SophieJS: A transparent user effort capture system for Sophie P2P search

by

Stephen Purcell, B.A.(Mod.)

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Stephen Purcell

September 9, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Stephen Purcell

September 9, 2010

Acknowledgments

My sincere thanks go out to my family, friends, classmates and all those who helped and supported me during the development of this project, not least my excellent supervisor, Stephen Barrett. The success of this project relied on his ideas, solutions, and assistance.

Stephen Purcell

University of Dublin, Trinity College
September 2010

Abstract

The ordering and quality of results that a search engine provides are crucial to that search engine's success. Today's Internet search engines are experiencing difficulty in maintaining their quality results, due to "search engine poisoning" - artificially raised rankings for specific pages, generally achieved through link farming. Traditional search engines assume that a page's importance is dependent on the number of inbound links to that page, but this methodology leaves rankings prone to manipulation.

Sophie is a peer to peer search engine that bases page importance on the effort expended by actual users getting to that page. Effort is determined through the actions a user has taken, where actions such as printing and form filling hold high value - and indicate that the page was of importance to the user. Actions are monitored through the use of an installable browser plugin, which must be developed for each browser and installed for each user.

SophieJS provides an alternative to the browser plugin solution, by monitoring users' actions without needing any software installation. By providing a HTTP proxy through which all browsing activity occurs, responses can be augmented with monitoring code, written in Javascript. This code executes in the user's browser, collecting data about user actions, and sending this data to a logging server. The system works on a variety of browsers, and does not require users to install any software. With transparent proxying, no configuration is required in the user's browser.

The data generated is stored in a database on the logging server, where it can be used to enhance user models being built for Sophie. This data can also be used to provide "browsing effort" information directly into the Sophie engine, improving results. The uses of the data extend beyond the scope of the Sophie project, and can be of use in a variety of other studies, both academic and otherwise.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

In the world of web search, there are two enormous tasks: indexing the contents of the Internet so that searches can be conducted, and sorting the relevance of the results of a search on this index in a meaningful way. The latter of these tasks is becoming an increasingly difficult proposition, with current link-based ranking algorithms, such as Google's PageRank [1] becoming victim to search engine poisoning, whereby link farms artificially increase a page's ranking, thus reducing the quality of the search results provided by the search engine.

Sophie, a peer to peer search engine in development at Trinity College, builds ranking information based on the "importance" of a page according to the actual users, rather than assuming that a link from another page represents importance. The user's view of the importance of a page is captured by the effort expended by the user in reaching that page, which in turn is determined by the actions the user takes to reach that page. As an example, a user searching for a specific topic who selects a result from the third page of results, scrolls to the bottom of that page, fills in a form and prints the resulting page has expended time and effort in reaching that page they

choose to print, and so that page is likely relevant to their search query, and should be placed higher in future results for that query.

Currently, Sophie gathers information about user's actions via a browser plugin installed by the user. This plugin outputs XML that is consumed by the Sophie engine, which analyses and reasons about this data, and publishes ranking information to the distributed Sophie datastore. This process is illustrated in figure 1.1.

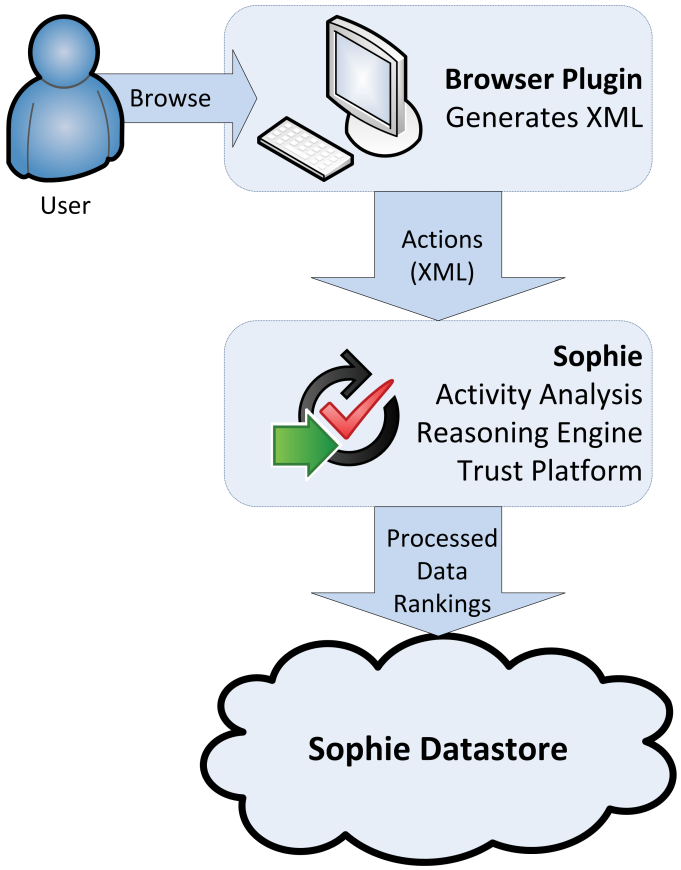


Figure 1.1: Architecture of the Sophie browser plugin

1.2 Motivation

There are a number of issues with the concept of using a browser plugin for data capture in Sophie.

Firstly, users must explicitly install the browser plugin in order for their data to be submitted to the Sophie datastore. This reduces the number of active users generating data for the Sophie system, thereby reducing the volume of data Sophie operates on and thus influencing the quality of search results. The very fact that the user has to install a plugin is a major barrier to the data gathering effort. Users need to be aware of the Sophie system and install the plugin, and many may not have the skills (or administrative access) to do this.

Secondly, each browser, and indeed each version of the browser, requires a different plugin. The Sophie project currently has a plugin developed for for Mozilla Firefox version 3 [2], but further work is required to port this plugin to all major browsers, and to add support for other versions of Mozilla Firefox.

Finally, it is a challenge with any capture solution to operate successfully on the the extremely variable Internet, where different web servers provide content in different ways, and published HTML is rarely specification compliant.

It is a key goal of SophieJS to provide a plugin-less method of capturing user data, and to enable this data to be used in ongoing research into Sophie, along with using the data as input into a production Sophie engine. It is easy to modify SophieJS to capture different types of data or change the way in which data is collected, as SophieJS is centrally administrable. This is not the case for the browser plugin architecture, where different versions of different plugins for different browsers can produce different data.

Beyond these issues, there is a demand for raw user activity data to support research into user models used in the Sophie engine. SophieJS seeks not only to provide a plugin-less method of capturing user data, but for this data to be usable both in research into

Sophie, and as input to a production Sophie engine.

1.3 Solution

The solution to these problems is to build a robust, cross-platform solution that requires no software installation. This involves creating a custom HTTP proxy capable of manipulating pages as they pass through the proxy. Pages are modified to include Javascript code which executes in the user's browser, monitoring and reporting user actions to a logging server. This can be seen in figure 1.2.

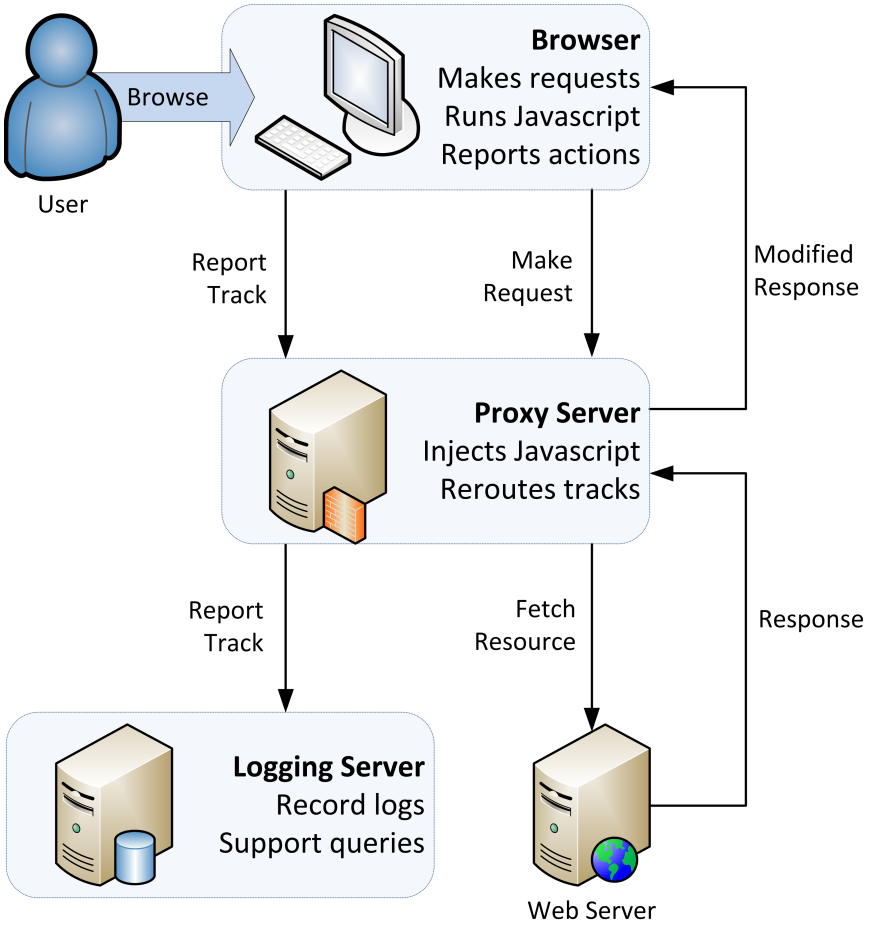


Figure 1.2: Architecture of the SophieJS solution

Using this architecture, pages are examined as they return via the proxy, and modified if the content is HTML. If so, Javascript code is inserted to the response, which, when executed in the user's browser, attaches onto elements in the page and receives events generated by the user on that element. This Javascript gathers relevant data from the event, such as the x and y coordinates of the mouse cursor on screen, and sends the data to a logging server asynchronously. The user's browsing is in no way interrupted in this process.

A number of challenges emerged during the design and implementation of the solution, including issues relating to performance, interoperability, user identification, and the security browsers enforce to protect users from cross site scripting, all of which were resolved. These challenges are discussed further in section 3.5.

1.4 Result

Sophie JS operates on a variety of browsers requiring only configuration of a HTTP proxy URL within the browser. SophieJS can also operate as a transparent proxy, enabling its use without any required configuration of the user's browser.

Running SophieJS generates vast volumes of data about a user's browsing activities, with as many as 60 updates logged per second. In its current state, SophieJS captures the location of the mouse on the screen, scrolling, clicks, form focus & submission, link hovers & follows, keyboard presses, text selection and printing. All of this data is sent to a remote server, and stored in a database.

This captured data has a variety of uses. The initial goal of the project was to gather a large amount of raw data for research into user models in the Sophie engine. The data is also suitable for exporting into the same XML format generated by the browser plugin, thus feeding a running Sophie system with user's browsing actions, enhancing reasoning and understanding of the important pages for a given search query.

Furthermore, SophieJS has applications in eye tracking studies, addiction studies, and in the commercial world, evaluating the usability of websites.

1.5 Outline

The remainder of this dissertation examines these areas in more detail:

Chapter Two examines the State of the Art: looking at today's search technologies and their weaknesses, and contrasting with the Sophie P2P search engine. The operation of Sophie is discussed, including the issues that face Sophie in collecting user data. The research question is asked.

Chapter Three delves into the design of SophieJS, covering the possible solutions considered and the architecture of the final solution. The behaviour of the components in SophieJS, and challenges that became apparent are discussed in this chapter also.

Chapter Four lists the steps taken to implement each of the SophieJS components: the proxy, the browser component, and the logging server. Deployment of SophieJS without user configuration is discussed in the *transparent deployment* section of this chapter.

Chapter Five evaluates SophieJS, discussing the events it can capture versus the browser plugin approach, and its browser compatibility. Performance of the solution is examined, and many uses of SophieJS-generated data are investigated.

Chapter Six concludes the report, listing a number of features and improvements to be considered in future versions of SophieJS

Chapter 2

State of the Art

2.1 Introduction to web search

2.1.1 Centralised search

Fundamental to modern use of the World Wide Web is the ability to conduct searches over its contents. Popular search engines, such as Google, Bing and Yahoo! provide their services by building enormous server farms to supply the capacity and infrastructure necessary in processing the huge numbers of web searches hitting the services each second. The centralised nature of these systems presents a multitude of scalability problems, with an ever-increasing number of servers to crawl the web, and to handle user queries.

Equally problematic is the number of results a given search returns, where the number of matching pages can extend into the millions or billions. This can be classed as an information overload: it is impossible for a human to process each of these results in order to find the page for which they were searching. The role of the search engine thus includes the task of narrowing down the results set, from the set of all pages available on the World Wide Web, to a subset of pages relevant to the user given their

search query.

Traditional ranking

Substantial efforts go into the development of algorithms to determine what should appear in the results set, and furthermore the ordering of said results. Initially, this process was relatively simple: when the search engine found a new page, it read its contents and built up a list of keywords present in the document [3], akin to the traditional information retrieval procedures used to categorise books. Whenever a search was conducted against this index, the results set consisted of the list of pages for which the maximum number of keywords matched. The number of times a keyword was present in the page determined how relevant for that keyword the page was, and thus the relevance of that page to a user searching for that keyword.

However, treating each page as an island is not sufficient to genuinely determine the importance of a given page, as it is easy for a rogue publisher to include a list of popular keywords several times on a given page in order to draw traffic to content not necessarily associated with the required keyword.

Link-based ranking

A more rigorous algorithm was required to determine a page's ranking, and thus where it would appear in the list of results for a given search. In addition to the contents of a page, the hyperlinks to and from it are considered during indexing [1]. This protects against the high ranking that would otherwise be attributed to a page consisting solely of popular keywords, but lacking any real value, since that page would likely not be linked to. This system adds a new metric to each page, namely how linked-to it is. This metric, however, proves to be a powerful one, as the act of "linking" to another page infers that the linking author found the linked page to be of interest, and it therefore must have some quality relevant to the keywords for which it has been indexed.

Unfortunately, three problems remain. Firstly, there is an assumption drawn between relevance and link-popularity [4]. For example, a well-known politician may have a website with hundreds of links pointed towards it from major news organisations; while a less well-known local politician may have just a few links pointed at his website from the personal websites of a subset of his supporters. According to link-based schemes, the first website is more relevant for its search terms, since it is rich with incoming links, but this conclusion is not necessarily the case.

Secondly, since the value of a link from page A to page B is (in part) based on the importance of page A, "rich get richer" effect comes to light [5], whereby high-ranking pages link to each other, further increasing each other's rankings, while a page outside of this loop remains largely unseen. This leads to a "bow-tie" shape of the web [6] - one where there is a central core of highly-ranked pages that frequently link to each other, a set of pages that link into this core, and another set that are linked from the core but do not link back, as seen in Figure 2.1.

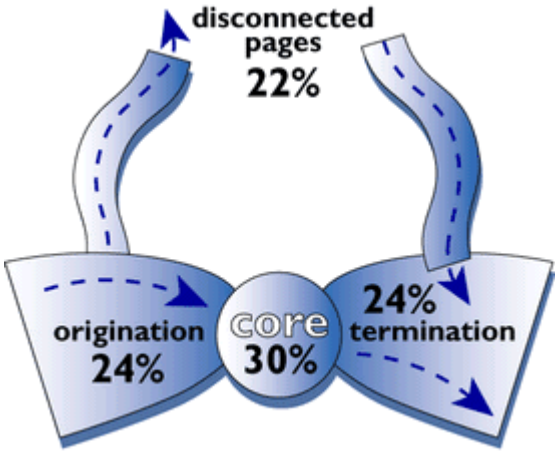


Figure 2.1: The bow-tie appearance of the World Wide Web [7]

This central core represents an area where most of the content of the web is consumed, as it contains the pages most likely to appear at the top of a set of search results. A page's presence in this core, however, does not necessarily accurately represent its

importance, quality, or relevance to a given subject area.

Thirdly, the link-based ranking algorithms can be tricked through the use of "link farms", thousands of inter-linked pages that serve only to increase their own ranking, and thus hold greater power when linking out to another page [8].This activity produces a tightly-knit community (TKC) effect [9], which acts in a similar way to the centre of the "bow-tie". Heavy interlinking leads to a greater perception of importance, and thus a stronger ability to link to non-relevant pages which will then appear artificially high in search rankings. While some efforts [10] have gone into removing link farms from search results, these methods are not perfect.

2.1.2 Peer-to-peer search

Solutions to these problems are possible when making use of peer-to-peer (P2P) technology to provide the search functionality. A P2P network made up of all of the world's web searchers will provide an enormous computation base, substantially larger than the power of the centralised server farms, that would be capable of processing large volumes of data [11]. As is the case with the majority of P2P technologies, the issue of scalability is ever-present, however research suggests that, when specific optimisations are made, P2P search can be feasible [12]. Indeed, a number of prototype P2P search systems have been developed [13, 14] and have bolstered confidence in the scalability of such systems.

Beyond scale, P2P search can completely change the way that the web is indexed. Using link-based ranking schemes, the importance of a page is given by the number of links to it, meaning that artificial links can be created to a page, or links may be incorrectly tagged. P2P search offers a solution to this problem. Consider a simple tfidf [3] keyword-based indexing system, where the input pages to the system were those that had been browsed by the user. Now, the user has the potential of searching over pages they have previously visited, with the implication that pages visited by the

user were considered valuable to the user. If a page is re-visited, its ranking for that keyword can be increased, as a re-visit adds weight onto the importance of that page to the user.

Now consider a system where the results of analysis of a given page are flooded out to the network. At this point, each user has an index based on what all users consider to be important. It is through such a system that the wisdom of crowds [15] can be harnessed. Instead of the search engines and page authors determining (artificially) how important a page is, the users do. It is only through intentional action - visiting the page - that a page is considered important.

The architecture proposed here is a simplistic solution, but more complex systems have been developed with the same mantra of "[giving] the Web back to the people" [11].

2.2 Effort computation

There is a flaw in the assumption that a person visiting a page deems that page to be of importance. There are any number of situations where this may not be the case: a link could be mis-clicked, the page could have been reached through a link-farm, or indeed the page may simply not be of interest. In the naïve approach proposed in Section 2.1.2, each of these circumstances is registered as a declaration of importance.

2.2.1 Social bookmarking

Social bookmarking is an interesting, present-day solution to this problem. Under this system, users explicitly submit pages of interest to a social bookmarking service, such as del.icio.us or digg, often associating tags with the submitted entry. This manual filtration leads to high quality, trustworthy page submission [16], and can be augmented with the ability for other users to add additional tagging information, and to up-rank

the submission, further reinforcing the importance and relevance of the page to its keywords/tags. In addition to this benefit of providing a good source of quality content, research has found social bookmarking to be beneficial to search results in a variety of ways. Social bookmarking services:

- Tend to provide more recently published content, which, in 25% of cases, has not yet been indexed by traditional search engines
- Index a small proportion of the web. However, 9% of what is indexed appears in the top 100 results for a corresponding term in a search engine, indicating high quality content
- Have 56% of submissions provided by 10% of users, indicating that relatively few people can produce high quality indexes
- Are not restricted to bookmarks at the core of the "bow-tie", as one in eight submissions are from domains for which there are no previous submissions

[17]

The key problem that exists with social bookmarking is the explicit action required to register a page as being of importance. While a page the user finds particularly interesting is likely to be submitted, many other pages the user finds important or relevant are less likely to be submitted. Consider a user searching for information on taxation. They are likely to conclude their search on their government's taxation website - a site containing pages that are both important and relevant. It is unlikely, however, that this page would be actively submitted to the social bookmarking service by the user. The content is important and relevant, but simply not interesting.

It is this gap between "interest" and "importance" that is the greatest barrier to the success of social bookmarking. While the quality of submissions tends to be excellent, the quantity of submissions is insufficient to build a substantive index of the web.

2.2.2 Traffic-based ranking

The explicit action of social bookmarking does not lend itself to an indexer finding pages of authority, or those that a user considers important. An alternative to this approach is to index the pages a user actually visits, and to base the calculation of importance of that page on the number of times that page has been visited by actual users. Using this technique, "popularity" is no longer determined by the number of inbound links to the page (as per PageRank), but instead by the genuine popularity of the page itself - the number of users visiting it.

Alexa makes use of traffic-based ranking when generating its results [18]. Under this system, users install a toolbar into their browser, which monitors the pages being visited by the user and reports these statistics to Alexa. The data is stored for three months, and used to add weight to the importance of pages when a search is conducted. While the action of recording access to a page is now automated, the user must still explicitly install the Alexa toolbar in order for this to occur.

Other solutions to traffic-based ranking have been considered in research. Meiss et al. [19] developed software to collect traffic at the edge of Indiana University's network. In real time, the traffic was inspected to check if it contained a HTTP GET request, and the destination site, destination page, and referring site are recorded, amongst other data. A weighted, directed, graph could then be built, consisting of all sites visited, and from where the user originated (i.e. the referring site), weighted by the number of users who took that path (i.e. followed that link). This information could be used in order to determine which hyperlinks on a page are of importance. In particular, the research found that the "assumptions underlying PageRank [are] violated: not all links from a site are followed equally". It is this realisation that confirms the benefits of using alternative techniques in gauging the importance of a page for ranking purposes.

However, these techniques are not perfect. The problem remains that the act of visiting a page does not necessarily represent an interest in, or the importance of that

page. Frequently, pages are visited in order to determine their content (and thus their quality, relevance and importance), and are quickly discarded before the user moves on to another result for their query. This action should, in fact, downrank a page, since the user found no value in it. Other neutral actions can lead to hits on pages without the associated intention of registering that page as important, such as mis-clicking a link or typing a URL.

2.2.3 Deeper user activity monitoring

An experiment was also conducted where a search engine was built such that the links to results pointed at a logging service which would then forward the user on to the required URL. For each search query, a unique ID is assigned, and represents part of the URL that is sent to the logging service [20]. Thus, the logging service can record those results selected by the user for a given query, and the order in which these results were visited (since clicking the browser's back button does not modify the unique ID for that query). Furthermore, the initial ranking is sent to the logging service, since it is known that high-ranking results are more likely to be clicked. A user that selects a result significantly further down the list is making a greater statement (w.r.t. their search for relevant, quality pages) than a user who consistently selects the first result. This information can then be used as part of a feedback mechanism which modifies the rankings of the pages that were returned for that query, such that the selected results move towards the top of the result set.

In order to further understand the value a user places on a page through their implicit actions, more of their actions must be monitored. It is interesting, for example, if a user moves the mouse cursor over a link, but chooses not to select that link. Letizia is an early browser add-on that monitors links being clicked and passed over, and makes recommendations to the user based on knowledge it captures [21]. Extensions to Letizia to capture scrolling have also been developed [22] that make use of neural networks in

order to predict the interest a user will have for a page, given past experience.

This research has continued into more complex user monitoring, specific to a user's actions following a search query. Following a search, browser instrumentation quietly monitors the user, checking if they partake in a variety of actions, such as whether the user has returned to click a link again, how long they have spent on a page, how long on a specific domain, if the followed link was the first selected one, how many clicks were made between executing the search query and finding the desired page [23]. These measurements are then compared to the average results for the specific user in order to refine the weight a specific action should be given. The result of this analysis can then be used in order to rank search results.

2.3 Validity of user actions in determining importance

An important question when considering the value of users actions when calculating the importance of the page is whether their actions actually map accurately onto the importance of the page. Work has shown that explicit actions, such as the submission of a URL to a social bookmarking site, can accurately relate to the importance of a page [19]. Checking the accuracy of implicit actions is more difficult to achieve, but studies have continued nonetheless.

Basic usability guidelines suggest that actions such as scrolling represent an effort being exerted on behalf of the user [24], and research into this area has confirmed the relationship between scrolling and the user's interest in the article [25]. Recently, client-side logging suites that include monitoring of bookmarking, printing, saving, form filling, cut and paste, scrolling, finding (searching for text within the page), and time spent on the page have been developed. This software was then used in an experimental setting, alongside requests for explicit feedback from the test subjects,

when asked to fulfil a specific task, and led to promising results in order to draw a correlation between the user model that can be generated based on these measured metrics, and the genuine value the user places in the page [26, 27].

Fox et al. studied the relationship between a user's actions and the user's actual impression of the situation [28]. This was achieved though the development of an add-on for the Internet Explorer web browser that monitored user's activity, such as clicking, spending time on a page, and scrolling. The results of this instrumentation were sent to a central database for analysis. Throughout the process, dialog boxes appear on screen to gauge the satisfaction of the user, in order to corroborate implicit evidence with fact. Following a study with 146 people over a 6 week period, the authors conclude that monitoring implicit actions can accurately capture a user's satisfaction levels, with the implication that users are satisfied when they have found a quality page, relevant to their search. These results can then be used to alter search rankings to achieve higher user satisfaction.

2.4 Use of proxies in monitoring user activity

Many of the previous examples of collecting user data involve installation of client software, be it a toolbar, browser add-on, separately running process that interfaces with the browser, or a customised browser entirely. This situation is not ideal, as it effectively requires the explicit action of a user actively installing the client software. A good alternative to this approach is to make use of a proxy to monitor the user activity in a transparent manner. While setting a proxy in a browser is normally a conscious effort undertaken by the user, it can easily be made a mandatory action before Internet access is available. It is also possible to have a proxy transparently introduced into the network, and have traffic sent to it according to routing rules, thus allowing for logging to occur without any configuration of the user's browser.

One example of such transparency was achieved when a logging service was placed at the edge of a university's network [19], however this is limited to analysing clicks made by a user, and does not extend into any further analysis. A customised search engine was also built which gauges the popularity of results by having outbound links from the search results traverse a proxy [20].

WebQuilt [29] is another proxy-based activity logging system, which re-writes the links, frames and other HTML-based entities such that the traffic is routed through the proxy before being forwarded onto the original URL. While this is an effective solution, it still fails to monitor any activity that remains strictly client-side, such as scrolling and hovering over links. Additionally, it requires the user to begin their web browsing session by entering their desired URL on a WebQuilt start page, which performs the initial modification of the HTML.

2.4.1 Proxy-level modification of page content

It is possible for a proxy, installed on a network, to perform some intelligent execution. Existing HTTP proxies frequently cache popular pages locally in order to decrease external bandwidth required and improve download time within the network. Other proxies provide authentication and logging functions to monitor access to the web by the users of the network. This method has already been used in order to enhance search results, either through monitoring user traffic [19] or providing a portal page through which the browsing session begins [29].

The aforementioned solutions capture only movement from page to page via the user's clicks, and do not fully capture the user's interaction with the page, as is achieved with client-side software [26]. It is possible to construct a HTTP proxy such that it modifies the page content that is returned to the requesting user, for example to remove sensitive words or pages when proxying requests for a school's network - that is, acting in a filtering role.

It is also possible for a proxy server to modify a page by inserting additional content into the page, in particular by inserting executable code into the page for it to be run in the user's browser. Work has already taken place to allow users to annotate pages, dynamically, in their browser [30]. These annotations are submitted to the proxy server who includes them in future requests, allowing co-workers to share annotations of the web.

WebMacros is a proxy-based logging system which serves to record a user's interactions with a page, and allow for these actions to be repeated. When a page is requested, the HTML is modified by the proxy such that the links and form submission URLs are modified to point towards the proxy. The user can then click through the site, and submit POST forms, in a way that is recorded by the proxy. When recording is complete, the macro is saved and can be replayed as required [31].

Proxy-level content modification has also been used in order to provide protection against web-based vulnerabilities. This can be achieved by introducing additional Javascript code to encapsulate accesses to elements of the page. These accessors ensure that no security policies are violated. Additionally, they could be used to profile the browser's performance [32]. Of particular interest to this profiling is the performance of browsers on mobile devices.

2.5 Sophie P2P search

Sophie [2] is a peer-to-peer search engine currently under development, combining the aforementioned technologies of P2P search with the technologies surrounding capturing user interest. It makes use of effort computations to determine the importance and relevance of a page, and adds that page (with appropriate ranking) to its database accordingly. This database is distributed across the peers in the network in a peer to peer manner, so the wisdom of crowds is exploited, resulting in high quality rankings for

all users. Additional trust computations are also performed in order to assure quality of data being fed into the system (and thus avoiding the effects of rogue users). The general architecture of Sophie can be seen in figure 2.2.

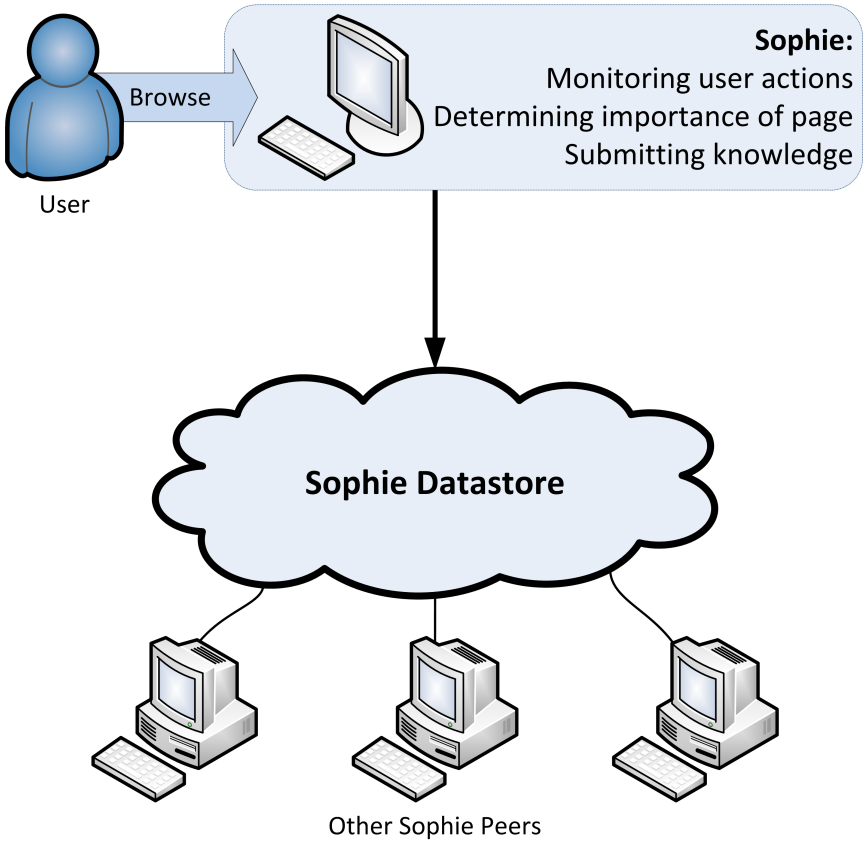


Figure 2.2: Architecture of Sophie

Sophie monitors user activity in order to determine the effort expended by a user in reaching a particular page, and thus the importance of that page to the user. Activities monitored include scrolling, time spent on a page, printing, link-clicking and others relevant to the calculation, and the ranking is then determined by the DANTE inference engine.

Sophie's data about user actions is gathered through software installed on the user's computer, as a plugin in their browser. Currently, support is limited to Mozilla Firefox, but work is continuing on extending the reach of the plugin out to other browsers. The

plugin outputs XML files which are consumed by the Sophie engine, also running on the user's computer. The architecture of the Sophie browser plugin system can be seen in figure 2.3.

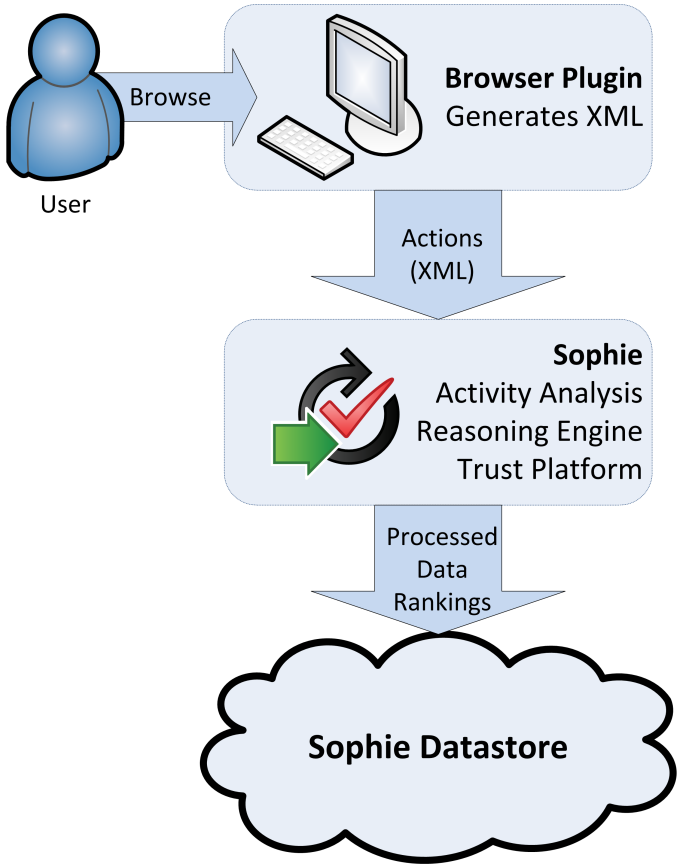


Figure 2.3: Architecture of the Sophie browser plugin

Different plugins must be written for different browsers, and for different versions of these browsers. Furthermore, this plugin architecture presents a problem for the mobile platform, for three reasons. Firstly, there is little support in mobile browsers to allow for installation of plug-ins. Secondly, the computation required by the trust, personalisation and reasoning engines is likely to exceed the capabilities of a mobile device. Finally, the P2P nature of the system requires nodes to, generally, be contactable. Providing this always-online functionality on a mobile network is expensive in terms

of bandwidth/spectrum usage, and in terms of power consumption of the radio on the mobile device.

2.6 Research Question

The goal of this project is to evaluate the potential of developing a plugin-less alternative to the current methods of capuring user actions in Sophie. This system must gather information on user's behaviour in a transparent manner, in order to build high quality user models in Sophie. Specifically, there is a requirement that no client-side software installation is required - the service does not necessarily require a per-user opt-in and no explicit action is needed.

Ultimately, the goal is to provide a solution that operates on all browsers without the need for software installation, and for the system to be capable of capturing and storing vast amounts of data for use in further research into user models in the Sophie engine.

Because of the transparent nature of the code insertion and logging functionality, the system has the potential to monitor the behaviours of users when using web browsers on mobile platforms, such as their mobile phone. This will allow for a future comparative study between the behaviour of users when using desktop and mobile browsers.

Chapter 3

Design

3.1 Requirements

The SophieJS system, as envisioned, has a number of key requirements. In order to successfully answer the research question, these requirements must be fulfilled. The system must:

- Be capable of capturing users' actions as they browse the web
- Not require any additional software installation
- Store captured data for analysis
- Replicate, as much as is possible, the monitoring capability of the existing Sophie browser plugin

Additionally, there are a number of non-functional requirements that should ideally be fulfilled. The system should:

- Capture user data in all major web browsers
- Fail silently, and not impact on the browsing experience

- Induce a minimum amount of latency to the user
- Avoid high resource usage, of CPU, memory or bandwidth

3.2 Possible approaches

As there is a specific requirement for no explicit software installation by the user, it is clear that the system must intercept the actions of the user in some manner. There are a number of possible approaches that could be taken to achieve this.

3.2.1 Plugin solutions

The project requirements stipulate that no software be required in advance of using the system, including plugins. However, by having the user browse through a HTTP proxy, it is possible to present the plugin for installation at the start of their first browsing session. By having the installed plugin add a header to each HTTP request, the proxy can check that the plugin is installed in that user's browser, and only redirect the user to a page to install the plugin if it is not already installed. The architecture of this solution is given in figure 3.1.

To solve cross-compatibility issues between browsers, a plugin framework could allow the plugin to be written in a generic way for all browsers. Greasemonkey [33] and Greasemetal [34] are plugins for Mozilla Firefox and Google Chrome respectively, that are capable of running scripts within the browser. By forcing the download of the appropriate plugin and browser-agnostic scripts, the browser can be successfully instrumented.

Forcing the installation of a plugin is not an ideal solution to the problem, in that it requires a specific opt-in on behalf of the user, and requires them to install software into the browser. This presents an issue for various versions of browsers, for users who do not have privileges to install software on the computer from which they are browsing,

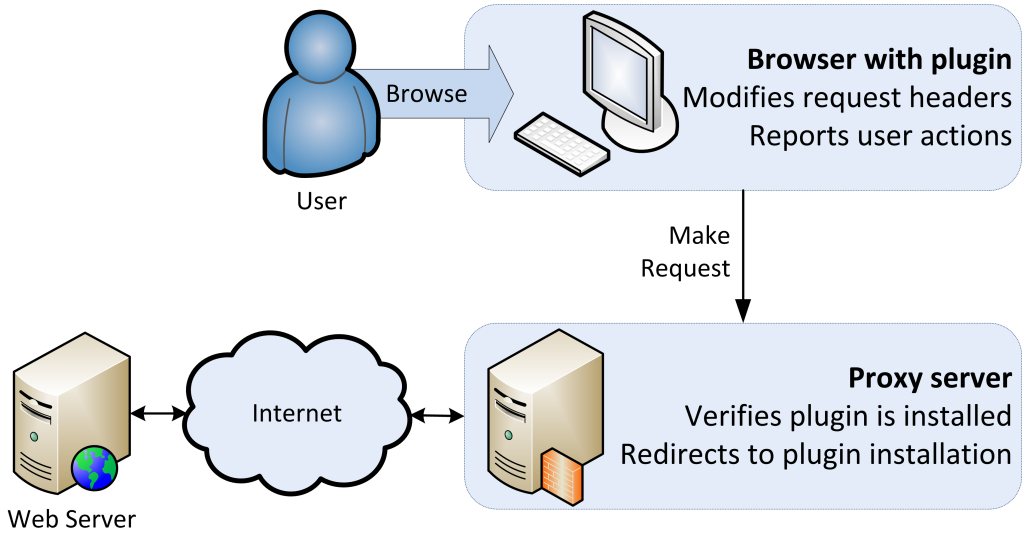


Figure 3.1: Forcing plugin installation

and for mobile devices, where plugins are generally not supported. As a result, plugin installs via a proxy do not represent a viable solution to this problem.

3.2.2 Logging HTTP messages

By logging traffic generated by the user's browser, it is possible to extract the HTTP messages that are sent, in particular HTTP GET and HTTP POST. By monitoring the HTTP GET messages, the list of resources viewed by the user can be deduced, thus providing an opportunity to measure the number of pages that were visited from running a search through to finding the desired information, in addition to the duration of time spent on each of the pages along the route. The HTTP POST messages provide an indication of form submission, and include the data contained in the submission. The architecture of this solution can be seen in figure 3.2.

However, this information tends to be insufficient in determining the effort expended by the user in order to reach a page. It is difficult to ascertain if the user has found the resource they were seeking, or if they gave up on their search. In today's modern web applications, the presence of dynamic content may result in both false positives and

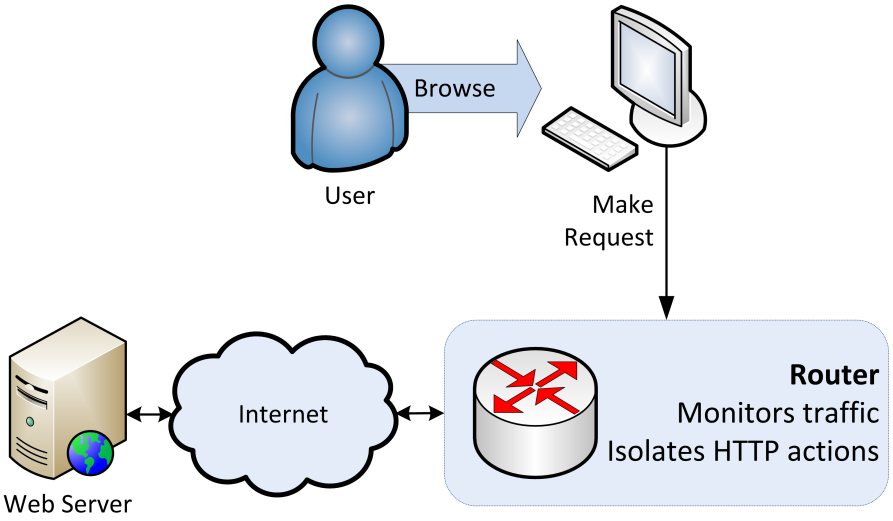


Figure 3.2: Monitoring HTTP messages

false negatives when using this method, as clicks may not involve a HTTP GET, or asynchronous HTTP GETS proving information for the page not relevant to the user's goal may be registered. Furthermore, the range of events captured by this method - visiting pages and submitting forms - does not provide sufficient information about the effort the user has expended in reaching a page. Ideally, actions such as scrolling, printing and clicking on dynamic elements should also be captured. As such, HTTP message logging is insufficient for this task.

3.2.3 Browser scripting

It is clear that in order to capture events such as scrolling, the system must execute code in the user's browser. There are a number of technologies that allow for this to occur, such as VBScript, DHTML, Flash, Java, and Javascript. VBScript and DHTML generally lack support beyond Microsoft Internet Explorer, and are therefore unsuitable for this task. Both Flash and Java require the user to install software for the code to run, and access to browser actions such as scrolling is limited.

Javascript is a well established, standard method of executing code directly in web

browsers. Modern web applications rely heavily on Javascript for their operation, and as such, it is supported by all major browsers, who have focussed on developing high-performance Javascript engines to interpret the code. In addition to executing code, Javscript has access to the browser's document object model (DOM), which provides hierarchically-organised objects representing the elements that make up the page on display in the browser, and indeed details about the browser itself [35]. Using the DOM, it is possible to examine elements of the HTML document as if they were objects, with attributes and operations. Among these attributes are pointers to functions that execute under certain conditions, such as when the element is clicked. This interface to access browser actions is suitable for use in this project.

3.3 Architecture

In order to execute Javascript in the browser without forcing the installation of a plugin, it must be sent to the user's browser as part of the response to each request. In order to achieve this, the response must be modified by a proxy, which inserts the required Javascript code, before being sent to the user as the response. The modifications should only occur on HTML content, and not any other content typically transmitted during a browsing session, such as CSS stylesheets, images and other downloads. In order to ensure that only HTML is modified, the proxy must check that the MIME type is "`text/html`". This information is provided in the HTTP headers of the response. Additionally, there are two possible ways in which the page could be modified to support event listening: by attaching events listeners at the proxy, or by attaching events listeners in the browser.

3.3.1 Attaching event listeners at the proxy

Two steps are required to modify a page entirely in the proxy. Firstly, the Javascript functions responsible for handling each event must be inserted into the head of the response. Next, each HTML element requiring an event to be fired must be modified to assign the appropriate function to run on each type of event. For example, modifying a link element on the page:

```
1 <a href="www.tcd.ie">Trinity College Dublin</a>
```

to be capable of capturing click events, must be rewritten as:

```
1 <a onclick="handleClick();" href="www.tcd.ie">Trinity College Dublin</
    a>
```

Matters become more complicated if there is an existing onclick attribute registered with the element, as written by the webpage's author:

```
1 <a onclick="doSomething();" href="www.tcd.ie">Trinity College Dublin</
    a>
```

This cannot be overwritten safely, as it may be crucial to the operation of the webpage. As a result, the SophieJS function must be prepended to the onclick attribute, as follows:

```
1 <a onclick="handleClick(); doSomething();" href="www.tcd.ie">Trinity
    College Dublin</a>
```

This is not an ideal situation. For modification to operate successfully, a full XML tree must be built from the response, the appropriate changes made, and the resulting tree collapsed back to a text format. For the proxy to be scalable and cause a minimum of latency, this is not a viable option. Furthermore, HTML is not sufficiently well defined to be parsed correctly by a standard XML parser, a HTML-specific parser is required. The variability of HTML online, in particular HTML that is not well-formed, means that building a tree is a difficult task to achieve.

3.3.2 Adding events at the browser

An alternative to adding events at the proxy is to insert Javascript code that attaches itself to the HTML elements in the browser. The browser already builds a DOM tree of the HTML page it is rendering, so there is little additional computation required for the Javascript to navigate the tree in the browser.

As before, the proxy inserts the required functions into the head of the page. Additionally, it adds Javascript code to be executed when the page has loaded in the user's browser. This code can attach event listeners to the required elements as follows:

```
1 element.addEventListener("click", handleClick, true);
```

This method of monitoring events permits multiple event handlers, does not interfere with existing author-defined handlers, and can be done in an efficient manner in the browser. As a result, this is the method used in SophieJS.

3.3.3 Architecture diagram

Figure 3.3 shows the architecture of the system. The browser makes requests via the proxy server, which forwards them on to the required host. If the response is HTML, the content is modified to include the Javascript tracking code and additional elements to handle printing events. The content length is recalculated, and the response sent to the client. The client's browser displays the page and begins executing the Javascript, which reports usage details over HTTP. These usage messages are redirected to the log server by the proxy.

3.4 Behaviour

There are three core components to the system: the proxy, the browser and the logging component.

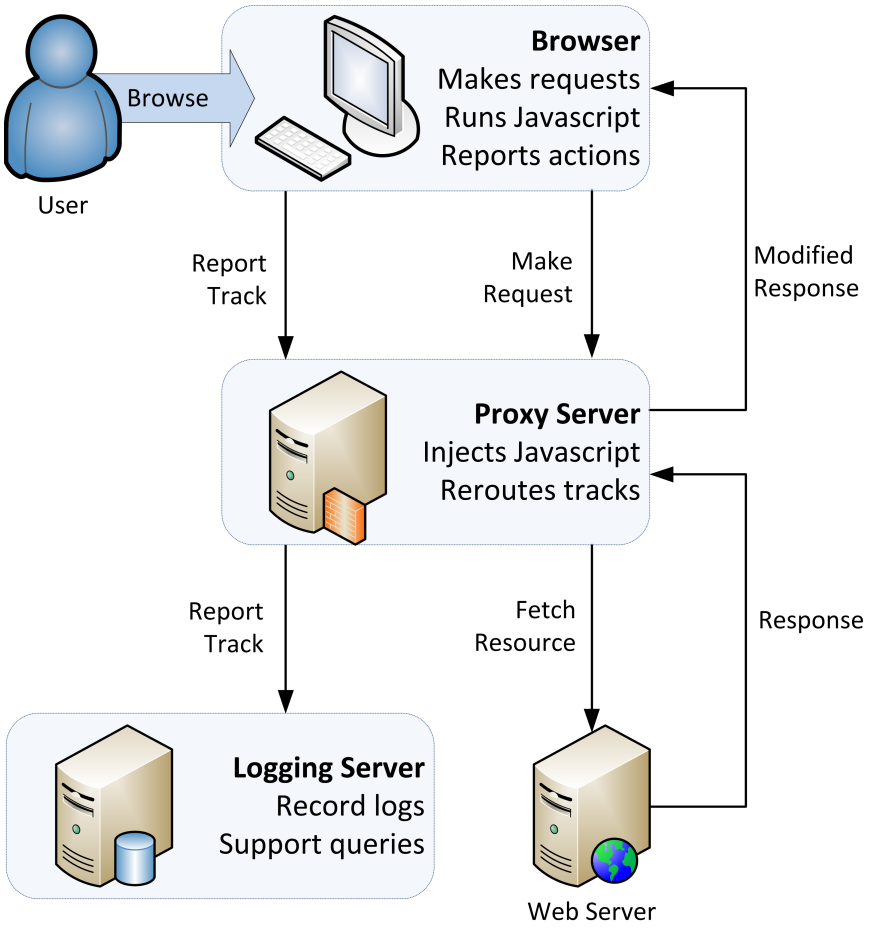


Figure 3.3: Architecture of SophieJS

3.4.1 Proxy component

The behaviour of the proxy component is as follows:

- Listen for incoming requests from the browser
- Read request from the browser
- Open the request, determine destination host
- Open connection to the destination, forward the request
- Read the response from the destination
- Open the response, check if MIME type is `text/html`
- If so, modify the page
- Send the (modified) response to the requester

Modification involves inserting Javascript code into the head of the response, along with some additional HTML into the body to support certain monitoring, such as printing. The "`Content-Length`" header of the HTTP response must also be modified to reflect the new size of the response, including the additional code. This is discussed fully in the implementation chapter, in section 4.2.

3.4.2 Browser component

The browser component has three responsibilities: to attach event listeners onto the appropriate DOM elements in the browser, to handle the events that are generated, gathering appropriate information, and to send these events back to the logging server in an efficient manner. The time at which events occurred must be logged with microsecond accuracy in order to evaluate the length of time an action took, and the gaps

in time between actions, in addition to ordering the events as they occurred (microsecond accuracy is more than sufficient for this purpose).

The actions must be sent for logging asynchronously - the user must not be hindered in their browsing experience. AJAX techniques, specifically XMLHttpRequests provide this functionality, by allowing HTTP messages to be sent by the browser in the background, unbeknownst to the user, as they browse the page.

3.4.3 Logging component

The logging component accepts a list of events that occurred in the browser. As the browser's means of communication is via HTTP actions (rather than simple socket-based communication), the logging component runs as an application on a HTTP server, and is invoked via a HTTP GET action by the browser, passing a list of events that have occurred since the last update. These events are semicolon separated, and attributes of the event being logged, such as time, event type and parameters are comma separated. A logging message thus takes the form:

```
1 GET /track.php?12345,click,http://www.tcd.ie,Trinity College Dublin
    ;12346,click,http://cs.tcd.ie,TCD CS;...
```

This is then stored in a MySQL database, and can be used for building user models, or for exporting to XML to be fed into the Sophie search system. A simple "view" function allows for outputting of the last 500 events to a web browser for quick analysis, and supports basic filtering functionality to isolate specific actions.

3.5 Design challenges

Throughout the course of designing and implementing the system, an number of challenges arose that required special consideration in the design:

3.5.1 Cross site scripting

Javascript, as a scripting language with code that is downloaded and executed while users browse the web, is inherently open to some security issues. One such type of attack is known as "cross site scripting" (XSS). This involves an attacker adding Javascript code to some website, for example by uploading the code as part of a comment that can be submitted by a user.

By adding this malicious code, the attacker could get access to the authentication token stored in a cookie by the website for the genuine user. This authentication token could be sent to the attacker via an off-domain XMLHttpRequest, thus enabling the attacker to authenticate as that user to the website on which the attack took place. This is of particular concern on financial services websites, where the authentication token could allow the attacker to transfer funds from the victim's account. In order to protect against these types of attacks, Javascript institutes a "same origin policy" [36], that only allows communication with the domain from which the Javascript was downloaded.

This represents a problem, because a user browsing `http://www.tcd.ie` through SophieJS will necessarially have Javascript code that sends logging updates to some other host, for example `logger.sophie.com`. By default, the browser's security measures will prevent this from occurring, thus preventing the system from logging the user's actions successfully.

In order to resolve this issue, the logging updates are instead directed at the original domain, but to a path that is known to the proxy and likely to be unique - such as `http://www.tcd.ie/track12345678.php`, where the digits are generated at random with each instance of SophieJS that is running. The proxy then detects HTTP requests from the client containing this path, and instead reroutes the request to the logging server. This assumes that the host website does not contain a resource named track12345678.php out of coincidence, but with each additional random digit, the prob-

ability of this occurring is reduced.

3.5.2 Transfer encodings

Two potential issues appeared with regards to HTTP transfer encoding:

Chunked transfer encoding

Chunked transfer is a method of splitting up parts of the response of a webpage, and is present in HTTP version 1.1 [37]. It is useful in dynamic web applications where the length of the content is not known in advance, and thus allows content to be sent as is it generated. This is problematic for the purposes of SophieJS, which requires the full page, so that it can be modified, before sending it to the client. This issue can be solved by downgrading the version of HTTP from 1.1 to 1.0 when the request is sent, at the expense of a slight increase in latency, as the server must now generate all of the content before it is sent.

Gzip

Gzipped transfer encoding involves the server compressing the response text in an attempt to reduce file size and thus bandwidth usage. However, this content would then need to be unzipped (and possibly rezipped) at the SophieJS proxy in order to permit modification, adding to load at the proxy. To solve this, the proxy checks for and removes gzip from the "`Accept-Encoding`" header of the outbound request. This comes at the expense of increasing bandwidth usage, but the saving of having to unzip content in the proxy.

3.5.3 Batching of updates

Ideally, messages notifying the log server of events should be sent as soon as the event occurs. However, the frequency at which events fire, in particular with mouse movement events, results in a large number of updates, and thus a large number of XMLHttpRequests. For performance reasons, browsers limit the number of concurrent XMLHttpRequests to be between 2 and 8 active requests (depending on browser) [38]. If an additional request is generated, it is typically queued.

This presents three issues. Firstly, the volume of traffic generated by creating a new HTTP connection for each log message is large. Secondly, if the user browses away from the page while there are queued requests, those requests are lost. If the mouse is being moved around the page, this can easily create more messages than can be served in the time. Finally, the clogging up of the XMLHttpRequest slots by SophieJS can prevent "genuine" requests being made by the page the user is viewing, resulting in a degraded experience for the user.

The solution is to batch updates to the log server. Updates are saved in a Javascript queue as the events happen, and a separate thread of execution loops between serialising the updates in the queue (removing them in the process), and sending the batched update to the logging server. As a result, only one XMLHttpRequest "slot" is used by SophieJS. When the page is exited, a final function to handle the unloading is called, which forces flushing of the queue before returning, ensuring that no event is lost.

3.5.4 Identifying users

A problem emerged with regard to the identification of users as they use the proxy, as it is crucial to see the path that an *individual* user took to reach a resource. Ideally, a cookie could be set in the user's browser to uniquely identify them, however this is not an option. As discussed in section 3.5.1, cookies can only be read from the website

that they were set for. As a result, the user could only be tracked on an individual domain. This is of little use for tracking their usage from a search engine through to a final site.

Tracking by IP address is the other option. It is not possible to determine the IP address of the host at the logger, since all requests for the logger will be sent via the proxy, and so the proxy's IP address will appear for all events. The solution is to add the IP address of the client to the end of the update sent via the proxy. This requires detection of the update and manipulation of the HTTP request to include the IP address, and thus a change to the "`Content-Length`" header.

Chapter 4

Implementation

4.1 Language and environment

There are three components to the implementation of SophieJS: the proxy, the monitoring code, and the logging server. The proxy is implemented in Java, and is thus cross-platform. The monitoring code is designed to operate in a wide variety of web browsers, though is specifically targeted for use in Mozilla Firefox, Google Chrome, and Apple Safari. These browsers operate on a variety of platforms, meaning that the monitoring component too can operate on these platforms. Finally, the logging server is implemented in PHP with a MySQL back-end, running via Apache on Linux.

The ability for SophieJS to operate in many browsers running on various platforms is crucial to the success of the project - the greater the compatibility with software used by users to browse the web, the more data can be collected for use in Sophie.

4.2 Proxy component

The proxy component accepts requests from the browser, fetches the resource, performs appropriate modifications, and returns the result to the browser. A UML class diagram

for the proxy component is given in figure 4.1

4.2.1 Accepting requests

The proxy listens for incoming connections on TCP port 8888 by default, though this can be configured by passing the required port number as an argument. The request is read from the connection and the appropriate `HttpCommand` and `HttpHeaders` objects are created from the data. This is required for three reasons: to disable chunking and gzip, and to reroute tracks.

Disabling chunking

Firstly, the HTTP version must be downgraded in order to stop chunked transfer encoding, which provides responses in chunks, thus complicating modification of the page. This problem was discussed in section 3.5.2, and is solved by reverting from HTTP version 1.1 to version 1.0. The version of the protocol that is being used is declared in the command (first) line of the request, which takes the form:

```
1 GET /path/to/resource.html HTTP/1.1
```

This is instead modified to request version 1.0:

```
1 GET /path/to/resource.html HTTP/1.0
```

Disabling gzip

Secondly, gzipped tranfer encoding is disabled, as it would require unzipping and rezipping in the proxy, adding additional load. This can be disabled by removing "gzip" as an accepted tranfer encoding in the "`Accept-Encoding`" header of the outbound request.

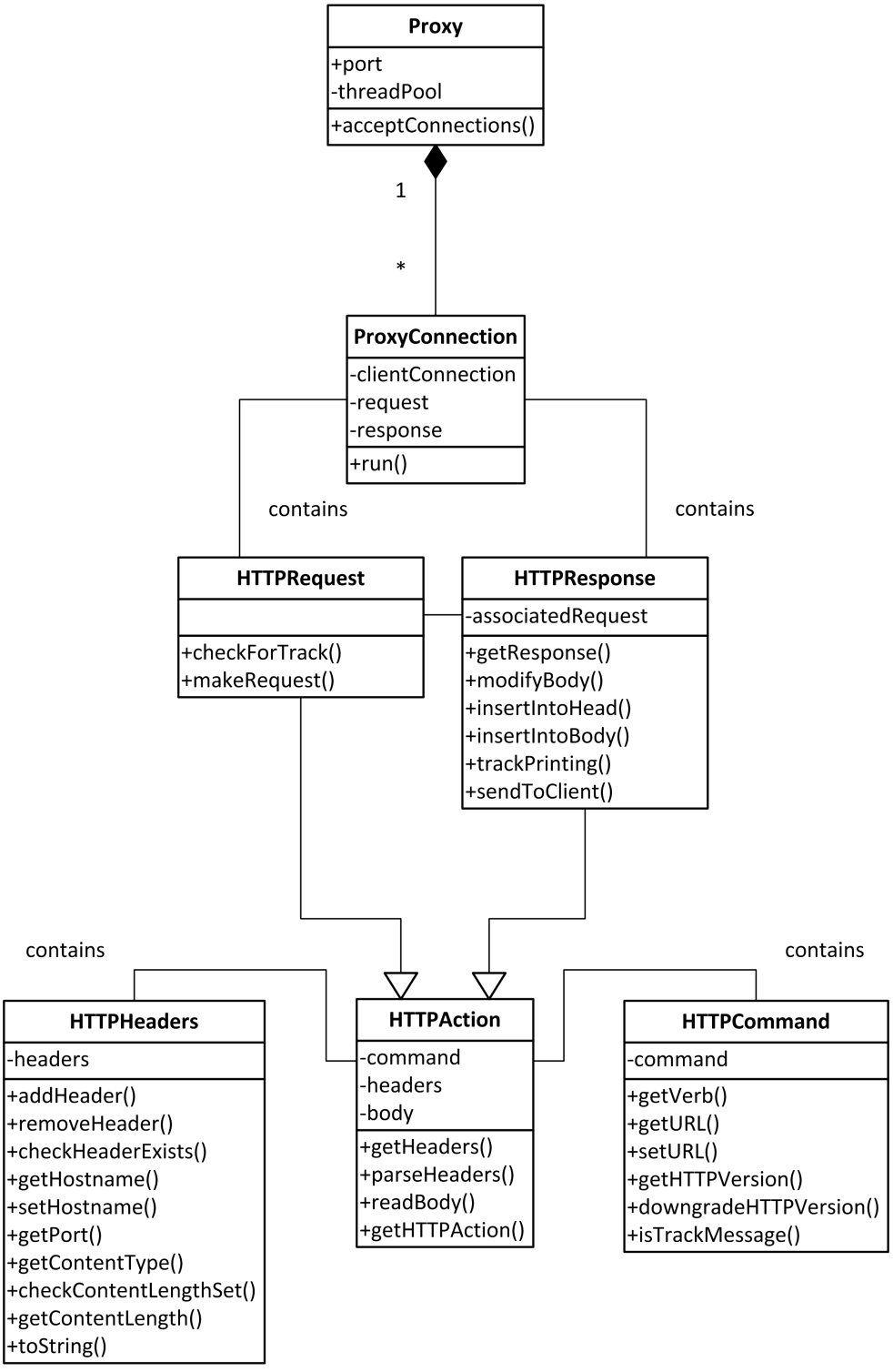


Figure 4.1: UML class diagram of proxy component

Rerouting tracks

Finally, due to Javascript's "same origin" policy, as discussed in section 3.5.1, the proxy must redirect logging messages to the log server. To solve the problem of XMLHttpRequest messages sent to other domains being denied, the log messages are sent to the domain that the user is visiting, and then intercepted at the proxy. The detection is achieved through the addition of a randomised trackID with each instance of the SophieJS proxy. This trackID appears in the URL for all logging messages. If the trackID is found in the request URL, the URL of the request and the host header are changed to direct the request to the logging server.

Figure 4.2 illustrates these modifications, which are made possible through methods provided on the `HttpCommand` and `HttpHeaders` classes. Once the modifications have been made, the request is forwarded to the appropriate web server, as defined in the `Host` header.

4.2.2 Modifying responses

When the modified request is sent to the host, the socket on which the response will be provided is passed to a `HttpResponse` object, which reads the response and builds the corresponding `HttpCommand` and `HttpHeaders` objects. Modifications to the page begin if the content is of MIME type `text/html`, as defined in the "`Content-Type`" header of the response. For non-HTML content types, such as for included images, CSS stylesheets and other downloads that occur over HTTP, no modification occurs, and the response is passed directly to the browser without change.

Because SophieJS uses Javascript to attach event listeners to DOM objects in the browser, the full HTML page does not need to be parsed in the proxy, enhancing performance and maintaining compatibility. Three modifications occur: Javascript is

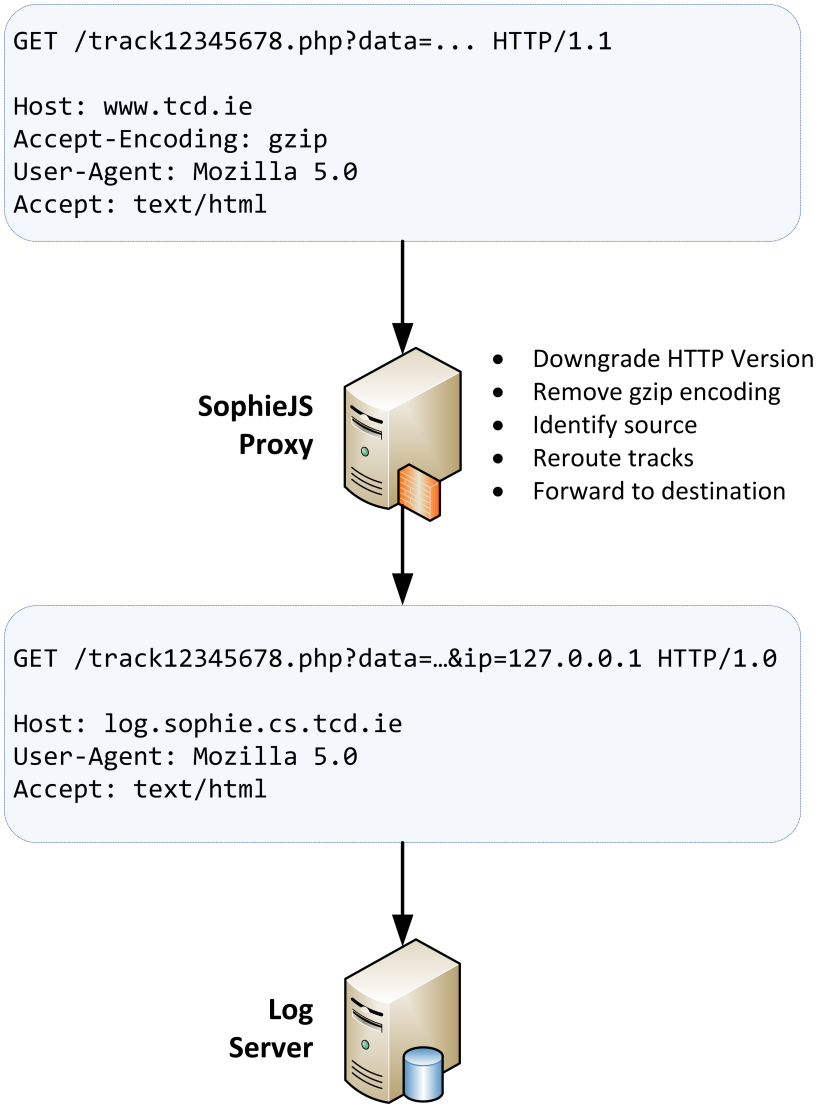


Figure 4.2: SophieJS proxy request handling

included, print tracking is added, and the content length is recalculated.

Include Javascript

The tracking Javascript must be inserted into the head of the HTML response. This is achieved by locating the closing `</head>` tag, and inserting the required code, wrapped in `<script>` tags inside the head of the document. The Javascipt code that is inserted into the document is read from text files, allowing simple future modification of SophieJS via configuration, rather than recompilation. This included Javascript is loaded and executed by the browser in order to provide the tracking functionality.

Track Printing

There are no standard event handlers that can detect a printing action. Given that printing a document is a high-weight action - i.e. that the page is of great interest to the user - a secondary method of capturing print actions is used.

The Cascading Style Sheets (CSS) specification [39] defines a media type for CSS included in HTML documents, such that different styling can be applied to the page contents in different situations, such as on-screen, on mobile devices, when being projected, and when printed. CSS can apply a background image to specific page elements.

In order to capture print events, a hidden, empty `div` element is added to the page, just before the closing `</body>` tag. This element does not appear on screen or on paper, it is invisible to the user. However, when the browser is rendering a page for print, it will use the CSS specified for print media. SophieJS also includes a CSS definition for the hidden element that defines a background image, which is loaded during the rendering for print. This loading action can track the print event.

The CSS required to achieve this result on the empty `div` element named "track12345678" is:

```
<style type="text/css">
  @media print {
    div.track12345678 {
      background-image:url(track12345678.php?action=print);
    }
  }
</style>
```

This CSS is included in the head of the document, which causes the browser to load the "`track12345678.php?action=print`" resource when rendering a page for printing. This URL is part of the logging server, which records the event.

Figure 4.3 illustrates these modifications. Once the modifications have been made, the response is forwarded back to the requesting client.

Modify Content-Length

As new data has been added to the response, the length of the response will change. The `Content-Length` header must be recomputed to alert the browser to the new length of the page.

4.2.3 User identification

In order to separate the logs generated by any two users, their hostname or IP address must be recorded as part of the log as identification. This must be done in the proxy, as the logging server cannot identify the origin of the logs (since the logs pass through the proxy), nor can the user's browser provide reliable information. If a tracking message is detected, the hostname/IP address of the message sender is appended on to the end of the message, and the content length of the message is recalculated.

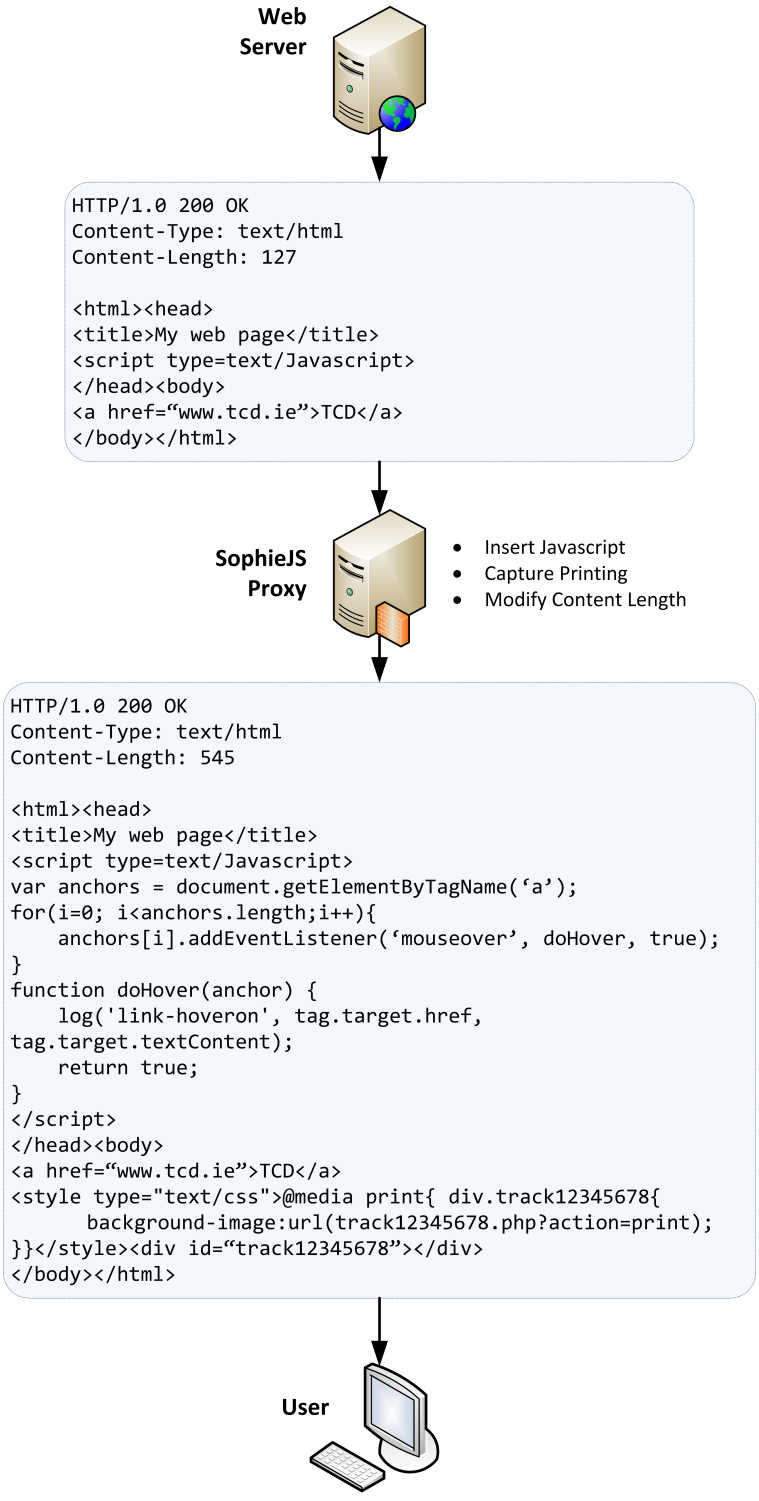


Figure 4.3: SophieJS proxy response handling

4.3 Browser component

The browser component consists of Javascript code, which performs three functions: attaching event listeners to the elements in the page, handling the events that are generated, and sending information to the log server.

4.3.1 Attaching event listeners

In order to receive event notifications from the browser, it is necessary to attach event listeners to the appropriate elements. Event listeners can be added using the Javascript code:

```
1 element.addEventListener("eventname", handlerFunctionName, true);
```

To monitor, for example, actions relating to links, a list of all anchor (<a>) elements is retrieved, and the appropriate event listeners are added to these elements, as follows:

```
1 var tagList = document.getElementsByTagName("a");
2 for(var i = 0; i<tagList.length; i++){
3   var currTag = tagList[i];
4   currTag.addEventListener("mouseover", trackLinkHoverOn, true);
5   currTag.addEventListener("mouseout", trackLinkHoverOff, true);
6   currTag.addEventListener("mousedown", trackLinkClick, true);
7 }
```

Where "trackLinkHoverOn", "trackLinkHoverOff" and "trackLinkClick" are defined event handlers, as described in the next section.

This procedure is used to attach appropriate event handlers to:

- The body element
  ⇒ captures mouse moves & clicks

- Form elements

  ⇒ captures focus, blur and submission

- Anchor elements

  ⇒ captures hovers, unhovers and follows

Additionally, there are some event handlers that are attached directly to the DOM window and document objects:

- The window object

  ⇒ captures scrolling, page unloads and text selection

- The document object

  ⇒ captures keyboard presses

4.3.2 Handling events

Each of the event handlers specified in the previous section are also defined in order to capture the event and meaningful information associated with it. In addition to the user's hostname/IP address, the URL of the page where the event occurred, and the time at which it occurred (to microsecond accuracy), the following attributes are recorded:

Mouse actions

Move	x and y coordinates of the cursor's location on screen
Click	Button used (left, middle, right), x and y coordinates of the click location on screen
Unclick	Button used (left, middle, right), x and y coordinates of the click location on screen, and selected text, if any
Scroll	x and y coordinate of the top-left visible pixel, relative to the top-left corner of the page

Through click and unclick information, it is possible to infer if dragging occurred.

Link actions

Hover on	Link text and URL pointed to
Hover off	Link text and URL pointed to
Follow	Link text and URL pointed to

Keyboard actions

Keypress	Character press and modifier keys

SophieJS detects keyboard shortcuts and identifies their use, including:

- Ctrl-C (clipboard copy)
- Ctrl-X (clipboard cut)
- Ctrl-V (clipboard paste)
- Ctrl-P (print)
- Ctrl-D (bookmark)

- Ctrl-F (find in page)
- Ctrl-S (save)
- Ctrl-R (refresh page)
- F5 (refresh page)

Form actions

Focus	Form element name/id and type, form name/id and submit location
Blur	Form element name/id and type, form name/id and submit location
Keypress	Form element contents, form element type and name/id, form name/id and form action location
Submission	All form element's contents, type and name/id, form name/id and form action location

"Form element" refers to the individual field in the form that the action occurs on, for example a specific textarea or selection box. "Form" refers to the form that the element is contained in.

Page actions

Load	Browser user agent, screen resolution
Unload	Page title and duration of time spent on page in milliseconds

4.3.3 Logging events

As was discussed in section 3.5.3, there is a maximum number of concurrent XMLHttpRequests permitted in browsers, leading to a possible loss of queued log messages if the user navigates away from the page. To prevent this from occurring, a "log" function is used. All event handlers call the log function to record their data, rather than opening individual XMLHttpRequests.

The log function appends the data onto a list, and starts a flush, if it is not already running. The flush function removes all items from the list and serialises them into a single string. It opens a single XMLHttpRequest, and sends the serialised log data in batch form. Once this XMLHttpRequest is complete, the flush function repeats (unless the log is empty). The items in the list are separated by a semicolon, while the attributes of a log message are comma separated. A logging message thus takes the form:

```
1 GET /track.php?12345,link-follow,http://www.tcd.ie,Trinity College
    Dublin;12346,link-follow,http://cs.tcd.ie,TCD CS;...
```

This solves the issue of lost updates, bandwidth usage, and denial of service to genuine XMLHttpRequests used by the website being visited by the user.

4.4 Logging component

The logging component is implemented in PHP with a MySQL database back-end. During implementation, this was deployed on a Linux server running lighttpd, though is it capable of running on any PHP-compatible web server on any operating system.

The logging server provides three functions: single event logging, batch event logging, and viewing. The database schema is given in figure 4.4.

Figure 4.4: Logging server database schema

4.4.1 Single event logging

The logging server accepts a HTTP GET action with the following parameters:

- **ip** - IP address or hostname of event origin
- **action** - name of the action that is being logged
- **x** - first parameter
- **y** - second parameter
- **time** - time in microseconds since Unix Epoch
- **url** - URL on which the event occurred

4.4.2 Batch event logging

To support batched updates, the log server accepts a HTTP GET action with the following parameters:

- **ip** - IP address or hostname of event origin
- **vals** - a semicolon-separated list of events, where each event is a comma separated list of the following values, in order:
  - time in microseconds since Unix Epoch
  - name of the action that is being logged
  - first parameter
  - second parameter
  - URL on which the event occurred

4.4.3 Viewing

A viewing function is available on the logging server for testing purposes. It provides a table showing all fields in the database, ordered by time. Basic filtering is available to show only certain types of events or exclude certain types of events. Due to the volume of mouse movement logs, this filtering is useful. A screenshot of the view function of the logging server can be seen in figure 4.6 at the end of this chapter.

4.5 Transparent deployment

SophieJS improves upon the Sophie browser plugin in three ways: it operates in a wide variety of browsers on a multitude of platforms, it does not require software installation, and optionally, it can be deployed transparently. By deploying transparently, no configuration of the user's browser is required by an administrator, thus increasing the number of SophieJS users, and thus the volume of data collected, dramatically.

This can be achieved by redirecting outbound HTTP traffic to the SophieJS proxy at the router [40], as can be seen in figure 4.5.

This is of particular use in an office or laboratory environment, where users access the Internet via a single router. In order to redirect the outbound HTTP traffic, routing rules must be setup on the router to forward HTTP requests to the proxy. If iptables [41] are used for routing, as is the case on most modern Linux distributions and the DD-WRT router firmware [42], applying the following rules fulfils this role:

```
1 iptables -t nat -A PREROUTING -i eth0 -s ! proxy.sophie.cs.tcd.ie -p
     tcp --dport 80 -j DNAT --to proxy.sophie.cs.tcd.ie:8888
2 iptables -t nat -A POSTROUTING -o eth0 -s 192.0.0.0/16 -d proxy.sophie
     .cs.tcd.ie -j SNAT --to localhost
```

The first of these rules applies to incoming traffic on the `eth0` interface on TCP port 80, and redirects that traffic to the proxy (sophie.cs.tcd.ie), assuming the traffic did

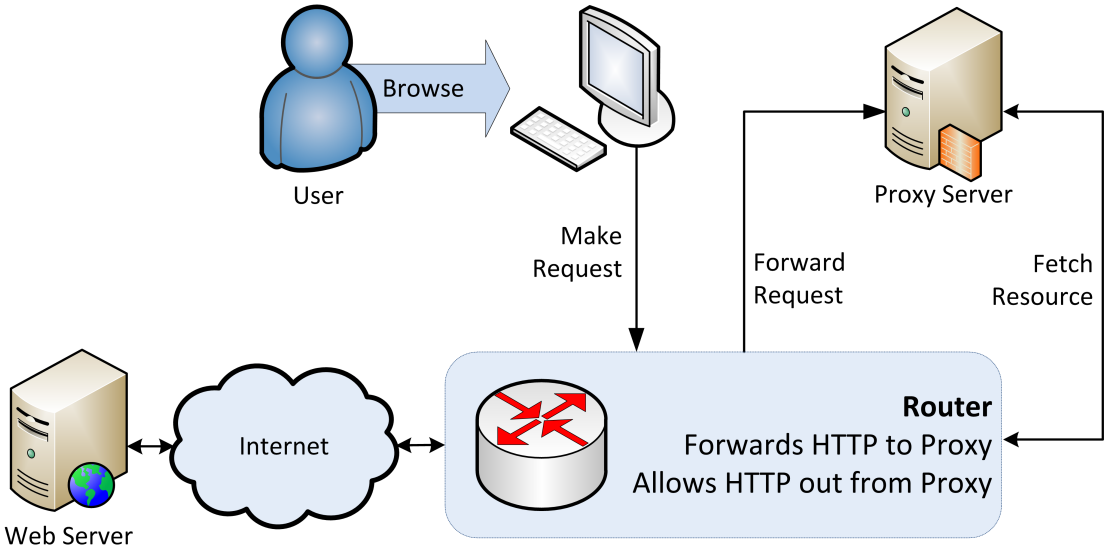
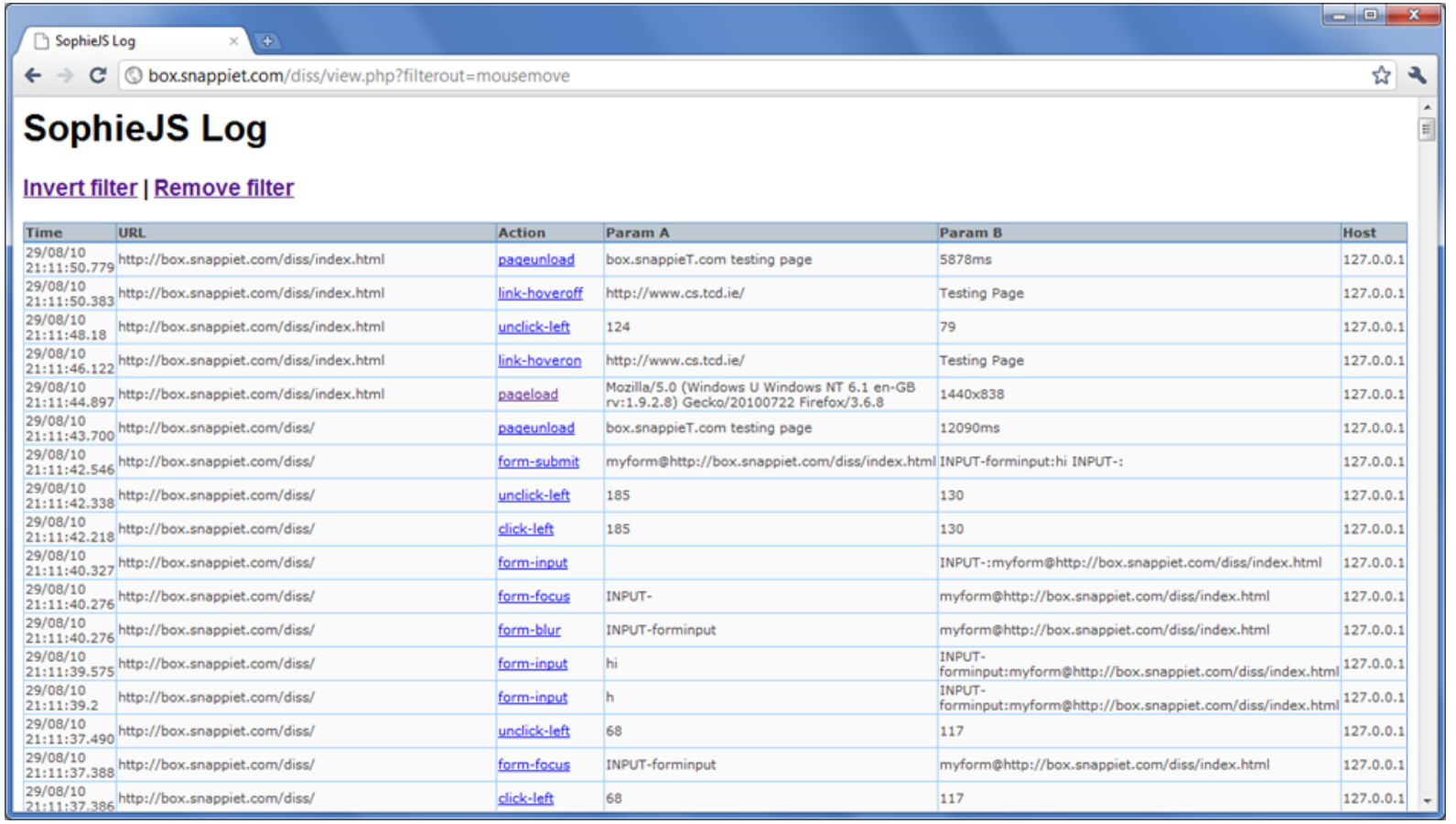


Figure 4.5: Transparent deployment of SophieJS

not come from the proxy itself. The second rule ensures that these forwarded packets appear to come from the router rather than the individual client. This is necessary when the proxy resides inside the same network as the users, as the reply must return via the router so as not to break TCP.

SophieJS can operate via manual configuration of the proxy in the user's browser, or as an intercepting proxy. In each case, HTTPS traffic can be considered separately from HTTP traffic, and be permitted to access the resource directly, rather than via the proxy.



Time	URL	Action	Param A	Param B	Host
29/08/10 21:11:50.779	http://box.snappiet.com/diss/index.html	pageunload	box.snappieT.com testing page	5878ms	127.0.0.1
29/08/10 21:11:50.383	http://box.snappiet.com/diss/index.html	link-hoveroff	http://www.cs.tcd.ie/	Testing Page	127.0.0.1
29/08/10 21:11:48.18	http://box.snappiet.com/diss/index.html	unclick-left	124	79	127.0.0.1
29/08/10 21:11:46.122	http://box.snappiet.com/diss/index.html	link-hoveron	http://www.cs.tcd.ie/	Testing Page	127.0.0.1
29/08/10 21:11:44.897	http://box.snappiet.com/diss/index.html	pageload	Mozilla/5.0 (Windows U Windows NT 6.1 en-GB rv:1.9.2.8) Gecko/20100722 Firefox/3.6.8	1440x838	127.0.0.1
29/08/10 21:11:43.700	http://box.snappiet.com/diss/	pageunload	box.snappieT.com testing page	12090ms	127.0.0.1
29/08/10 21:11:42.546	http://box.snappiet.com/diss/	form-submit	myform@http://box.snappiet.com/diss/index.html	INPUT-forminput:hi INPUT-:	127.0.0.1
29/08/10 21:11:42.338	http://box.snappiet.com/diss/	unclick-left	185	130	127.0.0.1
29/08/10 21:11:42.218	http://box.snappiet.com/diss/	click-left	185	130	127.0.0.1
29/08/10 21:11:40.327	http://box.snappiet.com/diss/	form-input		INPUT-:myform@http://box.snappiet.com/diss/index.html	127.0.0.1
29/08/10 21:11:40.276	http://box.snappiet.com/diss/	form-focus	INPUT-	myform@http://box.snappiet.com/diss/index.html	127.0.0.1
29/08/10 21:11:40.276	http://box.snappiet.com/diss/	form-blur	INPUT-forminput	myform@http://box.snappiet.com/diss/index.html	127.0.0.1
29/08/10 21:11:39.575	http://box.snappiet.com/diss/	form-input	hi	INPUT-forminput:myform@http://box.snappiet.com/diss/index.html	127.0.0.1
29/08/10 21:11:39.2	http://box.snappiet.com/diss/	form-input	h	INPUT-forminput:myform@http://box.snappiet.com/diss/index.html	127.0.0.1
29/08/10 21:11:37.490	http://box.snappiet.com/diss/	unclick-left	68	117	127.0.0.1
29/08/10 21:11:37.388	http://box.snappiet.com/diss/	form-focus	INPUT-forminput	myform@http://box.snappiet.com/diss/index.html	127.0.0.1
29/08/10 21:11:37.386	http://box.snappiet.com/diss/	click-left	68	117	127.0.0.1

Figure 4.6: Log server view function

Chapter 5

Evaluation

5.1 Event coverage

SophieJS is a viable alternative to the Sophie browser plugin for data capture. While the data that SophieJS can capture is limited by the information made available to the Javascript engine by the browser, it successfully captures a large proportion of the events monitored by the Sophie browser plugin.

Sophie JS can capture:

- Page loads and unloads
- Duration of time spent on page
- Mouse moves
- Mouse clicks and unclicks
- Scrolling
- Form contents

- Form item focus and unfocus
- Form submission
- Keyboard presses, including modifier keys
- Link hover and unhovers
- Link follows
- Selected text
- Searching for text in page
- Printing

Though it is not possible to access actions conducted via the browser's menu system, such as clipboard actions, bookmarking and page saves, keyboard shortcuts mapping onto these actions are captured and identified appropriately. As a result, SophieJS can match the Sophie browser plugin's event coverage with the exception of:

- Menu-driven clipboard actions
- Menu-driven bookmarking actions
- Menu-driven save actions
- Tab identification

This high degree of functionality allows for SophieJS to replace an installed browser plugin in many circumstances.

5.2 Compatibility

SophieJS operates on many platforms, and can collect data inside many browsers. This alleviates the necessity to write and deploy different browser plugins for each browser, for each version of the browser, and potentially for different platforms. SophieJS has been tested in the following popular desktop browsers:

- **Mozilla Firefox** version 3.6.8
- **Google Chrome** version 6.0.472.53
- **Apple Safari** version 5.0.1
- **Opera** version 10.61
- **Microsoft Internet Explorer** version 8.0

all of which were the most recent versions available at the time of writing on the target operating system, Microsoft Windows 7 Professional.

SophieJS has full compatibility with Mozilla Firefox, Google Chrome and Apple Safari, and can capture and report events as listed in section 5.1. Opera has near-full compatibility, and can monitor most events with the exception of scrolling and printing. Microsoft Internet Explorer has limited support, in that it can only monitor page loads and unloads. This is due to the different way in which event handlers are registered with the browser, and non-standard XMLHttpRequest behaviour on earlier releases of the browser. With additional work, SophieJS can be extended to support Internet Explorer in full.

Furthermore, compatibility with mobile platforms was tested, specifically Mobile-Safari on Apple iOS 4.0.2, which runs on Apple's iPhone, iPad and iPod Touch, the most popular mobile browsing platform, according to recent figures [43]. Accessing mouse movement information is not possible on touchscreen devices, but SophieJS still

detected page loads and unloads, scrolling, form focus and blurring, keyboard events and tap locations. This data capture would not be possible without a solution like SophieJS, as browser plugins are typically not installable on mobile devices.

SophieJS fails silently. While it may not be possible to always collect user data, this failure will never prevent the user from accessing and using the page as they intended.

5.3 Performance

Due to the nature of SophieJS, and of proxy services in general, there is a cost to their use.

5.3.1 Download latency

Because SophieJS makes modifications to pages, the whole page must be downloaded to the proxy, then modified, and then sent to the client. As a result, there is a delay between the browser requesting the page, and the page being displayed. Ordinarily, a browser can load a page as the contents are streamed to it, especially when chunked transfer encoding is being used. However, this is disabled in SophieJS (as described in section 3.5.2). In any case, the entire page needs to downloaded to the proxy in full before it can be sent to the client.

This is not of great concern if the proxy is located close to the client, in that the download from the web server to the proxy will take no longer than a download to that client, and that the time required to send the page from the local proxy to the client is minimal. A problem does arise when large files are being downloaded, for example PDF documents which are several megabytes - once again, the entire document is downloaded to the proxy before it is sent to the client, despite no modification being necessary. This is problematic for file types such as PDFs and images, which can typically be displayed as they load in the browser. In its current incarnation, SophieJS

does not support this, however future work (as described in section 6.2.1) can solve this problem with relative ease.

5.3.2 Download size

SophieJS inserts additional content into every HTML page in order to monitor user actions, thus increasing the download size of the page. When the proxy is close to the user, this generally does not cause an issue for download speeds, since the small amount of additional data can be transferred without great cost.

According to research by Google this year, which involved sampling 380 million of the "top" websites, the average webpage download today is 477.26 KB [44]. This represents the loading of the whole page, without compression, and including images, CSS and other resources used on the page. In its current state, SophieJS adds 79.19 KB of additional text onto the response to allow for tracking, an increase of approximately 16.5% on average page sizes.

This filesize can be reduced via Javascript refactoring, such as removing whitespace, reducing the length of variable names and compressing the code through alternative encoding [45, 1]. Using these techniques, the SophieJS code can be reduced to 45.67 KB, approximately 60% of the size of the original includes. This also serves to obfuscate the code. When the SophieJS included code is reduced in this manner, it is less than 10% of the total page size. As has been mentioned, if the proxy resides on the local network, this penalty is minimal.

5.3.3 Traffic generated

The browser component of SophieJS captures user actions and sends these actions to the logging server. These messages represent an increased volume of traffic that is generated as a user browses a website. Depending on the browser, as many as 60 events can be captured per second, all of which must be sent back to the logging server.

The length of each of these messages is variable, since the captured data is different for each update, and the URL is included in the message. A sample update, from the most frequently generated event, the mouse move action:

```
1283790448567,mousemove,998,183,http://scss.tcd.ie/;
```

is 52 bytes long. With as many as 60 updates per second, this generates approximately 31KB/s of additional traffic for logging purposes. Once again, this is generally not an issue if the proxy is on the same LAN as the user who is being monitored, and assumes that 60 updates per second are being generated, which tends only to occur during times when the user is moving the mouse.

In a test of realistic browsing, an average of 9.7 updates per second were generated, at 82 bytes per update, or just under 8KB/s of traffic. When logging of mouse moves was disabled, 2.1 updates per second were generated on average, at 97 bytes per update, totalling approximately 0.2KB/s of traffic. These figures show the proportion of updates that are related to mouse movements (78%), and that those updates are also among the most compact, as they only need report the x and y coordinates on screen.

In tabular form, the results are:

Test	Updates	Update size	Traffic generated
Maximum mouse moves	60/s	52 bytes	31 KB/s
Real-world (including mouse moves)	9.7/s	82 bytes	8 KB/s
Real-world (excluding mouse moves)	2.1/s	97 bytes	0.2 KB/s

The traffic generated can be reduced by removing redundant information from each update. Since updates are batched, it is possible to provide the URL of the page and the start time only once per batch, and provide deltas from these values for each update in the batch. Furthermore, the names of the events that are logged could be coded to some shorter name understood by the logging server. Finally, the entire batch update

message could be compressed before being sent to the log server.

5.3.4 Caching

Through the implementation of caching, the latency issues can be reduced, and the speed & general scalability of SophieJS can be improved. Caching can be implemented inside SophieJS as future work, or alternatively, an existing web caching solution, such as Squid, can be placed between Sophie and the users, as seen in figure 5.1.

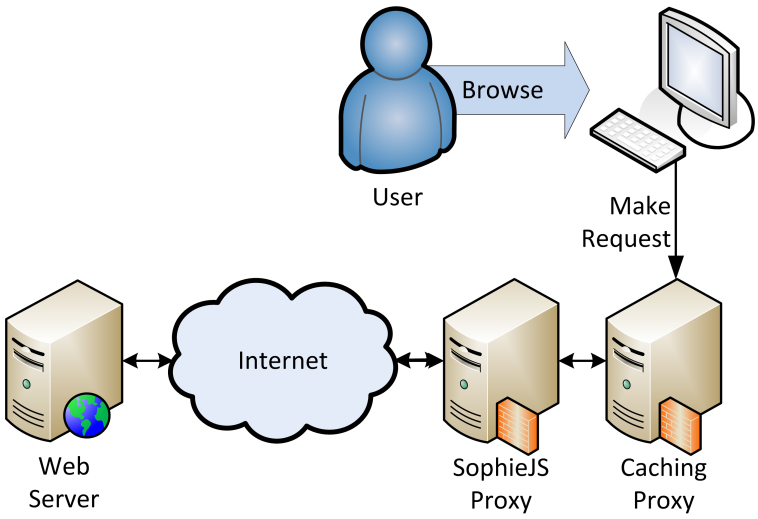


Figure 5.1: Caching SophieJS responses

Using this method, the modified results of requests that SophieJS processes can be cached by the caching proxy. When that resource is requested again, the caching proxy can serve the request without contacting the requested site, thus reducing the load on the SophieJS proxy and the internet connection. This reduces time taken to download (since the result is available locally), and latency (since the page needs to be modified once, but can be served several times). The SophieJS proxy can also support more users, since fewer requests need to be served by SophieJS, and are instead handled by the caching proxy server.

5.4 Uses of SophieJS-generated data

The data that is captured by SophieJS has many potential applications, including within the Sophie project itself, in other academic studies, and in a commercial environment.

5.4.1 Sophie research

The original scope of this project was to provide a method of generating vast quantities of usage data in support of ongoing research into user models in Sophie [27, 26, 46]. With direct access to the MySQL database, the large volume of data collected from each user, and the transparent deployment possibilities of SophieJS, this data is now readily available for capture and analysis, improving Sophie's user models.

5.4.2 Input to the Sophie engine

The Sophie browser plugin generates XML documents listing the events that occurred on a page as the user browses it. These XML documents take the form:

```
<page>
 <sophie_plugin_id>0</sophie_plugin_id>
 <window_id>0</window_id>
 <tab_id>0</tab_id>
 <previous_window_id>0</previous_window_id>
 <previous_tab_id>0</previous_tab_id>
 <url>
   <url_opened>http://scss.tcd.ie/</url_opened>
   <page_title>Home - School of Computer Science and Statistics :
      Trinity College Dublin</page_title>
   <file_id>249.txt</file_id>
   <opened_by_link>false</opened_by_link>
 </url>
 <focused_time>6407</focused_time>
 <start_time>31676060</start_time>
 <events_occurred>
   <event>
     <incremental_id_event>0</incremental_id_event>
     <id_event>5</id_event>
     <event_type>SubmitData</event_type>
     <detail_1>7 1</detail_1>
     <detail_2 />
     <event_occurred_ms>31682467</event_occurred_ms>
   </event>
 </events_occurred>
 <stop_time>31682467</stop_time>
</page>
```

While there is not a one-to-one mapping of data elements in the XML document and the events stored in the SophieJS database, building a tool to export database entries into this form would not be a difficult proposition. This tool could run periodically, taking data from SophieJS, building XML files from the data, and submitting those files to the Sophie engine. This allows SophieJS to be used directly in a production

Sophie system, contributing to the knowledge that Sophie has about the actions of users, permitting analysis and ultimately making an impact on the ranking data that Sophie provides.

5.4.3 Eye tracking studies

Increasingly, research is being carried out to evaluate the focus of user's attention on web pages [47]. Modern eye tracking techniques involve affixing a cap with mounted cameras to the subject's head. These cameras detect the eye's "gaze" relative to the screen, and can thus indicate what part of the page the user is looking at [48]. From this, maps indicating attention "hotspots" can be built, as seen in figure 5.2, and used to reorganise a webpage, or set prices for advertising units.

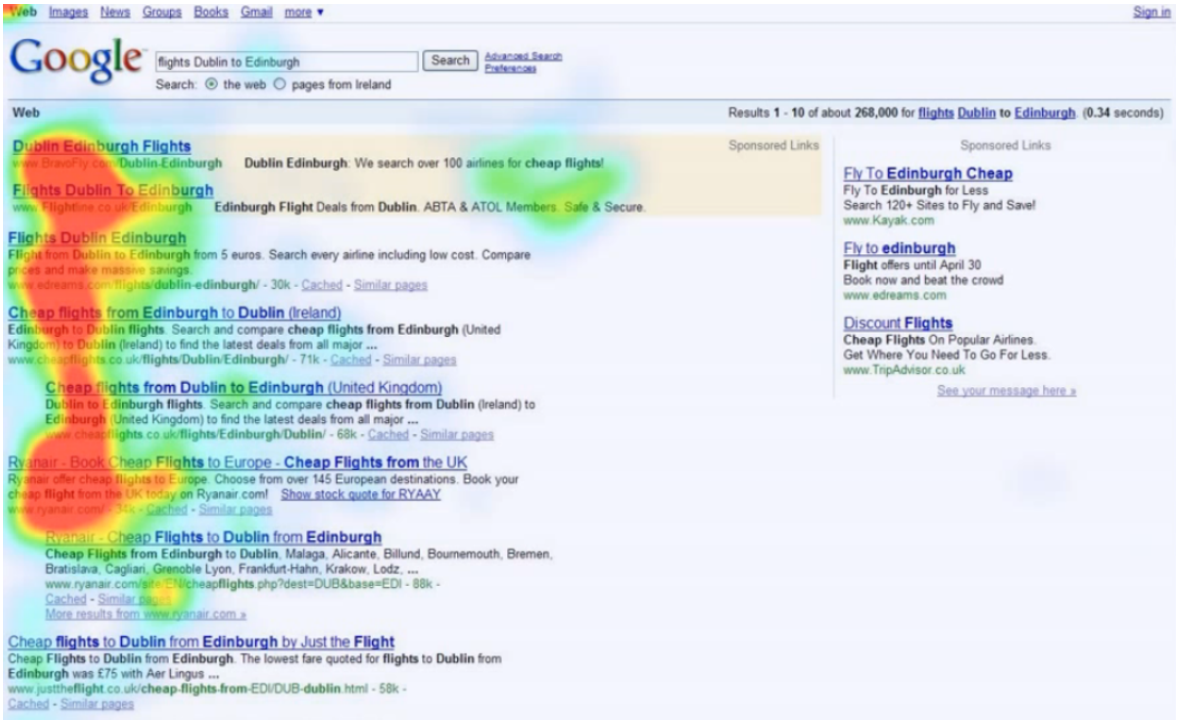


Figure 5.2: Eye tracking heat map [49]

While lacking the accuracy of dedicated eye tracking solutions, SophieJS can provide

an inexpensive, simple alternative to eye tracking by monitoring mouse movements, and requesting subjects to keep the mouse cursor at the location on screen that they are looking. This can be useful for initial research, or as a cost-friendly alternative to purchasing eye tracking equipment.

5.4.4 Addiction studies

As SophieJS tracks user's actions while they browse the Internet, it is possible to detect repetitive actions, such as frequent checking of email, moving the mouse while the page loads, selecting a block of text during reading, and clicking on form elements, such as checkboxes or radio buttons, multiple times. This data can be analysed automatically to detect patterns, or used to monitor the behaviour of a (consenting) individual for investigation of compulsive actions.

5.4.5 Usability studies

In a commercial environment, SophieJS technology can be used on individual websites, by placing the SophieJS proxy between the web server and the Internet, as shown in figure 5.3. This gathers data on behalf of the site owner that can then use used to improve usability, such as the number of clicks, scrolls and form entries required to achieve a goal.

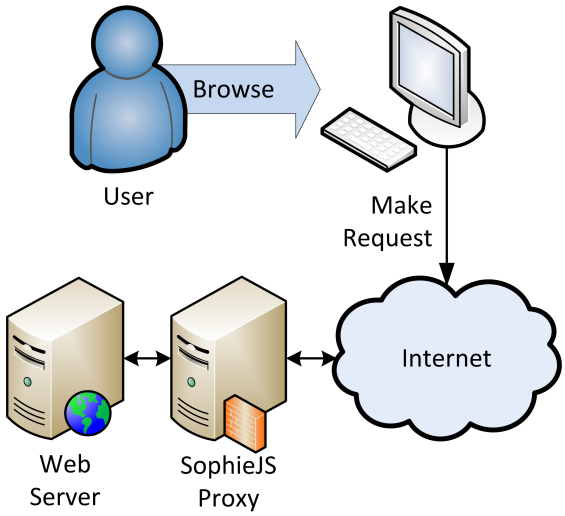


Figure 5.3: Monitoring private websites with SophieJS

Chapter 6

Conclusions & Future Work

6.1 Project evaluation

This project involved researching, designing, implementing and testing a solution to the limitations of the Sophie browser plugin. This has been achieved, and a fully-operational system was built, ready for deployment as intended.

Research included an evaluation of search technologies in use and being developed today, examination of the existing browser plugin, and evaluation of possible approaches to solve the problem. The solution is a ready-to-use, simple to deploy, proxy-based system, where monitoring can occur regardless of browser or operating system, and without explicit installation. The results of this monitoring rival the event coverage of the Sophie browser plugin, and in some cases extend upon it.

The result of this research is a tool with many uses, not least for generation of vast volumes of data on which Sophie's user models can be based. The requirements set out in section 3.1 have all been met, producing a system that captures and stores browsing data on multiple browsers and operating systems without serious performance issues or installation of plugin software.

SophieJS is also extensible - the included Javascipt is read from a series of text files

in the same directory as the proxy while it runs. As a result, many new events can be added to the system with ease, and unnecessary events can be disabled as required. While more complex events (such as printing) require modification and recompilation of the proxy itself, a great deal can be achieved through reconfiguration of the Javascript include files.

There are plans for SophieJS to be deployed in an experimental setting in TCD in the near future, leading to the data capture of thousands of user actions, thus improving the quality of user models in Sophie and consequently the quality of results provided by the Sophie search engine.

6.2 Future Work

While SophieJS in its current incarnation is a fully operational system, future work can develop the system further.

6.2.1 Performance improvements

There are a number of potential improvements to the performance of SophieJS:

Reduce latency

Currently, SophieJS downloads a file in full, checks if modifications are necessary, and then sends it to the client. This is unavoidable for HTML files, which need modification, but it introduces a needless delay on other files transferred over HTTP. For example, downloading a 10MB PDF will download the entire file to the proxy before sending it to the client.

However, the MIME type of the resource being fetched is known once the headers have been loaded and parsed. A potential optimisation would involve streaming the

data directly to the client as it is downloaded when the file is not a candidate for modification (i.e. not an HTML document).

Reduce SophieJS size

As was described in section 5.3.2, the SophieJS code adds 79.19 KB to the length of a page, but this can be reduced to 45.67 KB via Javascript shortening techniques, such as renaming variables to have smaller length names, removing extraneous whitespace, and compressing the text of the code. While this is possible to do offline, and to load the compressed text from the Javascript include files in the proxy, this would hinder future modification of said Javascipt.

Alternatively, the Javascript could be compressed by the proxy between loading it from the files and sending it to users. This approach maintains the adaptability of SophieJS while reducing the footprint of the SophieJS monitoring code sent as part of every response.

Reduce update size

Section 5.3.3 considered the traffic generated by updates bound for the logging server, which, in batch mode, take the form of:

```
1 http://www.tcd.ie/,1283791075125,link-hoveroff,http://mail.tcd.ie,
    Email;
2 http://www.tcd.ie/,1283791075148,link-hoveron,http://cs.tcd.ie,
    Computer Science;
3 http://www.tcd.ie/,1283791075205,link-follow,http://cs.tcd.ie,Computer
     Science;
4 ...
```

This format consists of individual updates, semicolon separated, but features a high level of overlap. For example, the page from which the events are generated (http://www.tcd.ie) is the same in all cases, and the time that the events occurred

(1283791075125 for the first event) change very little between events. To improve on this format, and thus reduce the size of update messages, an "initial" time and URL can be sent, and variations from that listed in logs. For example, the above update could become:

```
1 http://www.tcd.ie/,1283791075125,link-hoveroff,http://mail.tcd.ie,
     Email;
2 ,+23,link-hoveron,http://cs.tcd.ie,Computer Science;
3 ,+80,link-follow,http://cs.tcd.ie,Computer Science;
4 ...
```

This reduces the size of the update without the loss of any data. It may also be possible to compress the entire message, while considering the impact on the CPU usage of the browser, given that updates are frequent.

6.2.2 HTTPS support

SophieJS is a HTTP proxy, but does not support secure HTTP (HTTPS). Because different proxies can be assigned to different protocols in the browser, this is not an issue, and traffic bound for a HTTPS address does not go via the proxy, and therefore is not logged. It is possible to support HTTPS with a number of modifications, though HTTPS cannot be used when transparent proxies are in use.

The question remains, however, if logging websites delivered to the user via HTTPS is appropriate. In particular, because SophieJS can capture keypresses and form submission, operating SophieJS on Internet banking sites would be questionable from a security and ethical perspective. Nonetheless, it is a possible improvement to SophieJS that could be added in the future.

6.2.3 Increase browser coverage

SophieJS does not currently capture events in Microsoft Internet Explorer, simply because normal W3C DOM conventions are not followed with respect to attaching event listeners. Furthermore, Internet Explorer 6 (which has a usage share of 6.7% at the time of writing [50]) does not support the use of XMLHttpRequest, instead using an ActiveX object for asynchronous requests. SophieJS can be adapted to support these browsers in a future release.

6.2.4 Increase event coverage

Many browsers feature browser-specific or even version-specific events that can be captured. For example, Mozilla Firefox version 3.0 and above provides an `oncut` event [51] that provides access to clipboard events regardless of origination - be it toolbar menu, context menu or keyboard shortcut. These events can also be handled in SophieJS, but in a browser-specific manner.

Furthermore, as additional events become standardised, they can be added to SophieJS via manipulation of the included Javascript files.

Bibliography

[1] A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2006.

[2] S. Barrett, "Sophie: A Request for Comment for Peer Web Search Technology." White paper, Trinity College Dublin, May 2009. Available online at Available online at `https://www.tcd.ie/research_innovation/assets/pdocs/Sophie%20whitepaper.pdf` (last accessed September 9th 2010).

[3] D. Blei, A. Ng, and M. Jordan, "Latent dirichlet allocation," *The Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[4] J. Cho, S. Roy, and R. Adams, "Page quality: In search of an unbiased web ranking," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 551–562, ACM, 2005.

[5] J. Cho and S. Roy, "Impact of search engines on page popularity," in *Proceedings of the 13th international conference on World Wide Web*, pp. 20–29, ACM, 2004.

[6] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Computer networks*, vol. 33, no. 1-6, pp. 309–320, 2000.

[7] IBM, "Researchers map the web." Available online at `http://www.almaden.ibm.com/almaden/webmap_press.html` (last accessed September 9th 2010).

[8] V. Krishnan and R. Raj, "Web Spam Detection with Anti-Trust Rank," *AIRWeb 2006 Program*, p. 37, 2006.

[9] R. Lempel and S. Moran, "The stochastic approach for link-structure analysis (SALSA) and the TKC effect1," *Computer Networks*, vol. 33, no. 1-6, pp. 387–401, 2000.

[10] B. Wu and B. Davison, "Identifying link farm spam pages," in *Special interest tracks and posters of the 14th international conference on World Wide Web*, pp. 820–829, ACM, 2005.

[11] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer, "P2p content search: Give the web back to the people," Citeseer.

[12] J. Li, B. Loo, J. Hellerstein, M. Kaashoek, D. Karger, and R. Morris, "On the feasibility of peer-to-peer web indexing and search," *Peer-to-Peer Systems II*, pp. 207–215.

[13] T. Jo-Wen, W. Zhang, A. Long, and K. Shanmugasundaram, "Odissea: A peer-to-peer architecture for scalable web search and information retrieval," in *International Workshop on the Web and Databases*, Citeseer, 2003.

[14] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer, "Minerva: Collaborative p2p search," in *Proceedings of the 31st international conference on Very large data bases*, p. 1266, VLDB Endowment, 2005.

[15] J. Surowiecki, *The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies and Nations.* Doubleday, 5 2004.

[16] Y. Yanbe, A. Jatowt, S. Nakamura, and K. Tanaka, "Can social bookmarking

enhance search in the web?," in *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, p. 116, ACM, 2007.

[17] P. Heymann, G. Koutrika, and H. Garcia-Molina, "Can social bookmarking improve web search?," in *Proceedings of the international conference on Web search and web data mining*, pp. 195–206, ACM, 2008.

[18] Alexa, "About the alexa traffic rankings." Available online at `http://www.alexa.com/help/traffic_learn_more` (last accessed September 9th 2010).

[19] M. Meiss, F. Menczer, S. Fortunato, A. Flammini, and A. Vespignani, "Ranking web sites with real user traffic," in *Proceedings of the international conference on Web search and web data mining*, pp. 65–76, ACM, 2008.

[20] T. Joachims, "Optimizing search engines using clickthrough data," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, p. 142, ACM, 2002.

[21] H. Lieberman *et al.*, "Letizia: An agent that assists web browsing," in *International Joint Conference on Artificial Intelligence*, vol. 14, pp. 924–929, LAWRENCE ERLBAUM ASSOCIATES LTD, 1995.

[22] J. Goecks and J. Shavlik, "Learning users' interests by unobtrusively observing their normal behavior," in *Proceedings of the 5th international conference on Intelligent user interfaces*, p. 132, ACM, 2000.

[23] E. Agichtein, E. Brill, and S. Dumais, "Improving web search ranking by incorporating user behavior information," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, p. 26, ACM, 2006.

[24] J. Nielsen, "Scrolling and attention." Available online at `http://www.useit.com/alertbox/scrolling-attention.html` (last accessed September 9th 2010).

[25] M. Morita and Y. Shinoda, "Information filtering based on user behavior analysis and best match text retrieval," in *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, p. 281, Springer-Verlag New York, Inc., 1994.

[26] L. Longo, S. Barrett, and P. Dondio, "Toward Social Search: from Explicit to Implicit Collaboration to Predict Users' Interests," in *5th International Conference on Web Information Systems and Technologies (WEBIST 2009), Lisboa, Portugal*, pp. 693–696, 2009.

[27] L. Longo, S. Barrett, and P. Dondio, "Information Foraging Theory as a Form of Collective Intelligence for Social Search," in *Proceedings of the 1st International Conference on Computational Collective Intelligence. Semantic Web, Social Networks and Multiagent Systems*, pp. 63–74, Springer-Verlag, 2009.

[28] S. Fox, K. Karnawat, M. Mydland, S. Dumais, and T. White, "Evaluating implicit measures to improve web search," *ACM Transactions on Information Systems (TOIS)*, vol. 23, no. 2, p. 168, 2005.

[29] J. Hong, J. Heer, S. Waterson, and J. Landay, "WebQuilt: A proxy-based approach to remote web usability testing," *ACM Transactions on Information Systems (TOIS)*, vol. 19, no. 3, pp. 263–285, 2001.

[30] T. Ozono and T. Shintani, "An Online Method for Editing Any Web Page Using Proxy Agents," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 6, no. 5B, p. 166, 2006.

[31] A. Safonov, J. Konstan, and J. Carlis, "WebMacros - a Proxy-based System for Recording and Replaying User Interactions with the Web," *WWW Posters*, 2001.

[32] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov, "JavaScript Instrumentation in Practice," in *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, p. 341, Springer-Verlag, 2008.

[33] A. Lieuallen, A. Boodman, and J. Sundstrm, "Greasemonkey." Available online at `http://www.greasespot.net/` (last accessed September 9th 2010).

[34] Cybozu Labs, "Greasemetal." Available online at `http://greasemetal.31tools.com/` (last accessed September 9th 2010).

[35] J. Keith, *DOM Scripting: Web Design with JavaScript and the Document Object Model*. Friends of ED Publishing, 2005.

[36] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM symposium on Applied computing*, p. 337, ACM, 2006.

[37] B. Krishnamurthy *et al.*, "Key differences between HTTP/1.0 and HTTP/1.1," *Computer Networks*, vol. 31, no. 11-16, pp. 1737–1751, 1999.

[38] S. Soulders, "Roundup on parallel connections." Available online at `http://www.stevesouders.com/blog/2008/03/20/roundup-on-parallel-connections/` (last accessed September 9th 2010).

[39] B. Bos, T. Celik, I. Hickson, and H. W. Lie, "Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification," tech. rep., W3C, September 2009.

[40] A. Cohen, S. Rangarajan, and N. Singh, "Supporting transparent caching with standard proxy caches," in *Proceedings of the 4th International Web Caching Workshop*, vol. 134, Citeseer, 1999.

[41] G. Purdy, *Linux iptables: pocket reference*. O'Reilly Media, Inc., 2004.

[42] D. Heldenbrand and C. Carey, "The Linux router: an inexpensive alternative to commercial routers in the lab," *Journal of Computing Sciences in Colleges*, vol. 23, no. 1, p. 133, 2007.

[43] Quantcast, "August 2010 Mobile OS Share." Available online at `http://blog.quantcast.com/quantcast/2010/09/august-2010-mobile-os-share.html` (last accessed September 9th 2010).

[44] Sreeram Ramachandran, Google, "Web metrics: Size and number of resources." Available online at `http://code.google.com/speed/articles/web-metrics.html` (last accessed September 9th 2010).

[45] M. Burtscher, B. Livshits, G. Sinha, B. Zorn, B. Livshits, and B. Zorn, "JSZap: Compressing JavaScript Code," in *Proceedings of the USENIX Conference on Web Application Development*.

[46] L. Longo and S. Barrett, "A Computational Analysis of Cognitive Effort," *Intelligent Information and Database Systems*, pp. 65–74, 2010.

[47] L. Granka, T. Joachims, and G. Gay, "Eye-tracking analysis of user behavior in WWW search," in *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval, July*, pp. 25–29, Citeseer, 2004.

[48] J. Nielsen and K. Pernice, *Eyetracking web usability*. New Riders Pub, 2009.

[49] S. Weibelzahl, A. Reynolds, D. McDonnell, and D. Byrne, "An eye-tracking study investigating users search behaviour using Google search.," tech. rep., National College of Ireland & Mulley Communications, June 2009.

[50] W3C, "Internet explorer version statistics." Available online at `http://www.`

w3schools.com/browsers/browsers_explorer.asp (last accessed September 9th 2010).

[51] Mozilla Developer Center, "Gecko DOM reference: element.oncut." Available online at https://developer.mozilla.org/en/DOM/element.oncut (last accessed September 9th 2010).