SeedyN: Safely Delivering Content Across a Network of Untrusted Peers

Devon Rifkin, Ian Walsh, Rose Perrone

Abstract

SeedyN is a fully distributed browser-based P2P web cache that depends on signatures provided by origin servers in order to be secure. The main goal of this distributed web cache is to increase the reliability of users' browsing experience. Our browser add-on hosts a web proxy which notices when a site is taking a long time to respond. The site is then cached in several peers so that if the site can't respond or is slow to respond due to demand, users with our add-on installed can get the content from the distributed cache. Requests are hence offloaded from servers experiencing unsustainable traffic.

1 Introduction

Peer-to-peer web caches are generally designed to handle two major problems: mitigating the cost of providing popular content and handling flash crowds that exhaust origin server capacity. We focus on achieving the latter in a fully decentralized system.

Distributed web caches, such as CoralCDN [3], aim to relieve load on overburdened origin servers by using a network of volunteer caching nodes. More recently, browser-based P2P caches such as PeerCDN [9] and Firecoral [10] have blurred the line between clients and caching nodes in an attempt to gain massive potential capacity. However, these systems still rely on some degree of centralization for security. We eliminate the need for centralized security by requiring origin servers to opt-in: resources that should be cacheable must be accompanied by a cryptographic signature.

With SeedyN, we combine the DHT-based content discovery from CoralCDN with the browser-based approach of the Firecoral Firefox add-on. We avoid capacity issues faced by CoralCDN by having users of the system provide bandwidth and sidestep the use of a centralized tracker like Firecoral through our DHT implementation. Since SeedyN is a browser add-on, we are able to operate transparently to the user by proxying HTTP traffic. We are able to optimize for normal operation by only routing the user's request through our network when the origin server is slow to respond.

Encouraging adoption of a system such as SeedyN will be challenging, as it requires modifications to origin servers to sign their content. Additionally, users must install a browser add-on, which makes distribution far more difficult than systems like PeerCDN that run on individual webpages. However, we contend that installing a browser add-on is a lower barrier to entry than installing and configuring a separate HTTP proxy. We envision that we can positively influence server adoption by providing drop-in resource signing plugins for popular HTTP server frameworks such as Express.

2 System Architecture

2.1 Overview

The system is composed of three parties: clients that have a browser add-on installed, origin servers that have `X-CACHE-SIGNATURE` in their HTTP response headers, and several seed servers that play a minor role: helping clients join the peering network.

The client is entirely contained within the add-on and largely consists of an HTTP proxy server. Once installed, the client operates transparently to the end-user. As the user browses content, our add-on monitors cacheable requests to see if the corresponding origin server is having difficulty processing the request. If a particular cacheable request exceeds a threshold in time waiting for a response, or the server returns errors suggesting capacity failure, we search for a cached version of the response in the DHT.

The peers in our system can be anyone who has installed our add-on to their browser, so our system must operate without any trust between users. Other P2P web caches, such as Firecoral, achieve security in untrusted P2P networks by implementing a trusted centralized signing server, along with a trusted centralized tracker. However, we wish to build a fully decentralized service where trust is only required between the clients and the origin servers they are attempting to contact. Thus, we require origin servers to sign any content they wish to be cacheable and provide the SHA-256 signature as the `X-CACHE-SIGNATURE` header in the HTTP response. We also require that each origin server publish its public key in the DNS. Nodes that are caching content must also cache the signature, and provide the signature along with the resource to clients that ask for it. When the client retrieves cached data from a peer, it must first verify that the signature is valid before handing over that resource to the browser.

In order to bootstrap new clients, we must ship down some list of *seed nodes* with the add-on that are likely to already be part of the network. The client tries to connect to a subset of the seed nodes in order to discover non-seed peers. This is slightly counter to our notion of decentralization, but this is still much less onerous than a single centralized server; the list of seeds can include volunteer clients from a wide variety of backgrounds and affiliations.

2.2 Peer-to-Peer Communication

Historically, peer-to-peer communication on the web has been difficult to achieve, but a technology called WebRTC is now supported in both Firefox and Chrome. WebRTC is aimed at allowing realtime P2P communication in the browser, particularly for both voice and video chat [11]. However, WebRTC also allows for arbitrary data transmission, which seems promising for an application such as ours. However, after evaluating the technology, we have found it to be unsuitable for our use case. WebRTC was designed for long-lasting scenarios such as video calls, and has a fairly intensive initialization step that involves a third-party signaling server that must assist in exchanging tokens between clients. Clients are also unable to reconnect directly with each other after a connection is terminated; they must go through the signaling server again. This does not interact well with a DHT because nodes are constantly learning of new nodes and forming new connections.

WebRTC makes sense for web-based systems such as PeerCDN, as there are not many other options for P2P communication on a webpage. However, since SeedyN is running as a browser add-on, we are able to receive messages from peers by running a full HTTP server. We still can run into trouble with users behind NATs being unable to run servers, but there are well known techniques for NAT traversal, such as hole punching [2] and gateway support for protocols such as UPnP [5]. Additionally, WebRTC uses a NAT traversal technique called STUN that uses a third-party server to allow connections between peers [4]. There are public servers provided by third parties such as Google that implement STUN, so this technique is a possible fallback for us to use. We do not currently implement any NAT traversal techniques, but this is a feature that we would like to add in the future.

3 Implementation

The SeedyN client is currently implemented as a Firefox add-on. The client is written entirely in Javascript, but it is able to access privileged APIs not available to Javascript running the context of a webpage. These enhanced privileges let it use XPCOM [8] to leverage core Firefox components such as a built-in HTTP server that was originally designed to assist in running unit tests [7]. The extension is able to start up a server with its own request handlers, and then using more privileged APIs, dynamically proxy web requests originating from the browser.

In addition to using the add-on's HTTP server to act as a proxy to intercept requests to potentially fetch from the DHT instead, we also use the HTTP server to implement our DHT's peer-to-peer communication. The HTTP server adds a request listener at a predetermined path for other clients to use to initiate communication. We have implemented a pure Javascript version of Kademlia [6], which is the DHT used by our add-on.

We have also implemented a prototype origin server by providing an asset signing plug-in for Express, a node.js-based server framework. Of course, our client is compatible with any origin server that correctly signs resources, and in normal operation, we would expect a wide variety of origin server implementations.

4 Evaluation

4.1 Security Overhead

A potential concern that comes with our approach is the overhead required for servers to sign all cacheable assets. Additionally, clients using cached content must verify that the content was not tampered with by verifying the signature. We performed several experiments to show the impact of these operations.

To obtain statistically significant results, we ran each test many times using the benchmark.js [1] performance testing framework. First, we benchmarked server-side signing using node.js's built-in crypto library. We signed a variety of differently sized data and measured how long it took to generate the signature. We then measured the performance of . See figure 1 for our results.

For signing small sized resources, milliseconds per resource is very acceptable given how long server responses typically take. As the resources start hitting the several hundred kilobyte range, things get much slower. However, an optimization can be performed: large resources that we would want to be cacheable are typically static assets. The server only needs to calculate the signature once and then cache the result.

Our results show that verifying signatures in the add-on is quite costly, particularly for large resources. To compare, we found that using node.js with its built-in crypto library, the server was able to verify thousands of signatures per second. What accounts for this difference? For one, node.js is able to use native crypto functions, instead of performing all operations in Javascript, which suggests our results are at least partially due to the difference between well-used, highly optimized C code and niche Javascript code. In the browser add-on, we are using a port of node.js's crypto library. The speed of verifying resources is acceptable in the add-on for smaller resources, which we expect to account for most of our resources. Larger resources become troublesome, particularly with file sizes above 128 KB. There are various techniques we can use in the future to try to mitigate the cost of verifying in the browser:

Size	Signing (node.js)	Verifying (browser)
8 B	552 ops/sec	16.16 ops/sec
512 B	550 ops/sec	16.36 ops/sec
2 KB	544 ops/sec	15.66 ops/sec
4 KB	541 ops/sec	15.00 ops/sec
8 KB	541 ops/sec	13.70 ops/sec
16 KB	529 ops/sec	11.59 ops/sec
32 KB	501 ops/sec	8.68 ops/sec
64 KB	458 ops/sec	5.89 ops/sec
128 KB	387 ops/sec	3.53 ops/sec
512 KB	205 ops/sec	1.07 ops/sec
1 MB	128 ops/sec	0.55 ops/sec
4 MB	37.29 ops/sec	0.14 ops/sec

Figure 1: Number of crypto operations per second for different sizes of data.

- Use web workers to be able to verify multiple resources at the same time. Since cached webpages are likely to have many resources, verification could often be done in parallel.
- Split large resources up into chunks and provide signatures for each chunk. This would likely make Javascript performance better, but come at the cost of a more complicated interface for servers. Servers would need to split up files and

provide the signature for each piece in the response headers.

- Use native executables to perform verification. Firefox add-ons are able to ship executables and then use XPCOM to run them. We could likely achieve performance rivaling node.js by doing this, but this could come at a cost to distribution. It seems unlikely that Mozilla would accept an add-on executing native code to their add-on gallery, which provides high visibility and ease of installation for add-ons.

5 Related Work

Firecoral [10] is another browser-based P2P cache that instead of a DHT, it utilizes a centralized tracker to provide content verification and peer discovery. Firecoral uses digital signatures, but requires the clients to trust the centralized tracker. Trusting the centralized tracker does have advantages; for one, it means that unmodified origin servers can have their content cached, whereas SeedyN requires origin servers to sign their content.

PeerCDN [9] and Maygh [12] are similar P2P web caches that share a goal different from ours. Instead of being aimed at users, they are presented as solutions for website operators to reduce bandwidth costs. When a user loads a page, they are also delivered Javascript that connects with a centralized tracker to connect to other peers using WebRTC. This does not require users to opt-in, so just by visiting a site, a user may unknowingly be volunteering their bandwidth. Since these solutions are web based rather than add-on based, they cannot provide caching for other domains and also require users to remain on the page to provide their bandwidth. We instead present SeedyN as an opt-in solution where a user can decide to donate bandwidth in order to avoid problems coming from flash crowds.

6 Conclusion

In implementing SeedyN, we have shown that a fully decentralized peer-to-peer webcache is practical. By making it a browser add-on, we have made it easy to distribute and gained implementation advantages such as using a communication method more suited to DHTs than WebRTC. SeedyN is both opt-in for users and servers, but provides incentives for both — users get a better average browsing experience in exchange for their bandwidth, and by signing their content, servers become more robust against flash crowds. Both code and a binary for the browser add-on is available, along with a sample origin server that implements our content signing scheme.

References

[1] M. Bynens and J.-D. Dalton. Benchmark.js v1.0.0. `http://benchmarkjs.com/`, 2014.

[2] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[3] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[4] IETF. RFC 5766: traversal using relays around NAT (TURN): Relay extensions to session traversal utilities for NAT (STUN). `https://tools.ietf.org/html/rfc5766`, 2010.

[5] IETF. RFC 6970: universal plug and play (UPnP) internet gateway device - port control protocol interworking function. `https://tools.ietf.org/html/rfc6970`, 2013.

[6] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the

XOR metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[7] Mozilla. HTTP server for unit tests. `https://developer.mozilla.org/en-US/docs/Httpd.js/HTTP_server_for_unit_tests`, 2014.

[8] Mozilla. XPCOM. `https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM`, 2014.

[9] PeerCDN. Meet PeerCDN, a completely new kind of CDN. `https://peercdn.com`, 2014.

[10] J. Terrace, H. Laidlaw, H. E. Liu, S. Stern, and M. J. Freedman. Bringing p2p to the web: Security and privacy in the firecoral network. In *Proceedings of the 8th International Conference on Peer-to-peer Systems*, IPTPS'09, pages 7–7, Berkeley, CA, USA, 2009. USENIX Association.

[11] W3C. WebRTC 1.0: Real-time communication between browsers. `http://w3c.github.io/webrtc-pc/`, 2014.

[12] L. Zhang, F. Zhou, A. Mislove, and R. Sundaram. Maygh: Building a cdn from client web browsers. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 281–294, New York, NY, USA, 2013. ACM.