CS249i: The Modern Internet **Winter 2024**

Project 2: Web and Content Delivery

Parts 1–2 Due: February 16 11:59pm PT **Part 3–4 Due:** February 23 11:59pm PT

Introduction

In this project, you will instrument a headless web browser to collect the DOM tree of top websites on the Internet.

From there, you will document the technical services and resources that make up popular websites, ultimately uncovering the technology providers that make up the modern web.

You will leverage the Linux virtual machine (VM) that you were provided in Project 1 to crawl websites using Headless Chrome, a version of the Chrome browser that can be programmatically manipulated. You will need to instrument this browser using existing browser instrumentation tools (i.e., Puppeteer) to crawl a list of websites we provide you, collect metadata about them, and answer questions about how modern websites function.

Internet Measurement Caveats

As in the previous assignment, you'll find that questions about the Internet do not have a single exact or correct answer, as the real-world Internet is nuanced and constantly changing. Thus, it is important that you and your teammates describe the methodology you use to answer each question in addition to providing your final answer.

The Ethics of Web Crawling. In this project, you will be making *real* requests to websites using a crawling infrastructure that you build. As with any active scanning methodology, there are many ethical considerations at play. For this project, you MUST ONLY crawl websites that are in the list we provide to you, as these are very large, popular websites for which the traffic you generate is negligible. You are permitted to scan multiple times, but see that you limit this to the best of your ability; too much unwanted traffic could get your IP address blocked by websites you are trying to reach and may inhibit your ability to complete the assignment.

The Importance of Testing And Starting Early. At one part of this project, you will be asked to run a web crawl on a set of 1000 websites. These crawls may take a significant amount of time—on a single core, running a crawl of 1000 websites with a timeout window of up to 30 seconds per website can take upwards of 4 hours. You want to ensure that your code is generating everything you will need before running on the entire set of 1000 domains. We recommend you test your code thoroughly, and run it through smaller inputs (e.g., the first 25 domains) to make sure you are catching any potential edge cases that you may not anticipate.

Web Background

In this section, we detail the necessary background to understand the components of websites we'll be investigating and the high-level function of web browsers.

Web Pages and Resources

When your browser makes an `HTTP` request to a website, it will first download the HTML content of the page. The browser abstracts the HTML content as a Document Object Model (DOM), which is a nested tree structure where each object is part of the larger HTML content. Typically, websites will embed many different types of resources in the DOM, such as images, CSS, and JavaScript, and many of these will be served *remotely*, meaning your browser will need to fetch these resources via `HTTP` to properly render the page. These are called *web resources.*

Web resources are typically retrieved over the Internet, and are thus served from a myriad of ASes and locations around the world. In this project, you will be investigating the types of resources modern websites rely on, where those resources come from, and the implications of your findings on the web at large.

Web Browsers

Web browsers are the primary way that people interact with the web. They handle all the network-side and client-side code required for websites to properly function. Google Chrome, in particular, exposes an API called the Chrome Development Protocol (CDP) that enables developers to *instrument* the browser to perform automated tasks. For example, through CDP, developers can automatically visit webpages, interact with web elements, and log all requests and responses made during a page visit. Essentially, CDP exposes almost everything you can do as a regular user with your browser, just programmatically.

Part 1: Collecting Web Data

In the first part of the project, you'll need to instrument a web browser to log HTTP response traffic and annotate each resource with metadata, such as the URL it's served on, its content type, and the AS that serves it.

Required Software

For this project, you will primarily be writing JavaScript and Python. Do not be alarmed if you do not have experience programming in either of these languages before, as the assignment does not rely on a deep knowledge of the languages themselves. We will be using two main software packages for this project:

1. **Puppeteer**[1] is a JavaScript library that enables you to instrument a headless web browser (i.e., Google Chrome) to perform various browser tasks. Puppeteer is a lightweight wrapper around the Chrome Development Protocol. You can use Puppeteer to take screenshots of webpages or automate large scale web-application testing. We will be using it to measure the complexity of modern webpages.
2. **Pyasn**[2] is a Python library that enables fast lookups from IP addresses (or IP blocks) to the ASes that serve them. It can interface directly with the MRT that you collected from Project 1, or you can use it to download a fresh routing table from the RouteViews project.

[1] https://github.com/puppeteer/puppeteer
[2] https://github.com/hadiasghari/pyasn

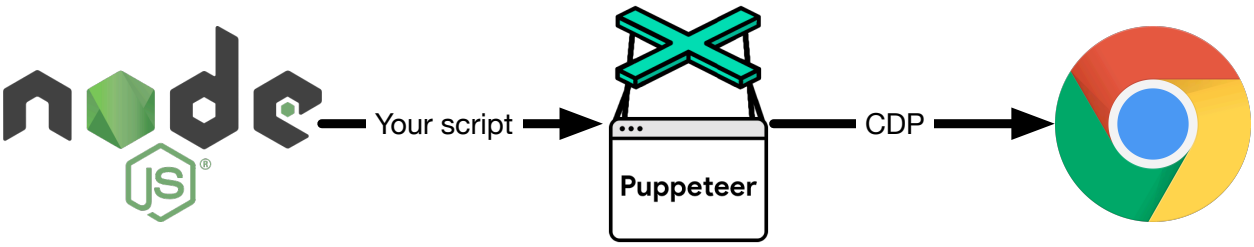

Figure 1: **Puppeteer Flowchart**

Setting up your VM for Web Crawling

To set up your cs249i VM for web crawling, you'll need to install a few packages and libraries.

First, in order to connect to the internet on your VM without having to run `gobgp`, run:

```
cs249i-student@cs249i-group-XX:~$ sudo ip route add default via 171.67.69.32
```

This will establish a default route to the upstream router without needing to read it in from an active BGP connection. You won't be able to advertise other IPs that belong to you anymore, but that's okay; you only need your own IP for this project.

To begin, download all the prerequisite files onto your VM with the following commands:

```
cs249i-student@cs249i-group-XX:~$ mkdir project2 && cd project2
cs249i-student@cs249i-group-XX:~$ wget https://cs249i.stanford.edu/projects/p2/p2.tar.gz
cs249i-student@cs249i-group-XX:~$ tar -xvf p2.tar.gz
```

These commands download a few files to your VM: `setup.sh`, `starter_code.js`, and `domains.js`. We will explain each in detail below.

To download all the prerequisite software, run the following command from the same directory you downloaded the starter code to:

```
cs249i-student@cs249i-group-XX:~$ chmod +x setup.sh
cs249i-student@cs249i-group-XX:~$ ./setup.sh
```

The setup script will download Puppeteer, the necessary libraries for the headless browser to function and `Pyasn`. When Puppeteer is installed, it also downloads a recent version of the Headless Chrome browser to your VM, which is how the library exposes the web crawling APIs it provides. You should ensure you have properly installed all the prerequisite software before proceeding—you may need to update node and npm on your machine.

Instrumenting the Browser

In order to instrument the browser, you will need to write some JavaScript code that leverages Puppeteer to collect metadata about the resources loaded when you visit a page. Figure 1 shows how your code will eventually interface with the browser. You will run your code via the `node.js` runtime, which will have direct access to Puppeteer and the headless Chrome binary. Puppeteer exposes an API[3] that allows you to directly interface with the browser via the Chrome Development Protocol (CDP), which you will use to collect the information you need from page loads. Specifically, you can look in the `HTTPResponse` fields to collect the relevant metadata.

[3] https://pptr.dev/api

We provide you with some starter code, `starter_code.js`, which contains a scaffold for browser instrumentation. Your main objective is to extend this script to collect information about every HTTP resource loaded by the pages you visit. The code looks something like this:

```
const puppeteer = require('puppeteer');

runningDomains = ["https://cs249i.stanford.edu/"]

async function crawlSite(site){
    outObj = {}
    outObj["site"] = site

    // TODO: Populate these fields
    outObj["site_ip"] = "TODO"
    outObj["resources"] = []

    const browser = await puppeteer.launch();
    const page = await browser.newPage();

    page.setUserAgent(
          "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6)
          AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4182.0 Safari/537.36"
    );

    // This event is fired when the browser detects an HTTPResponse
    page.on('response', async (response) => {
        // TODO: Store response data here
        responseObj = {}
        responseObj["url"] = "TODO"
        responseObj["ip_address"] = "TODO"
        responseObj["content_type"] = "TODO"

        outObj["resources"].push(responseObj);
    });

    await page.goto(site);
    await browser.close();
    return Promise.resolve(outObj);
}

/*
 * README: runOverDomains() a wrapper function that runs crawlSite asynchronously.
 * So long as the "domains" variable is populated correctly, you should not need
 * to modify anything else in this function in order to get the script to work.
 */

async function runOverDomains() {
    for (let i in domains) {
        await crawlSite(domains[i]).then(function(result){
            console.log(JSON.stringify(result));
        }).catch(function(error){
            console.log(error);
        });
    }
}

runOverDomains();
```

At a high level, the starter code orchestrates the headless browser (Chrome) to visit `https://cs249i.stanford.edu/` and logs information about the page visit. To do this, it loads the Puppeteer library, and runs two functions, `runOverDomains()` and `crawlSite(site)`.

`runOverDomains()` is a wrapper function that calls `crawlSite` asynchronously and prints out a JSON blob of the returned result. You should not have to modify this function at all in order to capture the appropriate fields.

`crawlSite(site)` contains the main logic of the script and is where you will have to add additional code. Specifically, you will need to leverage the Puppeteer API[4] to fill in the fields marked `TODO` in the starter code. We have already written the code required to capture *when* an `HTTPResponse` event is fired by the browser, all you need to add is how to collect the relevant metadata about each resource.

In summary, your code should collect the following pieces of metadata about the website:

1. The IP address of the root page of the website you are crawling

And for each resource:

1. The URL of the remote resource
2. The IP address of the remote resource
3. The content type of the remote resource

Tagging Resources with AS

After collecting a JSON object containing resource metadata for `https://cs249i.stanford.edu/`, you will then tag each resource with additional metadata, including the AS that the resource was served from.

For AS tagging, we recommend using `Pyasn`, a simple Python library that contains mappings from IP addresses to ASNs. You can reuse the `asn_names.txt` file we provided you from the previous project to also tag each resource with the name of the AS that serves it. `pyasn` can read the MRT file you collected from the previous project, or you can simply download a recent RIB file from RouteViews[5] using the tools provided by the `pyasn` library.

Evaluating Your Final Output

To test your final output, we provide a file, `cs249i_output.jsonl`, that contains the JSON output from our solution crawler. You should check whether or not you received the same resources (though the order and IPs may not match exactly).

Part 2: Crawling the Web

In this second part of the project, you will take the web crawler you built in Part 1 to crawl the top 1000 websites on the Internet, using data from Google Chrome's CrUX dataset. You can download the data from: https://github.com/zakird/crux-top-lists.

You should modify your JavaScript crawler to crawl these 1,000 websites. Note that some websites may produce *different* resources when you crawl them multiple times. This is expected, and a product of how dynamic the web has become. You do not need to crawl websites multiple times, but know that doing so may give you slightly different snapshots of the web. Additionally, a small handful of top websites may never load in your browser—this may be due to a number of factors, but for the scope of this project,

[4] https://pptr.dev/api
[5] http://www.routeviews.org/routeviews/

please use `"UNKNOWN"` as a placeholder for any data or IP addresses that fail to return. Do not exclude domains from your final output. You may want to log these errors and sanity check them after your crawl has completed.

Your objective will be to generate a single JSON Lines file called `output.jsonl` that contains full resource metadata for every website in the list, delimited by a newline. This will help with grading and your result will enable you to answer the questions listed in Parts 3 and 4.

Submission

Please upload the following files to Gradescope:

- `output.jsonl`: the file that contains the resources for all 1000 websites.
- `starter_code.js`: your modified starter code.
- Any other scripts you wrote (e.g., Python scripts to tag ASes)
- `README.txt`: A readme file that describes the files you submitted.

Please double check to make sure your final output is valid JSON Lines format.

Part 3: Understanding Web Resources

In this section, we will begin by investigating the most common resources that power the modern web. Please share your methodology (a high-level description will suffice) for answering each question.

1. What percent of sites were you able to successfully crawl? For the sites that failed, why did the crawl fail?
2. How many resources, on average, are loaded per site? What are the top ten sites that load the greatest number of resources? How many resources do those sites load? Why are so many resources loaded for these top sites?
3. What are the most common resource types across all websites in your crawl? You can determine this by investigating the content type of each resource loaded. What fraction of sites use JavaScript?
4. What are the top 10 most commonly included resources across all websites (as defined by file name)? How frequently do they occur in the Top 1000 websites, and what function do they serve?
5. What are the top *domains* that serve resources on the most websites? What fraction of websites rely on these domains? What are these domains being used for? Answer this in a table. We recommend using the Mozilla Public Suffix List[6] to group URLs by domain, for which there are many libraries.
6. For this and the next question, consider a third-party domain to be one that is not equivalent to the domain of the website you visited based on the public suffix list. For example: `google.com` and `blog.google.com` are first-party domains, whereas `google.com` and `googledomains.com` are third-party domains. How many websites depend on third-party content (i.e., a resource loaded from a third party domain)? On average, how many *third-party domains* do websites rely on? What are the top 10 most included third-party resources (as defined by URL), and what do they do?
7. What percent of sites load Javascript from third parties? How widely adopted is Subresouce Integrity? What attack can occur when SRI is not used? What are the ten most commonly included Javascript files from third-party domains and what role do those files play in the web ecosystem?

[6] https://publicsuffix.org/

8. What percentage of websites display ads? What about deploy tracking code? You can use the lists published at https://easylist.to/ to help answer these questions. Anecdotally, what patterns do you see emerge in the *types* of websites that deploy a large number of ad providers? What about patterns in the sites that display a large number of ads?

Part 4: Uncovering Web Centralization

In this section, you will investigate the *networks* that the web is most reliant on (versus the *domains* most relied on in the last question). Please share your methodology (a high-level description will suffice) for answering each question.

1. What are the top 10 ASes that are responsible for *hosting* the most websites in your crawl? How many websites does each top AS host? (hint: what is the IP address of the root page of the website itself?)
2. How many websites are hosted by ASN 13335? What AS is this, and what service do they provide?
3. On average, how many unique ASes host resources (both first party and third party) per website in your crawl? What ten websites rely on the largest number of unique ASes, and what websites rely on the least unique ASes?
4. What are the top 10 third-party ASes that websites rely on to serve resources, and what fraction of websites do they appear on? Consider a third-party AS to be one that is not equivalent to the AS that hosts the website.
5. What are the potential implications of having websites increasingly rely on a small number of ASes to serve and host content?