NoCDN: Scalable Content Delivery Without a Middleman

Junbo Xu
Case Western Reserve University
junbo.xu@case.edu

Michael Rabinovich[*]
Case Western Reserve University
michael.rabinovich@case.edu

ABSTRACT

Today's websites achieve scalability by either deploying their own platforms with sufficient spare capacity or signing up for services from a content delivery network (CDN). This paper investigates another alternative, where a website directly recruits Internet users to contribute their resources to help deliver the site's content. We show that this alternative, which we call NoCDN, can be implemented securely, transparently to the users accessing the site, and without changes to the content itself.

KEYWORDS

Content delivery networks, scalability, performance

1 INTRODUCTION

Scalable content delivery requires a widely distributed server platform with sufficient capacity to absorb any temporary demand surges. Today's content providers utilize such a platform either by deploying it themselves (e.g., Google) or through a third-party content delivery network (CDN) such as Akamai. In this paper, spurred by the prevalence of always-on residential broadband (and impending emergence of ultrabroadband), we explore an alternative for scalable content delivery that follows a shared-economy approach that is increasingly applied in other domains, such as transportation or hospitality services.

We envision that a content provider, say, cnn.com, would recruit volunteers (presumably regular Internet users but potentially any operator of a suitably connected host) who contribute their machines and connectivity to essentially become edge servers for cnn.com. The content provider would compensate the recruits for their service in some fashion (our working assumption follows current CDN's model based on bytes delivered). Multiple content providers vying for the volunteers and volunteers signing up with one or more content providers would create a new content delivery marketplace *directly* between content providers and volunteers, without a middleman in the form of a traditional CDN operator. Shared economy has reduced the costs and brought new opportunities in other domains, and we hope it may bring similar benefits in the content delivery arena.

While seemingly similar to a number of peer-to-peer CDNs that have been proposed in research (e.g., [8, 13, 16]) and used commercially [2], the key difference with our approach is that these systems use peers to augment a CDN platform whereas we eliminate the third-party CDN altogether. We call our system *NoCDN* to highlight this distinction. Although the underlying idea behind NoCDN is simple, realizing it involves a number of challenges.

- **User transparency** Individual content providers may lack the clout of large CDNs to compel users (content consumers) to change or reconfigure their browsers or install some sort of a download manager such as [1]. Thus, NoCDN must ensure complete transparency for users, which in practice restricts any user-side functionality to only what can be fully implemented in standard JavaScript.
- **Content integrity** Peer assistance in current peer-to-peer CDNs often comes with no awareness from the person who owns the resource. E.g., the agreement to act as a peer in Akamai's NetSession is buried inside the agreement to terms of use of various partners, such as Flash player or Autodesk [9]. When a user explicitly signs up to become a peer in NoCDN, there is more danger that an attacker would sign up with an intent of corrupting the content that flows through the attacker-controlled equipment. NoCDN must ensure content integrity despite untrusted peers.
- **Accurate accounting** Since peer compensation depends on the amount of their contribution to content delivery, an unscrupulous peer has an incentive to inflate their contribution they report to the content provider. NoCDN must be able to protect content providers from such behavior.
- **Peering for multiple content providers** To facilitate an open marketplace between potential peers and content providers as envisioned by the NoCDN approach, it is desirable to let a peer sign up with multiple content providers. While not strictly a requirement, avoiding exclusivity in peer-to-content-provider relationships will foster competition and flexibility in the ecosystem.

The rest of the paper describes the design of a system that addresses the above challenges. We further contribute a proof of concept prototype that shows that our design can in fact be implemented within the current technological landscape. Finally, we provide a preliminary evaluation of overheads, which represent the performance penalty the content provider would face in exchange for being able to scale to unpredictable loads without the expense of, and reliance on, a CDN service. These overheads will depend on the content organization and therefore vary among content providers. Based on these, as well on the monetary considerations, a content provider would be able to make a business decision on choosing

[*]This author's work on this project was supported in part by NSF through grant CNS-1647145.

HotWeb'17, October 14, 2017, San Jose / Silicon Valley, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5527-8/17/10…$15.00
https://doi.org/10.1145/3132465.3132476

the best way to achieve scalability. NoCDN expands their decision space by offering a new alternative.

2 RELATED WORK

Numerous approaches (e.g., [8, 16, 18]) have explored the possibility to assist a traditional CDN by adding a peer-to-peer content delivery component, mostly in the context of streaming media distribution. In [10], researchers quantify the potential gains of a CDN-P2P scheme. They find that a hybrid CDN-P2P design can significantly reduce the cost of content distribution. Commercially, Akamai employs NetSession [18] for peer-assisted content delivery and uses statistical inference to guard against billing misrepresentation by peers [4]. Unlike NetSession, our system requires no special software for end-user browser. LiveSky [16], which is deployed by ChinaCache, is mainly designed for video streaming sharing. This system allows a peer to share content with another peer only while the source peer is watching the same content as the recipient peer.

Several proposals explore direct web object sharing among browsers by turning the browser into a web cache. These systems typically require changes to the browsers, usually in the form of a plugin or a co-located proxy cache (e.g., [11, 14]). A significant challenge NoCDN overcomes is to have complete transparency to user, i.e., to require no browser modification.

The two systems most closely related to our work are Maygh [17] and Stickler [12]. Like our system, Maygh proposes a way for a content provider to achieve scalable content delivery through making browsers deliver the content to each other. Further, Maygh achieves this without any changes to browsers, leveraging browser-to-browser communication enabled by Adobe Flash's RTMP or Google's WebRTC. However, in Maygh, content sharing is only possible while both parties are browsing the same web object. NoCDN does not have this restriction.

In independent work, Stickler [12] proposes a technique to verify the integrity of content delivered through an untrusted CDN. We apply essentially similar technique in a different context of ensuring content integrity in the environment without a CDN. Further, Stickler appears to require content providers to drastically change content organization, by replacing regular embedded objects with recursively nested JavaScript scripts ("manifests") that obtain, verify, and incorporate any embedded content. NoCDN only adds wrapper meta-pages, leaving the actual content intact.

3 SYSTEM OVERVIEW

Terminologically, we distinguish a *user machine* (or just "user" for short) that accesses content from a *peer* who sign up with a *content provider* to help with content delivery. A a user may act as a peer as well but that is purely incidental. The content provider operates the *origin site*. We assume regular web content, where a webpage comprises a *container HTML object*, which includes embedded objects (CSS objects, JavaScript code, images, other web pages included into iframes, etc.), some of which may recursively embed further objects, resulting in a tree structure. Due to prevalence of HTTP usage for streaming, our approach directly generalizes to streaming content also.

There are two asynchronous aspects in our system operation. The first concerns peer management: supporting sign-on of new

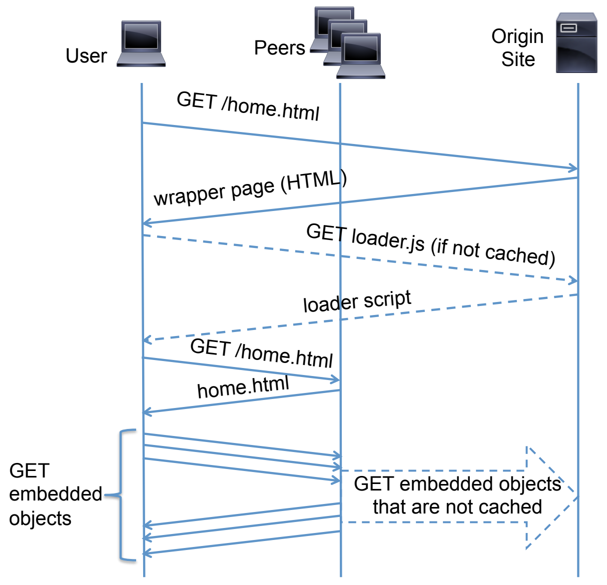

Figure 1: Webpage access in NoCDN

peers and tracking online status of existing peers through heartbeats (which can be implemented simply by periodic downloads of a small object through a peer). During sign-on, the peer host informs the content provider of the port number the peer will be using for delivering this provider's content (see § 6 for details on port selection), and the content provider also records the peer's public IP address observed in the communication. The peer then downloads a package that contains the Apache server and a script for discovering and configuring port forwarding on a NAT box (if any) as well for configuring Apache in the reverse proxy mode with the port specified during the sign-on (see § 6.

The other aspect deals with the actual content delivery. It would seem that we could use classic DNS-based request routing employed by traditional CDNs to direct users to peers for content delivery [6]. However, this would make it impossible for the content provider to verify the integrity of the content delivered by the peer, and to ensure peers' truthfulness when they report the amount of delivered bytes for billing. Thus, users must download content via a (trusted) script provided directly by the origin site. Furthermore, as discussed in § 5, this script cannot be embedded directly in the container object – the container object must also be downloaded by the script.

With these considerations, the workflow of a page download is depicted in Fig. 1 as follows. When a user accesses a URL of a content provider utilizing NoCDN, the content provider returns a *wrapper page*, which (a) lists the IP address of a peer from which to fetch the container object, (b) for every recursively embedded object reachable from the container object, maps the object URL to the IP address of a peer to use for fetching the corresponding object[1], (c) a for every object (including the container object), a cryptographic

[1]While the format of the wrapper page used in our current prototype specifies a single peer for all objects on the page to be accessed, it is straightforward to allow different peers to be used for different objects, which would improve the overall access time by aggregating peer upload bandwidth. We use this more general assumption in this paper except in the evaluation section where we employ our actual prototype.

hash of its content and (d) a *loader script* in JavaScript[2] that fetches the above objects from the peers, verifies them by computing their hashes and comparing them to the hashes listed in the wrapper page, assembles the objects into an integrated webpage and invokes the rendering function on the browser to display the page for the user. A peer acts as a normal reverse proxy when processing user requests – the peer serves the requested object from its cache if available or obtains the object from the origin server, forwards it to the user, and caches it locally for future requests.

This mechanism improves scalability of the origin site because it only has to deliver a small wrapper page, with the loader script eminently cacheable and the rest of the page content fetched from the peer(s). The wrapper page would typically be generated dynamically since its contents reflect the peer selection for the user. However, depending on the peer selection policies employed by the origin site, even the wrapper page may be reused among users and/or allowed to be cached by the user for a certain time.

4 NAT TRAVERSAL

Many peers can be expected to be behind gateways that perform network address translation (so-called NAT boxes), and any peer-to-peer system including NoCDN must provide a way for incoming connections to traverse the NAX boxes. Peer-assisted CDNs (including both Akamai's NetSession and ChinaCache's LiveSky) use the STUN protocol for NAT traversal between the user and the peer. This is a complex solution that requires an external STUN server, imposes delay for establishing port mappings at the time of communication, and relies on NAT boxes to use *consistent endpoint translation* [3]. In contrast, because of an explicit sign-on in our case (rather than an implicit installation of the code as part of an unrelated package such as Flash player [9]), the content provider can worry less about compatibility with every residential environment – incompatible peers can simply be rejected. Consequently, we are able to use a much simpler approach based on UPnP to discover the NAT box and configure permanent port mapping on it if it exists. Our automatic configuration script has been successfully tested on three different UPnP-compliant wifi routers (Netgear wnr2000v4, TP-link tl-wr841n, and Netgear wgr614v10).

5 THE LOADER SCRIPT

The two key functions of the loader script are to ensure content integrity and mitigate potential inflation of the bytes served as reported by a peer to content provider for billing. This section discusses interesting lessons we learned from implementing these functions and optimizing the script performance.

5.1 Content Integrity

Our implementation introduces an extra step into the critical path of page access – the fetching of the wrapper page, which is a "meta-object" that must be obtained before any actual content starts flowing to the browser. A natural question is – why can't this extra step be avoided by including the loader script directly into the container object (which in this case would be fetched from the origin site and therefore would not need to be checked for integrity)?

To answer this question, consider possible locations within the container object where the loader script could be inlined. Let the loader script appear at the top of the container object, right after the list of all the embedded objects and their hashes. The script can fetch and verify all the objects recursively embedded in the container object. But because the browser in this case executes the script before the rest of the container object is loaded, the script would not know how to assemble all the verified objects into an integrated webpage.

If the script is at the bottom of the container object, an external script within the container object may be loaded and executed before the loader script at the bottom runs, and may modify the loader script or redirect the browser to a completely different page.

5.2 Accounting

We assume the peers would be compensated based on the amount of bytes they serve to users. Since peers are unlikely to be thoroughly vetted, it is important to give content providers a mechanism to acquire accurate information about peer utilization. Note that because the user always contacts the origin for the wrapper page, by logging which peers are picked to deliver which objects, the content provider can have an upper bound on the amount of data served by a given peer (modulo any distortion from wrapper page caching, the extent of which is again at the origin site's control). However, this may inflate the payment incurred by the content provider because the user may have unknown fraction of the objects in the browser cache, and never request those objects from the peer.

A more accurate bookkeeping would be possible if the loader script sent usage record to the origin server after loading all objects from the page. But this would double the number of requests fielded by the origin server.

Our solution is as follows. When processing the user request for a page, the origin site includes into the wrapper page a set of secret keys, with each key unique for a given user-peer IP address pair. When the loader script finishes obtaining embedded content from the peers, the loader script generates a usage record for each peer (the amount of bytes downloaded from the peer as the result of the page access), produces an HMAC for this record, along with a nonce to prevent a replay, using the corresponding key, and uploads this record to the peer at the end of the interaction. The peers accumulate these records and periodically upload them to the origin site for payment. Making the keys specific to given user-peer pair complicates key trafficking by unscrupulous users who could otherwise obtain the keys (that would be valid for any peer) and share them with peers for subsequent misuse. Further, collusion of a user with a peer is complicated because such a collusion would succeed only if the colluding peer happens to be selected to serve the colluding user.

5.3 Further Implementation Details

To work with unmodified user browsers, NoCDN must implement the loader script entirely in JavaScript. We use the Cross-Origin Resource Sharing (CORS) [7] mechanism to overcome the same origin

[2]The wrapper page actually includes a link to a separately obtained JavaScript object with the loader script – since this script is generic, this object to be cached, thus minimizing the size of the wrapper page.

policy that would prevent the loader from accessing data downloaded from a peer. Specifically, each peer-served object includes an Access-Control-Allow-Origin header with the origin site's domain, thus allowing a script from this domain to access the contents of the object.

We use JavaScript's Promises API [5] to download objects in parallel. On a tree representing recursively embedded objects on the webpage, our implementation downloads in parallel all top-level subtrees emanating from the container object, and then within each top-level subtree, downloads in parallel all subtrees at the next level down, and so on recursively. Note that no matter how many concurrent downloads the loader tries to execute, the browser would still only open up to six concurrent TCP connections to the peer that will be used by all the downloads. Paralel download improves performance even when using one peer for all the objects, but especially useful when downloading from different peers. Indeed, while a regular CDN can be expected to have well-connected edge servers, NoCDN's peers reside mostly in residential networks with limited upload bandwidth. Downloading from multiple peers in parallel would allow the loader to aggregate bandwidth of several peers.

If the user fails to fetch any web object of the webpage from the peer, due to causes such as an object verification failure or the peer not responding, the loader will fetch the web object from the origin server directly. For simplicity, the loader script simply aborts the current attempt to fetch the page and retrieves the whole webpage from the origin site by using a distinct port number (different from the one normally used to access the website). A different port is needed because otherwise the origin server will return another wrapper page.

6 SIGNING UP WITH MULTIPLE CONTENT PROVIDERS

In the shared economy marketplace envisioned by NoCDN, one can expect a peer to sign up for content delivery with multiple content providers. Our prototype demonstrates how this can be supported by leveraging existing technologies.

As mentioned, we use Apache in the reverse proxy mode. To support sign-on with multiple content providers, we use the reverse proxy mode in combination with virtual hosting. Normally virtual hosts are distinguished by their domain names in the requests' Host header. However, because the wrapper page lists peers by their IP addresses to avoid an extra DNS query, the Host header in the requests arrived at a peer will carry peer's IP address. Thus, we assign different virtual hosts different port numbers, and Apache distinguishes between the virtual hosts accordingly.

We assume that during the sign-on procedure, the peer host will download and run locally a script that uses UPnP to (a) discover a NAT box if any; (b) view existing port mappings on the NAT box; (c) pick an unused port and add it to the port mappings on the NAX box. The script then (a) communicates with the content provider to send the picked port to the content provider and receive the port and hostname of the origin web server for fetching cache misses; and (b) augments the Apache configuration file with a new virtual host section, mapping it to the peer's IP address and selected local port number and specifying the ProxyPass directive to contain the origin's hostname and port number[3].

	Min (s)	Max (s)	Ave (s)
Without NoCDN Total	0.52	0.62	0.54
Fetching	0.17	0.23	0.20
Rendering	0.31	0.40	0.34
With NoCDN Total	1.74	1.92	1.83
Fetching wrapper	0.02	0.03	0.03
Running loader	0.46	0.57	0.51
Rendering	1.24	1.35	1.30

Table 1: NoCDN Overhead (delay components may not sum up to totals due to rounding)

7 PERFORMANCE

We assess NoCDN performance from two angles: the overhead imposed on the user due to extra processing and the end-to-end effect on user experience. We use a complex real home page of a real high-volume website for our experiments. Specifically, we clone the home page of 360.cn – a popular website ranked 24th in the Alexa global top-500 list. The home page's container object is 271KB and it has 135 embedded web objects totalling 2.5MB. We also have studied how the amount of local processing is affected by page structure, using synthetic pages whose structure we can control. Refer to [15] for these experiments.

7.1 Overheads

Our first goal is to assess the overhead of NoCDN, which comes from two main sources: the need to fetch the wrapper page prior to fetching any actual content, and the execution of the loader script. We compare the time it takes to access this page in two scenarios: the current situation (browser just access the container object plus all the embedded objects natively) and the NoCDN scenario (browser accesses the wrapper page, then executes the loader to fetch container object, and embedded objects). All the components in both scenarios are cached by the browser, so we factor out any network delays and focus just on the overhead of NoCDN. We access our test page using a laptop with Intel i5 6267u (base 2.9GHz) CPU with 8GB RAM running Chrome Version 55.0.2883.95 (64-bit) browser. We run the experiment eight times under each scenario.

Table 1 shows the total time it takes the browser to access and render the page with and without NoCDN, as well as the main components contributing to the total access time. We use Chrome's DevTools to separately obtain the time to fetch all the content in the page for the case without NoCDN, and the wrapper page for the case with NoCDN. For the NoCDN access, we further record the execution time of the loader script, which includes the time to fetch the actual content. We attribute the remaining time (total time minus fetch time in the case without NoCDN and total time minus

[3]Our current prototype automatically configures port forwarding to demonstrate the critical part of this script, but leaves port selection and Apache configuration augmentation to manual manipulation.

the wrapper fetch minus the loader execution time) to rendering time.

The results show a substantial performance penalty due to NoCDN. Overall, local processing takes 3.4 times longer when accessing our test page through NoCDN. Note, however, that the origin server using NoCDN can always decide whether to return, in response to a given request, a wrapper (thus redirecting the browser to peers) or a regular page (thus serving all content directly as normal). In particular, the content provider can redirect users to NoCDN peers only when under heavy load, when it would not otherwise be able to satisfy all the demand or when its own response time degradation would exceed the NoCDN overhead. See § 7.3 for more thoughts on content provider strategies.

Another interesting observation is that most overhead comes not from executing the loader script but from rendering the content. We speculate the reason has something to do with the data URI scheme we use in the integrated page that inlines all the embedded content into the container object. Considering most of the content web objects are images, and the data URI scheme of binary image data uses base64 character encoding, rendering the integrated object would entail a large amount of decoding for the browser. If it was possible to integrate binary data directly, we believe NoCDN overhead would decrease substantially.

7.2 End-to-end Performance

We now turn to evaluating the end-to-end performance of a fully deployed prototype of NoCDN. We consider two extreme scenarios. First, we consider a situation where the user and peer are in close proximity to each other (in terms of network latency) and connected with a high-bandwidth network path as compared to the origin server. This scenario, which we call "nearby peer" case, is the most beneficial to our system. Second, we consider a situation where the user and the peer are still close to each other geographically but the connection has higher latency and lower bandwidth. This situation, to which we refer as "remote peer", is the worst case for NoCDN.

To emulate the two scenarios, we set the peer (Intel i7 4700MQ 2.4GHz CPU with 8GB memory) at one of the authors' home network In Columbus, OH and the origin server (a virtual machine with Intel Xeon 3.3 GHz CPU with 1GB memory) at a well-connected datacenter in Oregon that is part of Amazon's AWS cloud (average measured bandwidth is 70Mbps for both upload and download[4]).

In the nearby peer scenario, the user machine (Intel i5 6267u 2.9GHz CPU with 8GB memory) is co-located with the peer in the same residential local area network. In this case, both the user and peer have the same connectivity to the Internet (up to 1Mbps upload/15Mbps download according to the Internet service plan but measured to be 2.3Mbps upload/20Mbps download) and round-trip latency to the origin server (97ms) and a much higher-capacity connection between themselves (50Mbps and 3ms latency).

In the remote peer scenario, we move the same user machine to a well-connected network at Ohio State University. The user's measured connectivity to the Internet now is 29 Mbps upload and 86Mbps download, the latency to the origin server is 68ms latency,

[4]The bandwidth of all Internet links is measured using an online service http://beta.speedtest.net/. The bandwidth within the home LAN is measured using a large file transfer.

	Direct fetch (s)	Peer cache hit (s)	Peer cache miss (s)
Nearby Peer Total	3.74	2.67	6.82
Fetching wrapper		0.23	0.24
Running loader		1.18	5.36
Rendering		1.27	1.23
Remote Peer Total	2.03	11.31	12.82
Fetching wrapper		0.15	0.16
Running loader		9.93	11.41
Rendering		1.24	1.25

Table 2: End-to-end time to access the test page (delay components may not sum up to totals due to rounding)

and the latency to the peer 36 ms. Thus, the bandwidth of the path for downloading content from the origin server is likely to be an order of magnitude higher than from the peer (limited by the 2.3Mbps peer upload bandwidth).

In both scenarios, we host the cloned 360.cn home page (with all its embedded objects) on our AWS instance, which runs both the web server and the authoritative DNS server for our domain. We measure how long it takes the user browser to fetch and render the web page with the NoCDN setup in two cases – when all content served by the peer is available in the peer cache ("cache hit") and when the peer cache is disabled ("cache miss"). In all cases, we ensure the browser cache is not utilized for any object fetches.

Table 2 summarizes our results. When the user is well connected to the peer (the "nearby peer" row), it enjoys a significant reduction in access time over the direct page access from the web site assuming the peer caches the content. A cache miss does incur response time penalty, due to the execution of the loader script and especially the longer rendering time of NoCDN. However, because a large part of the delay is due to the transmission of content from the origin to the home network (present in all cases), the penalty is relatively modest.

The situation is dramatically different with a remote peer. In this case, the limited-capacity network path between the peer and the user is the bottleneck dominating the NoCDN download time on either peer cache hit or miss, and this bottleneck is absent when the user accesses the page directly from the origin. Thus, access time through NoCDN is similar in the peer cache hit and miss cases, but is over XXX times higher with NoCDN than with the direct download. With a poorly connected peer in place of a well connected origin site obviously degrades the performance of an individual access, and NoCDN's main benefit in this case is scalability to a large volume of accesses. This result also highlights the importance of being able to aggregate bandwidth of multiple peers by downloading different embedded objects from different peers in parallel. As mentioned, this extension is straightforward to implement but is currently not supported by our prototype.

7.3 Discussion

Our performance results suggest a general content delivery strategy for content provider along the following lines.

- The origin site should monitor the throughput of content transfer to users and peers and record this information as input for its content delivery decisions. A subtle point is that from its interactions with peers the origin can only learn peers' download bandwidth but not the upload that is relevant to the user performance. This suggests that the loader script should include into the user's usage report not just the number of bytes for accounting but also the observed performance of downloading from the peer.
- During periods of low demand, when the origin site is able to serve all the demand, serve all requests directly, without engaging NoCDN.
- During periods of high demand, serve well-connected users preferentially directly and serve poorly connected users through NoCDN peers (to preserve capacity for serving well-connected users).
- If the demand is such that even well-connected users must be offloaded, use well-connected peers preferentially for well-connected users.
- "Pool" bandwidth of poorly connected peers by assigning different objects for download from different peers, since the NoCDN loader will often perform these downloads in parallel (depending on the structure of the page).

The investigation of content delivery strategies with NoCDN is a subject for future work.

8 LIMITATIONS

NoCDN will not obviate traditional CDNs. First, while the overheads discussed earlier could likely be reduced by tuning of the implementation and evolution of JavaScript API, there will always be some performance cost that traditional CDNs do not face. Second, NoCDN can only outsource delivery of objects whose URLs are easily parseable from the content of the embedding objects. In particular, this excludes objects whose URLs are constructed dynamically by a JavaScript script or buried deep inside JavasScript variables. Hence, although we believe this limitation is not going to be practically significant for most websites, the scope of content that NoCDN can deliver is fundamentally narrower than with traditional CDNs. We consider NoCDS a "poor man's approach" to scalable content delivery, which trades these limitations for removing the cost of and reliance on a middleman between providers and consumers of content.

From a peer's perspective, in addition to contributing its resources to deliver content for content providers, the peer opens a permanent port to externally initiated communication for each provider the peer signs up with. Fundamentally, external ports can only decrease security of the peer's network. NoCDN's marketplace will presumably arrive at prices that would compensate the peers for added risks.

9 CONCLUSION

This paper investigates a new approach to scalable content delivery that does not require a website to sign up for CDN services. Instead, a website directly recruits Internet users to contribute their resources to help deliver the site's content, and compensates the participants based on the amount of data they help to deliver. This creates a new open marketplace *directly* between content providers and Internet users, without a middleman in the form of a traditional CDN operator. We show that this approach, which we call NoCDN, can be implemented securely (despite untrusted recruits), transparently to the users accessing the site (without any configuration or modification of the browser), and without changes to the content itself. A preliminary performance evaluation shows that, depending on relative connectivity of the browser, origin server, and the peers, and whether the peer has the requested content in its cache or has to fetch it from the origin, the performance of a stand-alone user access may improve or greatly diminish compared to accessing the origin server directly. While the latter can be viewed as a reasonable cost for being able to scale to a large volume of accesses, future work will develop peer selection strategies to minimize penalized user accesses.

All code comprising NoCDN is available at https://github.com/aloell/NoCDN.

REFERENCES

[1] Akamai Download Manager. https://www.akamai.com/us/en/products/media-delivery/download-manager-overview.jsp (Accessed on 8/13/2017).
[2] Akamai NetSession interface overview. https://www.akamai.com/us/en/products/media-delivery/netsession-interface-overview.jsp (Accessed on 8/09/2017).
[3] Hole punching (networking). https://en.wikipedia.org/wiki/Hole_punching_(networking). Accessed on 08/04/2017.
[4] P. Aditya, M. Zhao, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, and B. Wishon. Reliable client accounting for P2P-infrastructure hybrids. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012.
[5] J. Archibald. JavaScript Promises: an Introduction. https://developers.google.com/web/fundamentals/getting-started/primers/promises; Accessed on 08/05/2017.
[6] A. Barbir, B. Cain, R. Nair, and O. Spatscheck. Known Content Network (CN) Request-Routing Mechanisms. RFC 3568 (Informational), July 2003.
[7] Cross-origin resource sharing. W3C Recommendation, 16 January 2014. Available at https://www.w3.org/TR/cors/.
[8] M. El Dick, E. Pacitti, and B. Kemme. Flower-CDN: a hybrid P2P overlay for efficient query processing in CDN. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT)*, pages 427–438. ACM, 2009.
[9] S. Hanselman. CSI: My Computer - What is netsession_win.exe from Akamai and how did it get on my system? Available at goo.gl/wgM1CK (Accessed on 08/03/2017), 2011.
[10] C. Huang, A. Wang, J. Li, and K. W. Ross. Understanding hybrid CDN-P2P: why limelight needs its own Red Swoosh. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 75–80. ACM, 2008.
[11] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC)*, pages 213–222. ACM, 2002.
[12] A. Levy, H. Corrigan-Gibbs, and D. Boneh. Stickler: Defending against malicious content distribution networks in an unmodified browser. *IEEE Security & Privacy*, 14(2):22–28, 2016.
[13] S. Seyyedi and B. Akbari. Hybrid CDN-P2P architectures for live video streaming: Comparative study of connected and unconnected meshes. In *Int. Symp. on Computer Networks and Distributed Systems (CNDS)*, pages 175–180. IEEE, 2011.
[14] 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 Int. Workshop on Peer-to-Peer Systems (IPTPS)*, page 7, 2009.
[15] J. Xu. Scalable content delivery without a middleman. Master's thesis, Case Western Reserve University, 2017.
[16] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li. Livesky: Enhancing CDN with p2p. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 6(3):16, 2010.
[17] 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*, pages 281–294. ACM, 2013.
[18] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec. Peer-assisted content distribution in Akamai NetSession. In *Proceedings of the Internet Measurement Conference (IMC)*, pages 31–42. ACM, 2013.