Rethinking Web Caching: An Optimization for the Latency-Constrained Internet

Mohammad Hosseini
Shahid Beheshti University
Tehran, Iran

Sina Darabi
Università della Svizzera italiana
Lugano, Switzerland

Patrick Eugster
Università della Svizzera italiana
Lugano, Switzerland

Mahmood Choopani
Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran

Amir Hossein Jahangir
Sharif University of Technology
Tehran, Iran

ABSTRACT

Caching is a fundamental web technique for reducing Page Load Time (PLT) by reusing previously fetched resources. We highlight the drawbacks of the current caching approach, especially in the context of high-speed networks where latency, rather than bandwidth, is the primary bottleneck for web performance. We discuss how the current design of web caching suffers from inefficiencies, particularly due to the latency involved in re-validation requests, which diminishes the potential benefits of caching. To address this inefficiency, we present a novel solution in which web servers proactively provide clients with the latest validation tokens for resources during the initial step of page loading, allowing browsers to use unchanged cached content without unnecessary round trips. This method significantly reduces PLT, with preliminary evaluations showing a 30% improvement.

CCS CONCEPTS

• **Networks** → **Network protocols**; **Application layer protocols**.

KEYWORDS

Web, Cache, HTTP, Latency, RTT, PLT, Browser.

HOTNETS '24, November 18–19, 2024, Irvine, CA, USA

ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696873

ACM Reference Format:
Mohammad Hosseini, Sina Darabi, Patrick Eugster, Mahmood Choopani, and Amir Hossein Jahangir. 2024. Rethinking Web Caching: An Optimization for the Latency-Constrained Internet. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3696348.3696873

1 INTRODUCTION

The Web, alongside its underlying protocol HTTP, emerged as the killer application in the early days of the Internet and has continued to dominate the online world. The widespread use of the Web and the dependence of businesses and users on it have made its performance critically important. Page Load Time (PLT) is one of the key performance metrics of the Web. Numerous reports have highlighted the impact of reducing PLT on increasing business revenue [8, 9, 11, 41]. According to the reports, even small improvements, like a 100-millisecond decrease in PLT, can lead to a significant increase in revenue. For this reason, numerous research efforts are focused on reducing web page load times [5, 10, 13, 14, 16, 18, 21, 23, 25, 33, 39, 46, 49].

Caching represents one of the most effective strategies for improving PLT by enabling client browsers to reuse previously fetched resources. When a browser accesses a webpage, it stores reusable resources, which can then be utilized in future requests to the same page or other pages within the same website. This eliminates the time spent downloading these resources, thereby significantly reducing PLT.

Despite its benefits, the current web caching mechanism exhibits significant limitations that inhibit its full potential. Design flaws in the existing caching system lead web developers to use it conservatively [19, 20, 30]. This means that many resources are either not made cacheable or have their cache duration set to a short period. This conservative approach, at best, leads to the underutilization of caching benefits and, at worst, results in frequent unnecessary data transmissions [18, 24, 29]. Hence, there is a significant gap between

the ideal and actual performance of web caching, highlighting the suboptimality of current caching approach [19, 29].

Apart from the conservative use of caching, another issue is that the HTTP cache re-validation mechanism, which is used to avoid retransmitting unchanged content, is not optimal. In this mechanism, the client sends a request for a resource along with the validation token (*ETag*) of the resource stored in its cache. The server checks the token and, if the resource has not changed, it sends a short response informing the client that the resource remains unchanged, thus eliminating the need to download the file. However, this solution is rooted in the limitations of the early Internet era, when bandwidth was low, and download speed was the bottleneck. Nowadays, with high download speeds, the primary bottleneck has shifted to latency. Re-validation requests still necessitate a round-trip time (RTT), which can be significant and sometimes even longer than the transmission time of downloading a resource.

This issue becomes highly significant considering that, with today's high-throughput access links, end-to-end latency plays a crucial role in page load time. The research of Sundaresan *et al.* revealed that at a downstream throughput of 16 Mbits/s, the latency of access link becomes the main bottleneck for web page load time [42]. They have shown that even a 10-millisecond increase in last-mile latency can lead to an increase of several hundred milliseconds in page load time. This issue is even more pronounced with mobile web access. Cellular networks, while offering high download speeds, also have considerably high latency; even 5G links often exhibit latency similar to that of 4G [27]. Recent research on mobile web performance has demonstrated that latency, rather than bandwidth, is the determining factor for PLT [21–23, 30]. Therefore, minimizing the number of RTTs required to load a web page is essential. One way to achieve this is to eliminate the unnecessary RTTs involved in the cache re-validation mechanism, and it requires a rethinking of web caching.

To address the inefficiency of the current caching design, we propose a novel approach that eliminates unnecessary round trips caused by the cache re-validation mechanism. The key idea of our solution is that, at the beginning of the web page loading process, the server provides the client with the validation tokens for the resources the client will need. This method allows the browser to check resource versions upon receiving the base HTML file, use cached content when versions match, and only fetch updated resources when necessary. This approach eliminates the need to set cache duration (`max-age`) for resources. Our preliminary evaluations suggest that this approach can reduce PLT by an average of 30 percent, offering a promising solution to the latency issues posed by traditional caching mechanisms. It is noteworthy that the proposed solution can be deployed without any changes to the existing client browsers. To encourage further research, we have made our implementation open-source[1].

In the rest of this paper, we first discuss in more detail the motivations for rethinking web caching. Then, we present the proposed method along with its preliminary implementation and evaluation results. We also discuss the advantages of the proposed method over other web acceleration approaches, such as Remote Dependency Resolution (RDR) and Server Push. Finally, we outline our plan to complete this solution in future work.

2 BACKGROUND AND MOTIVATION

Before delving into the motivations behind this work, we briefly review how web caching is deployed using HTTP cache headers.

2.1 Overview of the current web caching

Each web page is composed of a base HTML file and numerous other resources linked within this HTML file. A client browser retrieves the web page and its resources from a server using the HTTP protocol. Figure 1a shows an example of HTTP interactions to retrieve a web page. The browser first sends an HTTP `GET` request to the server to fetch the base HTML file (`index.htm` in the figure). By parsing the HTML file, the browser identifies links to other resources (`a.css` and `b.js`) and sends requests for them as well. Fetching and evaluating other resources (such as CSS and JavaScript files) may lead to the need for additional resources (`c.js` and `d.jpg`), in which case the browser fetches them too. As illustrated in the figure, the time required to fetch each resource consists of two components: round-trip time (RTT) and the transmission time of downloading the resource. The RTT is influenced by the distance between the client and the server, while the download time depends on the bandwidth of the Internet links between them and the size of the resource.

Since many resources may be requested again during subsequent visits to the same page or other pages on the same website, the browser caches these resources based on cache headers sent by the server with each response. The most important cache-related header fields are `no-store`, `no-cache`, `max-age`, and `ETAG`, which must be set by the website's developer. These headers inform the browser how long to keep each resource in the cache and what actions to take when the resource expires. When the browser needs a resource again, it first checks its cache to see if a copy is already available. If the resource is found in the cache and has not expired, the browser quickly serves the cached content. If the resource

[1] https://github.com/toorin-lab/CacheCatalyst

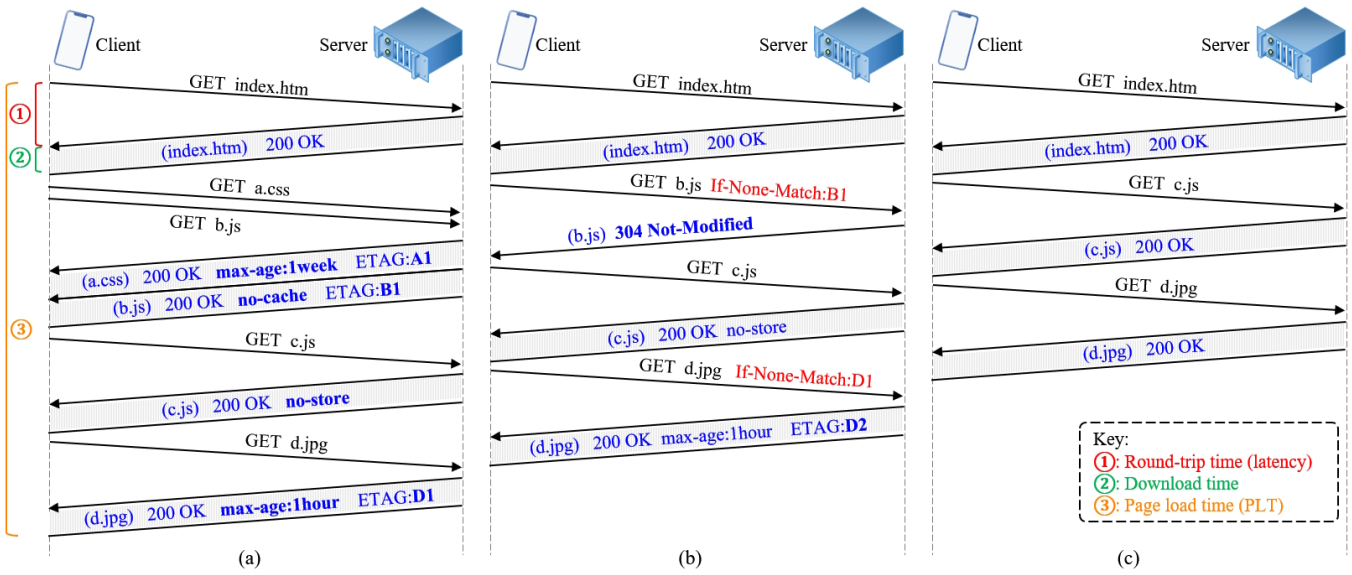

Figure 1: Web caching mechanism using HTTP cache headers

(a) The first visit to a web page. "`a.css`" and "`b.js`" are identified after parsing "`index.html`". "`c.js`" and "`d.jpg`" are fetched as a result of evaluating "`b.js`" and "`c.js`", respectively. **(b)** Another visit to the same page or a similar page on the same website two hours later. **(c)** Optimized scenario for a revisit, assuming "`a.css`", and "`b.js`", and "`d.jpg`" have not changed at the time of the revisit.

has expired or if the browser was instructed not to cache it, the browser requests a fresh copy from the server.

The simplest cache header is `no-store`. It means that the browser must not cache the resource and must download it every time it is needed. The `max-age field` specifies the time-to-live (TTL) for a resource, indicating how long its content will remain unchanged. The browser can cache and reuse a resource for the duration specified by its `max-age` field. Figure 1b shows a revisit to the mentioned page two hours after the first visit. Since `a.css` was retrieved with a `max-age` of one week in the first visit, the browser does not send a request for this file in the second visit and instead uses the cached version. If a resource is expired (`d.jpg`), the browser sends a conditional request known as the re-validation mechanism. The `no-cache` header (in `b.js`) results in the same outcome. Despite its name, `no-cache` means that the browser can cache the resource, but it must re-validate it before each reuse. A re-validation request asks the server to send the resource if its content differs from the cached version in the browser. If the resource has not changed, the server sends a short `Not-Modified` response, thereby eliminating the transmission time of downloading the resource. This is the case for `b.js` in Figure 1b. If the resource has changed, the server sends the new version of the resource, which is the case for `d.jpg` in Figure 1b. Checking for changes is done using a validation token called `ETAG`. The server includes a validation token in each response that can be cached. The browser includes the validation token in the `If-None-Match` field of each re-validation request to inform the server which version of the resource is cached in the browser.

2.2 Why web caching needs rethinking?

The latency bottleneck. A previous study has found that increasing throughput beyond 16 Mbits/s does not have any considerable effect on reducing web page load time [42]. At this throughput rate, latency becomes the bottleneck and the determining factor in PLT. The study revealed that although clients in East Asia have higher average throughput than those in Europe, they do not experience corresponding improvements in PLT due to latency bottlenecks negating the benefits of higher throughput. Another study by Asrese *et al.* [2] has also confirmed that webpages perform differently across regions and service providers without a direct correlation with throughput.

Latency is influenced by the distance between the client and the server. Bringing all servers closer to all users is not feasible. Additionally, access links like mobile connections inherently have high latency, and reducing this latency is a challenging issue. Therefore, to reduce PLT, we must minimize the negative impact of latency as much as possible. One of the best solutions is to use caching. However, as will be discussed in the following, the current caching mechanism is designed to mitigate the negative impact of low throughput, not high latency.

Significant redundant transfers in web traffic. Basically, setting cache headers for resources should be done by the website developer. The website developer must decide, based on the type, usage, and content of each resource, whether a resource should be cached and for how long (TTL). However, since this task is difficult, developers often neglect it. As a result, default headers are applied to resources by the site's content management system (CMS) based on the type of each resource. Since CMSs are not aware of the usage of every resource, they often send many resources with settings that prevent them from being cached by the client. Consequently, many resources that could have been cached are not cached in practice, and the same content is repeatedly sent to the user on each visit. Previous studies have shown that, on average, only about 50 percent of the resources that can be cached are actually cached [18–20]. The significant redundant transmissions in web traffic confirm this issue [1, 18, 24, 29]. These redundant transmissions not only increase PLT but also result in higher power consumption and data costs for mobile users. Hence, we need a more straightforward caching solution that can be optimally deployed without developer interaction.

Conservative TTLs. In the current caching approach, web servers must set a TTL value for each resource that can be cached. However, estimating how long a resource will remain unchanged is challenging, and accurately predicting when a resource will change is usually impossible. Setting a TTL that is too long can cause the client to use a stale version of the resource. Conversely, setting a TTL that is too short causes the resource to expire prematurely, leading to unnecessary re-validation requests to the server even though the content has not changed. These re-validation requests impose additional RTTs on the page load time, which make the current caching scheme ineffective for mobile web browsing [47, 48]. Previous studies have shown that developers often set TTLs conservatively, meaning they are much shorter than the actual duration. For instance, the research by Ramanujam *et al.* has shown that 47% of resources expire in the cache even though their content has not changed [30]. Another study found that 40% of resources have a TTL of less than one day, but 86% of these do not change within that period, meaning they expire in the cache without changing [19]. The most difficult scenario is when the TTL for a resource cannot be estimated at all, in which case the developer uses a `no-cache` header. This means that although the resource can be cached, it is always considered stale, and the client must re-validate it each time it is used.

This caching policy was beneficial when the Internet bandwidth of clients was low and downloading was a bottleneck, as it improved PLT by eliminating the transmission time of downloading unchanged resources. However, with current high download speeds, this approach is insufficient, especially considering that the size of resources on web pages is small. Web pages, while containing hundreds of resources, have a total size of about 2.5MB [15]. In other words, resources are around a few kilobytes in size, which means that their download time is comparable to RTT. Therefore, simply eliminating download time is not enough; the re-validation RTT must also be eliminated. Hence, we need a solution that not only works without the need for setting TTL but also eliminates or minimizes re-validation requests.

Summary. The current caching policy is far from optimal. An optimal caching solution should ensure that a client's request for a new version of a resource is only sent when the content of the cached resource has changed on the server. In other words, as long as the resource has not changed, the client should be able to use it immediately without incurring an RTT. Figure 1c illustrates the optimal scenario for the mentioned example.

3 PROPOSED APPROACH

We propose a simple solution to eliminate the RTT spent on the re-validation process. The solution is to provide the client browser with the most recent validation tokens for resources it requires to load the page as quickly as possible, i.e., at the beginning of the page loading process.

When the server receives a request from the client browser for the base HTML file (`index.htm`), it should send, along with the HTML file, the validation tokens (`ETags`) for the resources that the browser will need to load the page. The validation tokens can be placed in the HTTP response header. Upon receiving and parsing the base HTML file, the client browser identifies the links to other resources. Before sending requests to fetch these resources from the server, it can compare the validation tokens of the cached resources with the newly received tokens. If they differ, it means the resource has changed on the server and needs to be fetched again. If they match, the browser can use the cached content. Therefore, if the resource has not changed, the browser can use the cached resource without incurring an RTT. Additionally, there is no need to specify the TTL value or set `max-age`, making caching much easier to use.

There are two challenges in implementing this solution. The first is that the web server must determine which resources will be needed to load the desired page. Most resources are deterministic and can be identified by parsing HTML and CSS files [4]. Therefore, each time the web server wants to send an HTML or CSS file to the client, it first inspects the file, identifies the links to other resources within it, and then sends the validation tokens for all those resources along with the requested file. However, some resources must be identified through the execution of JavaScript code, which

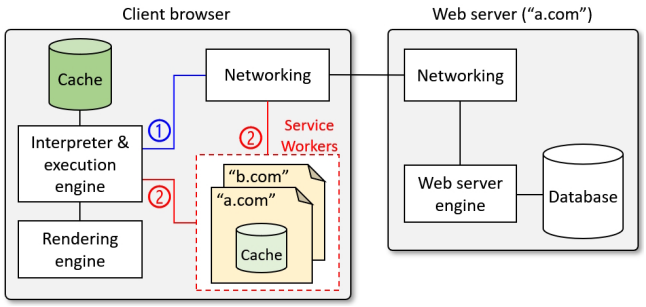

Figure 2: Service Workers
①: the path that requests and responses normally take
②: the path taken if a Service Worker exists for a domain

is not deterministic. Many of the links to resources that need to be fetched during the execution of JavaScript are not explicitly defined within the code and require execution to be generated. Additionally, many of the generated links depend on the user and their interactions. A method for identifying all resources that might be fetched by JavaScript code is symbolic execution [17].

A different solution entails the server capturing a list of resource URLs that the client requests during a user's first visit to a webpage. This list functions as a guide to the specific resources required for rendering the page for that particular user. When the user returns to the same page in later sessions, which can be recognized through session management techniques, the server includes validation tokens for the previously listed resources along with the primary HTML file in its response. The advantage of this method is that it also covers dynamic and user-specific resources. For now, we focus on the resources identified outside of JavaScript, which typically cover most of the resources on a page. We leave the consideration of resources within JavaScript code for future work.

The second challenge is to implement this caching model on the client side. Before sending a request for a resource, the browser must compare the validation token sent by the server for the resource with the validation token stored in its cache, and only send the request if they are different. Obviously, this method should be inherently integrated into the browser. However, we propose a solution to implement this method with existing browsers using the Service Worker API [43].

A Service Worker, as shown in Figure 2, is a proxy that resides in browsers and intercepts requests and responses between a website (on the client side) and its origin server. Service Workers are domain-specific, meaning that a Service Worker is registered by the origin server of a website and only intercepts requests to the corresponding domain. A Service Worker, which is written in JavaScript, can control and modify requests, and also respond to requests on its own. Each Service Worker can have its own cache and use it to respond to requests when the origin server is not accessible (for example, in offline mode). All major desktop and mobile browsers support Service Workers.

In the proposed solution, the web server registers a Service Worker on the browser for each website during the first visit of any client. This Service Worker stores all resources received from the server in its cache, provided they do not have a `no-store` header. Additionally, at the beginning of each visit (request for the base HTML file), the web server provides the Service Worker with the validation tokens of all resources so the Service Worker knows the latest validation token for each resource. Therefore, during each user visit to the website, the Service Worker can immediately serve the resources from its cache if they have not changed.

In our preliminary implementation, we used Caddy [6, 7], a popular open-source web server. We modified Caddy so that, when serving a web page, it first traverses its entire DOM, extracts all resource links, and creates a map of links to `ETags`. This map only includes resources that are on the main server. Considering resources hosted on third-party servers (cross-origin resources) is planned for future work. The map is placed in a header field named `X-Etag-Config` in the HTTP response for the base HTML file. The web server also inserts the registration code of the Service Worker in the HTML file. During each subsequent visit to the website and after receiving the base HTML file, the Service Worker intercepts client requests for other resources and compares the `ETag` in each request with the `ETag` specified in the map received from the server. If the `ETags` match, the Service Worker immediately serves the response from its cache; otherwise, it forwards the request to the server, delivers the response to the browser, and stores it with the new `ETag` in its cache.

4 PRELIMINARY EVALUATION

To evaluate the proposed solution, we selected the 100 most visited websites [36]. We cloned and hosted their homepages on the modified Caddy web server and used Selenium [35] with ChromeDriver to load the pages and measure PLT[2]. The measurement of page load time was performed using the `OnLoad` event. Additionally, we simulated different latencies and throughputs using the browser's throttling feature

First, we measured the PLT for each page without deploying the proposed solution. For this purpose, we loaded each page in a cold cache state. Then, with time delays of

[2] The evaluation of additional performance metrics, such as First Contentful Paint (FCP), Speed Index (SI), and Time To Interactive (TTI), which more comprehensively reflect user experience, has been deferred to subsequent research.

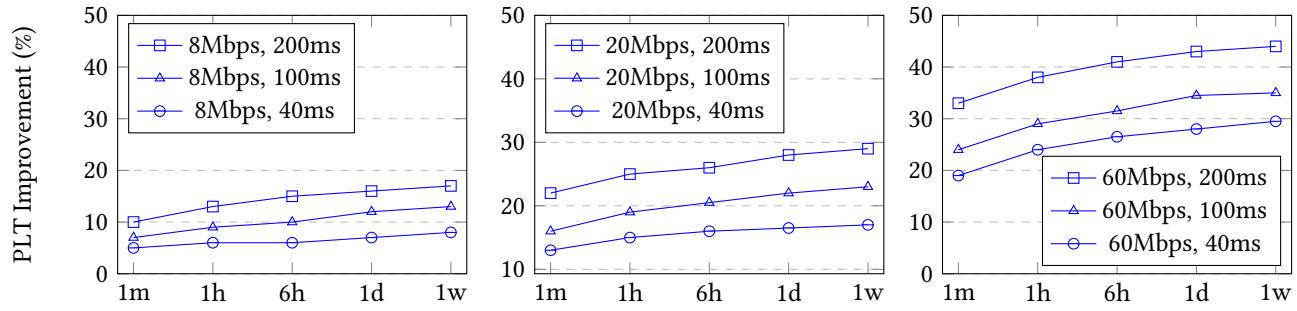

Figure 3: Reduction in PLT by the proposed approach under different network conditions (throughput and latency)

one minute, one hour, six hours, one day, and one week, we reloaded the page and measured the PLT for each one. To simulate the time delays, we advanced the system clock by the desired amount. Next, we repeated the previous steps with the proposed method enabled and compared the resulting PLT with the PLT obtained from the previous experiment. Figure 3 shows the average improvement (reduction) in PLT with the proposed solution compared to the current caching approach under various network conditions in terms of throughput and latency.

As can be seen in the figure, while the proposed method does not significantly reduce PLT at low throughput (8Mbps), a substantial reduction in PLT is observed at high throughput (60Mbps). This is because, under these conditions, the bottleneck in web page loading is latency rather than throughput, and by eliminating unnecessary RTTs, the PLT is significantly reduced. Additionally, at a constant throughput, the improvement in PLT is more pronounced for higher latencies. Therefore, this method provides greater benefits for users in regions that are farther from web servers. It is noteworthy that a throughput of 60 Mbps and a latency of 40 ms represent the median condition for global 5G Internet access [40].

5 RELATED WORK

To clarify how the proposed method is situated within the literature, we review PLT improvement methods that aim to mitigate the negative impact of network latency and compare them with our approach.

Cache-related methods. Since many web publishers do not configure the cache-control headers correctly and carefully, Raza *et al.* [31] have proposed Extreme Cache, a cache proxy between clients and servers that sets the cache headers by estimating the change rate of objects. However, as mentioned, estimating the change time of a resource is not straightforward, and this paper does not provide any report on the estimation accuracy. Moreover, the change time of many resources is not predictable at all (*no-cache* header). Wang *et al.* [45] demonstrated that although many resources change frequently, the changes are often limited, with large portions remaining constant. Based on this, they proposed the Micro-caching method, which allows for caching only the parts of a resource (such as JavaScript or CSS statements) that do not change, instead of the entire resource. This way, only the changed parts are sent to the client. While this method reduces the download size, it does not eliminate RTT. Fawkes [21], a server-side module, captures all content that remains unchanged across versions of a page's top-level HTML and uses it to create a static template. This template is cached on the client side to maximize the cached content for the page. During each client page load, the server generates dynamic patches that express the updates (i.e., DOM transformations) required to convert the template page state into the latest version of the page. Rewap [18] acts in a similar manner. This method does not eliminate the RTT of unpredictable dynamic resources. Geol *et al.* [10] proposed a browser-based solution that extends the cache of browsers to reuse identical computations (e.g., JavaScript execution) from prior page loads. This method is orthogonal to our work, and an optimal caching approach can further augment the benefits of using an execution cache.

Remote dependency resolution. An effective approach to resolving the problem of last-mile latency is to deploy a remote dependency resolution (RDR) proxy [25, 38, 39]. An RDR proxy is implemented using a headless browser on a cloud server with low-latency network paths to origin web servers. It moves the resolution of resource dependencies closer to web servers and performs resource fetches on behalf of the client. After fetching all resources of the requested web page, it sends the entire web page in bulk to the client. While this method is very effective in reducing PLT, it poses security and privacy issues since it acts as a man-in-the-middle for TLS connections. This method not only violates end-to-end TLS security but also requires clients to provide proxies with the clear form of their private HTTP cookies.

WatchTower [26] is a workaround for this problem that requires each HTTPS origin to run its own RDR proxy. RDR only has a positive impact on the initial load of the website. Resources that need to be fetched dynamically through user interaction still experience high latency. In contrast, with our proposed method, if a resource is in the cache and is valid, it is used immediately and without latency. It is also noteworthy that some proxies perform other optimizations, such as minification and compression, to accelerate the download time of resources and also reduce mobile data usage [1, 37].

Server push. Server Push is a feature of HTTP/2 [3] that allows a web server to send resources to a client browser before the client requests them. The web server can anticipate future client requests and push the resources that the client will likely need, thereby avoiding unnecessary RTTs. However, anticipating a user's actions on a web page is not straightforward. Therefore, various policies are employed for pushing resources. The simplest policy is for the server to push all resources. Numerous studies have shown that this policy can waste user bandwidth by sending resources that are either not needed or already cached by the client [44, 50]. Additionally, it can increase the page load time due to the increased computational load on the client side [51, 52]. It has also been shown that network performance characteristics play a major role in the effectiveness of Server Push [32]. Klotski [5], Shandian [46], and WebGaze [16] identify high-priority resources of pages in terms of user utility and prioritize pushing them. Another problem with Server Push is that the content of many web pages is often served by multiple domains, and the main web server cannot securely push resources from other domains. Hence, some authors have proposed providing the client with hints in the form of URLs for resources that the client should fetch before its own dependency resolution [12, 23, 33]. Using our caching approach, many of these prefetches will not be necessary, and less bandwidth will be used.

The RTT issue in other protocols. The shift in communication bottlenecks from bandwidth to latency in many internet applications has led to changes in numerous internet protocols to minimize additional RTTs wherever possible. An example of this is the mechanism known as Stapling [28] in the Online Certificate Status Protocol (OCSP) [34]. Browsers utilize OCSP to verify the revocation status of TLS certificates they receive. In the standard OCSP protocol, each time a browser obtains a certificate, it queries an OCSP server to check if the certificate was revoked. This process incurs a round-trip delay before the web page can be downloaded. With OCSP stapling, the web server attaches a recently signed statement from the certificate authority to the certificate itself, indicating that the certificate remains valid. When a browser receives a certificate with this stapled information, it no longer needs to query an OCSP server to determine the certificate's status. This approach eliminates the round-trip delay, thereby improving page load times.

6 CONCLUSION AND FUTURE WORK

In this paper, we discussed why the current web caching design is not optimal under the conditions of the modern Internet, where throughput is not a bottleneck. We then proposed a solution that eliminates unnecessary RTTs imposed by the cache revalidation mechanism, thereby improving PLT. Additionally, our solution makes using the cache much easier by removing the need to set TTL for resources.

There are three primary issues to address in order to finalize the proposed solution. First, to expand the coverage of the proposed solution, a method to extract links to resources fetched by executing JavaScript code is necessary. This method should be designed in such a way that it does not impose a significant computational overhead on the web server. As stated in Section 3, an effective approach might involve the server recording the set of resources that each user fetches during their first visit to each page. This method also covers dynamic and user-specific resources. However, it potentially incurs a significant memory footprint, necessitating an optimization strategy. The second important issue is how to cover cross-origin resources, i.e., the ones that reside on third-party servers. The main server does not have direct access to them and, as a result, cannot give their `ETags` to the client. One potential solution is for the main server to fetch those resources itself and obtain their `ETags`. The third issue pertains to sites that already have their own Service Workers. In such cases, the web server must add the cache-related Service Worker to each site in a way that does not interfere with the activities of the site's existing Service Worker.

The comparison of this approach with alternative methods, particularly Server Push, has also been deferred to future research due to the necessity of a multifaceted analysis of results. For example, as mentioned in the previous section, Server Push can impose additional processing load on the client browser, potentially affecting various performance metrics differently. Hence, this issue requires a more in-depth evaluation using not only Page Load Time (PLT) but also other performance metrics such as First Contentful Paint (FCP), Speed Index (SI), and Time To Interactive (TTI). This comprehensive analysis will be carried out in future work. The effect of this approach on the performance of web servers should also be analyzed.

ACKNOWLEDGMENT

Work supported in parts by Swiss National Science Foundation projects 192121 (FORWARD) and 197353 (BASIS)

REFERENCES

[1] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. 2015. Flywheel:{Google's} Data Compression Proxy for the Mobile Web. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 367–380.

[2] Alemnew Sheferaw Asrese, Steffie Jacob Eravuchira, Vaibhav Bajpai, Pasi Sarolahti, and Jörg Ott. 2019. Measuring web latency and rendering performance: Method, tools, and longitudinal dataset. *IEEE Transactions on Network and Service Management* 16, 2 (2019), 535–549.

[3] Mike Belshe, Roberto Peon, and Martin Thomson. 2015. *Hypertext transfer protocol version 2 (HTTP/2)*. Technical Report. IETF RFC 7540.

[4] Michael Butkiewicz, Harsha V Madhyastha, and Vyas Sekar. 2013. Characterizing web page complexity and its impact. *IEEE/ACM Transactions On Networking* 22, 3 (2013), 943–956.

[5] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V Madhyastha, and Vyas Sekar. 2015. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 439–453.

[6] Caddy. 2024. *Caddy source code*. https://github.com/caddyserver/caddy

[7] Caddy. 2024. *Caddy web server*. https://caddyserver.com

[8] Chrome DevRel. 2020. *Milliseconds make millions*. https://web.dev/case-studies/milliseconds-make-millions

[9] fastcompany. 2012. *How One Second Could Cost Amazon $1.6 Billion In Sales*. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

[10] Ayush Goel, Vaspol Ruamviboonsuk, Ravi Netravali, and Harsha V Madhyastha. 2021. Rethinking client-side caching for the mobile web. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*. 112–118.

[11] Jake Brutlag (Google). 2009. *Speed Matters for Google Web Search*. https://services.google.com/fh/files/blogs/google_delayexp.pdf

[12] Bo Han, Shuai Hao, and Feng Qian. 2015. Metapush: Cellular-friendly server push for http/2. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*. 57–62.

[13] Mohammad Hosseini, Sina Darabi, Amir Hossein Jahangir, and Ali Movaghar. 2023. Yuz: Improving Performance of Cluster-Based Services by Near-L4 Session-Persistent Load Balancing. *IEEE Transactions on Network and Service Management* (2023).

[14] S Mohammad Hosseini, Amir Hossein Jahangir, and Sina Daraby. 2021. Session-persistent load balancing for clustered web servers without acting as a reverse-proxy. In *2021 17th International Conference on Network and Service Management (CNSM)*. IEEE, 360–364.

[15] httparchive. 2024. *State of the Web*. https://httparchive.org/reports/state-of-the-web?start=2024_01_01&view=list

[16] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R Das. 2017. Improving user perceived page load times using gaze. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 545–559.

[17] Ronny Ko, James Mickens, Blake Loring, and Ravi Netravali. 2021. Oblique: Accelerating page loads using symbolic execution. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 289–302.

[18] Xuanzhe Liu, Yun Ma, Shuailiang Dong, Yunxin Liu, Tao Xie, and Gang Huang. 2016. ReWAP: Reducing redundant transfers for mobile web browsing via app-specific resource packaging. *IEEE Transactions on Mobile Computing* 16, 9 (2016), 2625–2638.

[19] Xuanzhe Liu, Yun Ma, Yunxin Liu, Tao Xie, and Gang Huang. 2016. Demystifying the imperfect client-side cache performance of mobile web browsing. *IEEE Transactions on Mobile Computing* 15, 9 (2016), 2206–2220.

[20] Yun Ma, Xuanzhe Liu, Shuhui Zhang, Ruirui Xiang, Yunxin Liu, and Tao Xie. 2015. Measurement and analysis of mobile web cache performance. In *Proceedings of the 24th International Conference on World Wide Web*. 691–701.

[21] Shaghayegh Mardani, Mayank Singh, and Ravi Netravali. 2020. Fawkes: Faster Mobile Page Loads via {App-Inspired} Static Templating. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 879–894.

[22] Javad Nejati and Aruna Balasubramanian. 2016. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web*. 1305–1315.

[23] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*.

[24] Ravi Netravali and James Mickens. 2018. Remote-control caching: Proxy-based url rewriting to decrease mobile browsing bandwidth. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*. 63–68.

[25] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 417–429.

[26] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. 2019. Watchtower: Fast, secure mobile page loads using remote dependency resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. 430–443.

[27] opensignal. 2020. *How AT&T, Sprint, T-Mobile and Verizon differ in their early 5G approach*. https://www.opensignal.com/2020/02/20/how-att-sprint-t-mobile-and-verizon-differ-in-their-early-5g-approach

[28] Yngve N. Pettersen. 2013. The Transport Layer Security (TLS) Multiple Certificate Status Request Extension. https://www.rfc-editor.org/info/rfc6961

[29] Feng Qian, Kee Shen Quah, Junxian Huang, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. Web caching on smartphones: ideal vs. reality. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 127–140.

[30] Murali Ramanujam, Harsha V Madhyastha, and Ravi Netravali. 2021. Marauder: synergized caching and prefetching for low-risk mobile app acceleration. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 350–362.

[31] Ali Raza, Yasir Zaki, Thomas Pötsch, Jay Chen, and Lakshmi Subramanian. 2015. Extreme web caching for faster web browsing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 111–112.

[32] Sanae Rosen, Bo Han, Shuai Hao, Z Morley Mao, and Feng Qian. 2017. Push or request: An investigation of HTTP/2 server push for improving mobile performance. In *Proceedings of the 26th International Conference on World Wide Web*. 459–468.

[33] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V Madhyastha. 2017. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 390–403.

[34] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Carlisle Adams. 2013. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. https://www.rfc-editor.org/info/rfc6960

[35] Selenium. 2024. *A browser automation framework*. https://www.selenium.dev

[36] Similarweb. 2024. *Top Website Ranking*. https://www.similarweb.com/top-websites/

[37] Shailendra Singh, Harsha V Madhyastha, Srikanth V Krishnamurthy, and Ramesh Govindan. 2015. Flexiweb: Network-aware compaction for accelerating mobile web transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. 604–616.

[38] Ashiwan Sivakumar, Chuan Jiang, Yun Seong Nam, Shankaranarayanan Puzhavakath Narayanan, Vijay Gopalakrishnan, Sanjay G Rao, Subhabrata Sen, Mithuna Thottethodi, and TN Vijaykumar. 2017. Nutshell: Scalable whittled proxy execution for low-latency web over cellular networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. 448–461.

[39] Ashiwan Sivakumar, Shankaranarayanan Puzhavakath Narayanan, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, and Subhabrata Sen. 2014. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 325–336.

[40] speedcheck. 2024. *Global 5G Speed Index*. https://www.speedcheck.org/5g-index/

[41] WPO stats. 2024. *The impact of web performance optimization (WPO) on user experience and business metrics*. https://wpostats.com/

[42] Srikanth Sundaresan, Nick Feamster, Renata Teixeira, and Nazanin Magharei. 2013. Measuring and mitigating web performance bottlenecks in broadband access networks. In *Proceedings of the 2013 conference on Internet measurement conference*. 213–226.

[43] World Wide Web Consortium (W3C). 2022. *Service Workers*. https://www.w3.org/TR/service-workers/

[44] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2014. How speedy is SPDY?. In *11th usenix symposium on networked systems design and implementation (nsdi 14)*. 387–399.

[45] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2014. How much can we micro-cache web pages?. In *Proceedings of the 2014 Conference on Internet Measurement Conference*. 249–256.

[46] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up web page loads with shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 109–122.

[47] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2011. Why are web browsers slow on smartphones?. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. 91–96.

[48] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2012. How far can client-only solutions go for mobile browser speed?. In *Proceedings of the 21st international conference on World Wide Web*. 31–40.

[49] Xu Zhang, Siddhartha Sen, Daniar Kurniawan, Haryadi Gunawi, and Junchen Jiang. 2019. E2E: embracing user heterogeneity to improve quality of experience on the web. In *Proceedings of the ACM Special Interest Group on Data Communication*. 289–302.

[50] Torsten Zimmermann, Jan Rüth, Benedikt Wolters, and Oliver Hohlfeld. 2017. How HTTP/2 pushes the web: An empirical study of HTTP/2 server push. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 1–9.

[51] Torsten Zimmermann, Benedikt Wolters, and Oliver Hohlfeld. 2017. A QoE perspective on HTTP/2 server push. In *Proceedings of the Workshop on QoE-based Analysis and Management of Data Communication Networks*. 1–6.

[52] Torsten Zimmermann, Benedikt Wolters, Oliver Hohlfeld, and Klaus Wehrle. 2018. Is the web ready for http/2 server push?. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. 13–19.