen and the legend which is responsible for serving up statistical information about the areas clicked. The legend has two possible tags that can fill it, either the data div or the no data p tag in case of missing data. It also includes a loading spinner in the center which is displayed for data that is still being loaded from an Ajax call. Data files that were small enough i.e. less than 1 megabyte, were simply added in an include tag. This is also the section that includes libraries that are going to be used including leaflet and GoogleâĂŹs chart API.

```
//Include tag
<script type="text/javascript"
src="data/econstate.json" ></script>

//CSS descrinbing the legned
#legend {
    border-style: solid;
    border-width: 1px;
    border-radius: 25px;
    border-color: rgba(81, 181, 229, 1);
    background-color: rgba(255, 255,
    255, 0.2);
    padding: 5px;
    display: none;
}

//Simplified html with the map,
//the loading spinner, and the legend
<div id="map"></div>
<div id="countyLoad" class="spinner"
style="visibility:hidden" > </div>
<div style="position: absolute;
    bottom: 5%; left: 5%;" align="center">
    <div  id="legend"  >
    <h2 id="name"> </h2>
    <p id="nodata" >No data available</p>
    <div id="data">
     <h4 id="population"></h4>
```

IV.2 Client-Side Javascript

The last step to creating the map and the core logic of how the map is going to operate, client-side javascript. The first step is to initialize both the google pi chart API and the Leaflet Map-API.

```
google.load("visualization", "1",
{packages:["corechart"]});
window.onload = function () {
map = L.map('map').setView([38,-105], 4);

//Adding open street map layer
L.tileLayer('http://{s}.tile.osm.org
/{z}/{x}/{y}.png', {
attribution: '&copy;
<a href="http://osm.org/copyright">]
OpenStreetMap</a> contributors'
}).addTo(map);

map.options.maxZoom = MAX_ZOOM;
map.options.minZoom = MIN_ZOOM;
//Restrict viewing to only united states
map.setMaxBounds([[0, -180], [75, -40]]);
```

This is when things become tricky, two layers are created. One for state level and the other for county level. Each layer consists of markers and geoJson, this is also the point where an ajax call is sent to the server to retreive the county geoJson data. A listener on the map zoom changed is also set, once the zoom-level goes beyoned a certain number the layers are switched, this ensures that the county level isn't overloading the map due to it being displayed in a low zoom-level.

```
//The ajax call
$.getJSON( "counties.json"
, function( counties ) {
    countyGeoJson =
    createGeoJsonLayer(counties);
    countyLayer
    .addLayer(countyGeoJson);
    document.getElementById
    ('countyLoad').style.visibility
    = 'hidden';
})


//The zoom listener
map.on("zoomend", function(e){
var zoom = map.getZoom();

if(!STATE_LEVEL && zoom < 7 ){
    STATE_LEVEL = true;
    map.addLayer(stateLayer);
    map.removeLayer(countyLayer);
} else if(STATE_LEVEL && zoom >= 7) {
    STATE_LEVEL = false;
    map.addLayer(countyLayer);
    map.removeLayer(stateLayer);
    }
});
```

The last step involves implementing the on feature click function which happens when a layer is clicked on after we set a listener for the map. This is the part where specific data is being served to the user and extracted from all the data that was created in the first section.

```
if(STATE_LEVEL){
    census = censusstate;
    econ = econstate;
    edu = edustate;
    code =  this.feature.properties.STATE;
} else{
    document.getElementById("name")
    .innerHTML =
    document.getElementById("name")
    .innerHTML
    + " County";
    census = censuscounty;
    econ = econcounty;
    edu = educounty;
    var fipLength =
    this.feature.properties.GEO_ID.length;
    code  =
    this.feature.properties.
    GEO_ID.substring
    (fipLength-5, fipLength)
}

var censusData = census[code]
if(censusData == undefined){
    //If data undefined show no data
    document.getElementById("nodata")
    .style.display = 'inline';
    document.getElementById("data")
    .style.display = 'none';
    return
}
document.getElementById("nodata")
.style.display = 'none';
document.getElementById("data")
.style.display = 'inline';

var population =
censusData.population

//Calculating male percent
//to two decimal places
var malePrecent =
((censusData.male/population)*100)
.toFixed(2);
var femalePerecent =
(100 - malePrecent).toFixed(2);

document.getElementById("male-percent")
.innerHTML = malePrecent + "%";
document.getElementById("female-percent")
```

Layer levels

`state.png`

Figure 1: *State level, markers show number of hate groups in each state.*

county.png

Figure 2: *County level, markers show individual hate groups with icon representing their class. When markers are clicked a caption appers, which displays the name of the hate group and includes a link leading to an SPLC page that describes the hate group class.*

```
.innerHTML = femalePerecent + "%";
//Will print population with commas
document.getElementById
("population").innerHTML =
"Population: "
+censusData.population.toString()
.replace(/\B(?=(\d{3})+(?!\d))/g, ",");

//Setting up pi chart
var data = google.visualization
.arrayToDataTable([
    ['Race',
    'Population'],
    ['White',
    censusData.white],
    ['African-American',
     censusData.black],
    ['Native-American',
    censusData.nativeAmerican],
    ['Asian',
     censusData.asian],
    ['Hawaiian',
    censusData.hawaiian],
    ['Other',
    censusData.other  ]
]);

var options = {
    title:
    'Racial Distribution',
    'chartArea':
    {'width': '100%', 'height': '70%'},
    'legend':
    {'position': 'bottom'},
    backgroundColor:
    'transparent'
};
var chart =
new google.visualization
.PieChart(document
.getElementById('censusPieChart'));
chart.draw(data, options);
```

This takes care of most of the core functionality needed to create the interactive-map.

I. Results and Conclusion

The map is publicly served on heroku and can be found at `https://hate-map.herokuapp.com/`. Hopefully, the SPLC and/or users looking for information about hate-groups in America will find it useful.

References

[1] Hemmer, Heidi. "Impact of Text Messaging on Communication." Minnesota State University, Mankato, 2009.

[2] "Telegram âĂŞ a New Era of Messaging." Telegram. N.p., n.d. https://telegram.org/

[3] "LAN Chat and Text Conferencing in Easiest Way with Vypress Chat." Chat Terminal. N.p., n.d. https://telegram.org/

[4] Wauters, Robin. "Supposedly Super Secure Telegram App Is Vulnerable to MITM Attacks, Cybersecurity Expert Claims." Tech.eu. N.p., 29 Apr. 2014. http://tech.eu/brief/supposedly-super-secure-telegram-app-possibly-vulnerable/