Spring 5-5-2015

GraphTracker: A User Opt-in Approach to Web Tracking

Kevin Daniel Aziz
Syracuse University

GraphTracker:
A User Opt-In Approach to Web Tracking

A Capstone Project Submitted in Partial Fulfillment of the
Requirements of the Renée Crown University Honors Program at
Syracuse University

Kevin Daniel Aziz
Candidate for a B.S. Degree
and Renée Crown University Honors
May 2015

Honors Capstone Project in Computer Science

Capstone Project Advisor: _____
Dr. Jae Oh, Professor of Computer Science

Capstone Project Reader: _____
Edmund Yu, Professor of Computer Science

Honors Director: _____
Stephen Kuusisto, Director

Date: May 8, 2015

Abstract

The web is a collection of interconnected documents and sites. A user will often jump not just from one site to another, but from one site to two or three others. How we explore the web is just as important as what we explore and what we discover as a result. In computer science, graphs are used to represent relationships between entities. By applying this concept to web tracking, graphs can capture the relationships between the sites we visit. We are building GraphTracker to tackle the difficult problem of tracking a user's web activity. GraphTracker builds a "browsing graph" to track the content a user accesses, and models how the user navigates the web. The browsing graph maps out the web as the user experiences it, to represent the user's personal browsing habits and interests. GraphTracker provides a valuable data source for modeling and understanding user behavior and makes that data source accessible, while also respecting user privacy.

Table of Contents

Executive Summary

Given the importance of digital data and privacy, web tracking has gained relevance over the last several years. Web tracking transcends the field of computer science, giving life to a new breed of digital advertising and spawning policy debates among lawmakers in Washington. Controversial means of web tracking, particularly tracking employed by individuals or groups of whom the user is unaware, threaten user privacy and trust. However, the data provided by web tracking is so valuable, that many are undertaking the challenge of exploring how to track users in a way that makes the user feel safe and secure, and directly benefits the user.

We've developed software called GraphTracker which advances the interest of responsible web tracking by putting that power in the hands of the user. Most methods of tracking originate from outside sources, often without the user's consent, thus putting user privacy at risk which leads to a loss of user trust. However, this can be ameliorated by putting the user in command, and making web tracking a user-driven process. The user must directly install and administer GraphTracker. The user has control over what data is being tracked, and controls what happens with that data. This puts the user, rather than a third party group, as the owner of the data, preventing data from being sold off to companies unknowingly.

Furthermore, from a technical standpoint, GraphTracker is able to access and collect a broader spectrum of data. Additionally, using the ideology that *how* a user gets from place to place on the web is important, we add another dimension to the metrics gathered, and at a level of detail which is difficult for third-parties to emulate. Thus, GraphTracker not only tracks the user in a way that is safe and trusted by the user, but it advances tracking technology by bolstering the quantity and quality of data collected.

GraphTracker works as a Google Chrome browser extension. The user can enable or disable web tracking with ease through a simple browser interaction. While GraphTracker is enabled, every time a user visits a site, or navigates from one site to another, GraphTracker retrieves browsing data about the page and navigational data about how the user is going from one page to the next. The extension sends the information to the user's personal storage location. The information can be viewed through the data visualization portal included with GraphTracker, or any other program which has been developed to leverage GraphTracker's data. We designed the program to allow other developers the opportunity to build their own programs on top of this platform using the data set logged by GraphTracker.

Most users of this current iteration will be highly technical users, typically developers looking to explore the benefits and potential applications of browsing data provided by web tracking. GraphTracker only tracks one individual, creating a small-scale, but highly user-specific data set. Therefore GraphTracker will not be useful in use cases such as providing data to large advertising networks, but will be useful in providing researchers with strong data sets when exploring new uses of web tracking as well investigating related user trust issues.

Acknowledgements

I owe my gratitude to Dr. Jae Oh, whose advice and mentorship over my undergraduate years has helped me to reach my goals as a computer scientist and has left his mark on time as a student. My sincerest thanks also extend to Edmund Yu and Dr. Susan Older, for their much needed input and encouragement. Finally, thank you to Eric Holzwarth and the Honors department for support throughout this journey and for making this possible.

Introduction

Data is power in computing. Much of computer science is predicated on the idea that well defined processes can be developed which systematically process and manipulate data. The field has evolved around performing this task faster, more efficiently, and on larger data stores. The problem arises of how to acquire this data in the first place.

Data concerning browsing history and browsing behavior is a particularly interesting category. A user's browsing history, for example, provides insights into a person's interests, wants, and needs. Browsing data can often contain very private personal information, usernames or passwords. These data, for their sensitivity, have become the focus of debate, especially over which information is collected, how it is collected, and who is collecting it. Tracking a user across the web is controversial for good reason, the information can be very dangerous in the wrong hands or when used for the wrong reasons.

However, it is possible to collect useful, safe data at the discretion of the user. We built GraphTracker on the premise that both what the user explores and how they access and discover that content are important. The system leverages a browser extension to build a "browsing graph," a directed graph which details user interactions while browsing, plotting the user's course as he or she explores the web. The browsing graph provides a model of user behavior indicative of user habits, interests, and exploratory patterns. The relational information between the pieces of web content offers additional context when drawing inferences from browsing data. The information collection and storage is kept local to the user's computer and the user is provided with a toolkit to leverage the collected data in beneficial ways. This puts the user in control of their own information. GraphTracker tackles problems in tracking users on the web,

delves into the nuances of data acquisition and manipulation, and shows that it can be done in a user-driven manner.

First we introduce the topic of web tracking in section 1, covering cookies, tracking pixels, and current protections from web tracking which are in place for user privacy. Section 2 will discuss GraphTracker and specify the browsing data which GraphTracker implements and how it is represented. Section 3 will cover the system architecture as well as design and implementation details each component.

Section 1

Background

Not long after the advent of the web, it became clear that tracking users would be advantageous to understanding a user's habits or interests. Companies have long understood the value of knowing their target demographic. This is easy with the personal interaction that occurs in offices or brick and mortar stores. The web makes the user invisible to the company, but web tracking lifts the veil. Web tracking informs companies of user interests, what the user is likely to do online, and how the user reaches companies' websites. Knowing this information puts companies in a position to cater their goods or services to the user, providing a better user experience. However, many web trackers have ulterior motives, seeking collect user data which it will sell to companies. The duality of intention draws concerns of protecting users from undesirable web tracking.

Cookies

Cookies tend to be at the heart of most tracking techniques. A server may set a cookie in a user's browser. When a user accesses a page, the server which hosts the page may retrieve any cookies that it has already set, thus identifying a returning user [1]. The lack of a cookie when the user visits a page is therefore an indicator that the user has not visited the page previously, or has deleted cookies since the last visit. A cookie can carry a unique identifier for the user which can allow a site to keep the user logged in, maintain user preferences, or keep items in a shopping cart for future visits.

Cookies typically expire when a browser session ends, but they can also be created with a lifespan specified by the server which set the cookie. These cookies, referred to as persistent cookies, can last over the course of multiple browser sessions [2]. While they can benefit the

user, such as allowing the user to stay logged into their email for an extended period of time, they can also be subversive, allowing a tracking program to identify the user over a number of browser sessions [3].

Domains only have access to the cookies which they set. Users tend to see most benefit from first-party cookies. These are cookies set by the domain which the user is currently visiting, and allow the site to "remember" the user for a more pleasant experience on the site. Third-party cookies, which form another category, are cookies set by domains other than the currently visited site. Sometimes this third-party contributes some beneficial service to the user, such as enabling a third-party video plugin or a social media widget on the site. Other times, it serves a purpose with no real or immediate benefit to the user, which is often the case with third-party web tracking.

Tracking Pixels

Tracking pixels provide a way for websites to track a user's visits to the site. They are usually implemented with `<img>` tags on a site. The URL of the image referenced by the tag will typically return a transparent, single pixel image with height and width parameters set to zero or the visibility setting set to "hidden," so it will not display [4]. Even though it does not visibly appear on the page, a request for the image is sent to a server. This request, which occurs unbeknownst to the user, carries user information such as the IP address of the user, and cookies set by that server. The image URL can also include parameters which encode additional information such as an email address or product identifier. Since it is a simple image HTML tag, this technique can also be used in emails. An email tracking pixel being pinged indicates that the recipient has received and opened the email.

Pixels enable tracking users across multiple domains. Tracking pixels can be on any number of pages on any domain, but the target URL of each pixel can point to the same server. Thus, the server pinged by the tracking pixel has access to the same third-party cookie across all those sites. This is how ad networks often track ad impressions or how affiliate sites track users across all sites which fall under their umbrella.

For example, consider two affiliate online stores hosting their sites on separate domains. Tracking pixels placed across pages on both domains can point to a single server. Thus, both sites would have access to the same tracking cookie which could then be used to implement a shared shopping cart between the two sites.

This method of user tracking is driven by the content provider and the tracking entity, though as in the example above, these two entities can be one and the same. The content provider must willingly embed a tracking pixel provided by the tracking entity. This is dependent on forming a mutually beneficial business relationship. This codependence limits the breadth of a tracking network to the sites which embed that network's tracking pixel.

Current Privacy Protection Methods

The growth of the market for user data has caused a growth in the number of third-party tracking companies. These third-parties often engage in tracking without user consent, which is a privacy concern, and furthermore often provide no benefit to the user, sometimes even tracking the user simply to sell user data to other companies. As third-party tracking continues to encroach on a user's privacy and sense of trust online, policy debate has spurred on in efforts to protect consumers' right to privacy.

One such method of protection, opt-out methods, are questionable at best. The user must first become aware of the tracking entity and then seek to opt-out, which is often a difficult task.

There is also the question of whether or not the opt-out policy has actually been implemented. Companies have been caught tracking users even after the user opts-out, which has led to legal battles and the destruction of user trust. Another method, blocking, employs blacklists to prevent web traffic to tracking sites. These can be technically difficult to administer and are only as strong as the corpus of tracking sites on the blacklist [3].

Perhaps the most successful method yet, has been “Do Not Track” (DNT). DNT adheres to the philosophy that the user should decide whether or not they wish to be tracked, and it is up to third-parties to honor that wish. The W3C is in the process of standardizing DNT under the “Tracking Preference Expression” specification, which defines a DNT request header to identify a user’s tracking preference [5]. The specification would allow for users to set a tracking preference in their browser. Web pages and third-parties would receive this preference in page request headers and respond appropriately. All major browsers currently support DNT.

The popularity of DNT makes the consensus clear that consumers should be in power of how they are being tracked. This bolsters motivation for GraphTracker, which enables self-tracking governed by the user.

Section 2

GraphTracker

GraphTracker is a user-driven web tracking system. The primary mode of tracking is a chrome extension and does not require cookies, which allows a user to track and log activity while preventing unwanted third-parties from doing so. GraphTracker's currency is the browsing graph, a graph representing the user's browsing data.

Browsing Graph

Traditionally, browsing data is synonymous with web history, both representing a collection of web pages with metadata, content information, and the timestamps of when the page has been visited. The browsing graph augments this data by including relational information between pages such as how the user navigated from one page to another. By focusing on the context around how a user got to a page, the browsing graph builds a deeper understanding of a user's browsing patterns.

Other tracking programs are limited in their ability to define the referring page. The referrer specified in an HTTP request can specify only the domain and not the full URL. GraphTracker is present to record each page exit and page load, and can capture the full URL of the referrer when creating a relationship between two sites in the browsing graph. Thus instead of connecting a site to its referring domain, as most tracking networks are capable of doing, the browsing graph connects URLs directly to the referring URL, which provides an improved and more granular level of detail.

A graph data structure by definition is a collection of nodes and edges. In the case of the browsing graph, the nodes encapsulate the standard browsing data for a web page. I will refer to these as PageNode objects, formally defined as the following structure:

```
PageNode {
     String url: “http://www.google.com”,
     String title: “Google”,
     Integer visitCount: 400,
     Timestamp[] history: [....],
}
```

The URL is the unique identifier of the PageNode. This is the URL which appears on the address bar of the browser and is obtained by accessing the `window.location.href` variable. The title property adds context and gives insight into the content of the page and is accessed through the `document.title` variable. The visitCount and history properties are used to track the visits the user makes to the site. Each time a user visits a site, the visitCount is incremented and the current timestamp is added to history.

The edges of the browsing graph represent a link from one web page to another. It can really be any form of navigation from one page to the next, the most obvious example of which being a hyperlink which a user clicks to navigate away from one page to the next. Another common example is a redirect. I will refer to these as Edge objects, formally defined as the following structure:

```
Edge {

     String source: “www.blog.com/weather”,
     String destination: “www.weather.com”,
     String type: “hyperlink”,
     String context: “weather”,
     String title: “It better be sunny.”,
     Integer visitCount: 5,
     Timestamp[] history: [....],
}
```

The source and destination represent the URLs of the PageNodes which this Edge connects, forming a directed edge which points from the source to the destination. The context and title variables provide valuable information that can identify the nature of the relation

between two pages. The context is gathered from the visible text of a hyperlink and the title is taken from the title attribute of the <a> tag. For example, the following hyperlink would generate the sample Edge above:

```
<a href=”weather.com” title=”It better be sunny.”>weather</a>
```

The visitCount and history represent the properties analogous to those which they represent for PageNodes, identifying the number of times a link has been followed, and the timestamps identifying when the link has been followed. The Edge’s type will either be “hyperlink” or “redirect”, depending on which of the two modes were used to navigate along that edge. Hyperlinks are identified by event listeners on <a> elements, and redirects are identified by page unload events correlated with a new page visit, but lacking the presence of a hyperlink event.

For example, a user may click a hyperlink on a page triggering the event listener on that <a> element and creating a hyperlink edge. The destination of that hyperlink could automatically redirect the user to another site. In that case, the original destination unloads and the page redirects to a new destination. Upon reaching this new destination, the system sees that the user did not follow a second hyperlink, and that this most recent navigation occurred through a redirect.

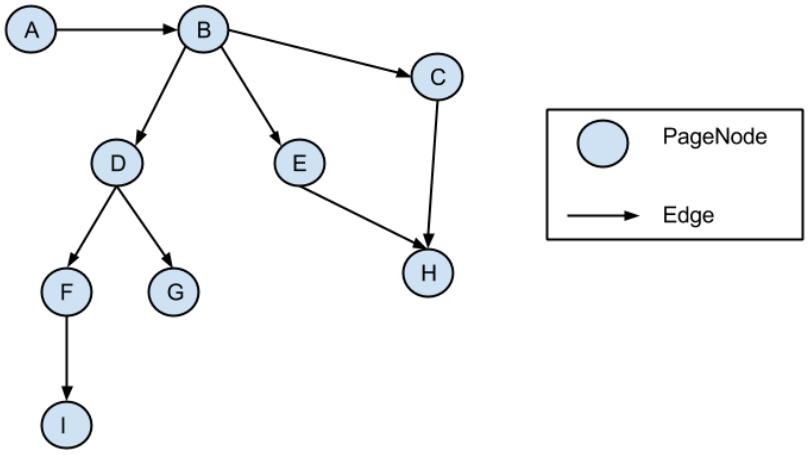

Figure 1. Sample Browsing Graph

In Figure 1 above, a user starts at PageNode “A”. The arrow between “A” and “B” shows that the user actively navigated from “A” to “B.” As defined, this represents an Edge connecting “A” and “B” that would have a source URL matching the URL of PageNode “A,” and a destination URL matching the URL of PageNode “B.” All other PageNodes represented in the Figure are a result of further navigation by the user.

In generating the browsing graph, we will aim to preserve the idea of only creating a PageNode if the user visits that site. It becomes evident, however, that it will become useful to create PageNodes with visitCount initialized to zero in immediate anticipation of visiting a page. The intuition behind this logic is that during the web browsing experience, the user clicks a link prior to actually visiting the page, thus when creating the edge, it is necessary to first create the node to which it will connect in the case that the node doesn’t already exist.

It could prove useful to track those nodes to which a page links but which have not been visited. For example, two nodes could link to the same unvisited web page and this could identify a connection between two previously unrelated nodes. However, this notion is outside the scope of this paper and is left to be explored in future iterations.

Section 3

System Architecture

The entire system is composed of five main components: the chrome extension, the front end client, the API, the back end server, and the database.

System Interactions

The interactions of components within the system are best illustrated by Figure 2 below.

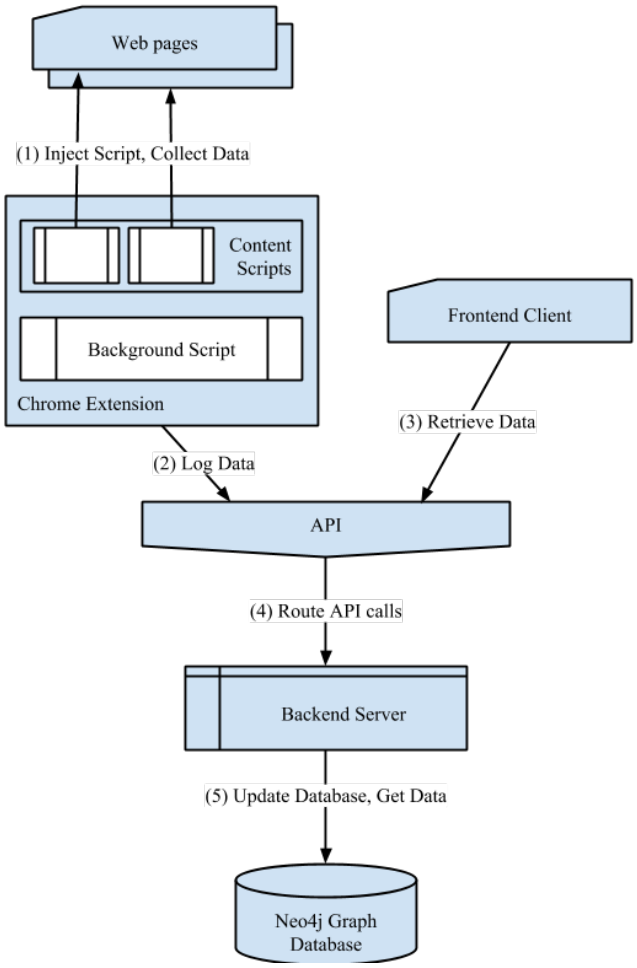

Figure 2. System Architecture Diagram

The extension must interact with the web page to inject its content scripts. Once injected, they collect data for the extension. The extension will make calls to the API to log the data. The

API also receives calls from the front end client for data retrieval. The API routes calls from both sources to the back end which has the implementations to properly process the requests, and then perform the necessary transactions with the database.

Chrome Extension

The Chrome extension is the bread and butter of the operation because it is what actively collects browsing data to be stored on the database. The extension has three components: the background script, the content script, and the browser action.

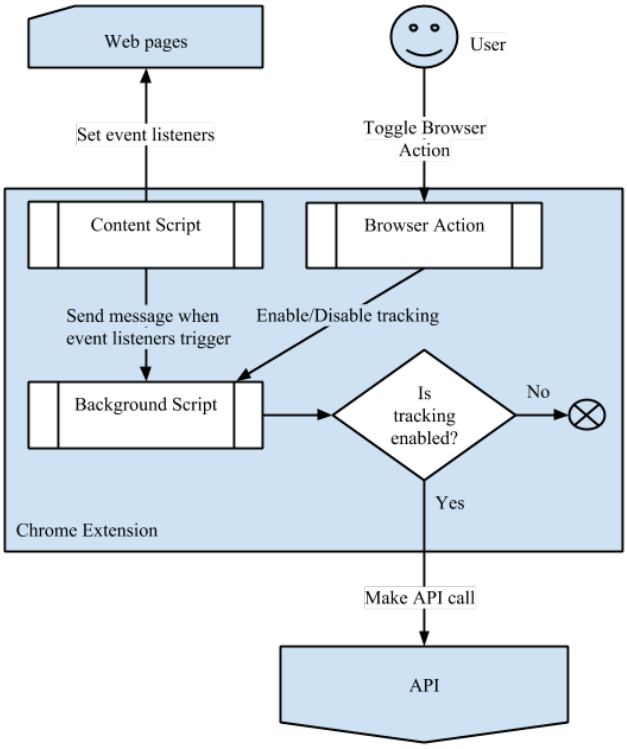

Figure 3. Chrome Extension Process Diagram

The simplest function is the browser action. An extension's browser action appears as a button on the toolbar in Chrome. This browser action serves as an on/off switch for tracking, allowing the user to control when their data is being stored. Clicking the browser action toggles a boolean variable accessible by the background script. All tracking operations in the background

script are governed by this boolean, and as shown in Figure 3 above, the process chain halts if tracking is toggled off. An “ON” or “OFF” label is visible to the user on the browser action as to identify the current state.

In order to collect data, we need access to the web page’s Document Object Model (DOM). In order to transmit the data, we need to be able to make calls to the API. Content scripts are able to perform the former action, but not the latter. The inverse is true of the background script. This is a security measure in place by Chrome extensions, because each has a separate set of permissions and extension APIs made available to it. It is then necessary to isolate these two tasks and communicate between the scripts [6] [7].

The extension injects the content script into each web page as it loads so that it executes locally on the page. Once injected, it is able to behave as a script on the web page normally would, meaning it has access to the DOM. This is necessary for gathering data from the page, as well as adding event listeners. The content script can communicate with the background script through extension messages.

A single instance of the background script runs persistently in Chrome. The background script is equipped with listeners which receive and handle incoming messages from content scripts injected. From the background script, we can communicate with the API.

Once the content script finishes loading onto a page, it sends a message to the background script containing data required of the PageNode object discussed in Section 2. The background script receives this message, sends a POST request to the “visitPageNode” API endpoint with the PageNode object. The script then attaches on-click event listeners to each link element on the page, as well as a page unload event listener to the document for when the user navigates away from the page.

When an event listener on a link is triggered, the content script retrieves page attributes necessary for creating an edge and sends this information in a message to the background. The background script receives the message and packages it into an Edge object, which it subsequently sends in a POST request to the “visitEdge” API endpoint.

When an event listener fires on page unload, the content script notifies the background script of the page which the user is leaving. On the next page load, if the background script recognizes that the user did not navigate to the page using a hyperlink, the background script will determine the referring site, if it exists, based on the timestamp and domain of the last page unload, and the timestamp and referrer for the recent page load.

The chrome extension manifest dictates the URLs which the extension may access. The default setup is that the extension has permission to access all URLs, however any person wishing to install and use the extension on their own may edit the list of permitted URLs in order to restrict the scope of the browsing graph. The user may block any domains which he or she may not want recorded.

Front End Client

The front end client serves as a web portal to access the information, and view it in a relevant and useful manner. This portal is accessible at http://localhost:8080/ provided that the back end server is running. This page contains a visualization of the user’s browsing graph and a utility panel which contains graph filters and a section which displays node details.

The page leverages D3.js and jQuery JavaScript libraries as well as the Bootstrap HTML, CSS, and JavaScript framework. D3.js does much of the legwork for graph rendering [9]. In addition to being a dependency for Bootstrap, jQuery is used in its standard manner for useful

DOM manipulation in much of the analysis tools which I have built into the site. Bootstrap takes much of the stress and time out of the aesthetic design and page formatting.

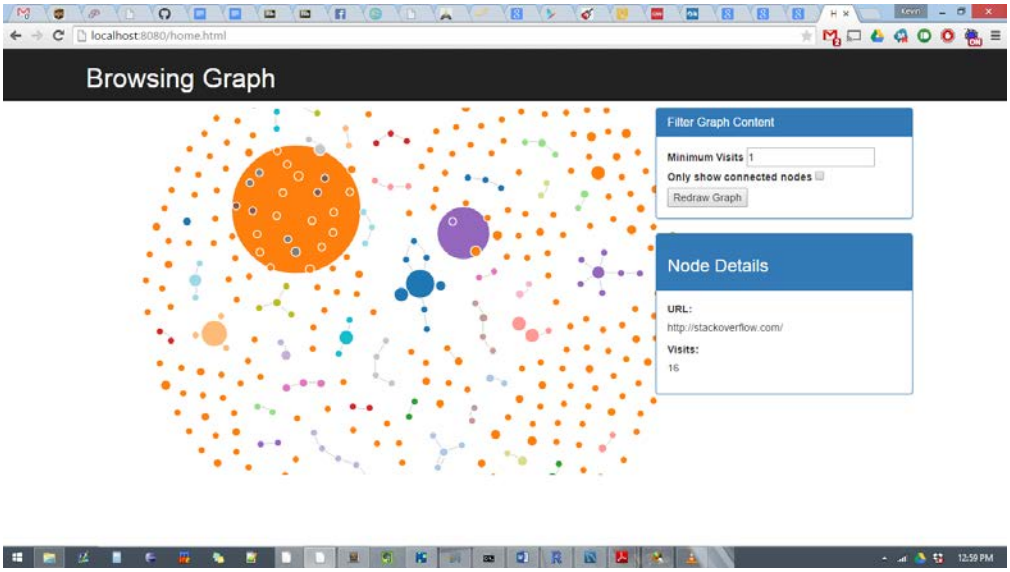

Figure 4. Front End Client Screenshot

When the page loads, it fetches the graph from the back end via the API. The graph is delivered as a list of nodes and a list of links, as outlined in the API specification. Then bidirectional edges are identified, and nodes and edges are weighted. The node and edge weighting is based primarily on number of visits. This is a rather elementary weighting system and in the future it will be beneficial to explore a new weighting system which incorporates more data points such as how recently the page or link has been visited and how closely the content on different pages is related. The processed graph is then passed through D3.js, which produces the interactive visualization.

The utility panel's filter panel allows the user to restrict the graph to nodes and edges meeting certain constraints. After the user sets the filter settings and pressed the "Redraw Graph" button, the same graph fetching and rendering process is executed. However, when the process is

modifying the graph prior to rendering, the graph is pruned to eliminate elements which don't fit within the constraints.

This front end client serves as a simple example of the capabilities of the system. Any developer can create their own front end client that manipulates the graph to suit their interests by applying their own weights or filters or running content analysis on the nodes and edges. The API and the back end make this possible.

API

The API allows the extension to log data, as well as enabling a platform on which developers can build new programs utilizing the browsing graph. Front end clients should not have unbridled access to the graph database; the API serves to restrict clients to only certain interactions. The API has two endpoints only available to the extension for the purposes of logging data, and three endpoints available to front end clients for retrieving data.

The data logging endpoints available to the extension are "visitPageNode" and "visitEdge," and should be utilized any time the user loads a web page or navigates between pages respectively. These API calls will result in either creating a new PageNode or Edge in the database, or updating an existing one. For visiting a PageNode, it is as simple as checking the existence of the PageNode in the database using its URL as the unique identifier, and either creating one in its absence, or updating the existing one. The process for visiting an Edge is better illustrated in Figure 5.

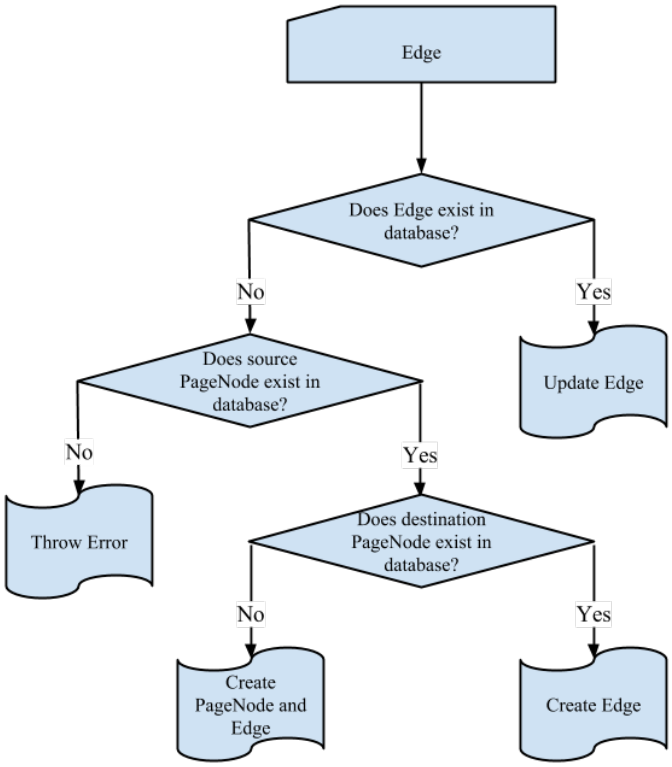

Figure 5. “visitEdge” API Call Process Diagram

When navigating to a previously unvisited page, it will be necessary to create the destination PageNode prior when creating the Edge so that the Edge does not point to a null PageNode. A scenario in which the user navigates away from a page which he or she has not visited is illogical and should throw an error. Both of these API calls will return the created or updated PageNode or Edge, however the extension makes no use of these responses outside of debugging.

The data retrieval endpoints available to front end clients are simple GET requests and can fetch either an individual PageNode, individual Edge, or the entire graph in the form of a collection of PageNodes and a collection of Edges. It is currently left to the client to perform any filtering and weighting of the graph suitable to its needs. It will become necessary in time to limit the number of PageNodes returned in the response as the graph grows, however, all work done

within the scope of this paper can be performed on graphs of a modest size. The next iteration of GraphTracker will implement a request parameter to limit the volume of PageNodes returned so that front end clients may experiment with performance changes with various graph sizes.

Back End and Neo4j Database

The back end server serves a rudimentary purpose in its current iteration. The back end runs a web server powered by Node.JS [10] with Express.JS, which handles the API endpoints as well as serving the client's web page for demonstrative purposes. The back end routes API requests to their proper implementations, identifies the necessary interactions with the graph database, and generates the database query to accomplish the task.

We selected Neo4j as the database to store GraphTracker's browsing graph. It is convenient on an intuitive level that the graph be stored in a graph database which provides easy access to nodes and edges. Neo4j is a leading graph database which is easy to scale and has a flexible structure which will make it easy to make updates to node and edge design as the project evolves [8].

Neo4j uses its own query language, Cypher. The API shields the database from direct user interaction in order to limit operations. The back end understands how to generate a valid Cypher query so that developers don't need to learn Cypher or understand anything about Neo4j in order to create their own front end client.

Conclusions and Future Work

GraphTracker is a small step in exploring user-driven tracking, which could eventually provide the benefits of web tracking, while absolving user trust and privacy concerns. We hope this will serve as the basis for future work, either improving GraphTracker as a form of user-driven tracking, building tools off of the GraphTracker data platform to demonstrate the benefits of small scale user-tracking, or deriving insights from GraphTracker which help develop other modes of user-driven tracking. There are a number of use cases we welcome researchers and developers to explore.

The data provided by this tool can be used to expedite repetitive internet tasks for users. The browsing graph could be used to identify frequently repeated hyperlink chains. As the user visits the first piece of that chain, the tool could prompt the user to proceed to the final destination. This could see several interesting challenges, such as auto-completion of intermediary tasks, but could also be a great time saver.

The browsing graph could be analyzed to identify communities of related sites among currently open tabs, which could then be automatically grouped or color-coded. This could be useful when researching a variety of topics to keep all related tabs in a logical order. This would require the ability to directly modify tab order or appearance, but could potentially provide a modest productivity increase.

GraphTracker could be a useful tool to improve usability testing done by web services. Since it is not uncommon to do very granular, small scale testing, a web site could have participants utilize a tracking technology such as GraphTracker in order to really understand how the user navigates around the site, without having to conduct an in-person, over-the-shoulder usability test.

A downfall of the current iteration of GraphTracker is a lack of scale. Scaling to more users, would provide challenges in preserving user trust and privacy, but would provide a robust data set perfect for recommendation engines. The recommendation engine could also potentially take the approach of accepting self-tracked web data in exchange for its recommendation services. This removes the burden of gathering data or tracking users from the service and puts it in the hands of the user, which if user-driven data tracking catches on, could have potential as a business model.

All code has been made publicly accessible on GitHub, allowing anyone to download and install GraphTracker. Source code, dependencies, and installation instructions can be found at www.github.com/kdaziz/Capstone. By continuing to explore modes of user-driven web tracking as an alternative to third-party web tracking, we may be able to reap the benefits of web tracking, while protecting user privacy and trust.

Works Cited

[1] Barth, A. "HTTP Management Mechanism." *IETF.org*. Internet Engineering Task Force, 1 Apr. 2011. Web. 1 May 2015. <http://tools.ietf.org/html/rfc6265>.

[2] "Means and Methods of Web Tracking: Its Effects on Privacy and Ways to Avoid Getting Tracked." *InfoSecInstitute.com*. InfoSec Institute, 23 July 2013. Web. 3 May 2015. <http://resources.infosecinstitute.com/means-and-methods-of-web-tracking-its-effects-on-privacy-and-ways-to-avoid-getting-tracked/>.

[3] Mayer, J. R., and J. C. Mitchell. "Third-Party Web Tracking: Policy and Technology." *Security and Privacy (SP), 2012 IEEE Symposium on* (2012): 413-27. IEEE. Web. 1 May 2015. <http://ieeexplore.ieee.org/xpl/abstractMetrics.jsp?reload=true&arnumber=6234427>.

[4] "Pixel Tracking in Third-party and Custom Creatives." *DoubleClick for Publishers Help*. Google, Inc., *n.d.* Web. 1 May 2015. <https://support.google.com/dfp_premium/answer/1347585?hl=en>.

[5] "Tracking Preference Expression (DNT)." *W3.org*. Ed. Roy T. Fielding and David Singer. W3C, 24 Apr. 2014. Web. 1 May 2015. <http://www.w3.org/TR/tracking-dnt/>.

[6] "Content Scripts." Google, Inc., *n.d.* Web. 1 May 2015. <https://developer.chrome.com/extensions/content_scripts>.

[7] “Background Pages.” Google, Inc., *n.d.* Web. 1 May 2015. < https://developer.chrome.com/extensions/background_pages>

[8] "Neo4j." *Neo4j*. Neo Technology, Inc. Web. 1 May 2015. <http://neo4j.com/>.

[9] Bostock, Mike. "D3 Data-Driven Documents." *D3js.org*. 1 Jan. 2013. Web. 1 May 2015. <http://d3js.org>.

[10] "Node.js." *NodeJS.org*. Joyent, Inc, 1 Jan. 2015. Web. 1 May 2015. <https://nodejs.org/>.

Appendix: API Request Reference

Visit Page Node

URL: /api/visitPageNode
Request Type: POST
Request Body: {pageNode: {*PageNode Object}*}
Response Type: JSON
Response Body: *{PageNode Object}*

Visit Edge

URL: /api/visitEdge
Request Type: POST
Request Body: {edge: {*Edge Object}*}
Response Type: JSON
Response Body: *{Edge Object}*

Get Page Node

URL: /api/getPageNode?url=*{url}*
Request Type: GET
Request Parameters:
{String} url - The url of the desired PageNode
Response Type: JSON
Response Body: *{PageNode Object}*

Get Edge

URL: /api/getEdge?toURL=*{toURL}*&fromURL=*{fromURL}*
Request Type: GET
Request Parameters:
{String} toURL - The origin url for the desired link
{String} fromURL - The destination url for the desired link
Response Type: JSON
Response Body: *{Edge Object}*

Get Graph

URL: /api/graph
Request Type: GET
Request Parameters: none
Response Type: JSON
Response Body: {nodes: [*{PageNode object}*,], links: [*{Edge object},*]}