PEER CACHE

SECURE PEER-TO-PEER WEB CACHING USING PRIVATE SET INTERSECTION

KRISTIAN BORUP ANTONSEN, 20105712

MASTER'S THESIS

March 2016

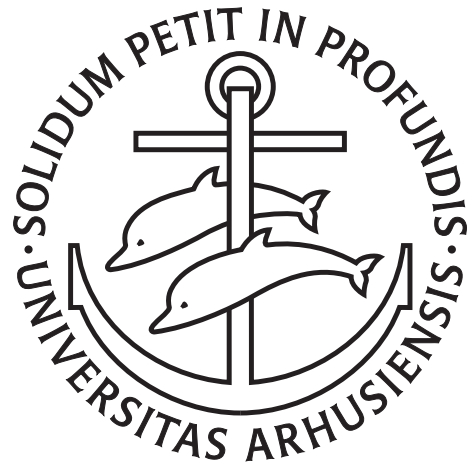
Advisor: Niels Olof Bouvin

Supervisor: Clemens Nylandsted Klokmose

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

PEER CACHE

KRISTIAN BORUP ANTONSEN



Secure Peer-to-Peer Web Caching
using Private Set Intersection

Master's Thesis
Department of Computer Science
Science & Technology
Aarhus University

March 2016

ABSTRACT

To prevent losing valuable information from websites that are being taken offline, the Internet Archive crawls websites to create an archive. However, if the Internet Archive itself one day gets taken offline, we will be no better off than before the Internet Archive was created.

This thesis introduces Peer Cache, a peer-to-peer-based browser extension for Google Chrome that lets users of the extension share cached versions of websites, either to make content remain available during server outages, or to permanently preserve content of websites that no longer exist, much like the Internet Archive does today. The peer-to-peer-based nature of Peer Cache makes it almost impossible to take down, unlike a server-based system like the Internet Archive.

Peer Cache is designed to automatically collect caches of the websites that the users of the extension visit. The cached websites may then be requested and shared with other users of the extension, or users of a dedicated web frontend.

In order for users of the extensions to not disclose private information from their visited websites when sharing caches, Peer Cache uses an adaptation of a two-party Private Set Intersection (PSI) protocol. PSI is a cryptographic protocol allowing two participants, each with their own set of elements, to find the intersection of their respective sets, without disclosing their sets to each other. The PSI adaptation – Private Tree Intersection – allows two users of Peer Cache to safely compare their caches of a website and compute an intersection that has been rid of both users' private information, which may then be shared to other users of the network.

While the Peer Cache prototype implementation does not adhere fully to the design presented, the implementation demonstrates that Peer Cache seemingly allows for users to collect and securely share caches. The cache collection is automatic, but somewhat obtrusive to the user's browsing experience, as Peer Cache causes minor freezes while the cache collection takes place. The caches that have been rid of private information using PTI maintain most of their content, but under certain conditions fail to create meaningful caches.

The problems that have been encountered during implementation and evaluation all seem to be solvable by minor changes to design and implementation. While some performance issues may not be immediately solved, it is expected that the continued progression of the JavaScript language eventually will catch up with the performance required for Peer Cache to function smoothly.

ACKNOWLEDGMENTS

I would like to express my gratitude towards my supervisors, Niels Olof Bouvin and Clemens Nylandsted Klokmose, for their engagement and suggestions throughout the project. Without their ideas and guidance, the project would not have turned out to be what it is today.

I would also like to thank Mathias Andersen for helping me procrastinate almost every day throughout the project. I couldn't have procrastinated as much as I did without you, I'm sure of it.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

GLOSSARY

API Application Programming Interface. A set of methods made available to third party developers, allowing them to utilize the underlying service programmatically.

CDN Content Distribution Network. A distributed network of proxy servers with the goal of serving static content to website visitors with high performance and availability.

CHURN Churn is the actions of peers joining and leaving a peer-to-peer network.

CORS Cross-Origin Resource Sharing. A set of HTTP headers making it possible to circumvent the same-origin security policy, thus allowing websites to request resources from other domains than the origin domain.

CSP Content Security Policy. An HTTP header to help prevent cross-site scripting and data injection attacks.

DHT Distributed Hash Table. A distributed data structure, allowing users to lookup key-value pairs.

DNS Domain Name System. A distributed naming system, translating hostnames into IP addresses.

DOM Document Object Model. A tree of objects, allowing JavaScript to edit HTML documents on the fly.

HTS Hash Tree Structure. A tree structure containing hashes of elements, generated from an HTML document. Used in conjunction with PTI.

JSON JavaScript Object Notation. A standard for serialization JavaScript objects, allowing them to be transmitted as plain text and later unserialized as JavaScript objects.

LAN Local Area Network. A network generally spanning a small area, providing both high transfer speeds as well as low latency.

LFU Least Frequently Used. An algorithm for determining which caches should be removed. Removes the least frequentlyused cache.

LRU Least Recently Used. An algorithm for determining which caches should be removed. Removes the least recently used cache.

PSI Private Set Intersection. A cryptographic protocol allowing two parties – each with a private set of elements – to find the shared set (or set intersection) of their sets, without disclosing their sets to each other.

PTI Private Tree Intersection. An adaptation of PSI that works on trees, specifically DOM trees (HTML documents), rather than sets.

SYBIL ATTACK A type of attack performed on peer-to-peer networks where a malicious user poses as multiple identities, thus gaining control over part of the network.

URL Unified Resource Location. A reference or link to a resource, usually on the internet.

VPN Virtual Private Network. A group of computers creating a private network overlay over a public network, allowing users to share and access resources as if they were on a local network.

WEB CRAWLER A piece of software that systematically visits websites, usually with the purpose of creating an index for search engines or saving copies of the visited websites to archive them.

1 INTRODUCTION

Thousand of new websites are born every day, but unfortunately, a great many are also taken offline at the same time. When a website goes offline, the website's content is at the mercy of the site owners, and that content will, more often than not, be lost forever.

To prevent this from happening, the Internet Archive has taken it upon themselves to archive the web; making sure content stays up for good by crawling websites and saving copies of them on their servers. But if the Internet Archive one day shuts down, we will be no better off than we were before the archive was created. On top of that, the Internet Archive is limited by the questionable virtue of using the client-server model, and will thus not scale as well as a peer-to-peer-based system. However, scalability is not the only benefits a peer-to-peer-based system would bring to the table. A peer-to-peer-based Internet Archive-like system would also become immensely difficult (if not impossible) to ever be taken down, making sure that content will remain available.

However, peer-to-peer-based systems outside of academia, have rarely succeeded, with the exception of some mostly infamous file-sharing systems such as Napster, KaZaA, and the more recent BitTorrent, which today is responsible for about 7.4% of daily internet traffic in North America[1], supporting over 100 million users simultaneously [2]. Recently, other attempts like IPFS have also seen the light of day, trying to create a peer-to-peer-based web. Unfortunately, IFPS requires the content providers to actively make their own content available on the network.

This thesis introduces Peer Cache: A peer-to-peer-based browser extension for Google Chrome that lets users of the extension share cached versions of websites, either to make content remain available during server outages or overloading, or to permanently preserve content of websites that no longer exist, much like the Internet Archive does today.

1.1 GOAL

The goal of this thesis is to create Peer Cache: A peer-to-peer-based web caching system, allowing users of the system to save, share, and retrieve caches from each other. Unlike the Internet Archive, Peer

1 http://www.theguardian.com/technology/2013/nov/11/netflix-youtube-dominate-us-internet-traffic

Cache is peer-to-peer-based, and can therefore not easily be taken offline or fall victim to censorship.

Sharing caches between users carry some risk, most notably the danger of accidentally disclosing private information. Therefore, the system should facilitate cache sharing without leaking private information. Furthermore, the system should be unobtrusive for the cache provider (the user sharing caches) and equally easy to use for the cache consumer (the user interested in accessing the cache).

The system should consist of a browser extension that collects the users' caches and facilitates sharing. The system should also have another part, allowing other users – who do not use the extension – to access caches through their web browser, much like the users of the Internet Archive does it today.

1.1.1 *Hypotheses*

With this goal in mind, the following hypotheses are examined in this thesis through the use of a proof-of-concept implementation:

1. Web Preservation can be achieved using peer-to-peer web caching while still remaining efficient and providing availability comparable to current client-server systems.
2. User-collected caches can be securely rid of personal information using Private Set Intersection and still remain meaningful.
3. Cache collection and sharing can be possible without becoming obtrusive to the cache provider.
4. Caches can be fetched by cache consumers from the the peer-to-peer system without requiring cache consumers to have specific computer or browser configurations, as well as not requiring them to install any third-party applications or browser extensions.

1.2 OVERVIEW

The rest of this thesis will be structured as follows: First, we examine related work in Chapter 2, followed by an analysis of the related work in relation to Peer Cache in Chapter 3. After the analysis, the technology used is presented in Chapter 4, after which the design is outlined in Chapter 5. The implementation of the design is then described in Chapter 6, followed by an evaluation in Chapter 7 and lastly a conclusion in Chapter 8.

Note that this thesis is about the technical challenges of creating2 Peer Cache, so we will limit the scope to not concern ourselves with legality, specifically copyright infringement and other issues related to redistributing other people's or companies' content. The legal issues are, however, briefly mentioned in Section 2.1.

2 RELATED WORK

The purpose of Peer Cache is to provide web preservation through peer-to-peer oriented web caching. A natural starting point will therefore be to examine the efforts made in web preservation, peer-to-peer oriented web caching and caching in general. Additionally, Private Set Intersection (PSI) protocols are in this chapter introduced and examined, as PSI can be useful for stripping web caches of sensitive data.

2.1 WEB PRESERVATION

Web preservation (or archiving) is the process of collecting and storing websites to ensure that the information on them will remain available for people interested in them the future. Simply collecting websites is therefore not sufficient; a distribution system for future users must also be made available.

In this section, some of the challenges in web preservation are summarized.

Web preservation can be difficult for three main reasons [4], namely technical, organizational and legal reasons.

The technical problem is the result of the ever-growing nature of the Web. While the Internet Archive has attempted to preserve the web, it is unrealistic for a single organization to do so [4]. Instead, web preservation should be a collective activity.

Another main technical issue is the fact that the Web is highly dynamic. Alexa Internet estimates that web pages disappear on average after 75 days. This rapid decay means that without some form of preservation, many invaluable resources – both scholar, cultural and scientific – may be unavailable to future generations.

In terms of legal challenges, the main ones relate to copyright and liability of content made available through archives. Additionally, there are also potential problems with defamation, content liability and data protection. The safest way of preventing these problems is by being very selective when it comes to what resources should be archived.

The decentralized nature of the internet causes some organizational challenges. Most web preservations initiatives tend to focus only on a subset of the entire web, for instance by only archiving resources from certain countries, subjects or organizations. Another organizational issue is about quality. The Web contains both content of high quality

like educational and research material as well as websites for political parties, but it also contains content of very low quality, such as inaccuracy or downright untrue information. Preserving everything might therefore not be ideal.

According to Day [4], three main approaches (and a combined approach) to data collection existed in 2003, and the same approaches seem to be the most prevalent today.

AUTOMATIC HARVESTING is based on web crawlers – programs that autonomously browse websites and download their content like the Internet Archive. Today, this is still by the far most common technique for web preservation, and is also how modern search engines like Google[1] provide web caches and index websites. This is also the cheapest way of collecting web content due to the minimal administrative work required.

SELECTIVE CAPTURING is a more expensive and administratively demanding technique where websites are selected according to specific guidelines are negotiated with their owners. Once agreements are in place, the data collection begins.

DEPOSITS are implemented by site owners or administrators that take a snapshot of their website and deposit it into a repository. While deposits can be difficult, they may require less space since creating incremental updates will be easier for the website administrators than an autonomous web crawler that does not have direct access to the website's underlying database.

As with so many other things, a combination of techniques is often the better solution. As an example, Netarchive[2] (a Danish web preservation initiative) employs both automatic harvesting for all Danish domains as well as selective harvesting of between 80 and 100 Danish websites, individually selected due to their importance to the Danish media public. The Netarchive also uses what they call *Event Harvesting* and *Special Harvesting* which are certain types of selective capturing. None of the data collected by Netarchive is available to the general public, however.

However, none of these approaches (combined or otherwise) match the kind of caching the examined peer-to-peer web caching systems use, nor does it match the caching technique intended for Peer Cache. To make up for this, this thesis therefore supplements the previous definitions with another data collection approach:

USER-DRIVEN HARVESTING is a caching approach used in the browser where websites navigated to by the user are cached. This

1 https://www.google.com/insidesearch/howsearchworks/crawling-indexing.html

2 http://netarkivet.dk/in-english/

form of caching happens automatically and requires little to no administrative work.

2.2 WEB CACHING-ORIENTED PEER-TO-PEER COMMUNICATION

Web caching-oriented peer-to-peer communication systems aim to provide caches of websites to the users of the system, without using a centralized server, but instead by utilizing every user on the network as a small server with its own cache store.

This section examines and summarize papers describing peer-to-peer systems. Most of these systems aim to reduce user latency and server load of the web server they are providing caches for. The systems also tend to provide some availability of unavailable resources, even though it is not their direct intent.

While Peer Cache does not share its goal with the examined systems, several similarities are observed, mostly the intention of making resources available to users without having to retrieve the resource from the original website – a website that may potentially be gone.

2.2.1 *Squirrel – A Decentralized Peer-to-Peer Web Cache (2002) [5]*

Squirrel is a decentralized peer-to-peer web cache with the purpose of allowing users to share their local caches. The paper demonstrates how it is possible to use a peer-to-peer web caching approach on a corporate Local Area Network (LAN) in favor of a local web caching server. They find that it is both possible, desirable and efficient to do so. Squirrel uses Pastry – a peer-to-peer routing network and distributed hash table (DHT) to store web caches – making Squirrel both scalable, fault resistant, and self-organizing.

Two different approaches to caching is used in Squirrel, namely *Home-store* and *Directory*. In both cases, all web requests go through a modified proxy that attempts to locate a cached version of the website by hashing the request and finding a node responsible for the request: a so-called *home node*. With Home-store, the home node looks in its own cache for the request, returns it if found, or otherwise makes a web request for the resource itself and returns it to the requesting node. With the second approach, the Directory, the home node does not make the Web request itself, but instead keeps a directory with pointers to other nodes that have recently accessed the resource. When a request comes in, the home node attempts to locate the resource in the directory and redirects the request to a random matching node.

In the paper (section 4.1), it is briefly described how they determine which things are cacheable. Squirrel basically only caches `GET` requests without SSL and query strings.

2.2.2 *YouServ – A Web-Hosting and Content Sharing Tool for the Masses (2002) [6]*

YouServ is a peer-to-peer-based web hosting system developed at IBM, requiring two server-side components, namely a dynamic DNS server and YouServ Coordinator. When the paper was released, the system had successfully been deployed inside the IBM corporate intranet for nine months, having more than 1200 active users each week, as well as 700-800 active YouServ sites available during peak hours.

To host a site using YouServ, users have to install the YouServ peer software, log in using their corporate ID, after which YouServ assigns the site a domain name based on their corporate email address. Users may then copy files they wish to share to a local YouServ folder on their hard drive. When a user, Joe, wants to host files, the YouServ Coordinator registers the IP of Joe's machine with the domain name `joe.youserv.com` using the dynamic DNS server. When another user wants to access Joe's files by accessing `joe.youserv.com`, the DNS server will return the IP of Joe's machine to the user. This, of course, requires that all users of the system use the dynamic DNS server that YouServ registers users with.

In case a user goes offline, another user, Alice, may agree to become a replica for the user. For instance, Alice and Joe may decide that Alice should become a replica for Joe's files. The YouServ peer software on Alice's machine will then periodically make sure that her machine contains all of Joe's files and that they are up-to-date. Should new files be added or old files become outdated, the YouServ software on Alice's machine will automatically download the files from Joe's machine. If Joe at any time decides to go offline, his machine will first contact the YouServ Coordinator which in turn will update the dynamic DNS server, so `joe.youserv.com` no longer points to the IP of Joe's machine, but instead points to the IP of Alice's machine. Thus, when a user tries to access Joe's files, the users will instead access a replica of Joe's files on Alice's machine.

Now imagine a user, Bob, who wants to publish files. Bob is behind a firewall or NAT, causing Bob's inbound port 80 to be closed, meaning nobody will be able to access Bob's files. YouServ solves this using peer-to-peer proxying. When Bob runs the YouServ peer software, the software will detect that Bob's inbound port 80 is closed and suggest a proxy, say Joe. Bob will then establish a persistent connection to Joe and the dynamic DNS server will make `bob.youserv.com` point to the IP of Joe's machine. When anybody tries to access Bob's files through `bob.youserv.com`, Joe's machine will forward the request to Bob through the persistent connection, acting as a proxy between the two parties.

2.2.3 *Kache – Peer-to-Peer Web Caching Using Kelips (2004) [11]*

Kache is a peer-to-peer web caching system based on Kelips, a peer-to-peer indexing protocol. The underlying Kelips system uses probabilistic schemes and self-regenerating data structures to handle churn. Due to this, Kache is demonstrated to maintain fast lookups and low overhead in networks of clients with cheap inter-client communication and expensive server communication, even with high churn rates.

Through the use of Kelips, Kache maintains a DHT where website URLs constitute the keys and the values specifying the location of the cached website. Kelips maintains a constant background network overhead in order to allow for fast lookups where nodes usually just need to ask a single other node to find the information they are looking for. The background communication in their test network had a continuous data stream of 3 KB/s per peer with 1000 nodes with high churn.

Kelips has a relaxed consistency goal, meaning a small percentage of key-value tuples may be missing from every node at any given moment. However, since each node is independent, this can be compensated for by simultaneously queuing multiple nodes.

When new information has been acquired by a node, an epidemic (or gossip-based) protocol is used to disseminate the new information to the remaining nodes. The node will broadcast the new information to a small set of nodes. These nodes then become "infected" and in turn each broadcast the same information to another small set of nodes. This process continues for a short time, until the majority of the other nodes have been informed.

The constant background network overhead is used to maintain the DHT. To ensure that stall DHT entries are deleted, each tuple in the DHT has an associated heartbeat counter. The node holding the resource associated with the specific tuple will periodically send an update to the DHT to keep the tuple alive by updating the heartbeat counter. When the tuple-holding node receives a heartbeat for a specific tuple, the counter on the tuple is reset. If a tuple in the DHT is not updated after a predefined time-out, the counter runs out, and the tuple is deleted from the DHT.

When a node requests a resource, it will look up the URL in the DHT for the tuple and contact nodes in the *affinity group*: the group of nodes responsible for the requested object. The requesting node then asks the nodes in the affinity group for the URL, and the affinity group nodes will return the address of the nodes holding the requested resource, if any such nodes exists. If a node exists, the requesting peer retrieves the cached object from the node, or otherwise makes a request to the original web server without using the peer-to-peer cache.

2.2.4 *Performance Evaluation of Peer-to-Peer Web Caching Systems (2005) [12]*

This paper intends to evaluate and analyze the performance of peer-to-peer web caching systems in three orthogonal dimensions: the caching algorithm, the document lookup algorithm, and the peer granularity.

Two caching algorithms are evaluated, namely the *URL-based caching algorithm* and the *content-based caching algorithm*. The URL-based algorithm simply concerns itself with the URL of the website in question and the freshness of the cache. In the paper, studies were found stating that 59.1% of all web requests that were traditionally perceived uncacheable, were repeatedly accessed, meaning they could in fact be cached with the right digest-based protocol. The content-based caching algorithm focuses on these digests. The idea is that the content-based algorithm exchanges content digests in order to decide whether any real communication is needed, and if so only transfers the new content (the content with different digests) to the requesting node. The paper concludes that the content-based caching algorithm could greatly improve cacheability (more on cacheability in Section 2.3.1) over the URL-based algorithm.

The paper also examines four document lookup algorithms: *home1*, *home2*, geographic-based (*Geo*), and the *Tuxedo algorithm*.

Using the home1 protocol, when a user requests a resource, the protocol first checks the local cache, then contacts an indexing server in its scope if nothing was found locally. If the indexing server finds a host containing the appropriate hash, the hash is delivered to the requesting user. Otherwise, the user requests the resource from the original server. home2 uses a DHT. When a user requests a resource, the local cache is checked first. In case of a cache miss, the URL is hashed and the request is forwarded to the node in the network responsible for the hash. If the resource is found, it is sent to the requesting user. Otherwise, the requesting user requests the resource from the original server.

With the Tuxedo algorithm, each node maintains two tables containing information about each web server and a list of nodes that are closer in proximity than the original web server. When a resource is requested, Tuxedo will attempt to retrieve a cached version from a node closer than the web server to improve performance.

The Geo algorithm only attempts to retrieve caches from users in the same subnet by multicasting the request, due to the assumption that people from the same geographic region will have visited similar websites. The paper finds that Geo has a significant latency reduction and good speed-up, but the overall performance is impaired by the use of multicasting. It is also found that Tuxedo and home1 could efficiently reduce user-perceived latency.

Four peer granularities are examined in the paper, namely host level, organization level, building level, and a centralized web cache. It is found that while the centralized level has comparable latency reduction to that of the building level, the centralized web cache will suffer from scaling issues in a real implementation. Further, it is found that building and organization level caching has very similar speed-up, and that even though host level has the highest speed-up of the four, the authors still prefer building or organization level for ease of implementation, reduced overhead for updating discovery information, and bigger total cache size than that of the host level.

2.2.5 *Towards a Smart, Self-Scaling Cooperative Web Cache (2012) [13]*

This paper provides an analysis, design, and prototype implementation of Cooperative Web Cache (CWC), which helps web services to scale by using clients in a self-organizing, peer-to-peer-based overlay network for content replication. The aim of CWC is to provide a speed-up for users and to decrease server load on both service providers and potential CDNs. According to Alexa Top 500, 90% of web page resources are static on average, and may therefore be cacheable.

Certain requirements for CWC are defined, including that the original services being cached should stay unchanged, and the process should be transparent, either through a browser extension or a proxy. CWC is implemented using a proxy and based on Pastry (like Squirrel).

CWC works by having the client send a request to the server through the proxy. The proxy inspects the request to predict whether the response is cacheable. If not cacheable, the request is sent directly to the original server. If deemed cacheable, the request is forwarded to the *CWC cloud* (the peer-to-peer network). If the result is found in the CWC cloud, the client fetches the result and joins the *CWC group* (a group of users containing the resource in the CWC cloud), now allowing future clients to retrieve the result from the requesting client. If no result is found in the CWC cloud, the requesting client will fetch the result from the original server and create its own CWC group from where it can serve the result to future users.

The paper concludes that CWC may provide improved download times for relatively common cases and significantly lower server costs for service providers.

2.2.6 *IPFS – Content Addressed, Versioned, P2P File System (2014) [2]*

The InterPlanetary File System (IPFS) is a peer-to-peer distribution protocol aimed at creating a global, distributed file system. The system is very similar to the DHT-based BitTorrent with Git-like version

control. The protocol is entirely peer-to-peer-based with no privileged nodes. The system is transport protocol-independent, but best suited for WebRTC DataChannels or uTorrent Transport Protocol (uTP).

IPFS can also provide reliability if the underlying networks do not, using either uTP or Stream Control Transmission Protocol (SCTP). IPFS may be used in many different settings, either as a global file system, a database, a CDN, or something entirely different.

2.3 CACHING

Caching is the act of saving retrieved data once acquired lest having to re-retrieve the data at a later point, should the data be needed again. In a computer, the CPU may cache data retrieved from memory to avoid having to re-copy data from memory; it may be caching in memory to avoid having to re-copy from a hard drive, or caching locally on a hard drive to avoid having to fetch the data from a web server again.

There are multiple purposes of caching, the most common being to reduce latency by caching data more locally than its origin, thus avoiding having to retrieve data from a far away server or from a slow hard drive. Often, computed data is also cached, to avoid having to spend both time and resources on performing the same computation again. In this project, however, the main purpose of caching will be to make data permanently available, should the primary resource no longer be available.

Some strategies are required to determine which caches to keep. While permanently keeping and caching every single retrieved object would be practical in terms of accessibility, it is often infeasible in practice due to data storage limitations. Therefore, strategies for deciding which items need to be replaced are required.

Podlipnig and Laszlo [10] present a classification of such replacement strategies. This classification is a combination of several other classifications discovered by Podlipnig and Laszlo during their survey of caching strategies. The combined classification consists of 5 classes:

RECENCY-BASED STRATEGIES use mainly recency of access (but also size and/or cost) to determine whether a cached object should be kept. For instance, it may be reasonable to replace cached objects that have not been accessed for a long time, because it seems reasonable that they are unlikely to be accessed in the near future either.

An example of a basic recency-based strategy is Least Recently Used (LRU) which simply replaces the object that was used least recently. Many other recency-based strategies exist, most being variants of LRU taking size or cost into account. Podlipnig and

Laszlo mention that a big disadvantage of recency-based strategies is that the simple variants do not combine recency size in a useful manner. Additionally, the recency-based strategies do not incorporate frequency, which may lead to very suboptimal replacements.

FREQUENCY-BASED STRATEGIES are mainly based on frequency of access to determine which objects to replace. A basic strategy is Least Frequently Used (LFU) that replaces rarely accessed objects. The main disadvantages (as suggested by Podlipnig and Laszlo) are the higher complexity of implementing the cache management, and cache pollution: With very dynamic content, the cache may end up containing aged objects that are unlikely to be accessed much in the future, taking up valuable caching space for newer, more relevant objects.

RECENCY AND FREQUENCY-BASED STRATEGIES are a class of strategies which combine the recency- and frequency-based strategies, combining advantages and disadvantages from both classes. If implemented correctly, this could mean better retention of objects that have not been accessed recently, but still frequently, while also not polluting the cache with aging objects that are unlikely to be accessed again in the near future. Unfortunately, the complexity of using frequency remains, and the outcome may be even worse by combining the two strategies, than using a purely frequency-based strategy alone.

FUNCTION-BASED STRATEGIES use a function to calculate a value for each object, determining the importance of keeping a certain object in the cache, replacing objects with the smallest value. Podlipnig and Laszlo describe certain specific algorithms, for instance the Taylor Series Prediction that uses a Taylor series to predict the temporal "acceleration" of requests for certain objects. The advantage of using function-based strategies is that the strategies may combine many parameters in the decision-making, and allow for optimization through weighting of these. The disadvantages are that choosing proper parameter weightings may be very challenging, and incorporating certain factors such as latency in the value calculation can introduce some problems.

RANDOMIZED STRATEGIES are strategies that base their replacement decision on randomization. An example of such a strategy is HARMONIC[10], which removes one item at random from the cache with a probability inversely proportional to its specific cost, meaning objects that will be expensive to reacquire are less likely to be replaced. The advantages of these strategies, according to Podlipnig and Laszlo, is the ease of implementation

and that the strategies do not require any special data structures for inserting and deleting from the cache. Additionally, randomized strategies are generally simple to implement. The main disadvantage is that these strategies are hard to evaluate due to their random nature.

2.3.1 *Cachability [sic] of Web Objects (2002) [14]*

Two types of caches exist (in terms of web caching), namely browser caches and proxy caches. Browser caches are stored locally on the client and proxy caches are caches stored on a proxy server. Proxies are servers sitting between the clients and the internet.

A resource is cacheable if it is practical (and safe) to save the resource for later retrieval. Some resources are uncacheable, because it can be predicted that they will not be useful later. Other resources can be uncacheable, because it will be unsafe to let other users than the original user see the cached resource due to privacy concerns.

Over all, requests are uncacheable due to: being dynamic URLs, having certain HTTP methods, HTTP status codes, or HTTP headers.

Dynamic URLs are characterized by the use of `?`, `=`, `/cgi-bin/`, etc. in the URL. Usually `?` occurs in a query string. 5.3% of collected web traces are dynamic URLs, according to the paper. These requests are uncacheable.

Uncacheable methods are the HTTP 1.1 methods: `POST`, `PUT`, `DELETE`, `OPTION`, and `TRACE`, allowing only `GET` and `HEAD` requests to be cached. In the trace, only `GET`, `HEAD`, and `POST` requests were found.

There are three categories of HTTP status codes: *the cacheable*, *the negative cacheable* and the *uncacheable*. The cacheable are most normal requests, such as `200 OK` and `304 Not Modified`. The negative cacheable are often bad requests that can be expected to remain that way for some time, for instance `204 No Content`, `404 Not Found`, or `503 Service Unavailable`. These are therefore cacheable for some time. The uncacheable requests include among others: `206 Partial Content`, `302 Moved Temporarily`, and `401 Unauthorized`. The content of these requests can change quickly and the reply may depend on context.

The HTTP headers of interest in terms of cacheability are `Last-Modified`, `Set-Cookie`, `Cache-Control`, `Pragma` headers, `Authorization`, and `Expires`. The `Last-Modified` header suggests either that: the object is dynamically generated, the original server wants the user to refetch the object, or the web server misconfigured. Either way, the page is uncacheable. If the `Set-Cookie` header is set, the resource is uncacheable. If the `Cache-Control` header is set to private, it indicates that the response is intended for a single user and must therefore not be shared by other cache users. If the `Cache-Control` header is set to no-cache, the request should not be cached at all. The `Pragma no-cache` header has the same meaning as the `Cache-Control no-cache` header and should

also not be cached. The `Authorization` header is used when authenticating a specific user to access a resource. Therefore, a resource with an `Authorization` header must only be accessed by the requesting user and should therefore not be saved in a shared cache. The `Expires` header indicates when an object becomes stale. Once an object has become stale, the object should no longer be used, unless it can first be validated by the original web server.

These uncacheable objects sum up to a total of 19.19% of all objects found during the trace performed by the author.

2.4 PRIVATE SET INTERSECTION

Private Set Intersection (PSI) is a cryptographic protocol for allowing two participants, each with their own set of elements, to find the intersection of their sets, without disclosing any non-intersecting elements to each other. If, for instance, Alice has a set $\mathcal{S}_A = \{A, B, C\}$ and Bob has another set $\mathcal{S}_B = \{C, D, E\}$, the intersection would be $\{C\}$. Private Set Intersection lets Alice and Bob compute this intersection, without either party becoming aware of what the other party's set consists of; e.g. Bob will not know that A and B are in Alice's set. Two example scenarios could be[3]:

1. Two real estate agencies want to compare customer records to see if any clients (homeowners) are double-dealing, i.e. signing contracts with both agencies. At the same time, neither agency wishes to disclose their clients to the other, competing agency. Both agencies would, however, like to identify the double-dealing clients.

2. A federal tax authority has a list of tax fraud suspects. The authority suspects that some of the suspects may have accounts in a foreign bank, but the bank may not legally disclose wholesale account holders. Likewise, the tax authority clearly does not want to disclose any of their suspects. In this case, the federal tax authority therefore needs to figure out which subjects have an account in the foreign bank, without disclosing any subjects to the bank, while the bank also does not disclose any clients that are not suspects.

As evident by the two example scenarios, two flavors of PSI exist: A two-way flavor like the former scenario and a one-way flavor like the latter scenario. Using a one-way flavor, only one party will end up with the intersection, whereas in a two-way flavor, both parties see the intersection. Often, however, it will be sufficient to only examine one-way PSI, because these protocols easily can be extended to become two-way, either by performing the PSI computation again with the parties switched, or simply by sharing the result after the transaction has concluded.

2.4.1 *Protocols*

PSI is a highly studied topic in secure computation[8], which is why many protocols have been designed since its inception. Next, a few common algorithms will be described.

NAIVE PROTOCOL

While insecure, the naive protocol is very fast[8]. Each party in the computation hash their elements individually and share the hashed set with each other.

Looking back at our example from before with Alice and Bob, now using the naive solution, Alice may hash each element of her set S_A to create $S'_A = \{\texttt{3bf3}, \texttt{69e2}, \texttt{5e80}\}$. Bob does the same to create $S'_B = \{\texttt{5e80}, \texttt{69e2}, \texttt{1fdd}\}$. When sharing these sets among each other, they will quickly find the intersection to be $\{\texttt{5e80}\}$, both knowing that that specific hash value represents the element C. This approach, however, is insecure, if the set elements have low entropy. If, as in this example, the sets for instance only consist of letters from the Latin alphabet, Bob could quickly bruteforce Alice's private set by hashing every letter from the alphabet and compare the result with Alice's hashed set. Likewise, when the hashed set is disclosed, information about set size is also leaked, even though this could be remedied somewhat by adding an arbitrary number of random hashes.

OPRF-BASED PROTOCOLS

Other protocols exist that work similar to the naive protocol, based on Oblivious Pseudorandom Functions (OPRF). As described by Cristofaro and Tsudik [3], with these protocols, we have a client C with a private set $\mathcal{C}$ and a server S with a private set $\mathcal{S}$ and a pseudorandom function $f_k(x)$. The exchange starts with C creating a key k and sending it to S. Afterwards, for each $s_j \in \mathcal{S}$ (of size w), S computes $u_j = f_k(s_j)$ and sends $\mathcal{U} = \{u_j, \ldots, u_w\}$ to C. C and S now compute an ORPF computation computation to calculate $f_k(c_i)$ for every $c_i \in \mathcal{C}$. As with the naive protocol, C now only knows that the intersecting elements are the elements c_i where $f_k(c_i) \in \mathcal{U}$, as well as the size of $\mathcal{S}$.

PUBLIC-KEY-BASED PROTOCOLS

Several public-key-based protocols exist, one being based on the Diffie-Hallmann (DH) key exchange protocol for securely exchanging cryptographic keys in a public channel. This protocol is based around the associativity of exponentiation[9], namely that if two elements, $\mathcal{S}_{A_i}$ from Alice's set and $\mathcal{S}_{B_j}$ from Bob's set, are the same, then $((H(\mathcal{S}_{A_i})^{\alpha})^{\beta} = (H(\mathcal{S}_{B_j})^{\beta})^{\alpha}$ where H is a hash function and α and β are primes from the same algebraic, cyclic group, each chosen by Alice and Bob, respectively.

The interaction starts with Alice sending $\{H(\mathcal{S}_{A_0})^{\alpha}, \ldots, H(\mathcal{S}_{A_k})^{\alpha}\}$ to Bob, where $\{\mathcal{S}_{A_0}, \ldots, \mathcal{S}_{A_k}\}$ constitutes Alice's private set. At the same time, Bob calculates $\{H(\mathcal{S}_{B_0})^{\beta}, \ldots, H(\mathcal{B}_{B_l})^{\beta}\}$ where $\{\mathcal{S}_{B_0}, \ldots, \mathcal{S}_{B_l}\}$ is Bob's private set, and sends it to Alice. She then computes $\{(H(\mathcal{S}_{B_0})^{\beta})^{\alpha}, \ldots, (H(\mathcal{B}_{B_l})^{\beta})^{\alpha}\}$ and Bob does similarly for the set he received from Alice. For any $\mathcal{S}_{A_i} = \mathcal{S}_{B_j}$, $((H(\mathcal{S}_{A_i})^{\alpha})^{\beta} = (H(\mathcal{S}_{B_j})^{\beta})^{\alpha}$ now holds true, and Alice and Bob can compare their new sets with each other openly.

SERVER-AIDED PSI PROTOCOL

Some very efficient server-aided protocols also exist, but they mostly require a *trusted* third party. One protocol[9] closely resembles the naive protocol, where both parties hash each of their own elements individually, shuffle the set, and send it to the server. The server then computes the intersection of the hashed sets and inform the participating parties. This method is only convenient if a trusted, unbiased server is available. If the server is not trustworthy – or is conspiring with one of the peers – the protocol will be vulnerable to bruteforce attacks just as the naive protocol. Unlike the other protocols, this algorithm does not reveal set size.

2.5 SUMMARY

In this chapter, web preservation along with data collection approaches have been presented.

Several peer-to-peer systems have been examined, mostly web caching-oriented peer-to-peer systems. Most examined systems were found to focus on reducing user latency as well as server load, not web preservation as Peer Cache. Several cache replacements strategies were also presented. Lastly, Private Set Intersection was introduced along with four PSI protocols.

3 ANALYSIS

In this chapter, the information gathered from Related Work is analyzed and related to Peer Cache. First, a list of focal points will be presented. These focal points will be used to create a comparison matrix of the web preservation and peer-to-peer systems from the Related Work chapter. After this, the comparison matrix will be analyzed in order to establish requirements for Peer Cache. Lastly, the Private Set Intersection algorithms will be analysed briefly.

3.1 FOCAL POINTS

INTENTION
The intention is simply what the purpose of the system is, either Web Preservation, Web Publishing, or Performance Enhancement.

ARCHITECTURE
The examined systems are either centralized or decentralized, using either a client-server architecture, peer-to-peer architecture or a hybrid architecture.

SCALE
Whether the system is meant for use in a small LAN, a larger organization or globally on the internet.

PLACE OF EXECUTION
Where the implementation takes place. For centralized systems, this will always be the server. For decentralized systems, there will be a client implementation, either through client software, a local proxy or a browser extension, potentially combined with a server.

RESOURCE LOCATION
Where the resources are stored, either at the user that owns the resource (collected the cache), at a server, or in a peer-to-peer network.

RESOURCE SEARCHING
How resources are located in the system. Resources can be either located by contacting a server and getting the location from the server, by looking up resource in the DHT, or by using dynamic DNS.

CACHE APPROACH
: Automatic harvesting, selective capturing, deposits or user-driven harvesting (see Section 2.1).

CACHEABILITY
: How it is decided whether a resource is cacheable or not.

ACCESSIBILITY
: How caches are made available for users to browse and access.

3.2 ANALYSIS OF COMPARISON MATRIX

3.2.1 *Intention and Architecture*

From the comparison matrix in Table 1, we can see that the usual systems used for web preservation are client-server-based. However, since the functionality of Peer Cache is closer to that of the peer-to-peer web caching systems than the web preservation initiatives, it makes sense to use a peer-to-peer-based system for Peer Cache. In "Preserving the fabric of our lives: A survey of web preservation initiatives" [4], Day also brings up the importance of making a collaborative effort to preserve the Web, and few things are more collaborative than peer-to-peer systems. Peer Cache should therefore be more similar to the examined peer-to-peer systems than Internet Archive in terms of architecture, even though the mission of Peer Cache is more similar to that of Internet Archive.

3.2.2 *Scale*

It makes sense for the peer-to-peer systems concerned with performance to limit the size of the network to keep retrieval speed up. With Peer Cache, however, we are less concerned with speed and more with availability, so limiting the size of the network (and in turn reduce availability), is not a viable solution here.

3.2.3 *Place of Execution*

All the examined peer-to-peer systems – except for YouServ – are implemented using a local proxy that handles all web requests, so this may immediately seem like the best solution.

However, with the continuously increasing focus on security, and the increase in computational power, more and more websites are transitioning to using web encryption (SSL/TLS). In 2014, 45% of the internet's top 1,000,000 websites were using encryption[1], a number

1 https://jve.linuxwall.info/blog/index.php?post/TLS_Survey

Name	**Intention**	Architecture	**Scale**	**Place of execution**	**Cache location**	**Cache searching**	**Caching approach**	**Cacheability**
Internet Archive, Netarchive, etc.	Web Preservation	Client-server	–	Server	Server	Server	Auto. harvest. and Selective capturing	Everything cacheable
Squirrel (Home-store)	Increase Performance	Peer-to-Peer	LAN	Proxy	Peer-to-Peer Network	Pastry (DHT)	User-driven harvesting	Only caches GET requests without SSL and query strings
Squirrel (Directory)	Increase Performance	Peer-to-Peer	LAN	Proxy	Owner	Pastry (DHT)	User-driven harvesting	
YouServ	Web Publishing	Hybrid	LAN	Client software	Owner or replica	Dynamic DNS	–	
Kache	Increase Performance	Peer-to-Peer	–	Proxy	Owner	Kelips (DHT)	User-driven harvesting	–
Cooperative Web Cache (CWC)	Increase Performance and reduce server load	Peer-to-Peer	Global	Proxy	Owner	CWC Group (DHT)	User-driven harvesting	Only caches static resources
IPFS	Web Preservation, etc.	Peer-to-Peer	Global	–	Peer-to-Peer Network	IPFS DHT (DHT)	Deposit	–

A dash (–) indicates that either there are no information available for the system on the focus point, or that the focal point is not applicable for the system.

Table 1: Comparison matrix of focal points.

far higher than what it was 10 years prior, around the time many of the examined papers were published.

New initiatives like Let's Encrypt[2] – that make setting up web encryption both easy and free – are also very likely to further increase the number of encrypted websites. When visiting an encrypted website, a proxy will only see the SSL handshake and the encrypted data, both with no caching value[3]. Using a proxy therefore becomes more and more infeasible as time goes on.

An alternative to the local proxy could be a browser extension, which is also suggested briefly by Tomáš Černý et al. [13] (2012). When deviating from previous practices, it is natural to wonder why others have not had similar ideas before. There are several reasons for why browser extensions may not have been used in place of local proxies earlier, the most noteworthy being:

1. Proxies give a lot of freedom of implementation, since there are no limitations set by the browser in terms of programming language or other restrictions to APIs.

2. Proxies are browser-independent, whereas individual extensions would have to be developed and maintained for all browsers that need to be supported.

3. Browser extension have been around since 1999 when they were introduced with Internet Explorer 5[4], but they did not gain widespread popularity until around 2005-2007, evident by the fact that Mozilla Firefox did not create its add-ons site until 2005[5], and neither did Chrome have their Chrome Webshop until 2010[6].

If the increase in web encryption had not been an issue, some may have still considered a local proxy to be advantageous. Browser extensions do also bring other things to the table, for instance by allowing for more in-browser functionality while also easing installation and distribution of the software. All major browsers have websites for distributing extensions, and installing and configuring an extension is usually nothing more than a single click. In contrast, a proxy has to be downloaded, installed, started, and the browser has to be configured to use it. And after all that, encrypted websites will still not be cacheable.

2 https://letsencrypt.org/

3 The encrypted data cannot be reused (and therefore has no caching value) due to replay attack prevention in the protocols, see https://tools.ietf.org/html/rfc4346#appendix-F.2.

4 https://msdn.microsoft.com/en-us/library/aa753587(VS.85).aspx

5 https://web.archive.org/web/*/http://addons.mozilla.org

6 https://web.archive.org/web/*/http://chrome.google.com/webstore/

3.2.4 *Cache Location*

Most of the examined systems cache the visited websites at the visiting user's computer with the exception of IPFS and Squirrel using Home-store mode where visited websites instead will be distributed to other users of the respective systems.

For IPFS, this makes a lot of sense, since IPFS is just a distributed file system protocol, not specifically intended for web caching.

For Squirrel using Home-store, this can also easily be defended by the fact that Squirrel is intended to replace a local web caching server used in an organization, so the organization's computers are likely to already be preconfigured with Squirrel, making it mandatory to use it. Additionally, the Squirrel users may already be more inclined to put a higher load on their own machines due to the fact that it will directly benefit the other people in their organization. Users in other systems may be less willing to use their own hard drive space to store website caches for other people. Lastly, Squirrel is intended to work on a LAN, so there will be no additional traffic outside of the local network.

Since Peer Cache is neither meant to be a distributed file system, nor to be used in an organization on a LAN, Peer Cache users should therefore store their own caches, just as Kache, Cooperative Web Cache, and Squirrel in Directory mode does it.

3.2.5 *Cache Searching*

The examined peer-to-peer-based systems – except for YouServ – use a DHT, and since there are no immediate disadvantages to this approach, Peer Cache should do the same. The purpose of YouServ is to be able to access a specific user's files easily, not search for files across users as a DHT would facilitate. YouServ therefore instead uses a dynamic DNS to locate a user – an approach not appropriate in the context of Peer Cache.

3.2.6 *Caching Approach*

Like the other peer-to-peer-based systems, Peer Cache should also use user-driven harvesting as this is a lot less demanding than automatic harvesting.

3.2.7 *Cacheability*

Determining cacheability is difficult and not immune to the effect of time. The net archives can safely cache almost everything, because their web crawlers contain no private state – no cookies with login or

other sensitive data that a regular user's cache may contain. Doing as the net archives will therefore be infeasible for Peer Cache.

The approach used for Peer Cache should therefore be closer to that of the other user-driven harvesting systems such as Squirrel and CWC. However, copying their approaches will also not be sufficient, because the Web today is vastly different from how it was when Squirrel and CWC was created. For instance, Squirrel does not cache encrypted websites, but, as previously mentioned, the number of encrypted websites is increasing every day. But the mere fact that a connection is encrypted, does not mean that the websites' contents necessarily have to be kept private. Disregarding everything encrypted will therefore not be very useful. Likewise, ignoring dynamic content will also not be very useful, because a lot more websites serve dynamic content today, but dynamic content also does not necessarily imply private content. Additionally, determining whether a request is dynamic also becomes increasingly more difficult due to the omnipresence of URL rewriting: the practice of making URLs "cleaner" by turning e.g. `/index.php?category=cars&itemID=5` into something as simple as `/cars/5`. The first URL, for instance, will not be cached by Squirrel, because it contains a query string, whereas the second will, even though they represent the exact same resource.

Kache and CWC can also afford to be more conservative with which caches they serve: worst-case scenario for a cache miss is a slight increase in user-perceived latency, whereas accidentally serving a private resource to the wrong user can be a devastating affair. With Peer Cache, we have the same consequences of misserving a private resource, but the worse-case scenario for a cache miss is also a lot worse: it could, for instance, mean a permanently lost website of historical importance.

It is therefore clear that Peer Cache needs a more thorough approach to cacheability than what the examined peer-to-peer systems have to offer.

3.2.8 *Accessibility*

The Internet Archive and other web preservation initiatives allow users to browse and access cached versions of websites directly from their browser without the need for any third-party software to be installed. In contrast, neither of the examined web caching peer-to-peer systems do this, solely because their purpose is to increase performance (and/or reduce server load), not to provide web preservation. Peer Cache, however, intents to provide web preservation. Therefore, allowing users to browse and access caches without having to install any software is of paramount importance, should Peer Cache be able to compete with the Internet Archive in terms of accessibility and usability.

3.3 PRIVATE SET INTERSECTION

Whether any of the specific PSI protocols examined should be used (Naive protocol, OPRF-based protocols, etc.) is mostly an implementation detail, seeing as most of the protocols perform the exact same task.

Although all one-way protocols can be expanded to become two-way protocols, this is not a necessity with Peer Cache, as only the cache-providing party needs to compute the intersection. Thus, one-way protocols will not be an issue for our needs – in fact, they may even be preferred, due to the presumably lower computational requirements.

The server-aided PSI protocols may appear desirable due to their efficiency. In a peer-to-peer network, they could also be implemented by using a third party as the server. However, due to the added threat of Sybil attacks (where one party assumes multiple identities and takes on the part of the server), they soon become rather disadvantageous. With the great availability of two-party protocols, there is arguably no reason to take the chance for the potential performance benefit they may offer.

Lastly, seeing as websites mostly constitute of HTML – which are trees rather than sets – it will be beneficial to choose a simpler protocol, so that it may more easily be modified to accommodate the slightly different setting.

3.4 REQUIREMENTS

In summary, the requirements for Peer Cache are as follows:

1. Must provide distributed web caching through a peer-to-peer system.
2. Must be scalable, because limiting scalability also limits availability of caches.
3. Must be implemented as a browser extension to facility caching of encrypted websites.
4. Must use local caching to reduce both bandwidth and space requirements of cache providers.
5. Must provide searching through a DHT, as a way of furthering scalability.
6. Must allow users without the browser extension to search and browse the caches in order to be as accessible as other web preservation initiatives.
7. Must allow users to exchange caches without private data leaks using an adaptation of a two-party PSI protocol.

3.5 SUMMARY

In this chapter, the related work has been analyzed and requirements for Peer Cache have been established based on the analysis.

It was found that Peer Cache should be implemented using a peer-to-peer-based web caching system implemented as a browser extension where each user of the system stores their own caches. The system should facilitate searching through a DHT and secure cache sharing using a two-party PSI protocol.

4 TECHNOLOGY

In this chapter, the technologies that will be used in the design and proof-of-concept implementation are presented.

The system consists of four part, the browser extension, a browser application, an indexing server and a web frontend, all developed in JavaScript.

A browser application is needed due to technical limitations which are explained further in the Design chapter (Section 5.1) and Implementation chapter (Section 6.2), but is also briefly brought up in this chapter (Section 4.3).

The system will be designed as if it used a DHT, but the actual implementation will be using an indexing server in its place to simplify the proof-of-concept implementation. The reasoning behind this can be found in the introduction of the Implementation chapter (Section 6). The web frontend will facilitate cache access to users without the browser extension and app installed as per Requirement 6 presented in the previous chapter (Section 3.4) and Hypothesis 4 in the first chapter (Section 1.1.1).

4.1 JAVASCRIPT

JavaScript is a general purpose, dynamic, untyped programming language, standardized by the ECMAScript language specification. JavaScript is available in all modern browsers and has extensive APIs for interacting with HTML and CSS – the languages used for building the websites that need to be cached, making JavaScript a great candidate.

Additionally, writing all parts of the implementation in the same language would ease development. JavaScript has become the de facto standard for browser extension development across most popular browsers (Chrome, Firefox, Opera, etc.). Likewise, JavaScript is gaining popularity as a server-side language, most notably through NodeJS, currently serving as backend for many major websites, including LinkedIn[1], PayPal[2] and Netflix[3]. For these reasons, it becomes the language of choice for both the browser extension, indexing server and web frontend.

1 http://venturebeat.com/2011/08/16/linkedin-node/
2 https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/
3 http://techblog.netflix.com/2014/11/nodejs-in-flames.html

4.2 BROWSER EXTENSION

Browser extensions are add-ons that allow developers to add additional functionality and features to a browser. Browser extensions are browser-dependent, meaning an extension for Google's Chrome browser will fail to work with Mozilla's counterpart, Firefox, and vice versa.

Browsers make a number of APIs available to the developer, allowing the extensions to inspect and modify websites as they are presented to the user, store data on the computer, make web requests, and sometimes search and modify browsing history and caches, etc.

Common browser extensions such as AdBlock[4] modifies web sites to remove advertisements and other unpleasantries from the browsing experience. Others, such as Hola[5], allow users to tunnel all their web data through other users, creating a peer-to-peer-based VPN.

4.3 CHROME PLATFORM FOR BROWSER EXTENSION

To reach the biggest possible audience, the browser extension should be developed for the most popular browser on the market. Determining which browser is definitively the most popular can be difficult since no central authority can monitor all browser users due to the decentralized nature of the Web. However, according to both W3Schools[6], SitePoint[7], and W3Counter[8], Google's Chrome browser seems to dominate the market. As a result, the browser extension will be developed for Chrome.

Due to technical limitations explored later in the Design chapter (Section 5.1) and Implementation chapter (Section 6.2), the browser extension will have two parts to it: A Chrome extension and a Chrome app.

4.4 INDEXING SERVER IN NODEJS

NodeJS[9] is a cross-platform runtime environment for JavaScript with a rich package manager (npm) hosting over 225,000 packages, severely speeding up development and prototyping, while still providing great performance, as evident by its popularity among big sites such as LinkedIn, PayPal and Netflix.

4 https://getadblock.com/
5 https://hola.org/
6 http://www.w3schools.com/browsers/browsers_stats.asp
7 http://www.sitepoint.com/browser-trends-january-2015-ie8-usage-triples/
8 https://www.w3counter.com/trends
9 https://nodejs.org/en/

4.5 COMMUNICATION USING WEBSOCKETS

All communication, both among peers and between indexing server and peers, will be done using WebSockets[10]. WebSockets is a full-duplex communication channel working through a single TCP connection. WebRTC[11] was also evaluated as an option, but deemed infeasible due to technical issues with the implementation used in Chrome extensions.

4.6 SUMMARY

In this chapter, the chosen technologies have been presented.

The system will be designed as a browser extension, a browser app, an indexing server and a web frontend. All parts will be developed in JavaScript with peer communication over WebSockets.

10 https://tools.ietf.org/html/rfc6455
11 https://webrtc.org/

5 DESIGN

This chapter presents the design of Peer Cache, developed based on the requirements established in the Analysis chapter and the technological assessments made in the Technology chapter.

Peer Cache consists of three parts:

- A Chrome extension for collecting and showing caches.
- A Chrome app for sharing caches.
- A web frontend for browsing and showing caches to users who have not installed the extension and app.

5.1 OVERVIEW

Accessing a cache of a website can happen in two different ways, either through the web frontend or directly through the Chrome extension. Using the extension, the activation is automatic, but other than that, the process is largely the same for either case.

Using the extension, a user may navigate to a website that is down. The Chrome extension detects this and promptly shows a list of available caches for the user to choose from. Without an extension, when a user finds a website that is down, the user will have to manually navigate to the web frontend and enter the URL of the resource that the user attempted to access. Once this has been done, the user will also be presented with a list of available caches, almost identical to the list shown to the user of the extension. The process can be seen in Figure 1.

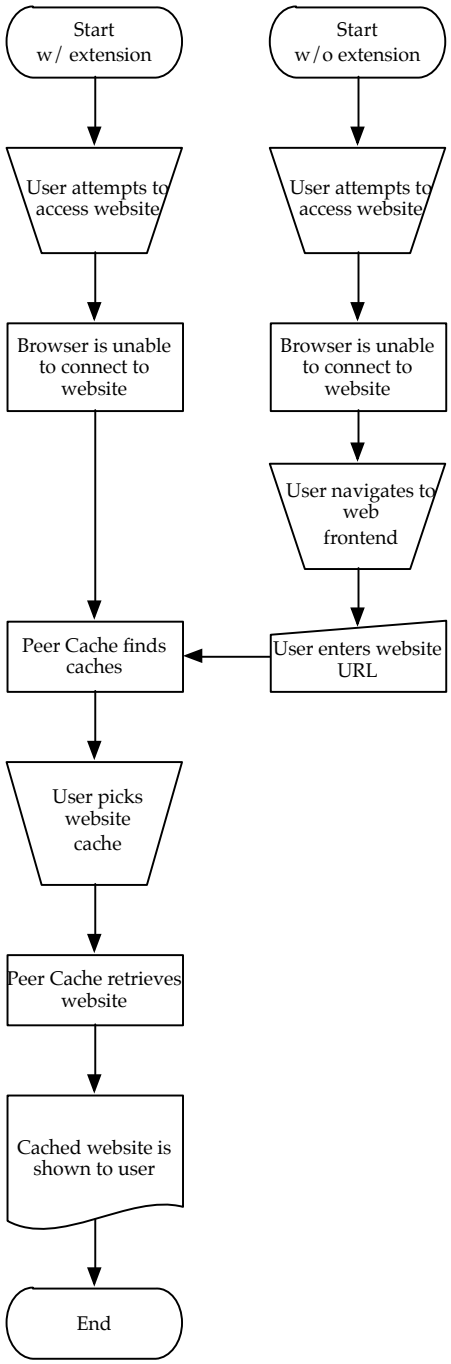


Figure 1: Flow chart of a user interaction with Peer Cache.

Due to limitations in Google Chrome's extension API, the Peer Cache browser extension consists of two parts: A Chrome extension and a Chrome app. Chrome extensions are not allowed to listen for connections, but in order to allow incoming connections from other peers, we must be able to listen on a port. Chrome apps, on the other hand, are allowed to listen on ports. The other part – the Chrome extension – we need, because Chrome apps are neither allowed to have *content scripts* (more on content scripts in Section 6.4), nor interfere with tabs and browser windows. This makes it impossible to see when the user encounters a website that is down, and more importantly, makes it impossible for a Chrome app to collect caches, create tabs and present the caches to the user.

Luckily, Chrome's Message Passing API is available to both Chrome apps and extensions, meaning we can send messages between the app and extension seamlessly.

5.2 THE CHROME EXTENSION

The overall responsibility of the Chrome extension is to collect, share and show caches to the user. This is achieved through a number of components, most notably the *Cache Table* and *Peer Table* shared among the Peer Cache users (Distributed Hash Tables), as well as the local components to handle caches (Cache Manager), communicating with peers (Peer Helper), and cleaning cached documents, making them fit for sharing (Private Tree Intersection).

5.2.1 *Cache Table and Peer Table*

When Peer Cache is run for the first time, the extension generates a unique browser identification number (client ID). This client ID is used whenever a cache has been collected and needs to be added to the Peer Cache network. Simply associating an IP address with a cache is not a viable solution, because multiple users may share the same IP address, or users may periodically be changing IP addresses for a plethora of different reasons (dynamic IP addresses, network roaming, etc.). Therefore, caches are associated with peers (client IDs) through a permanent Cache Table, and peers are associated with network addresses (IP address and listening port) in a more temporary Peer Table. An example workflow may be as follows:

1. A user, Alice, on IP address `10.1.2.142` runs Peer Cache for the first time. Peer Cache immediately generates a random client ID `dvn3oxy2` and adds this to the Cache Table in the peer-to-peer system along with her IP address.

2. Alice navigates to `stackoverflow.com`, which gets registered in the system alongside her client ID.

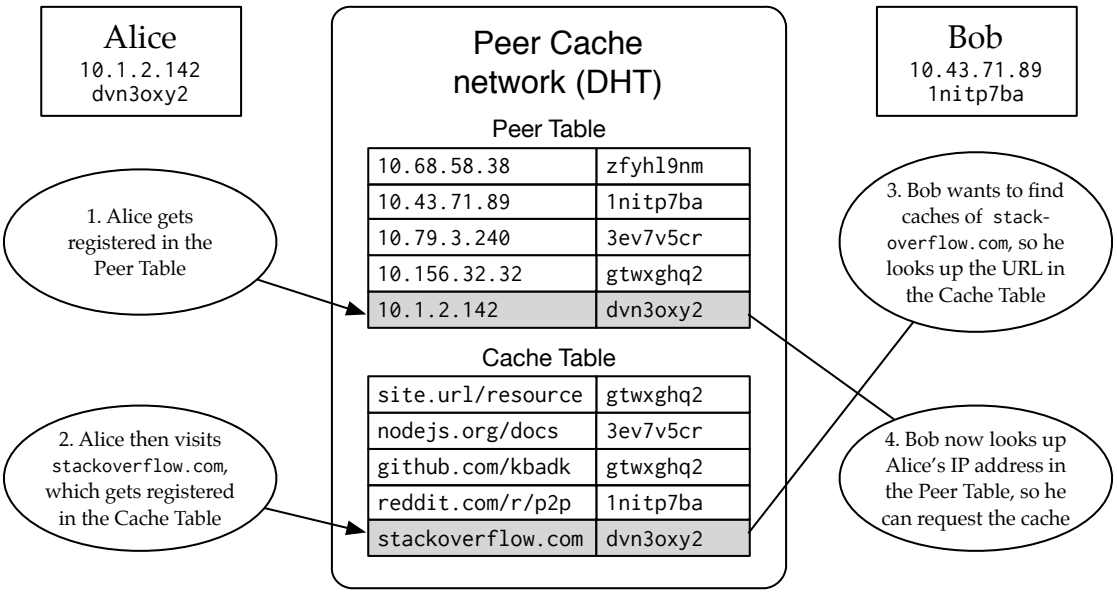


Figure 2: Cache Table and Peer Table in Peer Cache.

3. Now, another user, Bob, wants to find caches for `stackoverflow.com`, so he navigates to `stackoverflow.com` in his own browser, activates Peer Cache (or uses the web frontend) and sees Cache Table entries for the site. Bob picks Alice's cache from the Cache Table through the Peer Cache user interface.

4. Peer Cache in Bob's browser now looks up Alice's client ID `dvn3oxy2` in the Peer Table to get Alice's IP address `10.1.2.142`[1] after which Bob's Peer Cache extension contacts Alice's Peer Cache app to retrieve the cache.

The process can be seen in Figure 2.

When Alice leaves the network (by closing her browser, disabling the extension, turning off her computer, etc.), the network is informed and her entry from the Peer Table is removed. The next time she joins, possibly from a new IP address, the new IP will be added to the Peer Table along with her client ID `dvn3oxy2`, so users (such as Bob) once again can access her caches. Notice how the Cache Table is unaffected by users joining and leaving the network.

The two tables are stored in the Peer Cache network using Distributed Hash Tables (DHT). The specific DHT implementation is not important to the design, but for now, the reader can imagine an implementation similar to Chord and Kademlia.

1 Bob also gets the port number of the Peer Cache app running on Alice's machine.

5.2.2 *Privacy*

Sharing caches opens up for a bunch of privacy issues, most notably the fact that caches meant for sharing may contain private information, as has been mentioned earlier. Consider the detrimental effects of a user unintentionally sharing their emails, bank statements or health records, after having used the respective websites while Peer Cache was quietly running in the background, collecting and sharing caches with other users of the system.

To prevent this, collected caches will automatically be saved as *dirty*, meaning they are unfit for sharing as they may contain private information that ought not be disclosed. The user may manually declare that a cache is safe to share by *starring* it, but for the remaining dirty caches, we need a way to make them *clean*, i.e., ridding them of private information. This will be done using Private Tree Intersection.

5.2.3 *Private Tree Intersection*

Cache cleaning is done using Private Tree Intersection (PTI) – a modification of PSI (introduced in Section 2.4 in the Related Work chapter) working on trees instead of sets. All PTI logic is encapsulated in a PTI class. The basic functionality of the PTI class should enable a user to generate a Hash Tree Structure (HTS)[2] of a document, as well as generate a clean document if provided with a local, dirty document and an HTS from another peer.

An example, as well as the implementation details of how the PTI class works, is covered in the next chapter on implementation (Section 6.7).

5.2.4 *Cache Manager*

All cache handling is done by the Cache Manager class. The Cache Manager encapsulates everything related to saving, listing and accessing caches. Whether the cache is located locally or remotely, the retrieval will happen through the Cache Manager. This means that any component in the extension that needs to interact with caches may do so without worrying about how to retrieve the cache. Regardless of whether the cache is located on another peer in the network or in the local storage, it happens transparently through the Cache Manager. In order for the Cache Manager to be able to do so, however, it will have to communicate with peers. This is done through the Peer Helper class. The Cache Manager will also interface with the PTI class to compute

2 An HTS is a tree structure containing hashes of document elements, not to be confused with the persistent data structure: hash tree. A better name may be a document hash tree, but the obvious abbreviation (DHT) could easily become confusing in this context.

clean documents through means of local document caches and hash trees retrieved from other peers through the Peer Helper. Once a clean cache has been computed, the Cache Manager saves it, so it will be easier to serve next time the cache is requested.

5.2.5 *Peer Helper*

The responsibility of the Peer Helper class is to handle all peer communication, both searching the Cache Table DHT for caches, and retrieving caches or hash trees from other peers by locating them through the Peer Table DHT and connecting to them.

5.3 THE CHROME APP

The Chrome app's responsibility is solely to handle incoming requests for caches and HTSs. Since these requests both regard caches, it is convenient to delegate them to the Cache Manager in the extension. Rather than manually sending requests to the Cache Manager from the Chrome app using Chrome's Message Passing API, this communication has been abstracted away in a Cache Manager Stub that uses the Message Passing API to perform calls on the extension's Cache Manager.

The incoming requests (coming from the Peer Helper class of other peers) is handled by a Peer Listener class in the app, which in turns uses the Cache Manager Stub to respond to the requests.

5.4 THE WEB FRONTEND

To allow access to the caches without having either the browser extension or app installed, a web frontend needs to be made available. While some initiatives for distributed web hosting exist (YouServ[3], ZeroNet[4], BitTorrent's Maelstrom[5], etc.), they all require special-purpose software or custom configurations to achieve it, conflicting with the goal of allowing users to access the caches, equipped with nothing but an extensionless browser.

As a result, the web frontend is centralized, i.e. hosted on a server that is not actually part of the peer-to-peer network. The web frontend allows users to search and browse caches, much like the Chrome extension does. The Chrome extension does this through the Cache Manager and Peer Helper classes. Note that the PTI class is not necessary here, as PTI is only needed when sharing cached documents or HTSs of the cached documents.

3 See Section 2.2.2.
4 https://zeronet.io/
5 http://project-maelstrom.bittorrent.com/

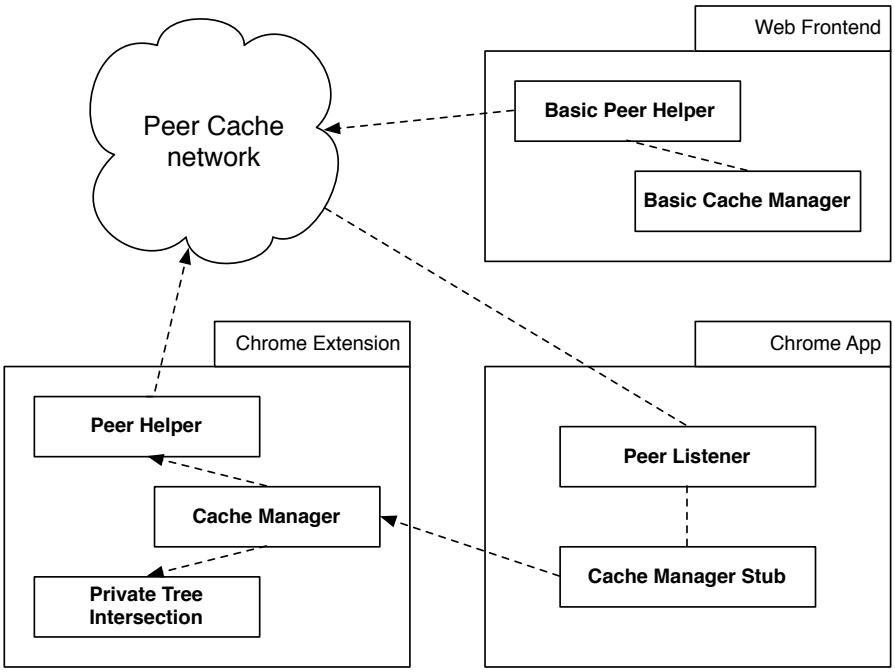


Figure 3: Illustration of how the components of the Peer Cache ecosystem interact.

By using slimmed-down versions of the Cache Manager and Peer Helper (namely Basic Cache Manager and Basic Peer Helper), a static web frontend can be created, allowing users to do most of what they were capable of with the Chrome extension. Since the web frontend can be made static, and all caches are still retrieved from the peers in the network, it is unlikely this centralized component of Peer Cache should become a bottleneck, considering that modern web servers can serve hundreds of static documents per second.

5.5 ARCHITECTURE

Now that the territory has been charted, we can delve into the architecture and how the components of Peer Cache interact.

As previously explained (and illustrated in Figure 3), the three main components (the web frontend, Chrome extension and app) communicate together through the peer classes, namely the Peer Helper in the extension, the Basic Peer Helper in the web frontend, and lastly the Peer Listener in the app that handles requests coming from the Peer Helper classes.

5.5.1 *Handling Cache Requests*

There are two kinds of outside requests that can be made to the Peer Listener: a regular cache request and a request for an HTS (see Sec-

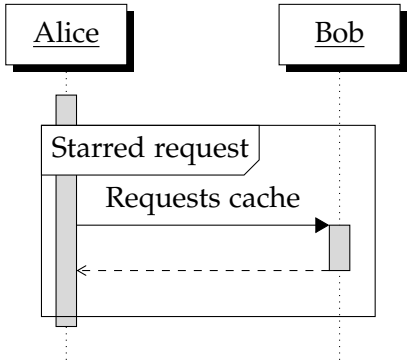


Figure 4: Sequence diagram of Alice requesting a starred or client cache from Bob.

tion 5.2.3). Some caches may already be clean, because they previously have been requested and thus undergone PTI transformation, making them safe to share. Other caches may have been manually deemed safe by the user through the starring functionality, thus not requiring any PTI transformation at all. These two kinds of cache requests do not need to involve any other peers than the cache consumer and provider making the exchange.

The remaining cache requests are requests to dirty caches. In order for these requests to be resolved, another peer has to provide an HTS for the the document that is being requested, so the PTI class can compute a clean cache that is safe to share.

Let us consider a user, Alice, who visits the web frontend to retrieve Bob's starred cached version of `www.stackoverflow.com`. When this happens, the Basic Cache Manager gets a request for `www.stackoverflow.com`. The Basic Peer Helper then retrieves a list of caches from the DHT. The list is handed back to the Basic Cache Manager and displayed to Alice. She then chooses a cache, which in this case happens to belong to Bob. A request is made to Bob's Peer Listener. Bob's Peer Listener requests the cache from the Cache Manager Stub, which in turns requests it from the Cache Manager in Bob's extension. The cache then travels back through the extension and Peer Listener in Bob's app to the Basic Peer Helper in the web frontend that Alice is using, after which it is presented to Alice. The only network involvement is a single request and reply between Alice and Bob, as can be seen in the sequence diagram in Figure 4.

However, if the requested cache is neither starred, nor has previously been cleaned, the process becomes a little more complex, and another user, Carol, needs to be involved in the transaction. As before, Alice has requested a cache and chosen a cache of Bob's, this time one that is dirty.

A request is sent to Bob's Peer Listener, which contacts Bob's Cache Manager through the stub. The Cache Manager detects that the document is dirty and looks through the DHT to find a similar cache.

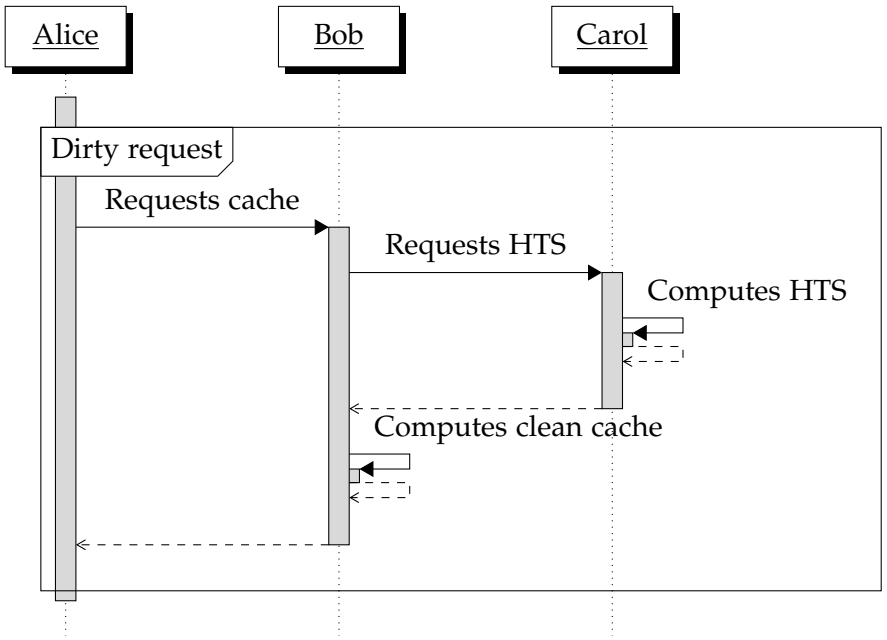


Figure 5: Sequence diagram of Alice requesting a dirty cache from Bob.

The more similar the caches are, the less information will have to be removed, creating more complete, clean documents.

If an identical cache exists in the DHT (determined by comparing hashes), it is assumed that no private information exists in the cache, or otherwise other peers would not have an identical cache. The cache is therefore marked as clean and sent off to Alice. A more likely scenario is that no identical caches exist, in which case the cache collected closest in time (least time difference) to the requested cache is selected. The HTS of this cache is then requested from Carol, after which Bob's Cache Manager computes a clean document using the PTI class, and finally returns the clean document to Bob's Peer Listener which sends it back to Alice. This process can be seen in Figure .

Instead of a single network request, this process requires 3 requests, as can be seen in the sequence diagram in Figure 5.5.1.

5.6 SUMMARY

In this chapter, the Peer Cache design has been presented.

The design consists of three main components: a Chrome extension for collecting caches and viewing them, a Chrome app for serving caches and a web frontend allowing users without the extension to view caches. The caches can be located through the Peer Cache DHT. Before caches are sent to a requesting user, they are being rid of private information by using PTI.

6 IMPLEMENTATION

This chapter describes the proof-of-concept implementation of the Peer Cache system. The implementation will somewhat deviate from what is presented in the Design chapter. Most notably, the system relies on an indexing server instead of a DHT for storing the Peer and Cache Tables (see Section 5.2.1). This functionality has been omitted, because it has been deemed too time-consuming to implement compared to the value it contributes to the implementation. After all, implementing a well-functioning DHT is not a trivial task[1]. On top of that, the functionality of a DHT can rather easily be simulated using an indexing server, allowing us to test the hypotheses presented in Section 1.1.1.

The structure of this chapter is as follows: First, the structure of Chrome extensions and apps are explained (Section 6.1), followed by a section on how network communication is facilitated using WebSockets (Section 6.2). Afterwards, the Chrome platform's cache APIs will be examined and Peer Cache's own caching implementation is introduced as well as some optimizations that were performed on the caching system (Section 6.3). Following this, Peer Cache's cache collection is described along with its challenges (Section 6.4). Continuing with the caching theme, Peer Cache's cache serving is explained and showcased (Section 6.5). After caching, a small section on a Token Callback Table that has been implemented to keep track of asynchronous calls is introduced (Section 6.6). The chapter concludes with an explanation and code samples of the Private Tree Intersection algorithms used (Section 6.7).

6.1 STRUCTURE OF A CHROME EXTENSION AND APP

Common for both Chrome extensions and Chrome apps is that they contain a `manifest.json` file ("the manifest") with meta information about the extension/app, as well as a list of background scripts. Background scripts are the backbone of Chrome apps and extensions – they are JavaScript files that are run during initialization. In these scripts, extensions set up event listeners for browser events using the Chrome platform APIs, or in the case of an app, sets up a listening ports for network events or other event listeners. Additionally, Chrome extensions may contain content scripts, which are JavaScripts that are executed at the end of every page load, much as if were they part of the loaded website itself (see Section 6.4). In the manifest,

the developer specifies a URL pattern for which websites the content script is to be run in. For Peer Cache, this is all websites.

6.2 NETWORK COMMUNICATION

The Peer Cache app listens on a port from where it may receive requests from others peer in the Peer Cache network and resolve them using its Cache Manage Stub, which in turn uses the Cache Manager in the extension. For the sake of simplicity, the implementation generates a random port number (in the interval 7000-8000) to listen on during startup, to more easily allow multiple instances of Peer Cache to run on a single machine while testing.

The mechanism used for all network communication in Peer Cache is HTTP WebSockets (introduced in Section 4.5). A BSD-licensed HTTP WebSocket server developed by Chromium developers (made available through Google Chrome's GitHub repository for Chrome app samples[1]) has been used. The WebSocket implementation relies on Google Chrome's socket API[2], which is only available to apps, not extensions[3], hence the need for an app. The Chrome extension uses the native WebSocket client implementation[4] to connect to the Chrome apps of other peers in the network. All requests consist of serialized JavaScript objects (JSON), containing a token and a payload. The payload is the actual data and the token is meta data, allowing us to identify and handle request responses appropriately (more on tokens in Section 6.6 on Token Callback Table). All communication is (according to design) encapsulated by the Peer Helper and Peer Listener in the extension and app, respectively. The interaction can be seen in Figure 3 in the Design chapter (page 34). In the prototype implementation, the functionality of the Peer Listener is, however, embedded directly in the app's background script due to its simplicity; the app's background script consists of a mere 100 lines of code. In the extension, Peer Helper correctly encapsulates the functionality as described in the Design chapter.

6.3 CACHING AND CACHE STORAGE

Google Chrome allows developers access to more than 60 APIs for developing extensions. In order to access the browser's cache, three seemingly helpful APIs spring to mind:

1 https://github.com/GoogleChrome/chrome-app-samples/tree/master/samples/websocket-server
2 https://developer.chrome.com/apps/socket
3 https://code.google.com/p/chromium/issues/detail?id=152875#c1
4 https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

HISTORY API

The History API[5] lets the developer search the user's browsing history, and allows for extensions to seach through URLs and retrieve information from the history. As can be seen in the code below, getting direct access to cached resources is not possible through the API. On top of searching and retrieving the very "thin" cache entries, it also allows developers to delete entries from the history, but nothing more.

Listing 1: History API sample.

```
chrome.history.search({text: ''}, function(results) {
  results.forEach(function(result) {
    console.log(result);
  });
});

{
  "id": "2",
  "lastVisitTime": 1445290962798.9968,
  "title": "StackOverflow",
  "typedCount": 3,
  "url": "https://stackoverflow.com/",
  "visitCount": 70
}
...
```

BROWSINGDATA API

BrowsingData[6] allows the developer to queue browsing data related settings and to delete browsing data based on some time criteria. Out of its 14 API methods, 13 of these start with the word "remove" (`remove()`, `removeAppcache()`, `removeCache()`, etc.), the last one simply being `settings()`.

SESSIONS API

Finally, the Sessions API[7] lets the developer restore recently closed sessions, i.e. reopen tabs that have just been closed. This very limited API has `getRecentlyClosed()`, `getDevices()` and `restore()` as its only methods.

After having exhaustively examined all available API, it was concluded that using Chrome's own cache is not possible through the APIs. This may have been obvious from the fact that every extension is sandboxed and therefore not allowed to access anything outside of their own, isolated file system, including caches.

The Cache Manager is therefore implemented using its own cache storage. Chrome's Storage API[8] has three *Storage Areas*, namely *local*,

5 https://developer.chrome.com/extensions/history
6 https://developer.chrome.com/extensions/browsingData
7 https://developer.chrome.com/extensions/sessions
8 https://developer.chrome.com/extensions/storage

sync and *managed*. Managed is administered by a domain administrator, sync is synchronized between all the user's browsers and is rate-limited, leaving just local, which therefore is used. The local API is similar in both design and functionality to `localStorage`, part of the HTML5 Web Storage API[9]. Like Web Storage, accessing the storage is asynchronous, utilizing callbacks for storage processing.

6.3.1 *Speeding Up Retrieval of Cache List*

The initial implementation stored all caches in an object `caches` in the local storage. The storage only allows the developer to select full objects, meaning whenever a lookup occurs, the entire cache object (containing all caches in the system) is loaded into memory. As this object grows over time, it will become rather big, making lookups take an increasingly long time. Benchmarks revealed that loading approximately 50 MB into memory using `chrome.storage.local.get()` took anywhere from between 600 ms to 4 full seconds. As a result, when the user tried to see which caches were available for a website, the remote caches found on the indexing server were presented to the user faster than the local caches.

To speed up the initial retrieval of the list of caches, the storage was split into two objects: an index `cacheIndex` and the full cache `cache`. The index object contains the same as the `cache` object, except for the website itself. Since a website itself usually accounts for more than 99% of the cache size (the rest being meta data), the optimization reduced the retrieval time of the cache list to about 2 ms on average with a similar cache storage population.

The retrieval of a specific cache from `caches` remained the same (unsurprisingly), but this is a more rare occurrence than retrieving just the list, and therefore more justifiable. Comparably, retrieving a cache from `https://archive.org/` may also take several seconds[10]. This process, however, can also be sped up by minimizing the cache sizes using compression.

6.3.2 *Compressing Data*

Another way of minimizing the local storage overhead would be to reduce the size of what is stored through compression. An added benefit of compression is that this also will reduce network traffic (thus increase perceived transfer speed) as well as reduce disk space usage.

Several compression algorithms were evaluated, but eventually `lz-string`[11] was chosen, because it is both more lightweight than Jame-

9 `https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API`
10 This is examined further in the discussion on end-to-end testing in Section 7.5.
11 `http://pieroxy.net/blog/pages/lz-string/index.html`

Site	Uncompressed Size (KB)	Compressed Size (KB)	Space Saving	Compression Time (ms)	Standard Deviation
Test site	185	91	51%	143	9%
Wikipedia	1163	503	57%	809	3%
StackOverflow	881	152	83%	403	9%
GitHub	733	130	82%	344	10%
Twitter	5764	2005	65%	3797	3%
Imgur	4597	1983	57%	3405	6%

All website visits were to the front page of the respective sites. Each test was performed 10 times and the mean values registered. For compression time, the standard deviation was also calculated. The compression algorithm is deterministic, so the sizes remained the same between runs.

Table 2: Evaluation of `lz-string` compression algorithm.

son Little's JavaScript GZip implementation[12] (weighing in at 70 KB as well as having `deflate.js` and `crc32.js` as dependencies against `lz-string`'s 15 KB), and a lot faster and more space-efficient[13] (Approximately 10 times faster and twice as space-efficient) than Nathan Rugg's JavaScript implementation of the Lempel-Ziv-Markov (LZMA) algorithm[14].

When testing `lz-string` on real-world examples, the implementation (using the valid UTF-16 mode) was found to compress most common websites by 51% to 83% in less than a second (See Table 2). For image-heavy sites, however, the compression could take up several seconds.

Using the "invalid UTF-16 mode", even better compression scores were achieved, but storing the data in the `chrome.storage-.local` corrupted the entire database, unfortunately. This is seemingly due to the fact that the "invalid" UTF-16 characters are not properly escaped before being inserted into Chrome's underlying SQLite database. Note that the documentation of `lz-string` states that invalid UTF-16 strings can be stored in `localStorage`. While this storage is similar to the extension API's local storage, its implementation must differ after all, evident by the corruption.

6.4 CACHE COLLECTION AND CONTENT SCRIPTS

A content script is a JavaScript file that is injected into every website that is visited and executed in a so-called *isolated world*[15]. In the iso-

12 https://github.com/beatgammit/gzip-js
13 http://pieroxy.net/blog/pages/lz-string/demo.html
14 https://github.com/nmrugg/LZMA-JS
15 https://developer.chrome.com/extensions/content_scripts#execution-environment

lated world, the content scripts have access to the DOM, but not any functions or variables defined in the site's JavaScript code. For the content script, it will appear as if no other JavaScripts are running on the page, and vice versa for the existing JavaScripts on the website. In return for the isolation, content scripts are allowed to access the Chrome APIs, making them good bridges between the extensions and the websites the extensions are meant to interact with.

Peer Cache injects a content script into websites to "collect" the website, so it can be saved for later retrieval using the Cache Manager. This is done by copying the entire DOM tree as a string and emitting a `CACHE_COLLECTED` event (containing the DOM string) that the background scripts will listen for and propagate on to the Cache Manager, which then registers and stores it.

Simply serializing the DOM, however, will not yield very rich caches, as websites often contain external resources such as images, stylesheets, and scripts. These resources therefore need to be embedded into the DOM, lest they would be missing from the caches. Since embedding these resources into the DOM may not always create a perfect representation of the website, all manipulations are instead done on a copy of the DOM, so the user experience will not be degraded for the user currently visiting the website.

6.4.1 *Embedding Stylesheets*

Stylesheets that are already embedded in the DOM will serialize gracefully and thus not call for any further action. External stylesheets, on the other hand, need to be redownloaded and included. Keep in mind that no data is actually transferred across the fiber when redownloaded, since the stylesheet will already be in the browser's own cache.

As an example of how an external stylesheet is embedded, consider a regular `<link>` tag in the `<head>` pointing to an external CSS file, like the one seen in Figure 6 (a). This file will be downloaded, base64-encoded and a new `<link>` tag containing the base64 string, as seen in Figure 6 (b), will take the old stylesheet's place in the DOM.

```
body {
  background: #ffa;
}
h1, h2, h3 {
  font-family: Helvetica, sans-serif;
}
```

(a) External CSS file.

```
<link href="data:text/css;base64,
  Ym9keSB7CgliYWNrZ3JvdW5kOiAjZmZh
  Owp9CmgxLCBoMiwgaDMgewoJZm9udC1m
  YW1pbHk6IEhlbHZldGljYSwgc2Fucy1z
  ZXJpZjsKfQo=" rel="stylesheet"
  type="text/css">
```

(b) Inlined base64-encoded CSS file.

Figure 6: Example of how an external CSS file is inlined in the HTML.

References to images (using `url()`) in the stylesheets will also have to be inlined as base64-encoded strings. This is done in a similar fashion to how `<img>` tags are handled, which is explained next. References to web fonts are not altered, and will thus not be cached. While technically possible, it was deemed non-essential to the functionality of the prototype.

6.4.2 *Embedding Images and CORS issues*

Images are refetched using the content script and drawn onto a canvas. Subsequently, the canvas is converted into a data URL, which will then contain the image as a base64-encoded data string. A simplified procedure for this can be seen in Listing 4. Notice how the `crossOrigin` is set to `anonymous` on line 3 to circumvent the same-origin policy using cross-origin resource handling (CORS).

Listing 4: Image URL to base64 data URL code

```
1  function imageUrlToDataUri(url, resolve, reject) {
2    var image = new Image();
3    image.setAttribute('crossOrigin', 'anonymous');
4    image.src = url;
5
6    image.onload = function() {
7      var canvas = document.createElement("canvas");
8      canvas.width = image.width;
9      canvas.height = image.height;
10
11     var ctx = canvas.getContext("2d");
12     ctx.drawImage(image, 0, 0);
13
14     resolve(canvas.toDataURL("image/png"));
15   };
16
17   image.onerror = function() {
18     ...
19   }
20 };
```

The same-origin policy is an essential part of the web application security model. The policy ensures that certain resources may only be requested from "the same origin", i.e. the same website. Without the same-origin policy, nothing would prevent a malicious website from performing cross-site scripting (XSS) attacks, e.g. by sending web requests to a visitor's mail or banking website, on behalf of the user.

HTML5 introduces Cross-Origin Resource Handling (CORS)[16] to complement the same-origin policy by allowing developers to relax the restrictions it enforces. When using CORS, a website may set a `Access-Control-Allow-Origin` (ACAO) header containing a list of websites from which its resources may be requested. Mind you, this is done to protect the user, not the website.

16 https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

For a banking site, the header would only contain websites from trusted partners or it may be left empty, in which case the same-origin policy applies without any relaxation. For content distribution networks, or other sites not serving personal information, it may be convenient to set the ACAO header to allow all origins, using a wildcard asterisk character (`*`).

The ACAO header is only served to requests including an `Origin` header. The `Origin` header includes the address of the parent website, i.e. the website on which the script issuing the request is being executed from. If the address in the `Origin` header does not match any addresses in the ACAO header, the client's browser will reject the request. Likewise, if no `Origin` header exists, no ACAO header will be returned, in which case the browser defaults to the same-origin policy, so the request is rejected as well.

Therefore, by setting a `crossOrigin` on the image request, we are allowed to bypass the single-origin policy[17], assuming the website uses CORS and the `Origin` header matches the ACAO header. Luckily, this is often the case as most websites and CDNs serving images have no reason to fear XSS or similar data injection attacks (as there is no personal data that can be leaked), so they can safely use the wildcard asterisk. Some websites still do not use CORS headers, most likely because it has not been widely popularized yet (first accepted as a W3C Recommendation in January 2014[18]), or because it has been deliberately left out to prevent JavaScript-assisted deep-linking.

The `crossOrigin` attribute can take two values: `anonymous` and `use-credentials`[19]. With the former, no user credentials are exchanged, whereas with the latter, cookies, HTTP authentication, etc. are exchanged. `anonymous` is used primarily because credentials are not compatible with wildcard ACAO headers, but also because no credentials are available as the content script is working in an isolated world.

Without CORS, both retrieving the image and drawing it on the canvas would still be possible, but the canvas would become "tainted", meaning data could no longer be pulled out from the canvas. Extracting the data URI using the `toDataURL()` method would therefore no longer be possible.

In cases where websites do not use CORS, the web browser defaults to the same-origin policy – regardless of the `Origin` header – thus refusing to download the image, as is the case with StackOverflow's CDNs, for instance, as seen below.

```
Image from origin 'http://cdn.sstatic.net' has been blocked from loading by Cross-
    Origin Resource Sharing policy: No 'Access-Control-Allow-Origin' header is
    present on the requested resource. Origin 'http://stackoverflow.com' is
    therefore not allowed access.
```

17 https://developer.mozilla.org/en-US/docs/Web/HTML/CORS_enabled_image
18 https://www.w3.org/TR/2014/REC-cors-20140116/
19 https://developer.mozilla.org/en-US/docs/Web/HTML/CORS_settings_attributes

When this happens, our cache will simply lack the image. Very little can be done to prevent this, but fortunately more and more websites are embracing CORS every day.

6.4.3 *Embedding JavaScript and CSP issues*

Since the serialized DOM is being saved in the cache, not simply the HTML source file, any initial JavaScript code affecting the DOM will already have fired when Peer Cache attempts to collect the website. This is quite fortunate, because JavaScripts cannot be inlined at all due to Chrome's Content Security Policy.

Much like CORS, Content Security Policy (CSP) is part of the web's security model. Where the intention of CORS is to protect users from malicious websites trying to send requests to friendly sites on behalf of the user, CSP works somewhat the other way around. CSP aims to protect users on friendly sites from accidentally executing code from malicious sites.

Consider a scenario where a malicious user has managed to include a `<script>` tag on a friendly website, for instance by exploiting the developer's ignorance of data sanitation. When an unsuspecting user then navigates to the compromised friendly website, the malicious user's JavaScript code is executed in the user's browser and all hell breaks loose.

To prevent this from happening, CSP allows website owners to set a `Content-Security-Policy` header on every response to the user[20]. The CSP header contains a list of sources that the developer has deemed safe. A CSP header containing `script-src 'self' https://apis.google.com`, for instance, tells the browser that it should only execute scripts from the sources `'self'` (meaning the website itself) and Google's API website. If the browser then at any point later on is asked to execute scripts from any other source, it will refuse and instead display an error in the browser console, explaining that the potentially malicious source is not part of the CSP. In this example, the CSP header only contains script-related privileges (due to the `script-src` directive), but a variety of other directives also exist to limit other resources, e.g. `img-src`, `style-src` and `frame-src`.

CSP is implemented in browsers to restrict data flow between websites, but the concept has also been carried over to Chrome's extension framework. Like websites serve a CSP header, a similar setting can be set in the manifest file[21]. Even if the setting is left out, a strict default will be enforced. When trying to inline JavaScript files from the content script, the CSP therefore steps in and the browser refuses execute to the code.

20 http://www.html5rocks.com/en/tutorials/security/content-security-policy/
21 https://developer.chrome.com/extensions/contentSecurityPolicy

```
Refused to execute inline script because it violates the following Content
    Security Policy directive: "script-src 'self'". Either the 'unsafe-inline'
    keyword, a hash ('sha256-Odxp/SRybDztYilJSoE/9d2Q4dlTkg9MQkyB+c+RYC8='), or a
     nonce ('nonce-...') is required to enable inline execution.
```

As per the documentation, adding `'unsafe-inline'` to the CSP is not possible as of Chrome 46. Adding a hash, nonce, or even the URL itself to the CSP, would also not be feasible, since the `manifest.json` file only gets reloaded when the extension is reloaded. On top of that, it would be less than desirable to fill the extension's configuration file with hundreds or maybe even thousands of URLs. As a result, including JavaScript code in caches is not doable in practice.

On the bright side, if JavaScript code was inlined, the current state of the DOM may not be compatible with the associated JavaScript code if it was to be run again. For instance, if the JavaScript code was saved with the cache, the later-to-be-served cache would contain a modified DOM that may already have been manipulated by the JavaScript the first time it was run during cache collection. When the JavaScript code would then try to manipulate the already manipulated DOM, undefined behavior is likely to ensue. After all, the JavaScript code found on modern websites is rarely idempotent, especially not with current JavaScript frameworks such as React and AngularJS, both relying heavily on templating.

6.5 CACHE SERVING

As mentioned in the Design chapter, caches can be served either through the web frontend or the Chrome extension (recall Figure 1 on page 29).

When the Chrome extension is enabled and a navigation error occurs, the extension is triggered and the error navigation page is replaced by the *Cache Selector* in Peer Cache. Error navigation code in Listing 5 shows how the Chrome's `webNavigation` API is utilized to detect navigation errors and create a Cache Selector Tab using Peer Cache's Tab Manager class (see the actual implementation for more details).

Listing 5: Navigation error handling

```
1 chrome.webNavigation.onErrorOccurred.addListener(function(
      requestDetails) {
2   if (["net::ERR_CONNECTION_REFUSED", "net::ERR_NAME_NOT_RESOLVED",
3     "net::ERR_INTERNET_DISCONNECTED", "net::ERR_CONNECTION_TIMED_OUT"
          ].indexOf(requestDetails.error) !== -1) {
4     tabManager.createCacheSelectorTab(requestDetails.tabId,
          requestDetails.url, requestDetails.error);
5   }
6 });
```

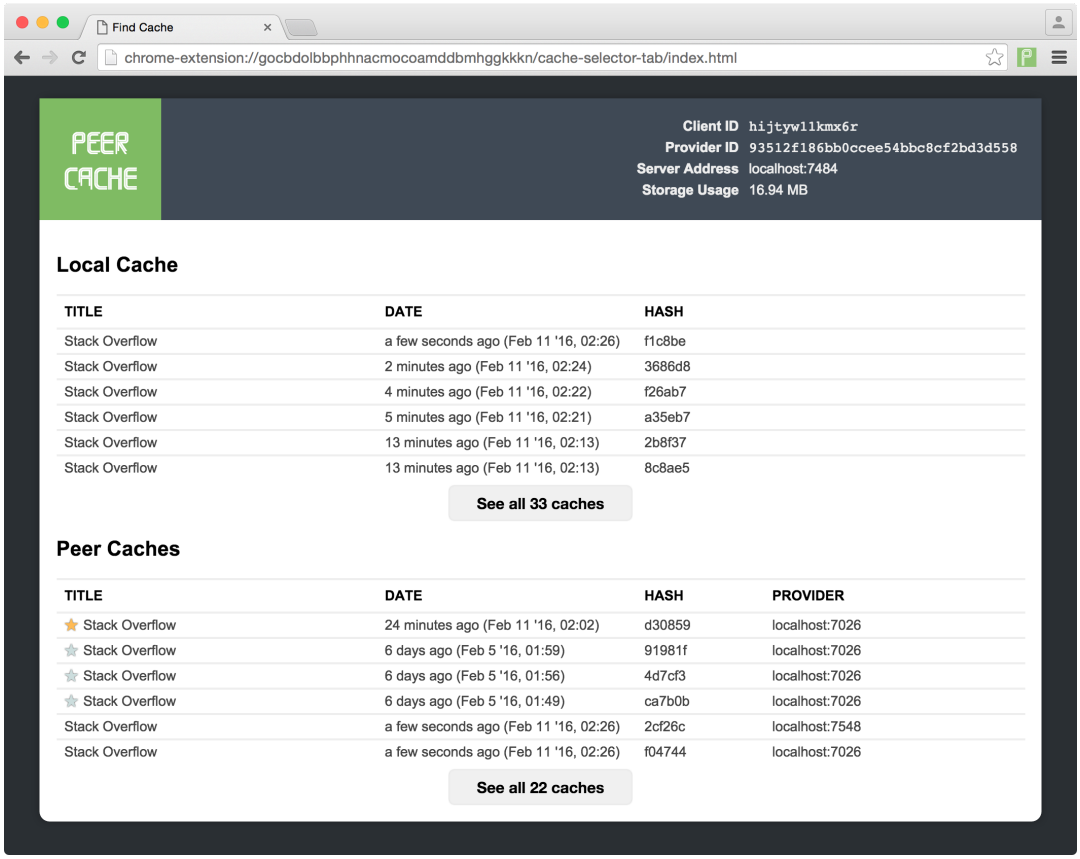


Figure 7: Cache Selector tab for StackOverflow.

The Cache Selector tab, as can be seen in Figure 7, presents the user with two lists of caches, local caches and peer caches, as well as some meta data about the client, specifically the Client ID, Provider ID[22], the address the Chrome app is listening on and how much storage the Chrome extension is using for caches.

The top list of local caches are all the caches the user has collected herself, presented along with their collection time and a hash of the document. The bottom list shows caches available on other peers. If a cache exists, but the peer is offline, the table entry will be dimmed. In Figure 7 all caches shown are online. Next to the website title on some of the caches, a star appears. The gold star symbolized that a cache is starred and the silver star: a cache that has already been cleaned (see Section 5.5.1). If any local starred or clean caches had existed, stars would also show up in the local cache table.

The lists are ordered by collection time (most recent first), but being either starred or cleaned is being prioritized above collection time, due to the assumption that a user would be more interested in a cache that is guaranteed to not be missing any elements (starred cache), or secondly a cache that does not need to undergo PTI transformation.

22 The provider ID is largely unused and is not part of the design, nor necessary for a DHT implementation to function.

Once a user has settled on a cache, she may click on it after which the cache is fetched from the peer and presented to the user in a new tab running a *Show Cache* script. The Show Cache script draws the document by replacing itself with the document. An example can be seen in Figure 8 of the StackOverflow website. Notice the bottom bar that has been appended, as well as how some graphical elements are lacking due to the fact that StackOverflow's CDNs lack CORS support as was demonstrated in Section 6.4.2.

A simplified version of the implementation of the Show Cache script can be seen in Listing 6. To correctly depict a cache, three things have to be done in the script before presenting the cache to the user:

1. Some websites may add `integrity` tags containing hash values of resource inclusions, e.g. `<link>` tags including stylesheets. In order to include the resource, the browser first verifies that the hash value corresponds to the retrieved resource. If the hashes differ, the resource will not be included. When converting resources to base64-encoded strings to inline them, the hash values will change. The `integrity` tag's value will therefore no longer match the resource, and the browser will refuse to include it.

   ```
   Subresource Integrity: The resource 'data:text/css;base64,LyohIG5vc...'
        has an integrity attribute, but the resource requires the request to
        be CORS enabled to check the integrity, and it is not. The resource
        has been blocked because the integrity cannot be enforced.
   ```

   The `integrity` tag therefore has to be removed from all `<link>` tags (line 3–6). The tag cannot be removed during cache collection for obvious security reasons.

Listing 6: Show Cache script

```
1  var unsafeDocument = document.createElement('html');
2  unsafeDocument.innerHTML = request.payload.document;
3  var links = unsafeDocument.getElementsByTagName('link');
4  for (var i=0, l=links.length; i < l; i++) {
5    links[i].removeAttribute("integrity");
6  }
7
8  document.open("text/html");
9  document.write(unsafeDocument.outerHTML);
10 document.write('<div style="position:fixed;bottom:0;left:0;right:0;height:20px;
       padding:5px;background:#000;color:#fff">' +
11   'You are viewing a cached version of ' + cache.url + ' from ' + new Date(
         cache.time * 1000) + '.</div>');
12 document.close();
13
14 var resetStyle = 'html, body, h1, h2, h3, h4, h5, h6, p, table { font-size:
       unset; font-family: unset; vertical-align: unset; line-height: unset; } h1
        { font-size: 2em; } h2 { font-size: 1.5em; } h3 { font-size: 1.17em; } h4
        { font-size: 1.12em; } h5 { font-size: .83em; } h6 { font-size: .75em; }
       body { margin-bottom: 50px !important; }';
15 style = document.createElement('style');
16 style.type = 'text/css';
17 style.appendChild(document.createTextNode(resetStyle));
18 document.head.insertBefore(style, document.head.firstChild);
```

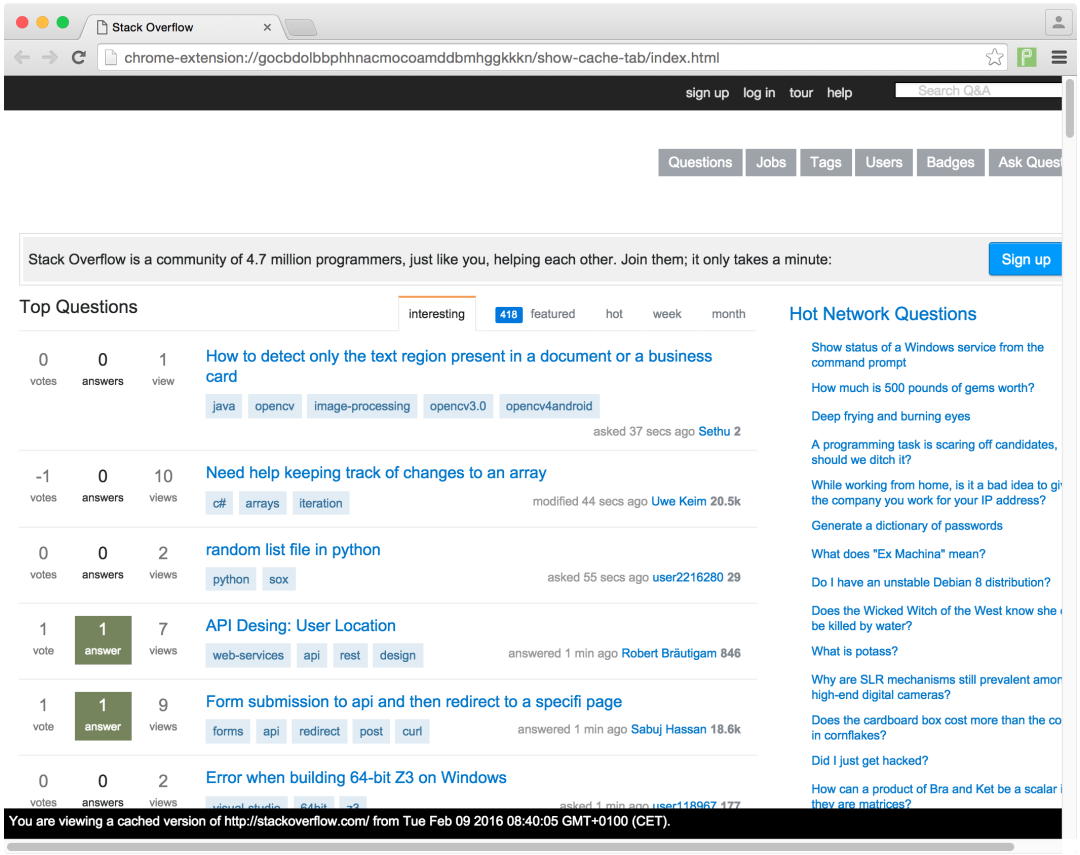


Figure 8: Cache Viewer tab showing StackOverflow.

2. Due to inconsistencies in `document.write`, the order of the write expressions is very important (line 9–10). If we write the `<div>` first, it will be appended to the `<body>` tag. When we then afterwards write the entire document, everything will be appended to the body, including the `<head>` tag of our document, which is clearly not what we want. When we write it the other way around, i.e. starting by writing `<html>`, the current `<html>` tag will be overwritten.

3. Chrome extensions come with some pre-defined styles, e.g. changing default font and size in the body. To avoid having these styles interfere with the layout of the cached document, we have to inject some CSS to cancel it out (line 14–18). The Chrome defaults cannot be disabled.

On the screenshot in Figure 8, a P icon can be seen in the top right corner. Pressing the icon will bring up a pop-up menu, showing some statistics and meta data to the user, as well as letting the user star the recently collected cache or view other caches of the website. The pop-up can be seen in Figure 9.

Cache serving through the extension has now been covered, leaving cache serving from the web frontend. The web frontend is very

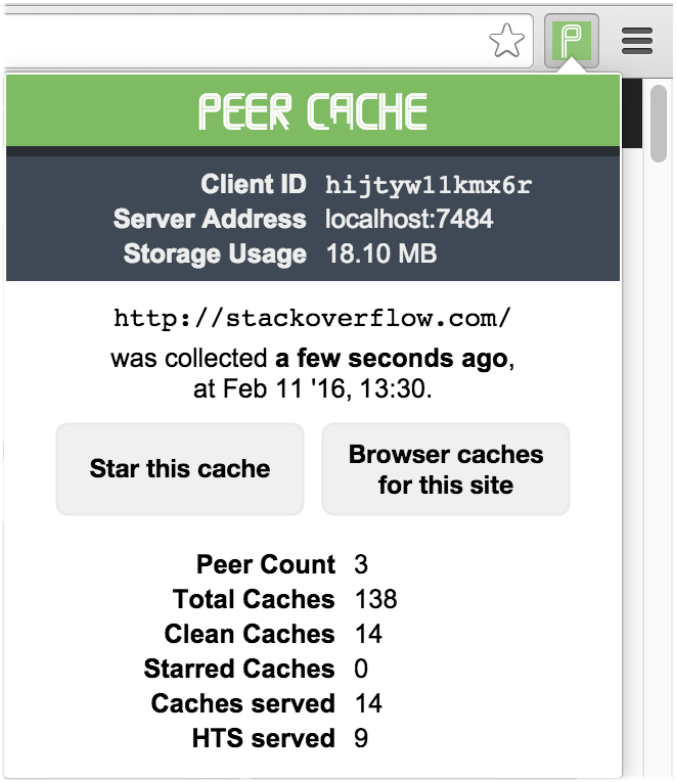


Figure 9: Pop-up window.

similar to the extension, except for the fact that navigation errors cannot be intercepted, and the user will not have a pop-up menu at their disposal. Instead, users of the web frontend will have to navigate to the web frontend's web server in their browser and then input the address of the website they wish to see caches for. This will present them with a Cache Selector very similar what is used in the extension.

6.6 TOKEN CALLBACK TABLE

Peer Cache is based on WebSockets, as previously mentioned. WebSockets allow developers to read and write to a stream, much like regular POSIX sockets, but without having to worry about message delimiters. However, WebSockets are still a lot more primitive than other protocols such as Remote Procedure Calls. A request, for instance, does not always dictate a response, making it hard to identify whether the incoming data is a response to the request that was just sent, or to an older request that still has not gotten a response.

If, for instance, two requests A_{req} and B_{req} are sent to the indexing server (or a peer) simultaneously, it may be difficult to identify the responses A_{res} and B_{res} and pair them correctly with the sender. With no protection in place, the request-response cycles might interleave, so A_{req} gets sent off to a server, then B_{req} gets sent off, but calculating the B process's response may be faster than calculating the A

process's response, resulting in the A process receiving B_{res}, which was meant for process B, and afterwards B receiving A_{res}, meant for process A.

JavaScript is well-known for its non-blocking asynchronous nature and heavy use of callbacks, which also carries over to Chrome's app and extension framework. Most browser interaction happens using event listeners (callbacks), and likewise Chrome's Storage Areas are all asynchronous. As a result, Peer Cache also heavily relies on callbacks, and the Peer Helper class is no different.

Every request that requires a response sent using the Peer Helper class it passed on along with a callback for the Peer Helper to run once the request has been resolved. In order to keep track of all these callback – and prevent mix-ups like seen with process A and B from before – the requests contain a token (or cookie). The token is a random string that gets attached to the payload (message) that is being sent. Before sending the request, Peer Helper adds the token to a table along with the callback. It is then the receiving end's responsibility to include the token in the response. When the response comes back, the Peer Helper looks up the token in the table and passes the response to the callback. A simplified version of the implementation can be seen in Listing 7.

Listing 7: Token Callback Table.

```
var tokenCallbackTable = {};
var server = new WebSocket("ws://" + INDEXING_SERVER_ADDR);
server.onmessage = messageHandler;

function messageHandler(event) {
  var message = JSON.parse(event.data);
  tokenCallbackTable[message.token](message.payload);
}

function sendMessage(socket, type, payload, callback) {
  var msg = {
    type: type,
    payload: payload
  };
  if (callback) {
    msg.token = Math.random().toString(36).substring(2);
    tokenCallbackTable[msg.token] = callback;
  }
  socket.send(JSON.stringify(msg));
}

function getCache(url, time, peerAddr, callback) {
  var peer = new WebSocket("ws://" + peerAddr);
  peer.onmessage = messageHandler;
  peer.onopen = function() {
    sendMessage(peer, "GET_CACHE", {
      url: url,
      time: time
    }, callback);
  }
}
```

6.7 PRIVATE TREE INTERSECTION

Two slightly different PTI implementations have been made, one potentially creating more "rich" intersections by being more relaxed in terms of sibling order. The relaxed algorithm is, however, also slightly slower than the strict algorithm.

The strict algorithm can be modelled by the recurrence relation $T(n) = mT(\frac{n}{m}) + O(1)$ where n is the number of elements in the source tree (not only leaf nodes) and m is the maximum "width" of the tree, i.e. the maximum number of siblings. Solving the recurrence relation yields a complexity of $O(n)$.

The relaxed algorithm can be modelled by $T(n) = mT(\frac{n}{m}) + O(m^2)$, also yielding an (expected) runtime complexity of $O(n)$. However, when m is very big, the algorithm becomes slow. Particularly, when $n = m$, $O(m^2)$ in the recurrence will dominate the recursion, giving us a worst-case time complexity of $\Theta(n^2)$. Seeing as HTML documents rather often can be very wide (nodes having many children), the complexity in practice is very likely to approach $O(n^2)$.

6.7.1 *Strict PTI Implementation*

Explaining how the strict PTI algorithm works may be easiest with an example. Imagine two users, Carol and Dave, visiting their home banking system. Since they are using the same bank, the website is almost identical to them, except for a header `<h1>` greeting them by their name and a paragraph `<p>` containing their account balance, as can be seen in Figure 10.

```
<html>
  <head>
    <title>Home Banking</title>
    <link href="style.css"
          type="text/css">
  </head>
  <body>
    <div id="top">
      <h1>Welcome to Home Banking,
        <span id="name">Carol</span>
      </h1>
    </div>
    <div id="main">
      <p>Your account balance is:</p>
      <p>250 USD</p>
    </div>
  </body>
</html>
```

(a) Carol's home banking source code.

```
<html>
  <head>
    <title>Home Banking</title>
    <link href="style.css"
          type="text/css">
  </head>
  <body>
    <div id="top">
      <h1>Welcome to Home Banking,
        <span id="name">Dave</span>
      </h1>
    </div>
    <div id="main">
      <p>Your account balance is:</p>
      <p>100 USD</p>
    </div>
  </body>
</html>
```

(b) Dave's home banking source code.

Figure 10: Source code for Dave and Carol's home banking front page.

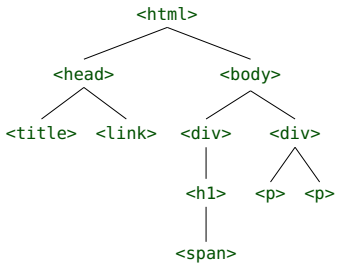


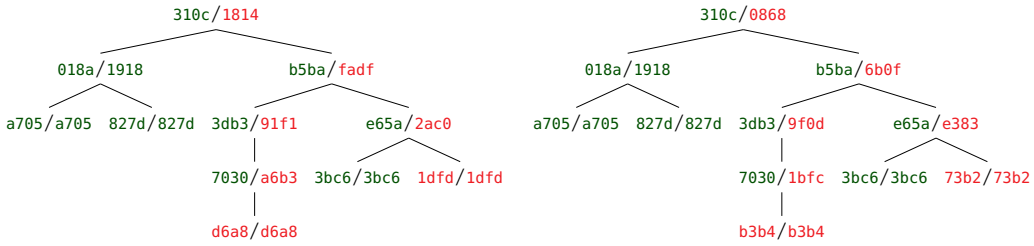
Figure 11: The home banking website as a hash tree structure.

We can consider each website source code as a tree (Figure 11). Note that in this particular figure, there is only one tree, because the trees for Carol and Dave's home banking front page happen to be identical. This is, however, not necessary for the Private Tree Intersection algorithm to work, but it does simplifies the example. If we recursively traverse the source code tree, replacing every element's contents with a structure containing a hash of the element itself without child nodes (a shallow hash) and a hash of the element including all child nodes (a deep hash), we will end up with a hash structure without any sensitive data, assuming our hashing function is cryptographically secure and the input is of considerable entropy.

When applying the recursive traversal to the tree generated from Dave's visit to the home banking system (Figure 10 (b)), we will get a new hash tree structure (HTS) as seen in Figure 12 (b).

For instance, the paragraph `<p>Your account balance is:</p>` on line 14 will be replaced with a structure containing `3bc6` as both the shallow and deep hash. The deep hash and shallow hash are here identical, because the element does not have any children. Particularly, this holds true for any leaf nodes in the tree. Intuitively, you could argue that the element itself is shallow, so any attempt at computing a deep hash will result in a shallow hash, nevertheless. Likewise, the element `<p>100 USD</p>` in Dave's structure has `73b2` as both shallow hash and deep hash. If we look at the containing element `<div id="main">...</div>` spanning line 13–16, however, the hashes will differ, having `e65a` and `c17d` for the shallow hash and deep hash, respectively.

Carol's tree structure will differ slightly from Dave's, just as the home banking front page shown to Carol differs slightly from Dave's. The hashes for `<p>Your account balance is:</p>` will be the same in both trees, but the second paragraph `<p>250 USD</p>` and `<p>100 USD</p>` will hash to `1dfd` and `73b2`, respectively, for both the deep and shallow hashes. Because the paragraphs in the `<div>` on line 13–16 differ between the trees – namely the second paragraph – the deep hash will also differ here, becoming `2ac0` in Carol's tree and `e383` in Dave's. Since the containing `<div>` element (if we disregard the children) remain the same, the shallow hashes will also be the same in both trees



(a) Carol's hash structure.

(b) Dave's hash structure.

Figure 12: The home banking website as hash structures. The first value is the shallow hash, the second value is the deep hash.

(`e65a`). Continuing this process for the rest of the trees, will yield structures similar to what can be seen in Figure 12. The red hash values in the trees signify the values that differ between the trees. Obviously, any node containing different values will differ, both in regard to deep and shallow hashes. Likewise, deep hashes of parent nodes whose children's values differ, will also differ. A parent node's shallow hash could also be different between trees if the node's text contents differed between the documents. In this example, this is however not the case.

To clean a cache – and make it safe to redistribute – all that is left now is to remove all nodes with differing (red) shallow hashes. The reader may now rightfully wonder what the purpose of the deep hashes is, seeing as how we only look at shallow hashes to produce a clean document. The deep hashes simply allow us to determine whether two trees differ quicker. Looking at just the deep hashes of the `<head>` elements (the left subtree in our trees), we can quickly declare the entire subtree clean, because the deep hashes match across the trees. Conversely, when we can see that two deep hashes differ, we have to continue the traversal. The optimization can also be seen on line 4 in Listing 10 on the next page, showing the implementation of the `findIntersectionFast` function. The deep hashes do not improve upon the expected runtime complexity of $O(n)$, but it does give us a best-case runtime of $O(1)$, assuming both trees are identical.

Listing 10: Strict PTI implementation for finding intersections.

```
function findIntersectionFast(treeA, treeB) {
  // If the trees are identical, the intersection will be either of the trees, so we
  // just return treeA.
  if (treeA.deepHash === treeB.deepHash) {
    return treeA;
  }

  // If the trees don't share shallow hashes, we won't know what to do with an
  // intersection of the children anyway, so we just declare that there's no
  // intersection.
  if (treeA.shallowHash !== treeB.shallowHash) {
    return false;
  }

  // If treeA doesn't have any children, we may be able to return the root element
  // itself as an intersection, assuming the root element is the same (i.e. the
  // shallowHashes are the same).
  if (treeA.children.length === 0) {
    if (treeA.shallowHash === treeB.shallowHash) {
      return treeA;
    }
    // If the shallow hashes differ and treeA has no children, there can't be any
    // intersection at all.
    return false;
  }

  for (var i=0, l=treeA.children.length; i < l; ++i) {
    // If the child of treeA doesn't exist in treeB, it clearly can't be part of
    // the intersection.
    if (!treeB.children[i]) {
      treeA.children[i].dirty = true;
      continue;
    }

    // If the hashes of the two elements are identical, it will be part of the
    // intersection and we therefore won't
    // have to mark the child as dirty, so we end the iteration.
    if (treeA.children[i].deepHash === treeB.children[i].deepHash) {
      continue;
    }

    // If the root element of the child however doesn't match, the element
    // clearly can't intersect at all. Even if treeA and treeB would have common
    // children, we wouldn't know how to organize them when the node they belong
    // to aren't the same.
    if (treeA.children[i].shallowHash !== treeB.children[i].shallowHash) {
      treeA.children[i].dirty = true;
      continue;
    }

    // Now the recursion starts. If there is an intersection, let's keep it.
    var intersectionTree = findIntersectionFast(treeA.children[i], treeB.children[i]);
    if (intersectionTree) {
        treeA.children[i] = intersectionTree;
        continue;
    }
    // Otherwise, mark it as dirty.
     treeA.children[i].dirty = true;
  }

  return treeA;
};
```

6.7.2 *Relaxed PTI Implementation*

The relaxed implementation is much like the strict implementation, except for the fact that the order of siblings is irrelevant. The rationale behind the implementation is that two users may have two identical trees, except for one tree having an additional child. If the extra child is the first among the siblings, none of the siblings will match its sibling-counterpart in the other tree due to the offset.

A DOM tree may be thought of as a recursive box structure, containing a list of boxes (children). By looking at box column X and Y in Figure 13, the problem with the strict algorithm immediately becomes clear. A good intersection of X and Y would be to keep A, B and C, but because D is at the top of column Y, the strict algorithm will compare A_X and D_Y, then B_X and A_Y, and so forth, yielding an empty intersection.

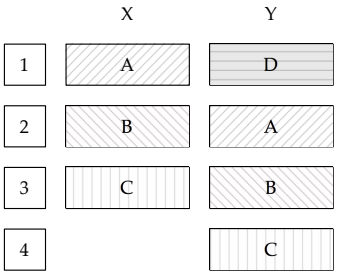


Figure 13: Two slightly different box columns.

The relaxed algorithm compares every child (box) to each other in order to find an intersection. The essence of the change can be seen in the loop starting on line 29 of the relaxed algorithm code seen in Listing 10.

Listing 11: Relaxed PTI implementation for finding intersections.

```
1  function findIntersectionSlow(treeA, treeB) {
2    // If the trees are identical, the intersection will be either of the trees, so we
3    // just return treeA.
4    if (treeA.deepHash === treeB.deepHash) {
5      return treeA;
6    }
7
8    // If the trees don't share shallow hashes, we won't know what to do with an
9    // intersection of the children anyway, so we just declare that there's no
10   // intersection.
11   if (treeA.shallowHash !== treeB.shallowHash) {
12     return false;
13   }
14
15   // If treeA doesn't have any children, we may be able to return the root element
16   // itself as an intersection, assuming the root element is the same (i.e. the
17   // shallowHashes are the same).
18   if (treeA.children.length === 0) {
19     if (treeA.shallowHash === treeB.shallowHash) {
20       return treeA;
21     }
22     // If the shallow hashes differ and treeA has no children, there can't be any
23     // intersection at all.
24     return false;
25   }
26
27   for (var i=0, l=treeA.children.length; i < l; ++i) {
28     treeA.children[i].dirty = true;
29     for (var j=0, k=treeB.children.length; j < k; ++j) {
30       // We do not want multiple children of treeA matching the same child of treeB,
31       // so if the child has already been matched, we ignore it.
32       if (treeB.children[j].matched) {
33         continue;
34       }
35
36       // If the hashes of the two elements are identical, it will be part of the
37       // intersection and we therefore won't have to mark the child as dirty, so we
38       // end the iteration.
39       if (treeA.children[i].deepHash === treeB.children[j].deepHash) {
40         treeA.children[i].dirty = false;
41         treeB.children[j].matched = true;
42         break;
43       }
44
45       // Now the recursion starts. If there is an intersection, let's keep it.
46       var intersectionTree = findIntersectionSlow(treeA.children[i], treeB.children[j]);
47       if (intersectionTree) {
48         treeA.children[i] = intersectionTree;
49         treeA.children[i].dirty = false;
50         treeB.children[j].matched = true;
51         break;
52       }
53       treeA.children[i].dirty = true;
54     }
55   }
56
57   return treeA;
58 };
```

6.8 SUMMARY

In this chapter, the Peer Cache implementation has been presented.

The implementation deviates from the design by depending on an indexing server rather than a DHT, but is otherwise a lot like the proposed design.

The implementation includes its own cache storage, as Chrome does not make any useful caching APIs available to developers. The cache storage implemented is utilizing compression to save place and speed up cache transfers. Cache collection happens through a content script, embedding stylesheets and images into documents, even though not all resources can be embedded due to CORS issues. Embedding JavaScript into the caches is not possible at all due to Chrome's CSP. Two PTI implementations have been made: a strict algorithm and a relaxed algorithm. The strict algorithm runs faster than the relaxed algorithm, but does not create as rich documents.

7 EVALUATION AND DISCUSSION

In this chapter, the prototype implementation is evaluated according to the hypotheses presented in Section 1.1.1. Each hypothesis is covered in the following sections.

The first hypothesis specifically claims that a system such as Peer Cache can supply both efficiency and availability on par with current client-server systems. The system's efficiency is examined in Section 7.3 on Performance and Obtrusion. In regard to whether the system can provide as much availability as current client-server systems, this has not been tested through the prototype implementation. However, the sufficient availability of properly designed peer-to-peer systems has been proven time and time again with other networks (such as Bit-Torrent [7][1]), assuming enough peers are using the network.

Availability of the *network* itself is not the only important aspect of availability in a peer-to-peer network. With Peer Cache, for instance, whether a specific website is available is equally important – what good is a well-functioning network if it provides no content? Testing content availability by using just a few peers – as would be the capabilities for a project of this scope – would not provide much insight into what availability a highly populated network may provide. As with other peer-to-peer networks, they generally tend to function better with a bigger population. BitTorrent, for instance, would have a poor availability of torrent content if there were just a handful of peers. This is no different for Peer Cache – with more peers, more websites is made available and thus provide higher availability for those specific websites. Specific tests on website availability have therefore not been performed, but it can reasonably be assumed that if Peer Cache became widely used, the network could provide similar available to The Internet Archive. Admittedly, The Internet Archive would have a leg up over Peer Cache, seeing as content that has already been taken down will not make its way into the Peer Cache system, except, of course, if the Peer Cache users visit The Internet Archive's caches. However, this would be the case for any web preservation initiatives created, peer-to-peer-based or otherwise.

The second hypothesis claims that user-collected caches can be securely rid of personal information, which also seems to hold true to some degree. This is examined in Section 7.1 on Security.

1 The paper mentions that BitTorrent's tracker system can cause low availability. This is somewhat irrelevant, though, since the tracker issue can be remedied (as explained in the paper), but more importantly, the designed Peer Cache system does not use trackers or similar mechanisms at all. Thus, the point about high availability stands.

Hypothesis 3 states that sharing can be possible without becoming obtrusive to the cache provider, which is further examined in Section 7.3, covering Performance and Obtrusion.

Lastly, the fourth hypothesis states that the cache consumers can access caches without having any extensions or third-party applications installed. Since this is the job of the web frontend in our implementation, this will appropriately be evaluated in Section 7.4 on the web frontend.

7.1 SECURITY

A consistent concern throughout the analysis, design phase and implementation, has been whether we could be certain that the system sufficiently anonymizes and sanitizes data (cleans documents). While we are no closer to a theoretical answer to the question, the implementation does give us some closure. Theoretically, it is unlikely that a system such as Peer Cache could ever be proven secure due to the somewhat random nature of deciding whether a cache is clean or not. By selecting a random peer on the network and comparing document contents, there is no guarantee that the particular selected peer may not share the same private data as the selecting peer, resulting in a "clean" document still containing sensitive data.

While the testing of the implementation does not appear to have disclosed any private information, this proves nothing more than just that: The tested websites with our specific testing conditions does not appear to have leaked any information. This cannot be generalized to all networks or all browsing conditions whatsoever. In fact, examples can rather easily be thought up where the current implementation will be able to leak *something*.

Consider an example where a husband and wife are visiting their shared banking website at approximately the same time, and a user requests a cache of the banking website from the husband. The husband's Peer Cache instance will then find other caches of the same website of users that has visited it at approximately the same time as the husband himself. In this case, it is quite likely that the wife's cache will be selected, causing their shared bank information to become be part of the intersection and thus get leaked.

This problem could possibly be solved by making the cache selection random. Currently, the cache is selected based on recency of collection, due to the assumption that two caches collected closely together in time are more likely to be similar, so more information is preserved. However, even by selecting a completely random cache, we may still end up comparing the husband and wife's cache, or two of the husband's own caches from different computers. Even if we expanded the cache cleaning to consider more than just two caches when computing the intersection, we may still end up in the unlikely

scenario where we exclusively select and compare caches of people that have a shared secret.

Most systems that are considered secure can be broken with an incredibly lucky "guess", for instance by guessing a user's password, or more unlikely, guessing a user's private key in a public-private key system. In order to be secure, security enthusiasts and cryptographers therefore advocate choosing strong passwords and keys with high information entropy. When the risk of having a malicious user break a system by "guessing" has been lowered sufficiently, a system is generally considered secure. Likewise, one can imagine that with enough randomness when selecting caches, and with enough caches involved in the intersection, the probability of accidentally leaking information could also be reduced to a point where it may be considered secure, because the probability of hitting a set of caches, all containing the same shared secret, is just as low as the probability of a user guessing another user's password.

Other than increasing the number of peers involved, or using more randomness in the cache selection process, a completely different approach to cache collection could also be useful: One could imagine a system that magically selected peers with no information in common. If peers had nothing in common – no shared secrets or similar – the resulting cache intersections would indeed be entirely clean. How this magical selection process would work is hard to say, but some approximations could be made. For instance, by making sure that caches compared are collected by users far away from each other geographically by looking at browser localization settings or simply by comparing IP addresses. Peer Cache could profile users into different demographic segments based on browsing habits and exclusively compare caches from different segments. When using an approximation, this could reduce the risk that the selected cache providers share some private information, but it could never completely eliminate it.

Inevitable, the more private data we try to remove from the cache, the more actual non-private data we will also end up removing. For instance, it is easy to imagine an extreme scenario where hundreds or maybe even thousands of caches for a website are compared, leaving nothing but a shell of the website behind for the cache consumer. In this case, the cache will likely not contain any private information, but it will also not be very coherent or useful for the user. Striking a balance between security and usefulness is therefore of vital importance in this matter.

Another issue that may compromise user data is the "star cache" feature, i.e. when a user manually decides that a cache is clean and therefore serves it without using PTI. A user may star a cache that seems clean, but with private information hidden in the source code. Some websites store the user's credentials as hidden fields in an HTML form to avoid using cookies. Sharing such a page may not

appear to contain any private information to a non-technical user, but any person with basic knowledge of HTML will be able to extract the login information from the cache. Removing all hidden fields and data properties could help prevent this, but some information may still slip through.

7.2 COHERENCE AND MEANINGFULNESS

In order for a website to be coherent and meaningful, it must be somewhat contiguous and contain most (if not all) of its original content. To determine whether the caches computed by the PTI implementations are coherent and meaningful, a variety of websites from different categories is examined below. Before delving further into the actual evaluation, let's start by defining two groups of websites, as well as what it means for a cache to be meaningful.

Strictly speaking, a website can either be static or dynamic. A static website will be identical to all users, regardless of who the user is, the time of visit, or the user's language settings. A dynamic website on the other hand has content that may change between visits – new articles may be added on a news site, the user may be presented with only local news or otherwise presented with content tailored for the specific user. These two groups of websites are, however, not fully disjoint, as many dynamic websites have static elements and vice versa.

Visiting a completely static website will always result in the exact same caches being saved by Peer Cache[2]. When cleaning a cache by comparing two identical caches, the resulting cache will therefore be perfect (i.e. retain all information), maintaining complete coherency and meaningfulness.

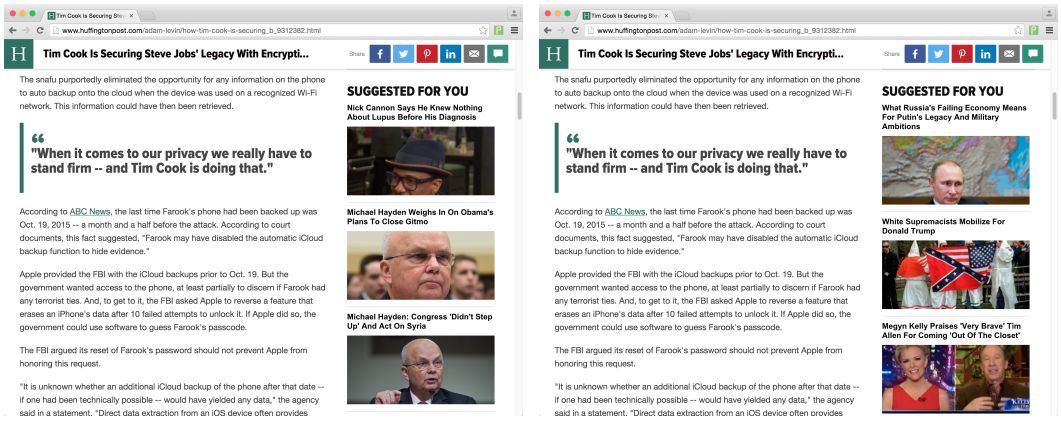


Figure 14: An article on The Huffington Post visited just minutes apart.

2 This is only true when two different peers visit the same, static website. Identical caches will not be preserved by a single peer in order to conserve space.

The diametrical opposite of a completely static website is a completely dynamic website. When computing the intersection of two completely dynamic websites, the intersection will be empty, leaving us with a result that is utterly unmeaningful. Fortunately, websites are rarely entirely dynamic – rather, they contain a combination of dynamic and static content, most of the time allowing us to preserve the main content when computing intersections.

Many websites frame their static content by dynamic content. For instance, a news article may have different ads and links to related news in the sidebar, depending on who is using the site and when. The primary content – the article itself – will largely remain the same, and will therefore remain intact in the cache. See Figure 14 for an example of a static website with dynamic secondary content. In this case, both PTI implementations would preserve the primary content (i.e. the article on the left).

Whether something is meaningful may be a very subjective matter, but working under the assumption that most people accessing caches only are interested in the primary content (e.g. the article itself on a news website), a good definition of meaningfulness is that the cached website must retain the primary content. Conversely, if a website exhibits so dynamic behavior that the PTI implementations are unable to safely preserve the primary content, the caches is declared unmeaningful.

Keep in mind that the PTI implementations use the caches saved by Peer Cache. These caches may not always be complete, so when evaluating the PTI implementations, Peer Cache's cache collection capabilities are implicitly evaluated as well. Figure 15, for example, demonstrates how many graphical elements on Twitter are not saved with the cache. Notice how the Twitter logo itself in the top right corner of the cache is missing, as well as all icons on the site. Rarely are caches perfect, but the missing content is mostly insignificant graphics as with this example.

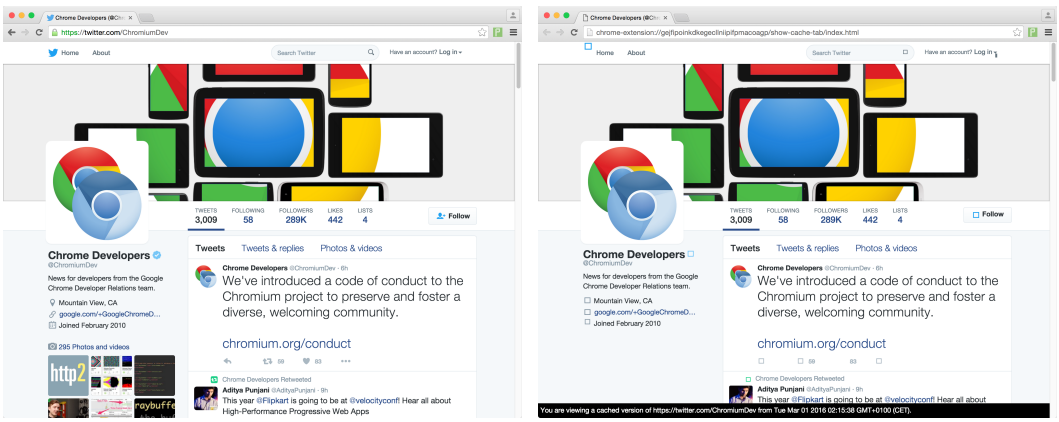


Figure 15: A screenshot of Twitter (left) and the resulting cache (right).

In an attempt to make an evaluation that is as exhaustive as our means allow (while being representative of real-world usages of Peer Cache), a test suite of websites from different categories has been assembled.

The websites in the test suite could have been evaluated by algorithmically calculating the similarity between the source code of websites and their cleaned caches using one of many available algorithms for string similarity measurements (e.g. Levenshtein distance algorithm[3]). However, saying that a cache contains 20% or 80% of the original document will not be a very useful metric, as the majority of a cached document either consist of secondary content (sidebars, etc.) or media (images, stylesheets, etc.). Instead, a less rigorous approach will be used where screenshots of websites and caches are compared manually. The test suite contains the following websites:

TEST SITE. A website created for the purpose of testing and comparing the PTI implementations. Has a combination of dynamic and static content.

WIKIPEDIA. User-driven encyclopaedia. Mostly static content.

STACKOVERFLOW. A question-and-answer website for developers where users subscribe to topics of their choice. Content listings may therefore vary significantly between users.

TWITTER. Social media website. Main content should be static across visitors, but dynamic across time.

THE HUFFINGTON POST. Regular news website. Main content should be static, but sidebar includes dynamic ads and links.

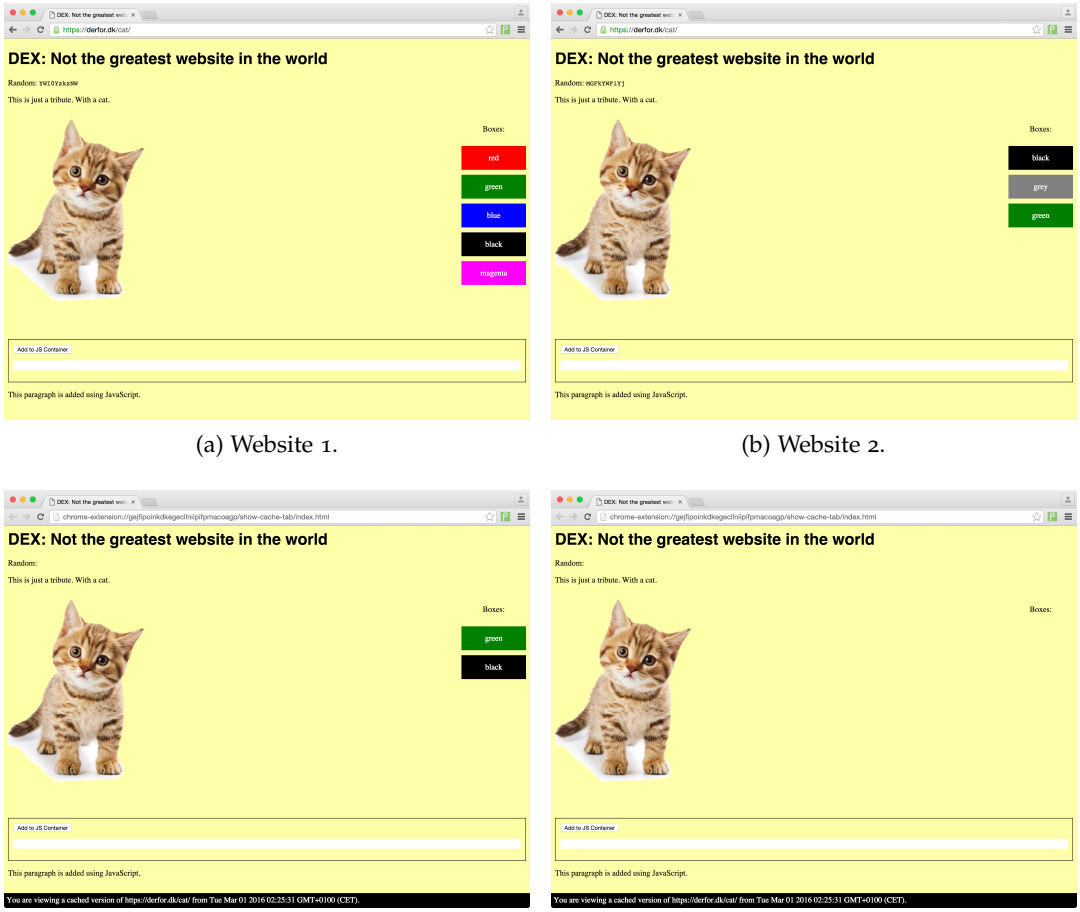
REDDIT. A user-driven news site where users subscribe to news items of their interest. Content listings may therefore vary significantly between users.

GITHUB. Git repository hosting service. Mostly static content.

On the following pages, a series of tile sets will be presented. Each tile set will consist of 4 frames of screenshots: First website visit (top left), second website visit (top right), the cache computed using the relaxed PTI implementation, and lastly the cache computed using the strict PTI implementation.

Each tile set was created using three peers. The two first peers would visit the specific website and screenshots of the websites would be taken. Afterwards, the third peer would attempt to receive the clean document from the peers using both PTI implementations. In order for this to be possible, the extension had to be modified slightly, since visiting a cache once would clean it and update the saved cache, making it impossible for the same caches to be cleaned twice using the different PTI implementations. Therefore, the extension was modified to not save clean caches during the evaluation.

3 https://en.wikipedia.org/wiki/Levenshtein_distance



(a) Website 1.

(b) Website 2.

(c) Relaxed PTI.

(d) Strict PTI.

Figure 16: Evaluation of Test Site.

The test site tile set in Figure 16 depicts a simple website designed specifically to test out the caching system and PTI implementations. The website contains an image, some CSS and a short paragraph added using JavaScript at the bottom. As can be see in the caches, these parts were all preserved through both Peer Cache's caching system and the PTI cleaning process.

Each visit to the website generates a random string as well as a set of random boxes. Together, these two components make up the "private content" of the website – content that must not be compromised.

Both PTI implementations correctly detect the different random strings and remove them from the caches. The strict implementation also removes all boxes as it is unable to detect the overlap (see Section 6.7 for further explanation). The relaxed implementation, on the other hand, sees that there is an intersection between the boxes and preserves the intersecting ones. Note that the resulting order of the boxes comes from Website 1, because the cache in this case has been requested from the peer pertaining to Website 1. The result is definitely meaningful.

(a) Website 1.

(b) Website 2.

(c) Relaxed PTI.

(d) Strict PTI.

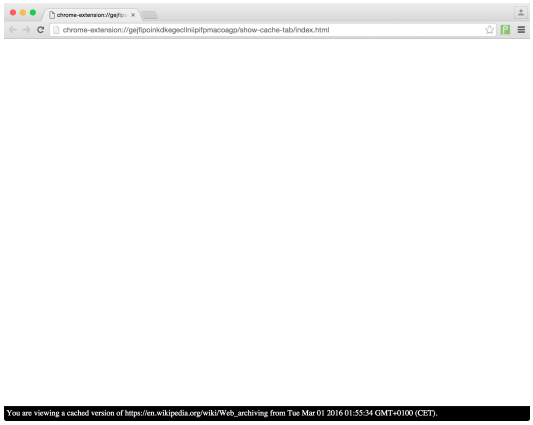
Figure 17: Evaluation of Wikipedia article.

Wikipedia articles are rarely changed (compared to the front page of a news site, for instance), so the caches should contain more or less everything from the original website. As evident by the screenshots in Figure 17, this does seem to be the case. Other than some minor graphical elements missing (not due to the PTI implementation, but the underlying caching system), the website seems to be almost intact. The relaxed PTI seems to have created a perfect cache, whereas the strict PTI is missing the sub-header "From Wikipedia, the free encyclopedia"; arguably not something very important. The result is therefore meaningful.

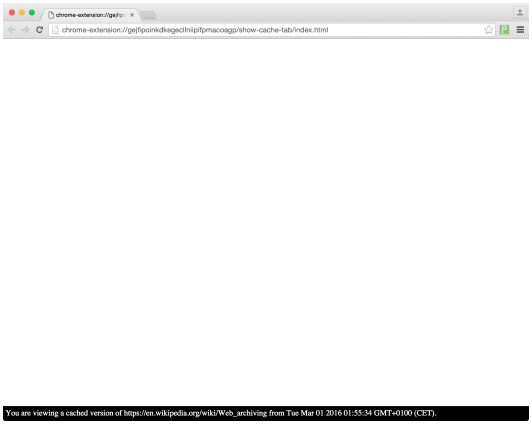
While the two caches compared here were collected very closely together in time, some problems may occur if the articles were modified, even slightly, in-between cache collection. Had a single typo or spelling mistake been corrected in one of the paragraphs, both algorithms would have discarded the entire paragraph. This would admittedly be quite poor, but may be remedied with more advanced PTI algorithms, which is also covered later in the discussion (Section 7.5).

(a) Website 1.

(b) Website 2.



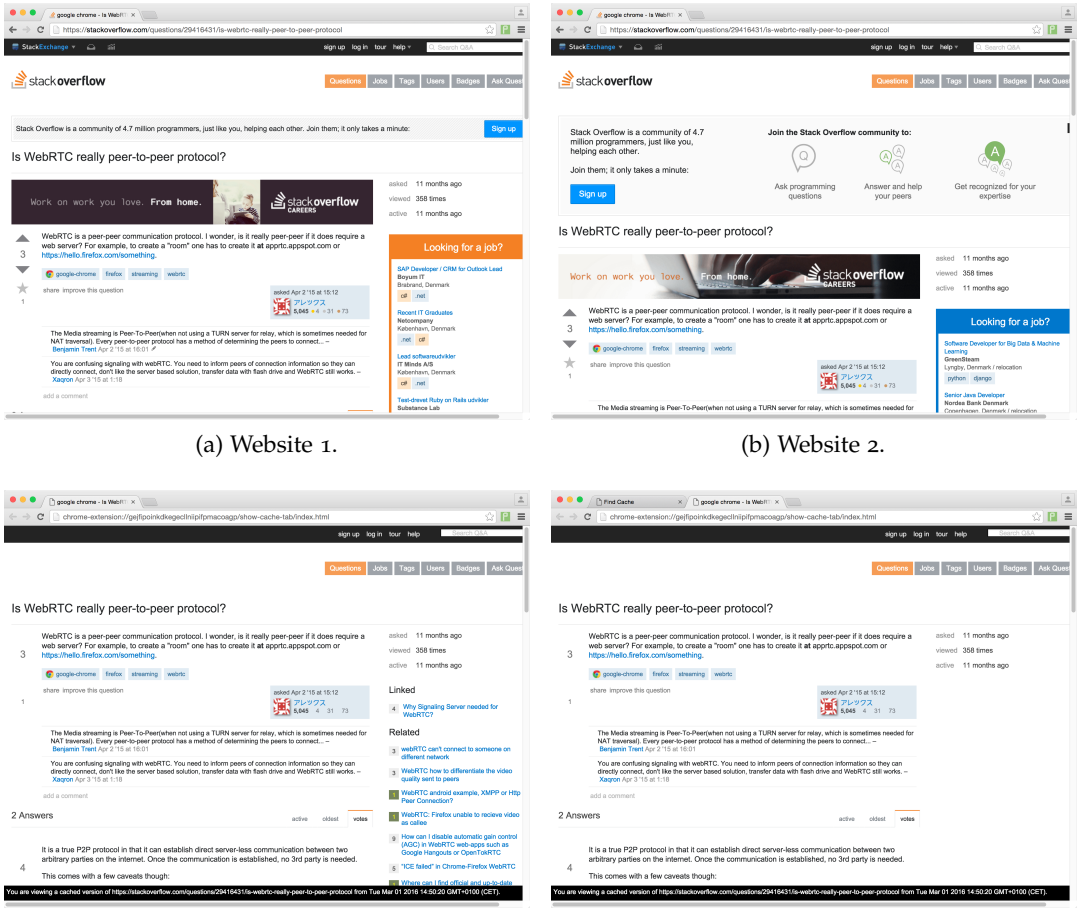
(c) Relaxed PTI.



(d) Strict PTI.

Figure 18: Evaluation of Wikipedia article with one user logged in.

In Figure 18, the previous test is repeated, but this time with one user logged in. In this case, both algorithms fail tremendously. When examining the differences between the source code, the problem seems to come down to the fact that the JavaScript on the site adds either the class `ve-available` or `ve-not-available` to the `<html>` element, thus making the entire document differ. `ve` stands for visual editor – a feature only available to logged in users. This also seems to be something that could be averted with a better PTI implementation that took HTML element attributes into consideration. Alternatively, it could be solved by extending Peer Cache with functionality that would prioritize comparing caches of other logged in users if the user itself was logged in and vice versa. This, of course, also brings forth other security issues. The result is not considered meaningful.



(a) Website 1.

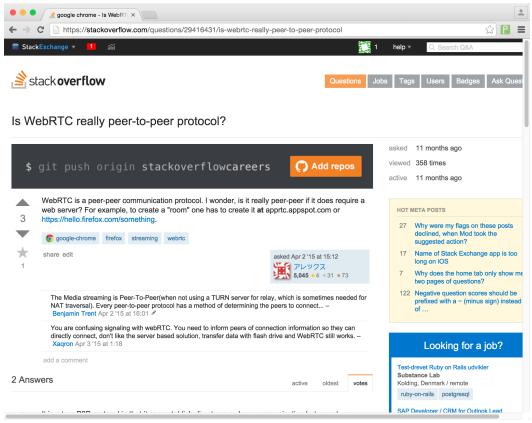
(b) Website 2.

(c) Relaxed PTI.

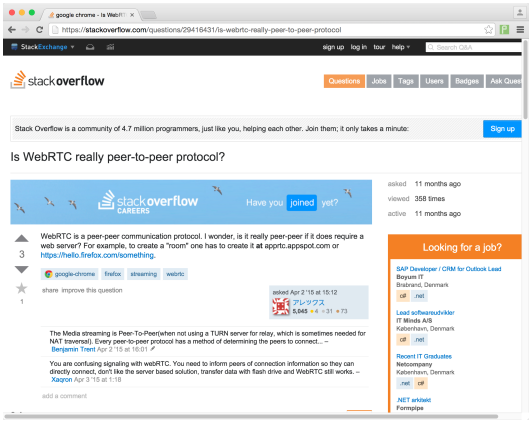
(d) Strict PTI.

Figure 19: Evaluation of StackOverflow question.

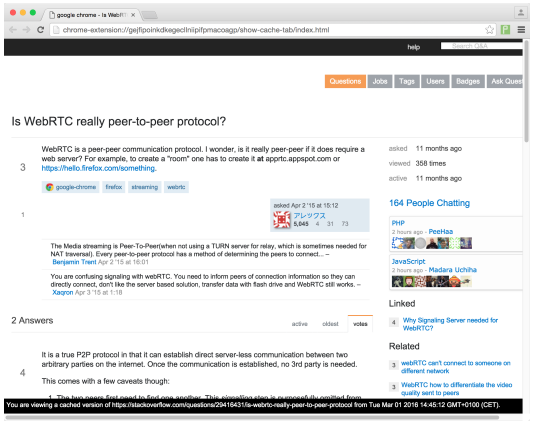
Like with Wikipedia, StackOverflow questions and answers are rarely edited. Additional answers and comments are, however, regularly added. In this case (Figure 19), even though the website visit differ quiet a lot, both PTI algorithms do fairly well. As per usual, some graphical elements are missing due CORS issues during cache collection. The only significant difference between the output of the two algorithms is the fact that the strict algorithm did not manage to preserve the related articles in the sidebar, but since this is secondary content, the cache is considered meaningful according to our previously defined metric, nonetheless.



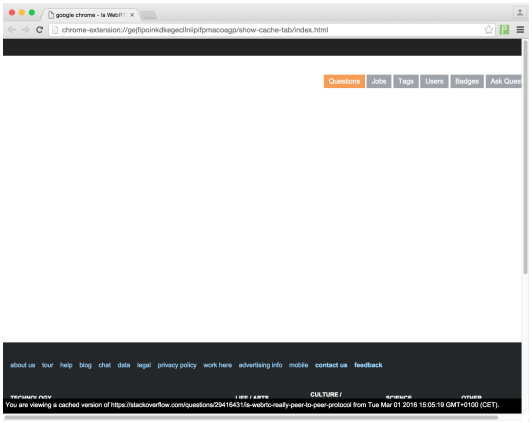
(a) Website 1.



(b) Website 2.



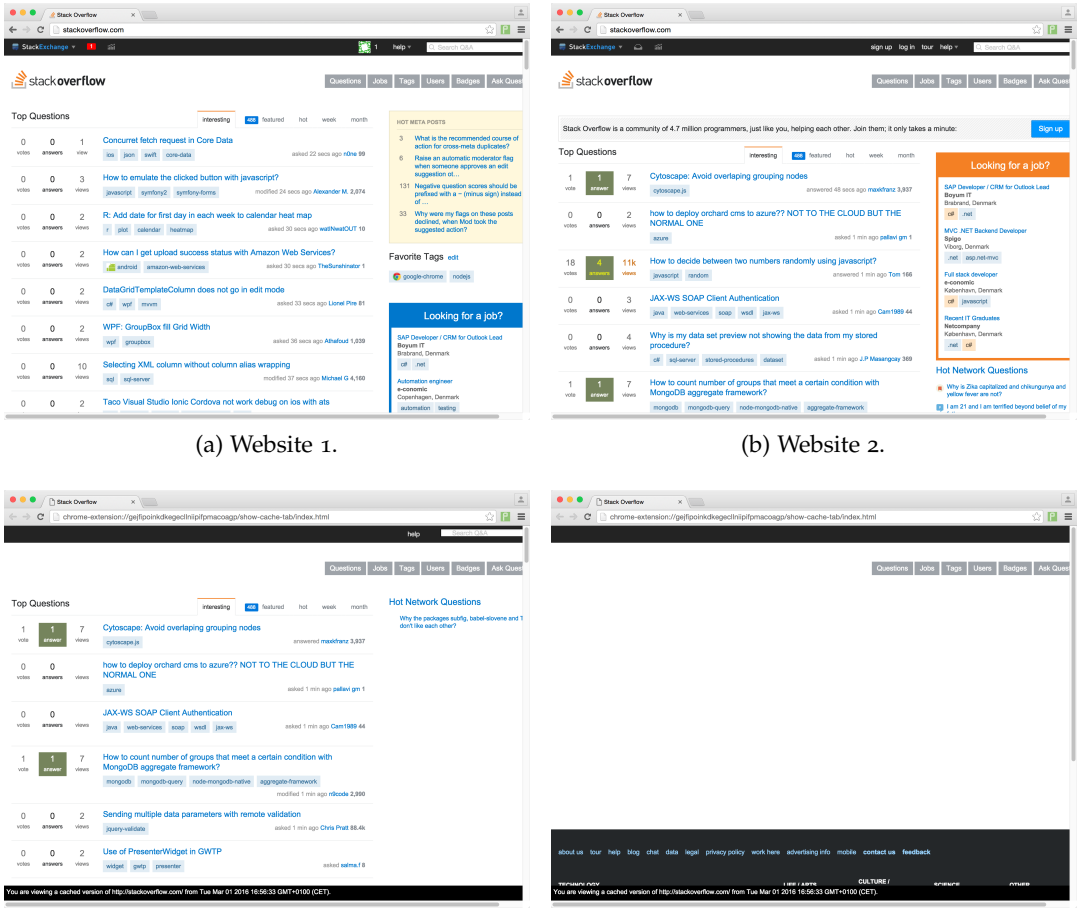
(c) Relaxed PTI.



(d) Strict PTI.

Figure 20: Evaluation of StackOverflow question with one user logged in.

As with our Wikipedia tests, we also consider the case where one user is logged in (Figure 20). The strict algorithm fails due to the gray sign-up block ("Stack Overflow is a community of 4.7 million programmers [...]") preceding the primary content. The relaxed algorithm, however, generates a meaningful cache, containing both right primary and secondary content, if we as per usual disregard the lack of graphical content.



(a) Website 1.

(b) Website 2.

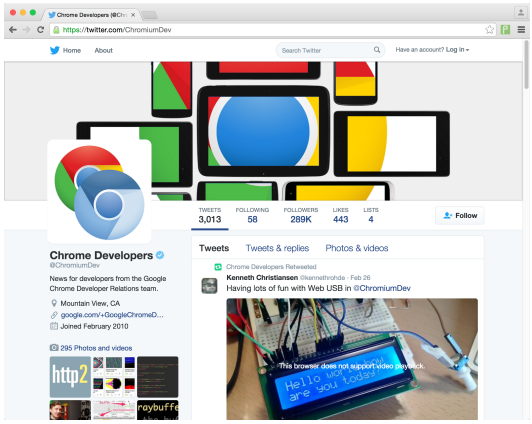
(c) Relaxed PTI.

(d) Strict PTI.

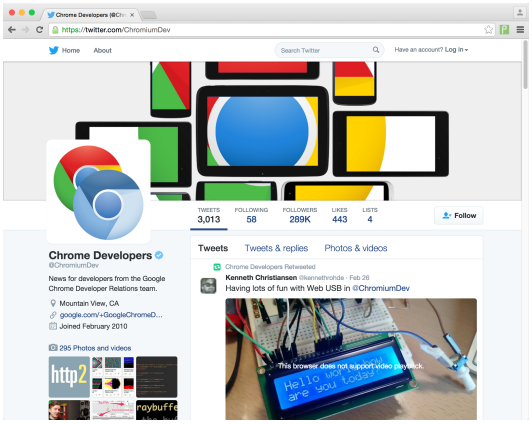
Figure 21: Evaluation of StackOverflow front page.

As the last test of StackOverflow (Figure 21), we will look at the front page instead of a specific question. The front page varies significantly more between users, because a logged in user may subscribe to certain categories. In this example, the user in Website 1 is logged in and subscribed to two categories, making him see slightly different questions. Naturally, the Strict PTI algorithm does not handle this well due to the gray sign-up block from before. But even when this was removed (not evident from the screenshots), the result is still not very useful. The relaxed algorithm on the other hand filters out the questions not shared between the users and over all presents a quite meaningful result.

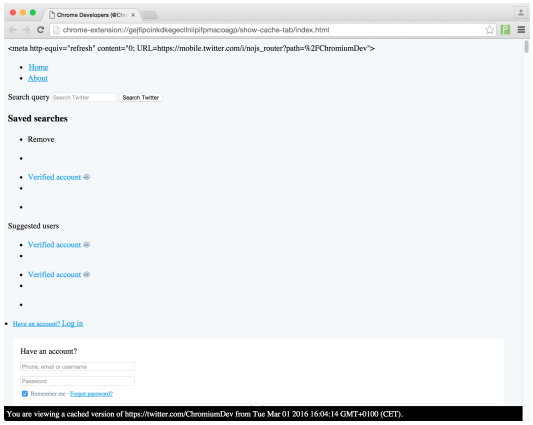
Even if the result was not meaningful, a future user of the system would rarely be interested in seeing which questions happened to be on the front page during the few minutes that the cache was collected in.



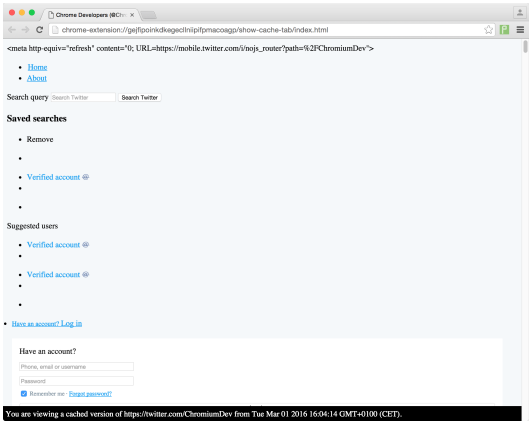
(a) Website 1.



(b) Website 2.



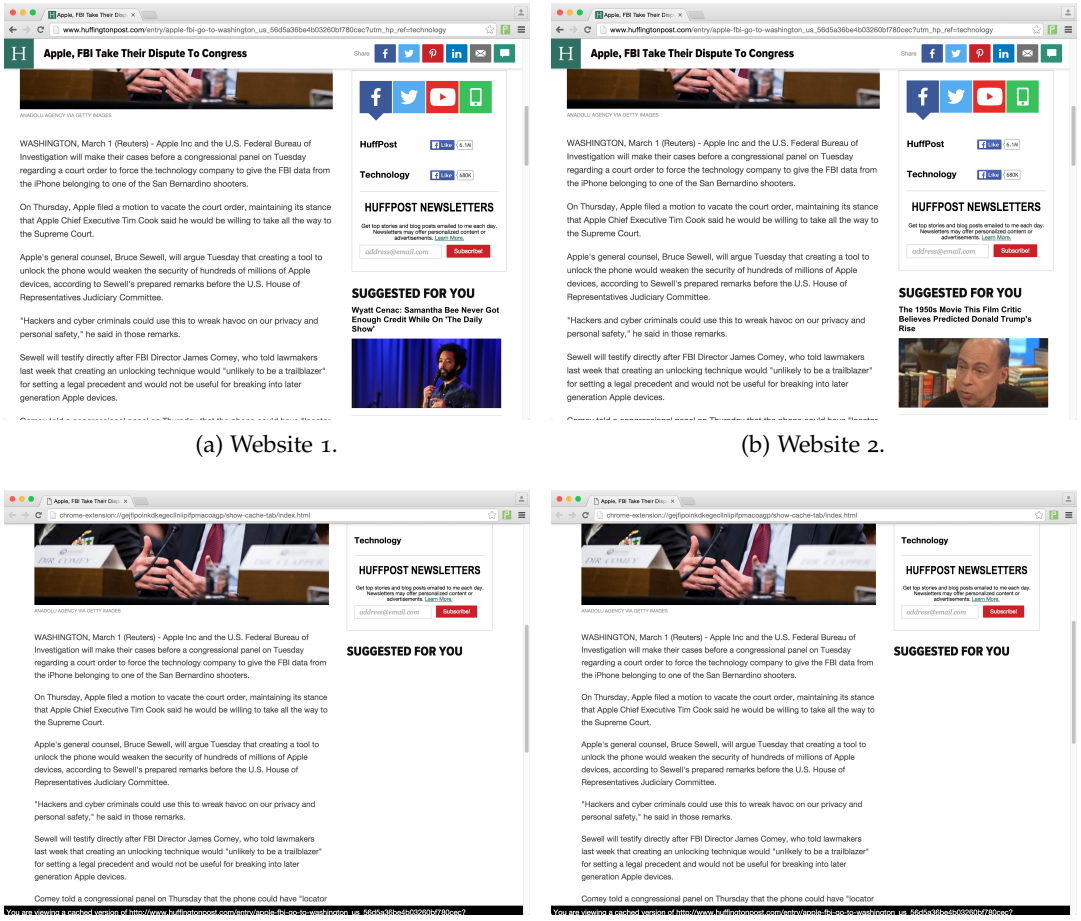
(c) Relaxed PTI.



(d) Strict PTI.

Figure 22: Evaluation of Twitter page.

Evident by the screenshots in Figure 22, neither PTI algorithm creates a good reproduction of Twitter, at least not consistently. Twitter seems to behave randomly, sometimes producing an empty cache (due to an empty `<style>` that intermittently is added to the `<body>` tag), other times producing what appears to be a perfect cache, but most commonly producing what is seen in the figure. When scrolling down on the cached page in the screenshots, the Tweets do show up, but all quiet malformed. The inconsistencies and occasional blank page renders this test case somewhat meaningful, as the primary content for the most part can be deciphered.



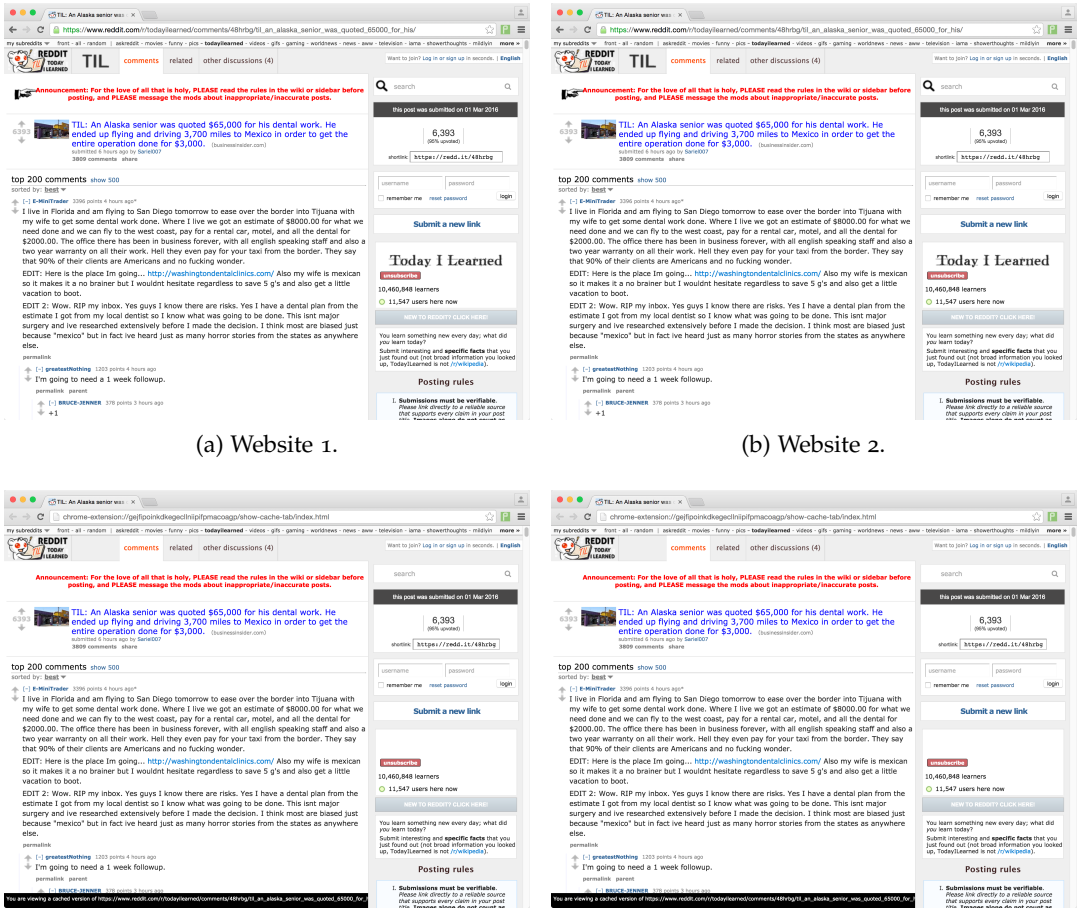
(a) Website 1. (b) Website 2.

(c) Relaxed PTI. (d) Strict PTI.

Figure 23: Evaluation of The Huffington Post article.

In the case of an article on The Huffington Post (Figure 23), the primary content is well-preserved. The secondary content – the suggested links in the sidebar as well the social media functionality and ads – has disappeared. One might expect this to be the result of the PTI algorithms, but it turns out the "Suggested for you" block is not being cached at all. The block consists of an `<iframe>` tag that is being injected into the DOM through JavaScript, seemingly after a delay, causing the page to already have been collected when it finally appears.

Indeed, if the iframe had been injected before the page was being collected, the result would not be much different, because the current implementation of Peer Cache does not cache iframes, but rather just includes them as-is. However, assuming no CSP violations (see Section 6.4), inlining iframes is very likely to be possible.



(a) Website 1.

(b) Website 2.

(c) Relaxed PTI.

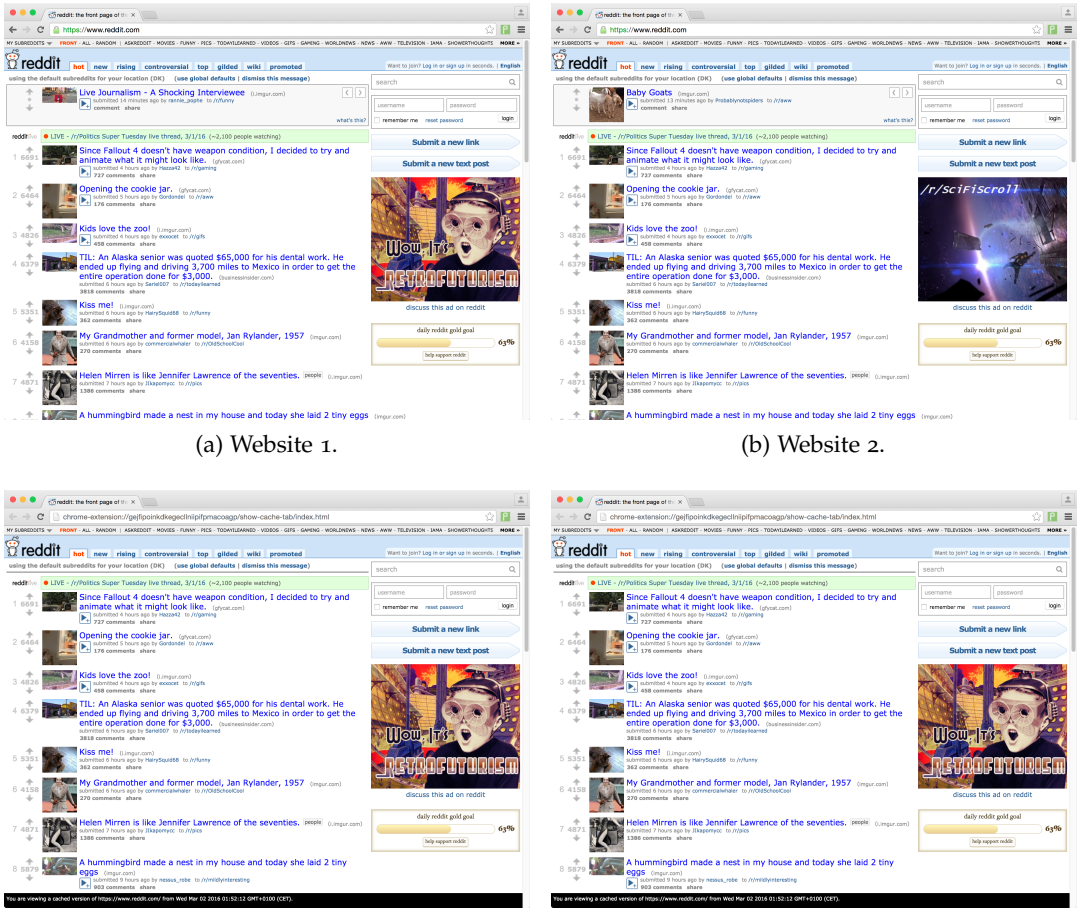
(d) Strict PTI.

Figure 24: Evaluation of Reddit post.

Since Reddit works much like a forum with a posts and comments, both are considered main content. The post is rarely edited, but comments are often added right after the post has been created. After a few days, the pages rarely change.

As can be seen in Figure 24, Reddit pages appear to create perfect caches, if we disregard the lack of some graphical elements. While this test does not demonstrate what happens when a user is logged in, this was also tested: As with Wikipedia, when a user is logged in, the classes of the `<body>` tag differ. Concretely, a logged in user will have the class `loggedin` added to the `<body>` tag, leaving only the `<head>` tag intact in the cache.

It was also tested how the PTI implementations handled pages accessed some time apart where the number of comments differed. In this case, the algorithms behaved as one would expect: The strict algorithm removed a lot of posts due to wrong ordering in the DOM, whereas the relaxed algorithm managed to preserve all unedited comments from the oldest of the caches. As long as both users are either logged in or not logged in, Reddit therefore creates quite meaningful caches.



(a) Website 1.

(b) Website 2.

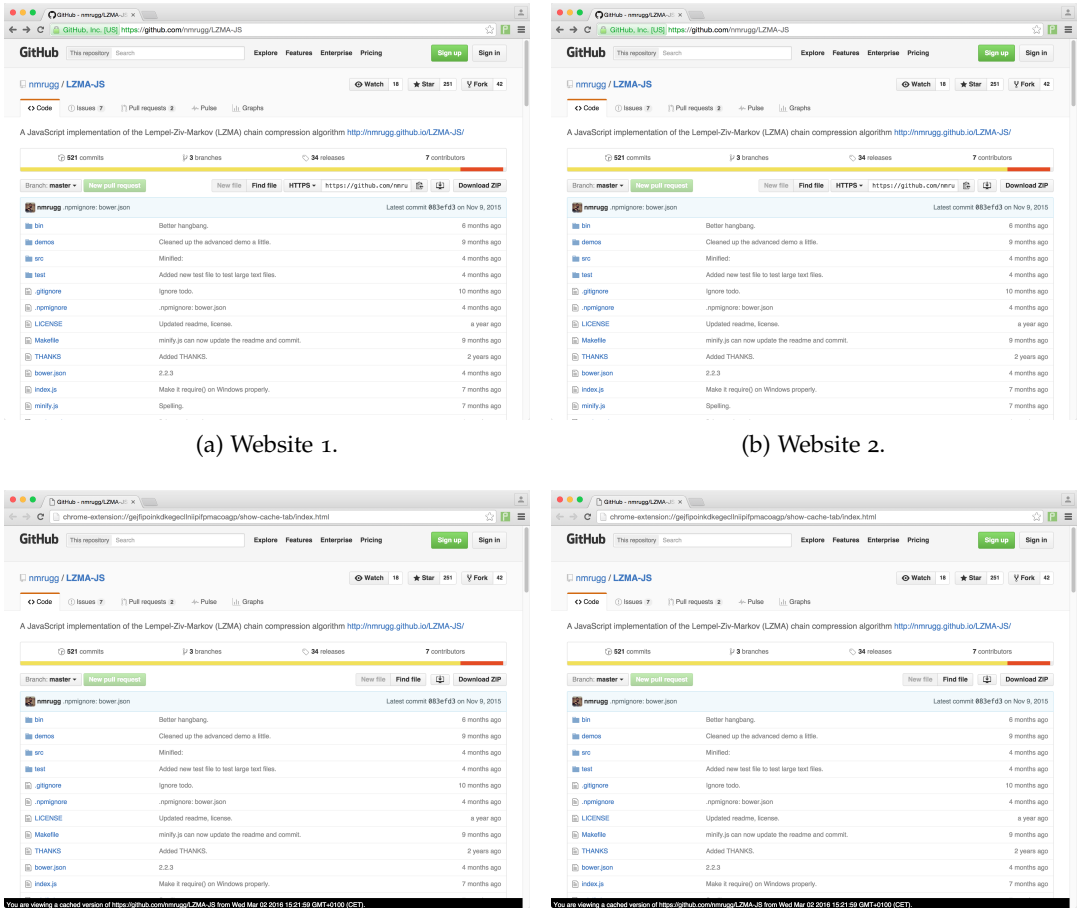
(c) Relaxed PTI.

(d) Strict PTI.

Figure 25: Evaluation of Reddit front page.

The front page of Reddit may (like StackOverflow) vary significantly between users, depending on what categories ("subreddits") they are subscribed to. The strict algorithm does not handle this very well, but the relaxed algorithm does. In Figure 25, the front page for two different (not logged in) users can be seen. The only thing differing here is the top posts ("Live Journalism - A Shocking Interviewee" and "Baby Goats"), which therefore correctly are removed with both algorithms. The rest of the website seems intact. Note that the ad persists in the cache, even though it differs between the users. This is because the ad lives in an `<iframe>`, which (as previously mentioned) is not cached by Peer Cache.

Over all, the resulting caches are meaningful, as long as both users are either logged in or not logged, exactly as with the Reddit post.



(a) Website 1.

(b) Website 2.

(c) Relaxed PTI.

(d) Strict PTI.

Figure 26: Evaluation of The Reddit front page.

Lastly, we have Github. As can be seen on Figure 26, GitHub also creates perfect caches using both PTI implementations, but only when both users are either logged in or logged out. As with both Wikipedia, Reddit and StackOverflow, the trend of having different classes on the `<body>` tag continues. When a user is logged in, the `logged.in` class is add and when the user is logged out, the `logged.out` class is added. The entire body section therefore gets removed. If both users are logged in or logged out, the caches are therefore meaningful.

7.3 PERFORMANCE AND OBTRUSION ON CACHE PROVIDER

This section covers the performance tests that have been conducted on the Peer Cache system as well us how the performance affects or causes obtrusion on the cache provider. The performance tests all measure the time it takes for a specific action to be completed.

First, we examine how long it takes to collect caches on a variety of websites; both how long the content script takes to execute and the underlying background scripts. Based on observations made dur-

ing the development and testing of Peer Cache, it was hypothesized that heavy work performed in background scripts had little to no impact on the user's browsing experience, whereas heavy work taking place in the content script drastically deteriorated the user experience. Since no documentation could be found on the topic, this was instead tested by creating a deliberately inefficient recursive implementation of the Fibonacci function and calling it first in the content script and then in the background script. When calling the Fibonacci function with the argument 42, the computation would in both cases take approximately 3 seconds. When the Fibonacci function was executed in the background script, there were no noticeable changes in performance. However, when the call was made in the content script, the website would freeze up for the entire duration of the execution, making it impossible to interact with the website at all. Therefore, having as thin a content script as possible is desired.

After evaluating cache collection time, the performance of the two PTI implementations is tested and compared. Based on observations from the previous section, it has already been determined that the Relaxed PTI implementation creates better and richer caches, but the better caches come at a price. As explained the PTI section from the Implementation chapter (see Section 6.7), the Relaxed PTI algorithm also has worse worst-case time complexity.

7.3.1 *Cache Collection*

Data was collected using `console.time` and Chrome's Development Tools. The collection times (and websites used in the test) can be seen in Table 3. The times shown in the table is the total time spent by the content script. Because all resources have already been cached by the browser when the content script is executed, there is almost no idle/wait time during the script's execution (observed with `console.profile`). The registered time in the table therefore reasonably represents for how long the page will be frozen.

The background script's execution can be seen in Figure 27. As the figure demonstrates, the background script's execution time increases roughly proportionally to

Site	Time (ms)	Std. dev.
Test Site	51	5%
Wikipedia	249	14%
StackOverflow	647	7%
GitHub	311	14%
Twitter	1254	4%
Imgur	776	10%

All website visits were to the front page of the respective sites. All websites had been cached in the browser before tests lest nework latency became another variable. Each test was performed 10 times and the mean values registered. The standard deviation was also calculated (Std. dev.).

Table 3: Evaluation of cache collection time in content script.

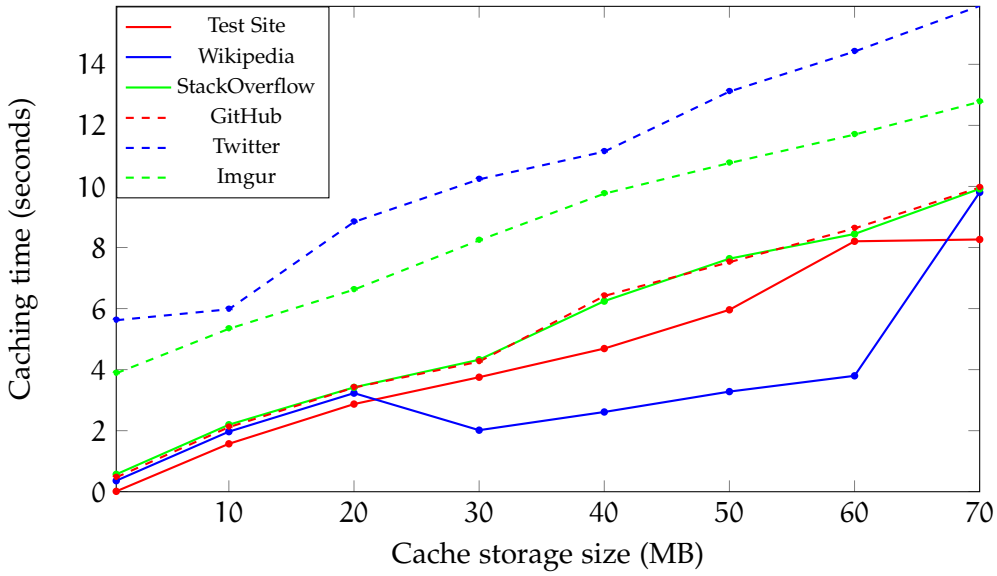


Figure 27: Background script execution time graph.

the cache size. On the graph, notice especially how Twitter and Imgur have much higher cache collection times than the remaining websites. This is due to the size of Twitter and Imgur's front pages – A single cache of Twitter's front page takes up about 6-7 MB in the cache and Imgur takes up about 5-6 MB whereas the other sites in the test suite generally stay below 1 MB. When the cache is empty, the collection times of all sites (except Twitter and Imgur) are less than 600 ms. As soon as the cache size reaches just 10 MB, the collection times have already increased to more than 1500 ms for all sites. When the cache size reaches 70 MB, most sites take upwards of 10 seconds to be collected.

7.3.2 *Cache Cleaning and Retrieval*

When a cache request comes in, the cache-providing client first needs to find the cache locally, decompress it, choose and retrieve an HTS from another peer, calculate its own HTS, find the intersection between the two HTSs, then compress it again and return it to the requesting peer (see Section 5.5.1 on handling cache requests).

Calculating the intersection can be done using either Strict PTI or Relaxed PTI. While Strict PTI on average performs 30% better than Relaxed PTI, as can be seen in Table 4 on the next page, the difference is in practice minuscule, as even Relaxed PTI in most cases computes the intersection in less than one-tenth of a millisecond. The remaining computation takes far longer than the execution of the PTI implementations alone.

Table 5 shows a simplified CPU profile of a cache request to the front page of Wikipedia. The CPU profile has been created manu-

Site	Strict PTI (ms)	Relaxed PTI (ms)	Performance Difference
Test site	0.035	0.058	39%
Wikipedia	0.042	0.079	46%
StackOverflow	0.043	0.056	23%
GitHub	0.052	0.085	38%
Twitter	0.123	0.139	11%
Imgur	0.044	0.091	51%

Performance Difference is how much faster the Strict PTI implementation is than the Relaxed PTI implementation. All website visits were to the front page of the respective sites. Each test was performed 10 times and the mean values registered.

Table 4: Evaluation of PTI implementations performance.

ally using `console.time`, but with some help from Chrome's profiler. Due to the fact that many calls in Peer Cache are asynchronous and callback-based, Chrome's profiler cannot properly identify call chains. The left column in the table shows the number of milliseconds spent in each of the function calls seen in the right column. All tests were made with an almost empty cache, because the problem with caching times described in the previous section carries over to cache retrieval as well: The larger the cache storage, the longer it takes to retrieve a cache from it.

In the example in Table 5, the full request for a clean cache (`getSpecificCleanCache`) took 2903 ms. Immediately after the request, `getCacheFromStorage` was called to retrieve the cached website from the cache storage. The inner request of `getCacheFromStorage` took $2903 - 2874 = 29$ ms, which is the time spent from `getCacheFromStorage` was called until `findBestCache` was called. `findBestCache` was invoked to find the best remote cache. In the Peer Cache prototype, this process uses the indexing server, but should in theory be using a DHT, as described in the Design chapter (Section 5.2.1). Making this request to the local indexing server generally takes about 5-8 ms (in this particular example $2874 - 2867 = 7$ ms).

After retrieving a list of potential caches, `findBestCache` runs through the caches to find the best candidate. This process takes another 1-2 ms. Once a proper candidate has been chosen, a request is sent to the relevant peer to retrieve the HTS.

Table 6 shows the calculation of an HTS on the other peer. In this case, calculating the DHT took a total of 755 ms on the other peer. During this time, the first peer was idly waiting for a response. As the table illustrates, time on the other peer was mostly spent decompressing the cache and calculating the requested hash structure. Notice that there is no compression time (only decompression) in Table 6,

Time (ms)	Function
2903	▼ `getSpecificCleanCache`
2903	▼ `getCacheFromStorage`
2874	▼ `findBestCache`
2867	▼ `getSpecificHashStructure`
184	► `LZString.decompressFromUTF16`
745	▼ `getCleanDocument`
20	▼ `getDocumentFromString`
18	`parseFromString`
716	▼ `getHashStructure`
716	▼ `traverse`
28	► `md5`
681	▼ `traverse`
–	...
826	► `LZString.compressToUTF16`

Table 5: CPU profile of clean cache request.

because HTSs are a lot smaller than cache documents, so they are not compressed before transferring. While this has only been tested briefly (and locally), the added compression and decompression overhead incurred did not make up for the transfer time saved.

Once the first peer has received the requested hash structure, the peer itself will also have to calculate its own hash structure and compute the clean document. Looking at Table 5, we can see that the remaining operation took 2867 ms of which approximately $184 + 745 + 826 = 1756$ was spent on actual work on the peer, leaving a gap of $2867 - 1756 = 1111$ ms for retrieving the HTS from the other peer. However, since this process only took 755 ms, we find $1111 - 755 = 356$ ms that have been unaccounted for. While not evident from the tables, the majority of this time was spent on setting up the WebSocket connection and performing the actual data transfer between the peers.

The function `getCleanDocument` takes a cache as a string and another peer's HTS. From Table 5, we see that most time in this function is spent creating the hash structure (`getHashStructure`), but also that some time is spent converting the HTML cache string to a DOM tree using `getDocumentFromString`. The function `getHashStructure` calculates the HTS recursively. The majority of this time is spent in the `md5` function (the hashing library), whereas `traverse` only is responsible for about 1% of the computation time in `getHashStructure`.

To keep the CPU profile simple, only functions which take up noticeable time have been included. For instance, `getCleanDocument` is

Time (ms)	Function
755	▼ `getSpecificHashStructure`
160	► `LZString.decompressFromUTF16`
477	▼ `getSpecificCache`
18	▼ `getDocumentFromString`
18	`parseFromString`
458	▼ `getHashStructure`
456	▼ `traverse`
23	► `md5`
426	▼ `traverse`
–	...

Table 6: CPU profile of hash tree structure request.

also responsible for calculating the intersection (PTI), as well as creating a clean document based on the intersection and returning it. These operations all take less than a millisecond combined, so they were left out.

While the results in Table 5 and 6 are from the front page of Wikipedia, this example is largely representative of other websites. With some sites, time distribution between compression, decompression and `getCleanDocument` is slightly different, however. In this example, compression takes slightly longer than `getCleanDocument`. On sites with less images, instead, `getCleanDocument` tends to dominate.

CPU profiles for retrieving starred caches have intentionally been left out as they are trivial. In the example presented in this section, `getSpecificCleanCache` took a total of 2903 ms. The same call for a starred cache takes about 30 ms, which is approximately the time it takes for Chrome's storage API to retrieve it from the cache storage.

7.3.3 *End-to-End Testing*

Lastly, we will look at end-to-end testing of the performance of the system. Table 7 shows the time it took from the user pressed the button to retrieve the cache in the extension until the cache had finished rendering on the screen (end-to-end). This data could not be tested using `console.time`, because the start and finish of the measurement were not in the same JavaScript scope. Instead, `console.log` was used for printing a timestamp at the start and end of the process, which were then manually compared.

In the table, *Dirty Cache Computation* is the time it took for the providing peer to process the request; this is the same time as was seen in Table 5. Notice that the computation for Wikipedia in Table 7 in

Site	Dirty End-to-End (ms)	Dirty Cache Computation (ms)	Clean End-to-End (ms)	Clean Cache Computation (ms)
Test site	536	260	242	6
Wikipedia	3849	2503	1262	26
StackOverflow	1823	1457	456	49
GitHub	1198	762	421	28
Twitter	17468	11259	4966	116
Imgur	12969	8496	4263	103

End-to-end is the time it takes from the user presses the cache button until the cache has been rendered on the screen. All website visits were to the front page of the respective sites. Each test was performed 10 times and the mean values registered. Standard deviation was also calculated and was consistently below 13% for measurements.

Table 7: End-to-end testing of performance.

this case only took 2503 ms, whereas it in Table 5 took 2903 ms. This is due to the dynamic nature of Wikipedia's front page: the page is different every day and the cache size (and therefore computation time) is heavily dependent on the number of images present.

Clean Cache Computation measures the time it takes to retrieve a cache that has already been cleaned (or starred). This process is vastly quicker than the Dirty Cache Computation, because no other peers have to be involved, and no compression, decompression or HTS computation is required in the process.

Most sites respond rather well to the end-to-end testing, but as we have seen previously, Twitter and Imgur are exceedingly slow, requiring more than 17 and 12 seconds to compute, transfer and draw the caches in the browser, respectively. This is expected as these sites are a lot bigger than most other sites in the suite. A cache of Wikipedia's front page on average takes up about 0.44 MB, whereas Twitter's front page is a massive 6.82 MB which all has to be decompressed, parsed, traversed/hashed and compressed again before being sent off to the requesting client.

The time it takes from the user presses the extension button until all caches for a specific URL are presented to the user has not been examined thoroughly as this would mostly measure the performance of the indexing server. Indexing server performance is irrelevant as this is not part of the Peer Cache design, which is instead DHT-based. For completeness, however, it can be reported that creating the tab and showing caches takes approximately 100 ms. This would have taken longer if not for the cache storage index (see Section 6.3.1).

7.4 THE WEB FRONTEND

Not much regarding the web frontend needs to be evaluated. The web frontend performs and behaves as the browser extension for listing and viewing caches, thus satisfying the requirements. This is not very surprising as both the Chrome extension and web frontend are built using HTML5 and JavaScript, relying on the same APIs and probably even the same underlying implementations. All evaluation relating to the extension therefore also largely applies to the web frontend.

7.5 DISCUSSION

Coherence and Meaningfulness

Looking at the evaluation results, we can see that meaningful caches can be created for most sites. The test site rightfully stripped the random message as well as the boxes from the clean cache, and most other sites created almost perfect caches as well. Wikipedia, StackOverflow, The Huffington Post, Reddit and GitHub all looked almost identical to the original sites. Twitter struggled, but even though the site's layout was severely misshapen, the main content remained readable most of the time.

Wikipedia, Reddit, StackOverflow and GitHub all seem to suffer when one of the users is logged in and the other is logged out, creating caches with empty `<body>` tags. In all tested cases, the empty caches were caused by HTML class differences on the `<body>` tag of the cache. A more advanced PTI implementation could either find an intersection of classes or contain a whitelist of classes that should never be stripped (such as `loggedin`, `ve-available`, etc.).

Some sites (Twitter especially) also had other stylistic issues. Many of these could likely also be remedied by better intersection algorithms, both on classes, but also for other attributes.

To prevent caching, it is not uncommon for websites to add the current timestamp as a `GET` parameter to resource references, such that `/style.css` becomes `/style.css?1458182730`, for instance. Two otherwise identical caches collected at different times would therefore have different timestamps, even though the stylesheets themselves would be the same. Ignoring timestamps could therefore also help create more rich intersections. Rather than completely removing differing stylesheets, finding intersections between them could also further improve caches.

Other stylistic issues were caused by the privacy policies used on the websites, denying the content script access to certain resources. Collecting these resources directly from the background script could therefore possibly resolve this issue.

One could also imagine issues caused by minor changes to text, even though no such issues were encountered during evaluation. For

instance, if a typographical error in a text paragraph was corrected between two cache collections, the entire paragraph would be left out of the intersection with the current implementation. A better private intersection algorithm could in these cases be used to preserve the entire paragraph, except for the differing word. When removing a differing word or block, it could also be useful to replace the missing words or box with a dummy text ("REDACTED") or box to make the user aware that they are missing content, as well as to help preserve the layout of the websites.

So while the implementation does fall short on some points in terms of creating meaningful caches, all the problems encountered (and imagined) seem fixable.

Cache Collection and Obtrusion on Cache Provider

In terms of performance, the system for the most parts tends to do fine, even though cache collection appears to be somewhat obtrusive on the cache provider. If we for the moment ignore the issues with the cache storage becoming slow as it grows in size, the cache provider's browsing experience was generally directly affected for less than a second while the content script was being executed. During this time, the website would freeze. The background script afterwards, however, did not directly impact the user's browsing experience, but may cause other inconveniences to the user. During the execution of the background script, the CPU load would increase, potentially causing noise by spinning up the CPU fan as well as reduce battery life for a laptop user. While some of this is due to the Chrome's local storage, some load time can also be attributed to the compression algorithm used.

The compression time may be reduced by using a C/C++ compression library through Chrome's Native Client[4]. Likewise, outsourcing both PTI (and as a result the heavy md5 hashing) to native libraries may also increase performance.

The fact that cache collection could take some time is not inherently a problem, though. If CPU load could be limited, having cache collection take even a couple of minutes would not be an issue, as making a cache available right when it is collected rarely is very important. The only problem that long collection times pose, is that people may close their browser before Peer Cache has finished saving the cache or navigate to other websites, causing new caching processes to begin. While caching multiple sites rarely caused any actual concurrency issues, there is a risk that parallel writes may cause data loss due to the simple nature of Chrome's local storage.

To make the content script finish faster, it may also be possible to move most of the work elsewhere. It was initially considered using

4 https://developer.chrome.com/native-client

Web Workers[5], but seeing as these are not allowed to work on the DOM, there would be no benefit to moving the work to a Web Worker in favor of the background script. Potentially, the content script could be reduced to just copying the DOM as a text string and sending it to the background script, thus hopefully shorten the execution time of the content script. As an added bonus, requesting resources from the background script may also help circumvent privacy issues (CORS and CSP).

Cache Cleaning, Retrieval and End-to-End Testing

The evaluation demonstrated in Section 7.3.2 shows that there is no practical performance difference between the two PTI implementations, meaning there is no performance penalty for using the Relaxed PTI algorithm – the algorithm creating richer caches.

Retrieving a clean cache for a website will for the most part take less than 4 seconds (see Table 7). It has been shown many times[6] that long page load times drastically increase abandonment (i.e. users cancelling the loading of a website before it has completed rendering). However, when a user is committed to finding a cache for a website enough to either install the Peer Cache app and extension or navigate to the web front, it is likely that a 4-second load time will not be a big issue. Besides, the loading time for caches on `https://archive.org` tend to be even longer – loading the front page of Wikipedia, for instance, takes between 5 and 10 seconds, depending on which cache is selected.

However, some websites – namely Twitter and Imgur in our test suite – caused severe performance issues. While having to wait more than 17 seconds to see a cached website is very undesirable, if the website is important enough to use Peer Cache to retrieve, the wait time may be something users will willing to endure.

Just as compression time may be reduced by utilizing native libraries, other parts of the implementation could likely also be improved this way. The creation time of HTSs is highly affected by the md5 JavaScript library. Moving HTS creation to a low-level library may therefore also increase performance on that front.

Cache Storage

Chrome's local storage does not seem to be the right tool for the job right now. The cache storage implementation is wonky, not only because accessing and manipulating the local storage takes a long time, but also because the storage, once it reaches a certain size, tends to just stop working altogether. Not only does the storage seem to

5 https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

6 https://blog.radware.com/applicationdelivery/applicationaccelerationoptimization/2013/05/case-study-page-load-time-conversions/

stop working, but it also causes the Peer Cache extension to crash at random times when the storage is too big.

A solution may be to solely rely on Chrome's local storage for indexing, and instead save all resources to disk. Doing so would severely limit the size of the local storage, and therefore increase performance. On top of that, it may also more easily facilitate data deduplication. The current implementation inlines all resources in the caches. Two caches of the same website may therefore contain duplicate data, e.g. having the same stylesheets and images inlined in both caches. When using Chrome's sandboxed file system, serializing resources (base64 encoding them) will therefore no longer be necessary.

As a result, an alternative cache storage system may therefore be a solution to caching issues.

Even though deduplication can save some space, caching every page visited will inevitably still end up taking a lot of space, assuming no caches are deleted. Employing a cache replacement algorithm (see Section 2.3) may therefore be necessary. Choosing the proper algorithm will not be an easy task, as it can be very hard to predict which caches will later become important and which are expendable. However, limiting the amount of almost-identical caches on the client could free up some space, as well as not keeping too many caches of websites that are already well-cached by other peers in the network. Even though there is no perfect solution to cache replacement (as evident by the many algorithms), this problem does seem solvable as well.

8 CONCLUSION

The implementation and design of Peer Cache show that modern browsers (Google Chrome in particular) allow developers to create an ecosystem on top of the Chrome platform that can both cache visited websites, rid them of private information, and share them to users of the system or users of a dedicated web frontend.

While the implementation does not fully meet the design requirements, nor presents a perfect solution, it demonstrates well that the hypotheses presented in Section 1.1.1 all can be confirmed.

Web Preservation can be achieved using peer-to-peer web caching and still remain efficient and provide availability comparable to current client-server systems (Hypothesis 1). It has been shown that even though the implementation may be slow in certain situations, these issues can mostly be solved with a slightly different design and implementation.

Likewise, it has been shown that user-collected caches can be securely rid of personal information, while still remaining meaningful (Hypothesis 2). While this has not been proven, all tests suggest that the system correctly removes all private information. At the current state, we cannot with confidence claim that the system is secure, but it is suspected that with more testing, the system most likely could be considered reasonably secure. The caches created have also for the most part been demonstrated to be meaningful. In the cases where unmeaningful caches were created, potential solutions have been presented to overcome the issue.

While cache collection and sharing is not entirely unobtrusive to the cache provider, all issues standing in the way of this seem to be fixable through changes in the implementation (Hypothesis 3).

Lastly, it was hypothesized that a system could be designed, so users could access the caches without installing any third-party applications or extensions (Hypothesis 4), which has also been confirmed by the existence of the web frontend.

Even though some issues may still be encountered if the suggested solutions (see the Discussion section) were implemented, the limitations mostly seem be caused by JavaScript's performance. JavaScript has progressed by leaps and bounds in recent years, and since the trend seems to continue, it is safe to say that the Peer Cache concept only becomes more viable every day.

BIBLIOGRAPHY

[1] Hari Balakrishnan, Scott Shenker, and Michael Walfish. Peering Peer-to-Peer Providers. In *4th International Workshop on Peer-to-Peer Systems (IPTPS '05)*, Ithaca, NY, February 2005.

[2] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.

[3] Emiliano Cristofaro and Gene Tsudik. *Financial Cryptography and Data Security: 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers*, chapter Practical Private Set Intersection Protocols with Linear Complexity, pages 143–159. Springer-Verlag Berlin Heidelberg, Berlin, Germany, 2010. ISBN 978-3-642-14577-3.

[4] Michael Day. Preserving the fabric of our lives: A survey of web preservation initiatives. In *In Proc. 7 th ECDL*, pages 461–472. Springer, 2003.

[5] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 213–222, New York, NY, USA, 2002. ACM. ISBN 1-58113-485-1.

[6] Roberto J. Bayardo Jr., Rakesh Agrawal, Daniel Gruhl, and Amit Somani. YouServ: A Web-Hosting and Content Sharing Tool for the Masses, 2002.

[7] Giovanni Neglia, Giuseppe Reina, Honggang Zhang, Donald F. Towsley, Arun Venkataramani, and John S. Danaher. Availability in bittorrent systems. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA*, pages 2216–2224, 2007.

[8] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on ot extension. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 797–812, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7.

[9] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *Proceedings of the 24th USENIX Conference on Security*

Symposium, SEC'15, pages 515–530, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-232.

[10] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003. ISSN 0360-0300.

[11] Indranil Gupta Prakash Linga and Ken Birman. Peer-to-Peer Web Caching Using Kelips, 2004.

[12] Weisong Shi and Yonggen Mao. Performance Evaluation of Peer-to-Peer Web Caching Systems. In *Journal of Systems and Software, Volume 79, Pages: 714 - 726, Year of Publication: 2006*, pages 0164–1212, Wayne State University, MI, USA, 2005. Elsevier Inc.

[13] Petr Praus Tomáš Černý, Slávka Jaroměřská, Luboš Matl, , and Michael J. Donahoo. Towards a Smart, Self-scaling Cooperative Web Cache, 2012.

[14] Xiaohui Zhang. Cachability of Web Objects, 2000.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`http://code.google.com/p/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`

Final Version as of March 30, 2016 (`classicthesis`).