Server-Side Dependency Resolution for Reducing Webpage Load Times

Adolfo Victoria
Carnegie Mellon University
avictori@andrew.cmu.edu

Bobbie Chen
Carnegie Mellon University
bobbiec@andrew.cmu.edu

ABSTRACT

Modern web pages often require multiple RTTs to resolve dependencies, e.g. when a CSS file refers to an image. This is a problem for users who experience high last-mile latency, like 3G mobile users and users in rural locations. We present a solution to this problem: server-side dependency resolution (SSDR) is a reverse-proxy that resides on the origin server that resolves dependencies using a low-latency local connection rather than the high-latency last-mile connection. The placement of the SSDR proxy on the same machine as the origin server allows significant reduction in the number of last-mile RTTs, without having to compromise the security of HTTPS traffic between the endpoints. We quantify the performance of SSDR on HTTPS traffic over relevant parameters such as number of dependencies and varying latency values.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems.* D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms

Measurement, Performance, Design, Experimentation

Keywords

Remote Dependency Resolution, Server-Side Dependency Resolution, Mobile Accelerators, Latency Reduction, Scalability, Last Mile Latency, Internet Services, Proxies, Protocols, HTTP/2, Server Push

1. INTRODUCTION

Responsiveness in Web services is a crucial component of the user experience. If page load times are slow, then the amount of time that a user spends using the services is reduced, and their likelihood of leaving the site entirely drastically increases [2]. This is exacerbated by the high last-mile latency inherent in mobile Internet connections: a 3G connection has a minimum one-way latency of 100 ms [10]. Although the first 5G networks are being deployed today, 3G is still prevalent: in 2019, 12% of wireless connections in North America were on 3G, while 45% of wireless connections in sub-Saharan Africa were still on 2G (300 ms) [8].

One common approach to improve page load times is the use of content delivery networks (CDNs) to bring content closer to the users. These networks essentially act as distributed caches with many nodes, improving latency because they will usually be closer to the end users than the origin servers are. However, CDNs do not fully solve the problem of last-mile latency; for example, if a webpage contains an external style sheet that refers to an image URL, a minimum of three RTTs are still incurred on the slow last-mile connection between the client and the CDN.

For significant improvement to be made for users with high last-mile latency, we must find a way to reduce the number of round-trips entirely. In this paper, we present server-side dependency resolution (SSDR), which allows the origin server to pre-resolve page dependencies, reducing client requests to a single RTT in the best case. The SSDR proxy works as a drop-in solution, without requiring changes to the webserver code or network protocol and without needing to add new trusted nodes to the network. Our evaluation shows significant benefits in page load times, albeit with negative effects on caching.

2. RELATED WORK

There are two primary techniques to improve page load times on slow last-mile connections: remote dependency resolution, and prefetching. Our SSDR proxy is directly related to the concept of the "content gathering network", which uses proxy servers with low-latency connections to desired origin servers to resolve dependencies on the proxy [6]. This use of a proxy for remote dependency resolution (RDR) is also seen in mobile accelerators like Shandian [23] and Silk [3].

When using an RDR proxy, a client must trust the proxy node to terminate HTTPs, which forfeits end-to-end encryption and exposes their traffic to the RDR proxy, or otherwise slowly resolves HTTPS sites in the traditional manner without RDR. Our SSDR proxy resides directly on the origin server, so HTTPS

may be terminated with no security implications; this comes at the cost of only being able to serve the single origin server. The WatchTower system describes this technique as an "HTTPS sharding proxy" and uses them as part of a larger RDR proxy system; this paper is an in-depth look at the performance characteristics of such a proxy in isolation, without the complexity of WatchTower [20].

In prefetching, a developer writes code to explicitly fetch data from the server before it is requested by the client. For example, Android developers are encouraged to prefetch data in the official Android developer guide [4]. This approach may work better for mobile applications, where use cases are typically narrow; but for our use case, website navigation patterns may be much more unpredictable. Like prefetching, in the HTTP/2 protocol web servers may use "server push" to pre-push dependencies to clients, piggybacking off an existing request [5, 11]. In practice, prefetching and server push can significantly improve page load times, but both techniques require significant developer effort to tune exactly which resources are prefetched [12, 21]. Additionally, HTTP/2 is a protocol-level change which requires adoption by both clients and servers; even today, five years after the protocol was published in RFC7540, less than 50% of websites have adopted HTTP/2 [19]. While we believe server-push is a good solution to the problem we face, an interim solution like our SSDR proxy can significantly improve the user experience until HTTP/2 adoption becomes more widespread.

Thus, there have been convincing attempts in the literature at addressing the problem of slow page loads caused by high-latency last-mile connections. Unfortunately, their implementation costs are prohibitive in terms of server and client code having to change or in terms of having to change infrastructure to support them.

3. DESIGN REQUIREMENTS

With a slow last-mile connection, any unneeded round-trips drastically increase the total page load time. Therefore, we want our solution to use a few RTTs independent of the number of dependencies that are in the page. Ideally, we want the user's load speed to be constrained by the bandwidth rather than the latency to the server. Our solution should also not affect the performance of users with negligible last-mile latency, because then we would be affecting the common case.

We also aim to minimize the integration costs of our solution. Existing techniques, described in the previous section, may require code changes for clients and servers or the addition of new nodes to a system. In recognition of the end-to-end principle, we prefer that clients and servers can benefit from our solution without having to know about our internals or change their behavior. Furthermore, if changes to client or server code were required then we would have to ensure that there is backwards compatibility and graceful fallbacks for clients and servers that did not implement the changes. With similar reasoning, changes to the protocol are undesirable. We envision a drop-in, turnkey solution that can improve performance with minimal changes for the server operator and no changes for the clients.

4. SYSTEM

Our solution is inspired by the concept behind RDR proxies, where a proxy with a high-speed connection to the origin server resolves the dependencies before sending them to the client. This proxy node can be at a variable distance from the origin server, and we simply take this idea to the limit. By placing the proxy node on the same machine as the origin server, we can transparently reverse proxy the origin server without changing any of its code (except perhaps the port that it listens on). This also allows us to terminate HTTPS without leaking data to any outside nodes and without having to provision additional SSL/TLS certificates.

The architecture is straightforward: the SSDR proxy is a reverse proxy that listens on a public-facing port and directs requests to the origin server. Upon receiving the first response from the server, the proxy will resolve dependencies like images and same-origin iframes using the high-speed local connection - effectively a RTT of zero. Once the dependencies are resolved, the SSDR proxy can return the result to the client, who receives a fully-renderable page with no unresolved dependencies. This solution has the advantage that it can work together with other solutions such as CDNs. To do so, we would simply place the proxy at the CDN edge server.

The design of a reverse proxy makes integration straightforward. An existing webserver publicly listening on port 443 (for HTTPS) switches to listen on a local port only. Then, the SSDR proxy can listen on port 443 and serve HTTPS using the same certificates as the webserver, and reverse proxy the

local port. This SSDR proxy is able to serve any content from the webserver, including personalized and dynamic content.

4.1 CACHING IMPLICATIONS

This solution disrupts client-side caching. Normally, each dependency is cached independently of each other based on its URL. In our solution, all dependencies are inlined, so the only resource that can be cached is the entire page. If any small part of the page changes, then the browser cache entry will be stale, and a full-page load would be required to retrieve the latest content. Furthermore, assets shared between pages will not be taken advantage of because they are bundled into requests and thus cannot be cached individually. As a result, overall performance may be harmed when visiting multiple pages on the same site.

To cache pages with our current system, the client would need to be aware of the proxy server to cache files individually, which implies a change in the client code. Although this opposes our design goals, we briefly explored changes to the overall system architecture to restore caching performance. In a cache friendly SSDR system, the SSDR proxy can mark each inlined dependency with a tag that indicates its origin URL. Then, a client can cache dependencies by origin URL. On a new request, a client sends headers indicating which dependencies it already has; the server may output a stub for the client to fill using its cache instead of inlining the full dependency. On closer examination, this theoretical system is essentially a reinvention of HTTP/2 server push, with additional overheads caused by the SSDR proxy; as mentioned earlier, we believe server push is a fundamentally sound approach that is limited by adoption.

4.2 PERCEIVED PERFORMANCE

Although our system decreases the total page load time, it also delays the initial HTTP response to the client to inline the dependencies. This means that the user sees a blank screen for longer, which is often perceived as a slow page even if the total page load time is lower [7, 18]. The traditional approaches to improving time to the first visible content are lazy-loading, in which a small stub of a page with basic content is first sent to the user, and then Javascript is used to load the rest of the page; increased caching of dependencies; and reduction in large responses. These approaches are unfortunately fundamentally incompatible with SSDR. One possible mitigation is to apply SSDR only for the "above-the-fold" section of the page; this would be a complex optimization, with many variables like client screen size and browser rendering differences, so it is beyond the scope of this paper to implement and evaluate.

5. IMPLEMENTATION

We implemented the SSDR reverse proxy using mitmproxy, a TLS-supporting proxy with a robust scripting interface [15]. The implementation script uses the Python requests library [14] to resolve dependencies in the following manner: (1) external CSS and Javascript are inlined as text, (2) images are inlined using the data:base64 URL scheme, and (3) iframes are inlined using the srcdoc attribute, and recursively resolved up to a configurable depth. There are various implementation details required to not break websites: absolute urls to the origin server must be rewritten to point to the SSDR proxy instead, and relative URLS sometimes need to be rewritten, e.g. CSS URLs are relative to the CSS file location, not the parent page. The implementation is fully available at https://github.com/bobbiec/ssdr/.

This implementation was written under time constraints, and so it has various flaws. These flaws are not fundamental to the technique and could be fixed with more work; they are described here for completeness. First, the use of the requests library, a simple HTTP client, means that Javascript is not parsed and evaluated; this means that if any dependency is generated by Javascript execution, it will not be resolved by our SSDR proxy. Second, the mitmproxy does not handle HTTP 301 or 302 redirects, which means that these dependencies also cannot be resolved. Third, dependency resolution must fully complete before any content is sent to the client, which adds unnecessary delay compared to a streaming approach. Finally, the implementation is not robust; when given malformed HTML, our simple parsing script fails to find dependencies.

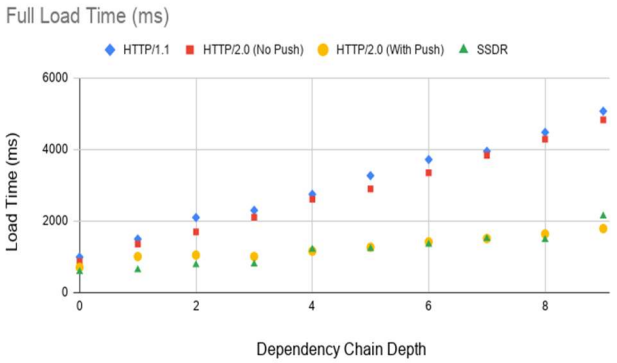

Figure 1: Time it takes for the browser to finish loading the page. Loading time increases linearly with the depth of the dependency tree.

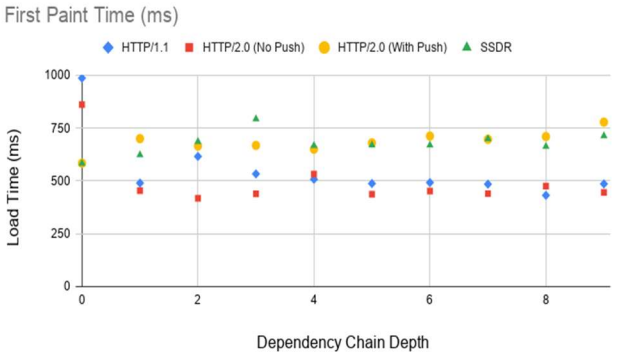

Figure 2: Time it takes for the browser to first display content at different depths dependending on the server. There is a slight upward trend in SSDR and server push due to them sending bigger responses to the browser. On the other hand, HTTP/1.1 and HTTP/2 with no server push always send the base page first, which makes their response time stay close to constant.

6. EVALUATION

We had a few fundamental questions about our SSDR proxy: To what extent does SSDR improve page load times compared to a baseline of HTTP/1.1? How does this scale with the maximum depth ("critical path") of the dependency tree? How does this scale with the RTT to the origin server? How does this compare against HTTP/2 server push? How does this affect the user experience in terms of time to first page paint? And what difference does the lack of caching make in terms of performance?

To answer these questions, we built a testing website using the Python framework Quart with the HTTP/2 compatible web server Hypercorn [17], with small pages of varying critical paths. This allows us to run the same code and web pages with four different web server configurations: (1) HTTP/1.1 baseline, (2) HTTP/1.1 with SSDR proxy, (3) HTTP/2 with server push, and (4) HTTP/2 without server push. Configuration (4) is included because we theorize the benefits of HTTP/2 in our use case are entirely derived from server push. Large pages were not tested, as the existing literature shows that latency effects dominate over bandwidth effects unless bandwidth is extremely limited [1, 20, 24, 25, 26]. This implementation is also fully available at https://github.com/bobbiec/ssdr/.

The web servers were hosted on a DigitalOcean droplet (VM) with 1 vCPU and 1 GB of RAM. Latency measurements were taken from our home computers in Pittsburgh. The original plan was to migrate the droplet between DigitalOcean's worldwide data centers, but we were thwarted by a lack of capacity to create new droplets due to increased demand. Instead, we simulated network latency using the Network Throttling capability in Mozilla Firefox [16]. While Google Chrome allows completely configurable network profiles compared to Firefox's presets, we found in testing that Chrome's throttling implementation did not correctly apply network latency to iframes. Firefox's rendering engine does not resolve iframes nested deeper than ten deep, so we were unable to test deeper levels of dependencies; further work is needed to see if such deep dependency graphs are common in the wild.

6.1 Load Time Results

As shown in Figure 1, we find significant improvements in the total page load time when using the SSDR proxy, compared to the baseline HTTP/1.1 server. As expected, the HTTP/1.1 server incurs an RTT for each layer of the dependency graph, because the browser makes a full round-trip to traverse a single link in the graph. The degree of improvement is linear with respect to the dependency depth as well as the RTT. This improvement comes directly from the dependency resolution; the HTTP/2 server with server push is also able to send dependencies without requiring extra round-trips, and it performs nearly identically to the SSDR proxy. We can isolate this improvement to server push in particular: the HTTP/2 server without server push performs as poorly as the HTTP/1.1 server does. Notice a small linear increase in the SSDR and HTTP/2 server push page load times with respect to dependency chain depth; we investigated this and found that the network time was actually constant, and the increase in page load time came entirely from browser rendering time for the

layers of nested iframes. A small amount of delay was also added due to the increase in the server's processing time due to dependency resolution.

6.2 Time to First Paint Results

Our second set of measurements provide data on how long it takes for the browser to first display content, in order to measure the impact on user experience. As was mentioned earlier, it is possible for users to start using a webpage without all the dependencies loading. In our testing, we found that both SSDR and HTTP/2 server push both suffer a time to first paint penalty at higher dependency depths. On the other hand, we saw a performance improvement in the time to first paint for smaller dependency depths due to the round-trip time savings because of JavaScript and CSS inlining.

Thus, there exists an impact on user experience because the time to first paint can be up to twice as large as without server push or SSDR on highly nested pages. The significance of this impact is not clear because the load time is greatly improved as well. This shows that there exists a tradeoff between page load time and time to first paint, where each one sees improvement on a different dependency resolution technique. A user study would be required to determine whether the benefits of SSDR and server push are perceived as detrimental or beneficial to user experience.

Server	Load Time (ms)
HTTP/1.1	430
HTTP/2 (no push)	430
HTTP/2 (with push)	473
SSDR	540

Figure 3: Load time with caching enabled at depth 0.

6.3 Caching Results

Our last set of measurements compare the performance of the four servers when caching is enabled in the browser. We found that iframes are not cached by browsers: this adds the factor of *dependency tree composition*, where the exact reason for dependencies affects the resulting performance. Due to the time limitations, we tested using the "depth 0" page, which directly references one CSS and one JS file with no further dependencies. As can be seen in Figure 3, caching greatly improved the performance of HTTP/1.1 and both HTTP/2 servers since they no longer need to fetch the JavaScript and CSS.

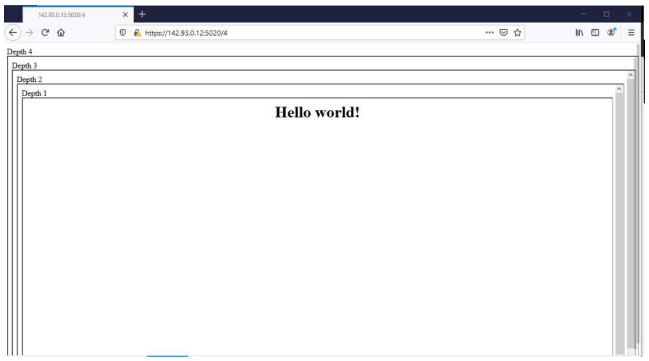

Figure 4: Testing website that emulates different dependency depths.

On the flipside, the performance of SSDR did not improve. This is due to dependency inlining, which makes the browser unable to cache the dependencies. Server push does not suffer from this limitation because it is able to account for this on the client-side protocol, reaping the performance improvements from caching and thus addressing the concerns raised in section 6.2.

Using these findings, we predict that the performance of SSDR will lag the others due to this limitation at different dependency tree depths. Even though we are assuming that bandwidth does not make a difference, we saw in section 6.1 that the time taken by the server to resolve dependencies can increase the load time of the page. Further testing at different dependency tree depths and compositions is needed to say whether this difference can have a noticeable impact on page load times.

7. FUTURE WORK

The choice of what content to inline or server-push is an active area of research. For example, WatchTower periodically measures latencies in the dependency graph to provide better dependency resolution. As mentioned above, time to first paint may be improved by restricting dependency resolution to only the "above-the-fold" content. If dependency resolution is not fully applied, this means that remaining content is resolved traditionally with additional round-trips. Additional work is required to determine an appropriate tradeoff between first paint and total load times. Time to first paint is also determined by undocumented browser-specific rendering behavior; WProf [22] has characterized some of these interactions, and this work can be applied to inform server push policies.

As shown in the evaluation, the page load benefits of SSDR come directly from a reduction in round-

trips. This means that SSDR may be combined with other techniques from mobile accelerator research which provide page load benefits through orthogonal changes like bandwidth or computation, such as proxies that also compress and transcode content as it passes through [1, 4, 9].

Finally, recent advances in commercial spaceflight have greatly increased the likelihood of a human colony on Mars in our lifetimes. An HTTP request from a Mars colony has a minimum latency measured in minutes rather than milliseconds; we predict that websites that implement SSDR will provide a drastically better user experience and capture the very valuable Mars colonist demographic. Under these extreme network conditions, SSDR can be further augmented with whole-site prefetching, possibly with a Coda-style user-defined "hoard profile" to prefetch sites completely unrelated to the initial request [13]. As humanity expands its reach further into the stars, even prefetching will be too slow; we propose the use of machine learning to generate potential webpages from the initial HTTP request alone, removing the need for costly network requests altogether.

8. CONCLUSION

In this paper, we present and evaluate the SSDR proxy, a specialized RDR proxy which is placed directly on the origin server as a reverse proxy. This approach resolves dependencies with a high-speed local connection rather than a potentially slow last-mile connection, and preserves the security guarantees of HTTPS. We find that the SSDR proxy produces performance benefits comparable to HTTP/2 server push, without requiring client or server protocol upgrades, and with only a small penalty in time to first paint.

9. ACKNOWLEDGMENTS

We thank our professor Dave O'Hallaron and our fellow classmates in 18-845 for their insights throughout discussion over the semester, we learned a lot of things that were not taught in 15-213. Furthermore, we would like to thank Dave and our TA Andrew Yang for their feedback in this project and in the individual project. We would also like to thank our roommates Janet Li, Sarah Chen, and Ishraq Bhuiyan for keeping us sane during the quarantine.

10. REFERENCES

[1] Agababov et al. 2015. Flywheel: Google's Data Compression Proxy for the Mobile Web. NSDI 2015. USENIX 2015. https://research.google/pubs/pub43447/

[2] Akamai. 2017. *The State of Online Retail Performance*. https://www.akamai.com/us/en/multimedia/documents/report/akamai-state-of-online-retail-performance-spring-2017.pdf

[3] Amazon. 2020. What is Amazon Silk? https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html

[4] Android Developers. 2020. *Optimize downloads for efficient network access.* *https://developer.android.com/training/efficient-downloads/efficient-network-access#prefetch*

[5] Belshe et al. 2015. *RFC 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2)*. https://tools.ietf.org/html/rfc7540

[6] Debopam Bhattacherjee, Muhammad Tirmazi, and Ankit Singla. 2017. A Cloud-based Content Gathering Network. HotCloud 2017. https://www.usenix.org/conference/hotcloud17/program/presentation/bhattacherjee

[7] Google Developers. 2020. First Meaningful Paint. https://developers.google.com/web/tools/lighthouse/audits/first-meaningful-paint

[8] GSM Association. 2020. The Mobile Economy. https://www.gsma.com/mobileeconomy/

[9] Han et al. 1998. *Dynamic adaptation in an image transcoding proxy for mobile web browsing*. IEEE Personal Communications, vol. 5. DOI: https://doi.org/10.1109/98.736473

[10] Ilya Grigorik. 2013. *High Performance Browser Networking.* https://hpbn.co/

[11] Ilya Grigorik and Surma. 2019. Introduction to HTTP/2. https://developers.google.com/web/fundamentals/performance/http2#server_push

[12] Instagram Engineering. 2017. Improving performance with background data prefetching. https://instagram-engineering.com/improving-performance-with-background-data-prefetching-b191acb39898

[13] James Kistler and M. Satyanarayanan. 1992. Disconnected Operation in the Coda File System. ACM Transactions on Computer Systems, Vol. 10, No. 1. https://www.cs.cmu.edu/~satya/docdir/kistler-tocs-coda-1992.pdf

[14] Kenneth Reitz. 2019. Requests: HTTP for Humans. https://requests.readthedocs.io/en/master/

[15] Mitmproxy Project. 2020. mitmproxy - an interactive HTTPS proxy. https://mitmproxy.org/

[16] Mozilla. 2019. Throttling - Firefox Developer Tools. https://developer.mozilla.org/en-US/docs/Tools/Network_Monitor/Throttling

[17] Phillip Jones. 2017. Quart: a Python web microframework. https://pgjones.gitlab.io/quart/

[18] Phillip Walton. 2020. First Contentful Paint (FCP). https://web.dev/fcp/

[19] Q-Success. 2020. Usage Statistics of HTTP/2 for Websites, May 2020. https://w3techs.com/technologies/details/ce-http2

[20] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. 2019. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. MobiSys 2019. DOI: https://doi.org/10.1145/3307334.3326104

[21] Torsten Zimmermann, Benedikt Wolters, Oliver Hohlfeld, Klaus Wehrle. 2018. Is the Web ready for HTTP/2 Server Push? CoNEXT 2018. DOI: https://dl.acm.org/doi/10.1145/3281411.3281434

[22] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. D*emystifying Page Load Performance with WProf*. NSDI 2013.

[23] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. *Speeding up Web Page Loads with Shandian*. NSDI 2016. https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-wang-xiao-sophia.pdf

[24] Mike Belshe. *More Bandwidth Doesn't Matter (much)*. 2010. https://goo.gl/PFDGMi

[25] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. *Supplementary Material For WatchTower*. 2018. http://web.cs.ucla.edu/~ravi/publications/watchtower_supplementary_material.pdf

[26] Srikanth Sundaresan, Nick Feamster, Renata Teixeira, and Nazanin Magharei. *Measuring and Mitigating Web Performance Bottlenecks in Broadband Access Networks*. 2013. DOI: https://dl.acm.doi/10.1145/2504730.2504741