Maygh: Building a CDN from client web browsers

Liang Zhang Fangfei Zhou Alan Mislove Ravi Sundaram
Northeastern University
{liang,youyou,amislove,koods}@ccs.neu.edu

Abstract

Over the past two decades, the web has provided dramatic improvements in the ease of sharing content. Unfortunately, the costs of distributing this content are largely incurred by web site operators; popular web sites are required to make substantial monetary investments in serving infrastructure or cloud computing resources—or must pay other organizations (e.g., content distribution networks)—to help serve content. Previous approaches to offloading some of the distribution costs onto end users have relied on client-side software or web browser plug-ins, providing poor user incentives and dramatically limiting their scope in practice.

In this paper, we present Maygh, a system that builds a content distribution network from client web browsers, without the need for additional plug-ins or client-side software. The result is an organically scalable system that distributes the cost of serving web content across the users of a web site. Through simulations based on real-world access logs from Etsy (a large e-commerce web site that is the 50th most popular web site in the U.S.), microbenchmarks, and a small-scale deployment, we demonstrate that Maygh provides substantial savings to site operators, imposes only modest costs on clients, and can be deployed on the web sites and browsers of today. In fact, if Maygh was deployed to Etsy, it would reduce network bandwidth due to static content by 75% and require only a single coordinating server.

Categories and Subject Descriptors C.2.4 [*Performance of Systems*]: Distributed Systems—Distributed applications; C.2.0 [*Computer-Communication Networks*]: Network Architecture and Design—Distributed networks

General Terms Algorithms, Design, Performance, Security

Keywords Content distribution network, distributed, JavaScript

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

1. Introduction

Over the past two decades, the web has enabled content sharing at massive scale. Unfortunately, the architecture of the web places substantial monetary burden on the web site operator, who is required to supply the resources (serving infrastructure and network bandwidth) necessary to serve content to each requesting client. Spreading the distribution costs among the users—the idea that underlies the success of peer-to-peer systems such as BitTorrent [9]—is difficult over the web, as the web was designed with a fundamentally client–server architecture. This situation exists despite the fact that significant amounts of web content are now generated by end users at the edge of the network, fueled by the popularity of online social networks, web-based video sites, and the ease of content creation via digital cameras and smartphones. Web site operators who wish to serve a large number of users are forced to make substantial investments in serving infrastructure or cloud computing resources, or pay content distribution networks (CDNs) such as Akamai or Limelight, to serve content.

Recent approaches have worked towards overcoming these limitations, aiming to allow end users to help the web site operator distribute the static objects (e.g., images, videos, SWF) that make up web pages. Examples include Akamai's NetSession [1, 4], Firecoral [50], FlowerCDN [13], BuddyWeb [55], and Web2Peer [39]. All of these systems are built either using browser plug-ins that the user must install, or client-side software that the user must download and run. As a result, the set of users who can take advantage of these systems is limited to those that download the software or plug-ins; clients have little incentive to install the plug-in or software, and clients without it are served using existing techniques. With the most popular [32] of all FireFox plug-ins (Adblock Plus) being installed by only 4.2%[1] of FireFox users, such techniques are still likely to provide only modest gains in practice.[2]

[1] AdBlock Plus reports approximately 15 million active users [3], out of approximately 350 million Firefox users [33].

[2] It is worth noting that Akamai's NetSession reportedly has over 24 million users [1] (the NetSession software is typically bundled with video streaming software). However, because these users run Akamai's software, web operators must still pay Akamai to use these clients to serve content.

In this paper, we build and evaluate Maygh,[3] a system that automatically builds a CDN from visiting users' unmodified web browsers. With Maygh, users visiting the operator's web site automatically assist in serving static content to others for the duration of their visit; once they leave the site, this assistance ceases. Maygh requires no client-side changes and only minor modifications to the operator's site, and can be deployed on the web sites and web browsers of today. One might expect that peer-assisted content distribution would not work well when implemented only using web technologies, as such an implementation necessarily results in short average session times, small cache sizes, high churn, a limited computation model, and limited network stack access. We demonstrate that even in this challenging environment, a useful browser-assisted CDN can be constructed and deployed today, allowing operators who wish to distribute significant amounts of static content to spread the costs of distribution across their users.

Maygh enables web browsers to assist in content distribution using two techniques. First, Maygh uses the storage APIs [56, 59] supported by modern browsers to persistently store content on end user machines. Second, Maygh uses newly-available browser-to-browser communication techniques [40, 57] to enable direct message exchange with other browsers. Maygh uses centralized coordinators to track the static content stored in browsers, serving as a directory for finding content.

We evaluate Maygh using five approaches. First, we demonstrate the potential for Maygh by surveying the fraction of web traffic that is static content; we find that despite large amounts of dynamically generated content, static elements like images, videos, and SWF (Flash objects) still represent a significant fraction of web bytes and could be served via Maygh. Second, using microbenchmarks, we show that the client-side performance of Maygh is acceptable, and that Maygh incurs little network and storage overhead. Third, we show that the central coordinators necessary to support Maygh can be scaled to support many thousands of content requests per second, easily handling the traffic that large web sites receive. Fourth, we demonstrate that Maygh can dramatically reduce the network traffic at web site operators by simulating a Maygh deployment at scale using real-world Akamai image access traces from Etsy [14] (a large e-commerce site that is currently the 50th most popular web site in the U.S. [5]). Our results show that Maygh would reduce the 95th-percentile bandwidth required to distribute image content by 75% while only requiring one four-core coordinator machine. Fifth, through a prototype deployment of Maygh within our department, we demonstrate that Maygh can be easily integrated into existing web sites and is practical on the web browsers and web sites of today.

The remainder of this paper is organized as follows: Section 2 explores the potential of Maygh by examining the fraction of static content on sites today. Section 3 details the design of Maygh, and Section 4 discusses the security/privacy implications. Section 5 evaluates Maygh, Section 6 details related work, and Section 7 concludes.

[3] "Maygh" is a rough phonetic translation of a Hindi word for "cloud."

Content Type	% Requests	% Bytes	% Cacheable
Image	70.5	40.3	85.7
JavaScript	13.1	29.0	84.8
HTML	10.7	19.9	30.1
CSS	3.5	8.7	86.5
Flash	0.9	1.3	96.0
Other	1.3	1.0	45.7
Overall	100	100	74.2

Table 1. Breakdown of browsing trace from the top 100 Alexa web sites. Cacheable refers to the fraction of bytes that are cacheable according to the HTTP headers.

2. Maygh potential

We begin by examining the potential for a system like Maygh that is able to assist in the delivery of static content. Over the past few years, dynamically generated web pages have become more prevalent. For example, advertisements are often targeted, online social networking sites customize pages for each user, and news web sites often provide suggested articles based on each user's browsing profile. While most of these dynamically generated pages cannot be served by end users, a significant faction of resources embedded in the page (such as images, videos, and SWF) represent cacheable static objects. We now conduct a brief experiment to measure the fraction of bytes that such static content typically represents, suggesting the potential for Maygh to help distribute static content.

We conduct a small web browsing experiment. We first select the top 100 websites from Alexa's [6] ranking (collectively, these sites account for 37.3% of all web page views [6]). For each of these sites, we use a web browser under automated control to simulate a browsing user, and we record all of the HTTP requests and responses. Starting with the root page of the site, the browser randomly selects a link on the page that stays on the same domain. The browser repeats this step five times, effectively simulating a random walk on the site of length five. To avoid any effects of personalization, we remove all browser cookies between requests. Finally, we repeat the experiment five times with different random seeds.

The result is shown at Table 1, aggregated across all sites by content type. We observe that 74.2% of the bytes requested are marked as cacheable based on the `Cache-Control` HTTP header.[4] This result, while by no means exhaustive, is in-line with other studies [23, 24] and suggests that reducing the bandwidth required to distribute static con-

[4] We only consider content that contains no `Cache-Control` header or is marked `Cache-Control: public`, as recommended by RFC 2616 [16].

tent is likely to provide significant savings to web site operators in practice.

3. Maygh design

We now describe the design of Maygh. At a high level, Maygh provides an alternate approach to building a CDN for distributing static web content like images, videos, and SWF objects. Maygh relieves some of the load of serving content from the web site operator (hereafter referred to simply as the *operator*). Maygh works by allowing the client web browsers visiting the operator's site to distribute the static content to other browsers. Maygh is compatible with dynamic web sites, as it can be used to load the static content objects that dynamic sites use to build their pages.

Maygh consists of two components: a centralized set of *coordinators*, run by the operator, and the *Maygh client code*, implemented in JavaScript that is executed in each client's web browser.

3.1 Web browser building blocks

In the design of Maygh, we use a number of technologies now present in web browsers. We assume that users have a web browser that supports JavaScript, and that users have JavaScript enabled (recent studies show that these assumptions hold the vast majority of web users [61]).

3.1.1 Persistent storage

To store content, the client-side Maygh JavaScript uses the storage APIs [56, 59] supported by modern browsers. In brief, these APIs allows a web site to store persistent data on the user's disk. The interfaces are similar to cookie storage, in that they present a key/value interface and are persistent across sessions, but are larger in size and can be programmatically accessed via JavaScript. When a user fetches content in Maygh, the JavaScript places this content into the browser's storage, treating the storage as a least-recently-used cache.

3.1.2 Direct browser–browser communication

Maygh is designed to use either of two existing protocols that allow two web browsers to establish a direct connection between each other. These two protocols, described below, are largely similar and are intended to allow video and audio to be exchanged in a peer-to-peer (p2p) fashion between web browsers. They also allow application-level messages to be sent; it is this messaging facility that Maygh leverages to communicate between clients. Both protocols are built using UDP and support network address translation (NAT) traversal using STUN[5] with assistance from the server.

Both protocols are built around a *protocol server* that assists in setting up direct connections between browsers. Each web browser selects a unique *peer-id* that other browsers can connect to with the assistance of the protocol server (the protocol itself handles the mapping from peer-id to IP address). In Maygh, the coordinator is implemented to also serve as a protocol server.

RTMFP (Real Time Media Flow Protocol [40]) is a closed-source protocol that is built into the ubiquitous Flash plug-in.[6] RTMFP enables direct communication between Flash instances. All RTMFP packets are encrypted (each client generates a key pair), and RTMFP implements flow control and reliable delivery.

WebRTC (Web Real-Time Communications [57]) is an open-source standard that is beginning to see adoption in popular web browsers.[7] WebRTC uses Interactive Connectivity Establishment [37] for communicating with the protocol server and setting up peer-to-peer communication channels. WebRTC can be accessed using a browser-provided JavaScript library.

3.2 Model, interaction, and protocol

We assume that the content that Maygh serves is always available from the operator as normal, in case Maygh is unable to serve the content. We also assume that all content to be distributed by Maygh is named by its content-hash;[8] since we are focusing on static content, this can be accomplished in an automated fashion by the operator before publishing content.

To use Maygh, the operator runs one or more coordinators and includes the Maygh client code (a JavaScript library) in its pages. The client code automatically connects the client to a coordinator and enables the client to fetch content from other clients later. Thus, the use of Maygh is transparent to users, who only observe web pages being loaded as usual. Users only participate in Maygh on a given web site as long as the user has a browser tab open to the site; once the user closes the browser tab, the participation ceases. Because the coordinator is run by the web site operator, the operator still receives a record of all content views and can use this information—as they do today—to target advertisements and make content recommendations.

3.2.1 Client-to-coordinator protocol

The Maygh client code communicates with the coordinator over either RTMFP or WebRTC.

Initial connection After the web page is loaded, the client initiates a connection with the coordinator. Once the RTMFP/WebRTC handshake is complete, the client informs

[5] In brief, Session Traversal Utilities for NAT (STUN) [38] enables two machines, each behind NATs, to establish a direct communication channel over UDP.

[6] Adobe claims [19] that the Flash player is installed on over 99% of desktop web browsers. RTMFP has been included in Flash since version 10.0 (released in 2008).

[7] WebRTC is available starting in Google Chrome 23 and Firefox 18.

[8] We avoid the worry of hash conflicts by using a hash function with a sufficiently large output space (e.g., SHA-256).

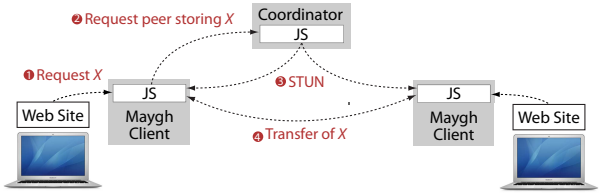

Figure 1. Overview of how content is delivered in Maygh, implemented in JavaScript (JS). A client requesting content is connected to another client storing the content with the coordinator's help. The content is transferred directly between clients.

the coordinator of the content that it is storing locally by sending a Maygh `update` message. This message contains a list of content-hashes. The client and the coordinator then keep the connection open via keep-alives.

Content request When a client wishes to fetch content, it sends a `lookup` message to the coordinator containing the content-hash of the content requested and the peer-ids of any other clients the client is currently connected to. The coordinator responds with a `lookup-response` message, containing a peer-id that is online and is currently storing that piece of content.[9] If there are many other clients storing the requested content, the coordinator attempts to select other clients that the requesting client is already connected to, or that are close to the requesting client (e.g., by using a geo-IP database).

Connect to another client When a client wishes to actually fetch content from another client, it requests a connection to the specified peer-id. The functionality of establishing the direct connection is handled by RTMFP or WebRTC with the coordinator functioning as a protocol server, and the client is informed when the direct client-to-client connection is available for use. In brief, if either of the clients is not behind a NAT, the connection can be made directly without coordinator assistance. If not, the connection is established using STUN, with the coordinator assisting.

New content stored When a client has a new object stored locally, it informs the coordinator by sending another `update` message. This message contains the content-hashes of any new objects stored, along with the content-hashes of any objects that are no longer stored.

3.2.2 Client-to-client protocol

The protocol between clients also happens over either RTMFP or WebRTC. Once a direct connection is established using one of these protocols, the client requesting the connection sends a `fetch` message containing the content-hash

[9] Clients are only able to communicate with others that support the same protocol (RTMFP or WebRTC); the coordinator ensures that only such clients are returned in the `lookup-response`.

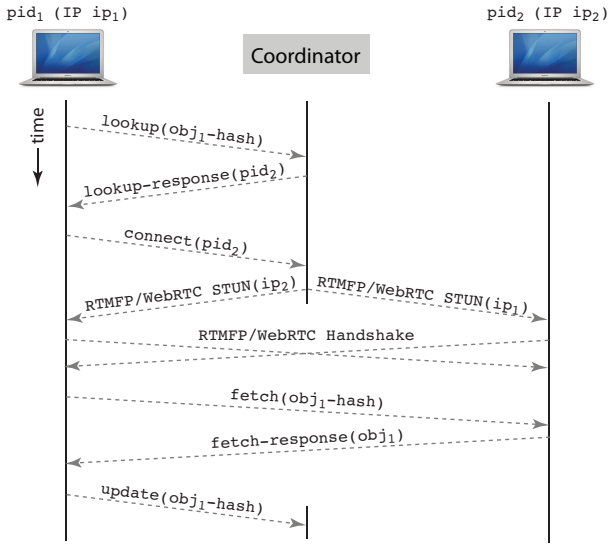

Figure 2. Maygh messages sent when fetching an object in Maygh between two clients (peer-ids $\texttt{pid}_1$ and $\texttt{pid}_2$). $\texttt{pid}_1$ requests a peer storing content-hash $\texttt{obj-hash}_1$, and is given $\texttt{pid}_2$. The two clients then connect directly (with the coordinator's assistance, using STUN if needed) to transfer the object.

that it wishes to download. The other client responds with a `fetch-response` message containing the corresponding content. A timeline of the messages exchanged in Maygh is shown in Figure 2.

3.3 Maygh client

The client-side code that implements Maygh is written in JavaScript. This code manages content and makes Maygh easy to integrate into an existing web site. With RMTFP, the Maygh code also has a small Flash object that conducts all communication (the Flash object itself is hidden, so the use of Maygh does not alter the layout or appearance of the operator's site). To deploy Maygh, the operator includes a reference to the Maygh JavaScript in their web page, which causes the Maygh JavaScript (as well as the Flash object, in the case of RTMFP) to be loaded and run.

The Maygh JavaScript code exports an API that the operator can use, shown below:

- `connect(coordinator)` Called when the web page is first loaded. Causes the Maygh code to connect to the given coordinator and establish an open session.
- `load(content_hash, id)` Called when the client code wishes to load an object. Causes Maygh to request the address of another client who is currently storing the given object, and then connect to that client, download the object, and verify its content-hash. If no peer has the content, if it cannot connect to the peer, if the content-hash is incorrect, or if downloading from the peer fails, the Maygh code loads content from the operator's web

site as normal. The id refers to the DOM id of the object; Maygh will display the object once loaded.

In brief, the operator can load static content via Maygh by slightly modifying its existing pages. For example, if the operator loads an image using the HTML

```
<img id="-id-" src="-src-">
```

it can instead load the image with Maygh using

```
<img id="-id-">
<script type="text/javascript">
 maygh.load("-hash-", "-id-");
</script>
```

where -hash- is the content-hash of the image. Once loaded, Maygh will display the image as normal. Similar techniques can be used to load other static content objects like videos, SWF objects, and CSS.

The Maygh library is configured to maintain only a single connection to each site's coordinator, even if the user has multiple browser tabs open to the same site (essentially, each site maintains a single connection with each browser, regardless of the number of tabs open). With WebRTC, this is done automatically using shared WebWorkers [58]. With RMTFP, this is accomplished using Flash's LocalConnection, where the multiple tabs from a single site can communicate with a single Flash instance and share a single coordinator connection.

It is important to note that many of the content-loading optimizations that are present in the web today are compatible with Maygh. For example, many web sites pre-fetch images that the user is likely to view next, or fetch images using AJAX instead of HTML <img> tags. Both of these can be easily modified to load the content with Maygh, replacing the existing loading logic with a call to Maygh's load function. Additionally, Maygh connects to other clients and can load objects in parallel, thereby avoiding additional latency on pages that have many objects loaded with Maygh.

3.4 Maygh coordinator

Maygh uses one or more centralized coordinators run by the web site operator. The coordinators have two functions: serving as a directory for finding other clients storing content, and serving as a protocol server for RTMFP or WebRTC.

Recall that once clients are connected to a coordinator, they inform the coordinator of any locally stored content (identified by content-hashes). The coordinator maintains this data in two data structures: First, the coordinator maintains a *content location map*, which maps each piece of content (identified by its content-hash) to the list of online peer-ids storing that content. Second, the coordinator maintains a *client map*, which maps each peer-id to the list of content that it is storing.

Maintaining these two maps allows the coordinator to ensure that the content location map contains only references to clients who are online. Whenever a client goes offline (either explicitly or through a timeout of the keep-alive messages), the coordinator determines the list of content that client was storing using the client map, and then purges that client's record from each of the entries in the content location map.

The coordinator also keeps track of the number of content bytes each client has downloaded and uploaded for a configurable time period (e.g., each week). Doing so allows the coordinator to ensure that no client is asked to upload more than a configurable fraction upload_ratio of what it has downloaded. Maygh also provides a global upper bound upload_max on the total amount of content that any client is asked to upload, regardless of how much it has downloaded. For example, the operator could set upload_ratio to 1 and upload_max to 10 MB per week, ensuring that no client has to upload more than the amount it has downloaded, and never more than 10 MB each week.

3.5 Multiple coordinators

One of our goals is to allow web sites using Maygh to scale to large numbers of users. In such a deployment, it is likely that a single coordinator will quickly become a performance bottleneck. We could trivially allow the operator to run multiple coordinators in parallel, but clients who are connected to different coordinators would be unable to fetch content from each other (as each coordinator would not be aware of the content stored on clients connected to the other connectors). This has the potential to preclude much of the potential savings of Maygh from being realized.

Instead, we enable multiple coordinators to work in tandem and allow clients connected to different coordinators to exchange content. We assume that the operator has deployed a set of N coordinators in a single location (e.g., the operator's datacenter), and has a load balancing algorithm where the clients are each randomly assigned to one of these N coordinators. We also assume that the coordinators are aware of each other and maintain open connections to each other.

Recall from above that the single coordinator maintains two data structures: the client map and the content location map. We keep the client map exactly the same as before (each of our N coordinators maintains a client map containing entries for its clients). However, we distribute the content location map across all of the coordinators using consistent hashing [25], effectively forming a one-hop distributed hash table [20]. Each coordinator selects a random coordinator-id from the same hash space as the content-hashes (e.g., by running the same hash function on its IP or MAC address). Each coordinator is then responsible for storing the content location map entries for which its coordinator-id is numerically closest to the content-hash key. Finally, for each peer-id in the content location map, we also store the coordinator-id that the peer is connected to. A diagram showing this distribution is shown in Figure 3.

Distributing the state of the coordinator in this manner allows for the multiple coordinators to have favorable scaling properties. Consider the operations that are necessary when

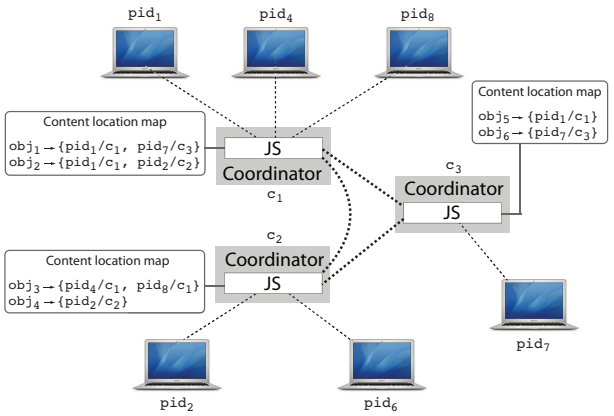

Figure 3. Overview of how multiple coordinators work together Maygh. The mapping from objects to list of peers is distributed using consistent hashing, and the coordinator each peer is attached to is also stored in this list. The maximum number of lookups a coordinator must do to satisfy a request is two: One to determine a peer storing the requested item, and another to contact the coordinator that peer is attached to.

a client issues a `lookup` message: The client's coordinator receives that message, and can immediately determine the coordinator who is storing that entry in the content location map, through the use of consistent hashing. The coordinator requests that entry from the remote coordinator, caches the result, and then returns the list of peer-ids to the client. When the client requests to be connected to another client, the coordinator uses the cached result to determine the coordinator to whom the remote client is connected, and then communicates with that coordinator to allow the two clients to directly connect. In fact, the establishment of the direct client-to-client connection proceeds exactly as in the single-coordinator case, except that each coordinator only sends STUN packets to its own client.

As a result, the coordinator is only required to send at most two messages to other coordinators in response to a `lookup` message, regardless of the number of coordinators that exist. As we demonstrate in the evaluation, this allows the performance to scale close to linearly with the number of coordinators, and enables Maygh to be used on sites that serve many thousands of requests per second.

4. Security, privacy, and impact on users

The design of Maygh changes many of the properties of the web, raising a number of concerns about security, privacy, and the impact that Maygh will have on users. We now address these questions, leading to a discussion of the deployments where Maygh is most appropriate.

4.1 Security

We first examine how Maygh handles malicious users. There are two primary concerns: malicious users might attempt to serve forged content to other users, or might attempt to perform denial-of-service (DoS) attacks by overloading the coordinator or violating the protocol.

In order to detect forged content, all content in Maygh is self-certifying [47], since it is identified by its content-hash (see Section 3.2). When a client receives content, the client immediately compares the hash of the content with its identifier. This enables the client to immediately detect and purge forged content; if forged content is detected, the client then downloads the content from the operator as normal.

In order to address users who attempt to violate the Maygh protocol (e.g., by claiming to have content stored locally that they later turn out not to have), Maygh uses similar techniques that are in-use by such sites today: Operators can block accounts, IP addresses, or subnets where malicious behavior is observed [45]. Additionally, since the coordinator is under the control of the operator, existing defenses against DDoS attacks can be deployed to protect the coordinator similar to the operator's web servers [27, 31].

4.2 Privacy

Next, we examine the privacy implications of Maygh. We first note that the Maygh coordinator tracks the content stored in each user's browser while the user is online, which could lead to privacy concerns. However, we note that the Maygh coordinator is run by the web site operator, who, today, is already able to log access requests and track downloaded content.

In Maygh, clients do receive information about *views* of content by other users (i.e., when a client sends or receives a `fetch` message for a piece of stored content, it can determine the IP address of the other user viewing that content). As a result, there may be sensitive content for which Maygh is inappropriate. In such cases, the operator can disable loading such content via Maygh, or allow the user to do so using privacy controls. Alternatively, the operator can add background requests to random pieces of content (sometimes referred to as *cover traffic* [17]), or can place content on clients before they have viewed it, in order to provide plausible deniability. Regardless, the privacy implications of Maygh, where users can sometimes infer information on the views of others, are similar to a number of deployed peer-assisted content distribution systems including Akamai's NetSession [4] (used by services like the NFL's online streaming [30]), Flower-CDN [13], Firecoral [50], and numerous IPTV systems [21] such as PPLive [43].

Additionally, it is difficult for an attacker to determine the contents of a specific users' LocalStorage in Maygh. This is because, for each `lookup` request a user issues, the coordinator returns a single remote client that the coordinator believes is close to the requestor. As a result, the choice of

which client is returned is the coordinator's, and the coordinator is able to apply randomization or other techniques to limit the ability for the attacker to target specific users. Accounts or IP addresses that issue spurious requests can be banned or blocked in the same manner as they are on today's web sites [45].

In order to ensure the users can only access content they are authorized to view, the coordinator can authenticate content requests in the same manner as existing web servers. If a client issues a `lookup` request for a content-hash it is unauthorized to view, the coordinator can simply deny the request. Moreover, using content-hashes for naming may enable Maygh to skip this authentication for many applications, as users can only request content if they know its hash. In fact, this is precisely the semantics that many web sites with sensitive content use today. For example, on Flickr, the URLs of images are obfuscated through the use of a per-image "secret" (analogous to our content-hash), but anyone who possesses the secret can construct the URL and download the image.

4.3 Impact on users

When deployed, Maygh reduces the bandwidth costs imposed on the web site operator. Our hope is that lowering these costs will both reduce the need for operators to rely on advertising revenue (often enabled by data mining end-user-provided content), as well as allow web sites to be deployed that are not currently economically feasible. For example, sites that deploy Maygh may choose to make Maygh opt-in, offering to show fewer ads to users who help with content distribution. Regardless, Maygh's tracking of the amount of content uploaded and downloaded ensures that no user has to contribute more resources than they use, and we demonstrate in the next section that Maygh imposes an acceptable bandwidth and storage cost on the clients (since the load is distributed over all clients).

Most cable or DSL systems offer users asymmetric bandwidth, with more downstream than upstream bandwidth [12]. This configuration is unlikely to significantly affect Maygh, as we demonstrate in Section 5 that each user ends up serving content infrequently and the content served is small compared to the available upstream bandwidth. Reducing content page loading latency is typically important for web site operators, and using Maygh imposes increased latency on content requests. Moreover, additional latency in Maygh can be caused by clients with asymmetric bandwidth. However, operators can hide much of this increased latency from users by using content pre-fetching techniques (in fact, many sites already do so).

4.4 Mobile users

The Maygh design so far has focused on users who are using traditional desktop web browsers. However, users are increasingly accessing the web from mobile devices. As the mobile web browsers are quickly catching up with their desktop counterparts, it is likely that Maygh will work without modification once WebRTC support is provided to these browsers. However, a potential drawback is that the user session times are likely to be much shorter, as smartphones typically only allow users to have one "active" web page at a time. Moreover, smartphones present two additional constraints: Limited battery life and data access charges. Previous work [60] has shown that mobile browser-based assistance can be implemented without significant battery consumption, and Section 5 demonstrates that per-user limits on the amount of data users are requested to upload does not significant impact the benefits of Maygh. However, we leave a full evaluation of the potential of Maygh on mobile devices to future work.

5. Evaluation

We now turn to evaluate the performance of Maygh. We center our evaluation around four questions:

- What is the impact of Maygh on web clients, in terms of increased latency and network overhead?
- What is the scalability of Maygh? How many content requests per second can a set of coordinators support?
- What is the impact of Maygh on the operator, in terms of the reduction in network traffic? How much network overhead does Maygh incur?
- How does Maygh perform when deployed on a real web site to real users?

5.1 Implementation

Our full Maygh implementation is written using RMTFP, as this allows us to deploy Maygh onto a wide variety of web browsers and to obtain a userbase quickly [19]. Support for WebRTC is in progress, and a proof-of-concept implementation is described in Section 5.2.2.

The Maygh coordinator is written as a heavily-modified version of the open-source ArcusNode [7] RTMFP server. ArcusNode is written in JavaScript, built on top of the Node.js [34] framework. As a result, our coordinator consists of 2,944 lines of JavaScript, and 372 lines of C for the encryption and encoding libraries. Our coordinator supports working in tandem with other coordinators, as described in Section 3.5. The code is single-threaded, event-driven, and uses UNIX pipes for communication (between coordinators resident on the same machine) or TCP sockets (if not).

The client-side Maygh implementation consists of 657 lines of JavaScript (with a total size of 4.2 KB) and 214 lines of ActionScript (compiled into a 2.6 KB Flash object). We use the Stanford Javascript Crypto Library's SHA-256 implementation [49] (6.4 KB) to implement hashing on the client side. Thus, each client has to download an additional 13.2 KB of code (which is likely to be cached, allowing each client to only have to download it once).

Loaded to		Loaded from: LAN	Cable	DSL
		BOSTON	BOSTON	NEW ORLEANS
LAN	BOSTON	229 / 87	618 / 307	1314 / 707
Cable	BOSTON	771 / 283	702 / 314	1600 / 837

Table 2. Average time (ms) to load first / second 50 KB objects using Maygh with RTMFP.

Loaded to		Loaded from: LAN	Cable	DSL
		BOSTON	BOSTON	NEW ORLEANS
LAN	BOSTON	72 / 16	364 / 120	544 / 354
Cable	BOSTON	284 / 57	577 / 107	765 / 379

Table 3. Average time (ms) to load first / second 50 KB objects using Maygh with our proof-of-concept WebRTC implementation.

In some of the experiments below, we wish to simulate a large number of clients using Maygh. It is challenging to run thousands of web browsers and Flash runtimes at once; instead, we wrote a command line-based implementation of the Maygh client. This implementation, written on the Node.js [34] framework similar to our coordinator code, follows the exact same protocol as the Maygh JavaScript and Flash does when run within a web browser; from a network perspective, the messages sent are identical. However, the simulated clients run entirely at the command line.

To implement this client, we reverse-engineered the client-side RTMFP protocol. We then implemented the client on the Node.js framework. In total, it contains 2,900 lines of JavaScript (2,053 of which are shared with the coordinator implementation).

5.2 Client-side microbenchmarks

We now examine the impact that Maygh would have on end users. We examine the latency of fetching objects, the additional network traffic, and the storage requirements.

5.2.1 Client-perceived latency

We first examine the client-perceived latency of fetching content. For this experiment, we deploy Maygh to a test web site with a coordinator located on our university's LAN in Boston. We connect two clients running Google Chrome 13.0 with Ubuntu 11.04 to the web site and one of the clients fetches two 50 KB objects from the other. We measure the time required to fetch each entire object, and all results are the average of ten separate runs. We report the time taken from the requesting client's perspective, including all messaging with the coordinator, connection setup with the other client, and hash verification. For all experiments, we configure additional command-line clients to create a background load of 200 fetch requests per second to the coordinator.

As we are interested in the latency experienced by clients in different locations, we run this experiment with a number of different configurations. We placed the client requesting the objects in two different locations: on the same LAN as the coordinator, and behind a cable modem in Boston. Then, we placed the client serving the objects in three different locations: on the same LAN as the coordinator, behind a cable modem in Boston, and behind a DSL modem in New Orleans.

The results of this experiment are presented in Table 2. We report the average time taken for the first object separately from the second object; the first is higher because it includes connection setup (including STUN, in the cases of Cable–Cable and Cable–DSL) with the remote client. For the second object, the connection to the remote client is cached by the Maygh library. For all intra-Boston connections, the time taken to deliver the 50 KB object is under 320 ms, with a RTMFP connection setup overhead of approximately 300 ms. Fetching content from New Orleans to Boston is more expensive, but we expect that clients will be able to find another online client within their geographic region the majority of the time. Additionally, much of this latency can be hidden through the use of content pre-fetching (which many web sites already do) and parallel downloads (which most browsers already do).

5.2.2 Latency with WebRTC

Our latency results in the previous section suggest that there is non-trivial latency overhead when using RTFMP (recall that RTFMP is primarily designed for audio and video streams to be exchanged, not application-level messages). To determine how much of the latency is due to RTMFP protocol overhead—and is therefore not fundamental to Maygh—we implemented a proof-of-concept version of Maygh using WebRTC. We use Chromium 26.0.1412.0[10], which has initial support for WebRTC's DataChannel. We repeat the same experiment as above, and report the results in Table 3. We observe that all cases, our prototype WebRTC implementation is significantly faster (typically around twice as fast as RTMFP). This result indicates that the performance of our RTMFP-based implementation is likely a lower bound on the performance of (in-progress) complete WebRTC-based implementation.

5.2.3 Network traffic

We now turn to examine the network traffic overhead caused by Maygh. As discussed above, the Maygh code itself is 13.2 KB, although this will be cached by the web browsers for clients' subsequent requests. To connect to the coordinator,

[10] This build of Chromium imposes a rate limit on each DataChannel of 3 KB/s; we removed this rate limit to perform our experiments. Additionally, this build only supports unreliable channels; we implemented a reliable sliding-window-based protocol on top of this interface.

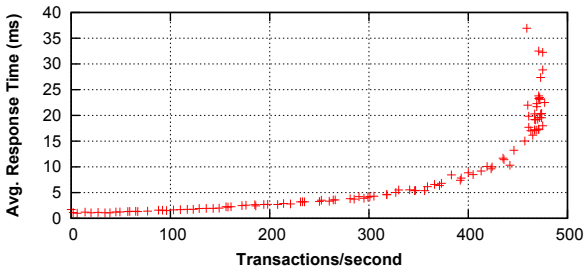

Figure 4. Average response time versus transaction rate for a single coordinator. The coordinator can support 454 transactions per second with under 15ms latency.

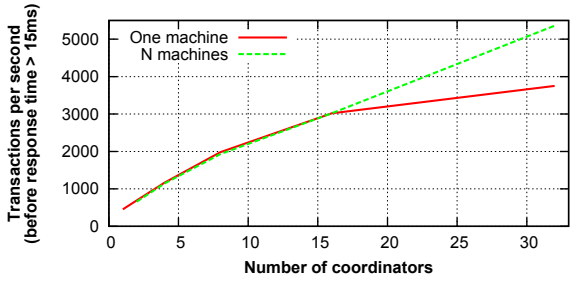

Figure 5. Average response time versus request rate for a multiple coordinators working in tandem, in two different placements of the coordinators across machines. Close-to-linear scaling is observed as more coordinator nodes are added (the one-machine set of coordinators show lower performance after 16 coordinators are placed on a single machine due to the effects of hyperthreading).

the client sends a total of 1.3 KB, and the client sends an additional 0.6 KB for each content request and subsequent connect request. Thus, even for very small objects, Maygh imposes very little network traffic overhead. Moreover, the majority of the overhead comes from the downloading of the client code and the initial connection to the coordinator; this cost will be amortized over all objects in the page.

5.2.4 Client storage capacity

Our Maygh implementation uses the LocalStorage [59] browser storage API, which is by default limited to only 5 MB of storage per site (the other available storage APIs offer greater defaults; our results are therefore conservative). We next examine is the number of objects that can be stored in each user's LocalStorage. Taking into account the 33% overhead induced by storing objects in base64 format, Maygh is able to store 3.3 MB of content per site in each user's LocalStorage. However, as we demonstrate below using real-world data from Etsy, even using only 3.3 MB per site on each client still allows significant savings to be realized.

5.3 Coordinator scalability

We now turn to explore the scalability of the coordinator nodes. Our goal is to determine the rate of content requests (referred to as *transactions*) that can be supported by the coordinators. Each transaction consists of a `lookup` and `lookup-response` message and—if the requested content was found on another online client—a `connect` message followed by the coordinator-assisted connection establishment between the clients.

For these experiments, we run coordinator node(s) on a cluster of machines with dual 8-core 2.67 GHz Intel Xeon E5-2670 processors (with hyperthreading), connected together by Gigabit Ethernet. We then run simulated clients on similar machines in the same LAN. We configure each of the clients to make content requests every five seconds, and measure the throughput (in terms of the number of transactions per second) across the coordinators. The clients are configured so that 70% of the transactions will have the content present on another client.

5.3.1 Single coordinator

Our first experiment examines the performance of a single coordinator process (i.e., a coordinator running only on a single core). In this experiment, we slowly increase the number of clients over time and record the number of transactions per second processed by the coordinator. We also record the request latency at the clients; as the coordinator reaches peak throughput, we expect the number of transactions per second to level off and the client-observed latency to increase sharply. The results of this experiment are presented in Figure 4. After 454 transactions per second, we observe that the response time increases above 15 ms and rises more sharply, indicating that the coordinator is having trouble keeping up.

5.3.2 Multiple coordinators

We now explore the scalability of the coordinators. We repeat the experiment from above, but deploy multiple coordinator nodes that work together to distribute content. The clients are randomly assigned to one of the coordinators and make requests for random content (i.e., there is no coordinator-locality in the requests).

We run two experiments, distributing the coordinators in two different ways: all resident on the same physical machine, and each located on their own machine. In all cases, we provide each coordinator with a dedicated CPU core. For each experiment, we slowly increase the number of clients present and calculate the number of transactions per second that can be supported before the average response time at the clients increases beyond 15 ms.

The results of this experiment are shown in Figure 5. We observe close-to-linear scaling, as expected from our design. We also observe that the set of coordinators located each on their own machine show similar performance to the set of coordinators on a single machine, up to 16 coordinators. After this point, the performance of coordinators on a single ma-

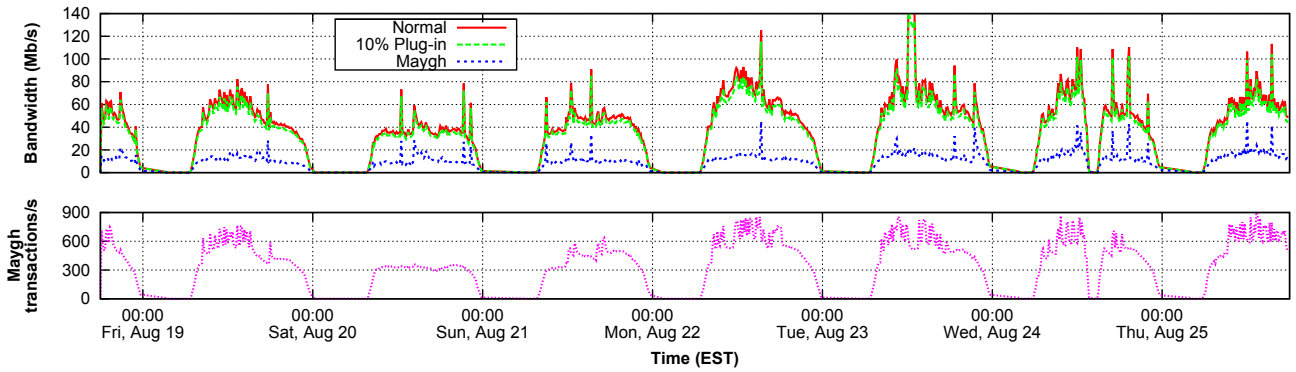

Figure 6. TOP: Bandwidth required at the operator, as normal (with no plug-ins or Maygh), with a 10% plug-in deployment, and with Maygh; a 10% plug-in deployment results in 7.8% bandwidth savings, while Maygh results in 75% bandwidth savings. BOTTOM: Request rate observed at the Maygh coordinator; the rate is almost always below 800 requests per second, and is easily handled by a four-core server.

chine increases more slowly, because coordinator processes begin to be assigned to the same physical core, though on different core threads (i.e., the effects of the hyperthreaded CPUs begin to become apparent).

Overall, we observe that the Maygh coordinators show very favorable close-to-linear scalability. With our 32-core server, the Maygh coordinators are able to support over 3,700 transactions per second, making it suitable for very large websites. In fact, in our simulations below using the access logs from Etsy, we observe a peak rate of 938 transactions per second; this load would require only a 4-core machine to host the coordinators.

5.4 Trace-based simulation

Our next evaluation concerns the benefits that Maygh can be expected to provide in practice to the web site operator.

5.4.1 Simulation data

Determining the benefits of Maygh in practice requires a real web access trace, as it is dependent upon the object request pattern, the object sizes, and the offline/online behavior pattern of clients. We use traces provided by Etsy, a large e-commerce web site that is the 50th most popular web site in the U.S. [5]. Etsy is a popular online marketplace for independent artists and designers; each seller is given an online "store" where they can post items available for sale. Each item listing typically contain of a number of images, and Etsy currently distributes these using Akamai.

We obtained anonymized logs for seven days of accesses to the static image content on the `etsy.com` website, covering 205,586,135 requests to 56,084 unique objects from 5,720,737 unique IP addresses. The logs only include the busiest 18 hours of the day, from 6:00am–11:59pm EST. In total, the requests represent 2.77 TB of network traffic, or 395 GB per day. To estimate the overall fraction of Etsy's bandwidth that is represented in our traces, we use the same random browsing methodology from Section 2 on Etsy's site. We find that image content represents 85.6% of the total bytes served by Etsy, meaning any savings we report is likely to represent significant overall bandwidth savings.

It is important to note that our logs are what the web site operator sees, after the browser caching and any in-network (HTTP proxy) caching. Thus, any bandwidth savings that we report are ones that would be observed in practice.

5.4.2 Simulation setup

Simulating a Maygh deployment requires knowing how long the clients' browser windows remain on the site (as this determines how long the client is available to serve requests to other users). The trace lists only HTTP requests, so we do not know the length of time that the client's browser window remains open to the site. Therefore, we simulate the clients staying on the site for a random amount of time between 10 seconds and 30 seconds after each request. This short online window is likely to be conservative [28] for a storefront like Etsy; the longer that users stay online, the better the network traffic savings that Maygh can provide (since users are available to serve requests for longer).

We simulate the clients' LocalStorage staying intact between sessions, as this would happen in practice. Unless otherwise mentioned, we set `upload_ratio` to 1 and `upload_max` to 10 MB. This means that no client is asked to upload more bytes than they have previously downloaded, and no client is asked to upload a total of more than 10 MB during the simulated week. In our bandwidth calculations, we also include the network traffic for clients to download the Maygh code, to connect to the coordinator, and to execute the Maygh protocol.

To compare Maygh to alternate approaches, we also simulate a deployment of a plug-in-based system (e.g., Firecoral [50]) to a random 10% of the Etsy users (recall from Section 1 that 10% is likely to be a higher fraction than would be observed in practice, as the most popular plug-in today is used by 4.2% of users). Due to the plug-in architecture, users

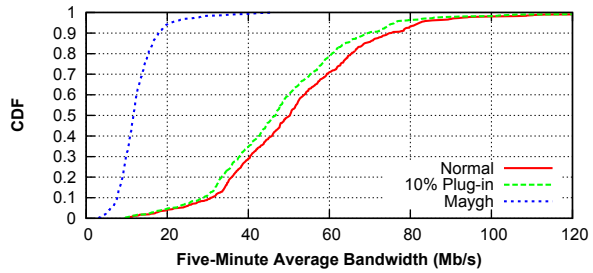

Figure 7. Cumulative distribution of five-minute average bandwidth required at the operator for the same trace as Figure 6.

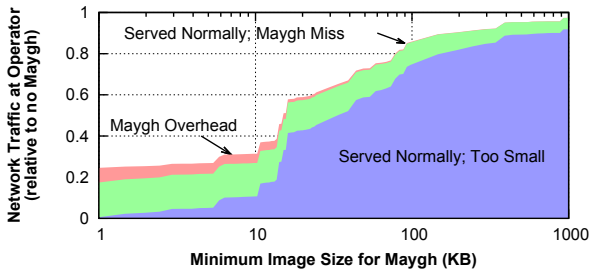

Figure 8. Network traffic at the operator, relative to the trace without Maygh, for different minimum Maygh image sizes. The traffic consists of three components: images that are served normally because they are too small, images that are served normally because Maygh cannot find an online client able to serve it, and overhead caused by Maygh (downloading the client code and protocol overhead). If Maygh serves all images, the operator would experience 75% less traffic due to images.

who install the plug-in would only be able to download from other such users; users who do not install it would continue to fetch content from the operator as normal. We simulate the plug-in system with a per-user 100 MB cache size, and allow users to serve to others regardless of the amount of data they have downloaded (i.e., we do not limit users to uploading only as much as they have downloaded).

5.4.3 Bandwidth savings

We examine the amount of network traffic at the operator under three configurations: normal (with no plug-in or Maygh), with 10% of users running installing the plug-in, and with Maygh. We record the network traffic experienced at the operator, aggregated into five-minute intervals over the course of the week-long trace.

The results of this experiment are presented in the top graph of Figure 6, showing the five-minute average bandwidth required at the operator with different configurations. Figure 7 presents the cumulative distribution of this same trace. We make a number of interesting observations: First, we observe that Maygh provides substantial savings: the median bandwidth used drops from 50.3 Mb/s to 11.7 Mb/s (a 77% drop). Second, we also observe that the 95th-percentile bandwidth—which often determines the price that operators pay for connectivity—shows a similar decrease of 75%, demonstrating that Maygh is likely to provide a significant cost savings to the operator. Third, we observe that the 10% plug-in deployment results in a median bandwidth decrease of 6.9% and a 95th-percentile bandwidth decrease of 7.7%; the savings is less than 10% due to cache misses and protocol overhead.

In the lower plot of Figure 6, we show the number of transactions per second that is experienced at the Maygh coordinator over the week. We observe that the average transaction rate is 482 transactions per second, with a maximum of 938. This shows that Maygh could be deployed at Etsy with a small number of coordinators.

5.4.4 Bandwidth breakdown

We now turn to examine the breakdown of the network traffic at the operator. In order to explore the contribution that images of different sizes have on the performance of Maygh, we configure Maygh to only serve images larger than a given threshold. We then vary this setting, examining the resulting performance tradeoff. In this setup, the network traffic at the operator can be broken down into three classes:

1. **Maygh overhead** consisting of downloading the client code and Maygh protocol overhead.
2. **Content served normally due to Maygh misses** when the coordinator could find no online client able to serve a request.
3. **Content served normally because it is too small** when content is smaller than the Maygh threshold configured by the web site operator.

The results of these experiments are presented in Figure 8, for different settings of the minimum image size that Maygh will serve. The results are presented as stacked curves and are expressed in terms of the total network traffic that Etsy experienced without Maygh. As an example, if Etsy chose to configure Maygh to only serve content larger than 5 KB, they would experience only 27% of the network traffic that they did in the original trace. Of this traffic, 19% would be comprised of images below 5 KB that are served normally, 62% would be comprised of images larger than 5 KB that Maygh cannot find an online client to serve, and 19% would be comprised of Maygh overhead (primarily downloading the Maygh code).

We observe that Etsy's workload consists primarily of small images; over half of the network traffic without Maygh is spent transmitting images smaller than 35 KB. As a result, the largest benefits of Maygh in our simulations only happen when the size threshold for being served Maygh is small. However, we note that this is an artifact of Etsy's workload;

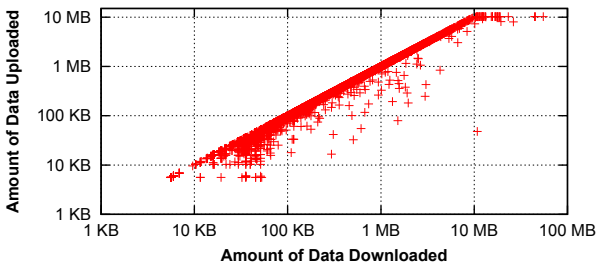

Figure 9. Network traffic at the clients, comparing amount of data uploaded to downloaded. We observe that the client workload distribution of Maygh is "fair", meaning most clients are requested to upload an amount of data that is proportional to the amount they download.

Maygh has lower overhead with larger objects, so a workload with a larger objects is likely to perform even better.

Finally, we observe that, for small thresholds, the largest fraction of the network traffic at the operator is caused by Maygh misses. Examining the logs, we find that these misses are usually caused by the enforcement of the upload constraints at the clients. In other words, there are often clients online who can serve a given piece of content, but all of the clients have already reached their upload limit. Re-running the same experiment from above and removing the upload constraints at the client (meaning clients may be asked to serve more content than they download) causes the network traffic due to images at the operator to fall to 18% of the traffic that would be experienced without Maygh (compared to 25% with the upload constraints).

5.4.5 Client network traffic

We now examine is the network traffic that is imposed on clients. For each client in the trace, we record the total amount of data downloaded and the total amount of data that the client is requested to upload. We then compare the two, to examine how "fair" the distribution of the upload workload is across clients. For this experiment, we configure the Maygh threshold to be 5 KB.

Figure 9 presents the results of this experiment, plotting the total amount of data uploaded vs. downloaded. We observe that the shape of the plot is largely defined by Maygh policy: no user is ever asked to upload more than they have downloaded, and no user is asked to upload more than 10 MB. However, even with those constraints, the workload distribution across users is quite "fair": Most users are asked to upload in close proportion to what they have downloaded.

5.5 Small-scale deployment

As a final point of evaluation, we deployed our Maygh prototype on a small scale within our computer science department to examine how it would work with real-world users.[11]

[11] Our real-world deployment was covered under Northeastern University Institutional Review Board protocol #10-07-23.

We set up a coordinator within our department, deployed Maygh to a special version of our department's web server, and then recruited users by emailing our graduate student population. The deployment ensured that all images on our department's web site would be loaded via Maygh.

In total, over the course of our 3-day deployment, we observed 18 users use Maygh on Google Chrome, Firefox, and Safari, on machines running Windows, Linux, and OS X. These users browsed a total of 374 images. 90 (or 24%) of these images were served from another Maygh client. For the remaining 76% of the images, there was no other client online storing the image; they were fetched from the origin site as normal. While the network savings of Maygh is lower than in our simulations, it is due to our deployment environment: For the simulations, we considered what would happen if a large, popular web site deployed Maygh; in our real-world deployment, we had to manually recruit users and the size of our user population is dramatically smaller. However, the deployment demonstrates that Maygh is can be feasibly deployed to today's web sites and web browsers.

6. Related work

We now describe related work covering CDNs, peer-to-peer systems, and cooperative web caches.

6.1 Optimizing CDNs

CDNs like Akamai, Limelight, Adero, and Clearway have emerged as a commonly-used platform for web content delivery. CDNs offload work from the original web site by delivering content to end users, often using a large number of geographically distributed servers. While much work on CDN architectures has examined different server selection strategies [2, 41, 53] and content replication algorithms [8, 51], most of it is not directly applicable to Maygh. CDNs generally assume a relatively stable list of servers under centralized control; in Maygh, each visiting web client is effectively a server. It may be possible to use content replication techniques to pre-fetch content onto users' machines; however, we leave such techniques to future work.

Alternatives to centralized CDN have been built that use resources under multiple entities' control to accomplish the same task (e.g., Coral [18], CobWeb [48], and CoDeeN [36, 54]). These solutions are impressively scalable and well-used, but they generally rely on resources donated by governments and universities, and are therefore not self-sustaining in the long run. In fact, the Coral system quickly overwhelmed the bandwidth resources available on PlanetLab within the first year of deployment [50], and was forced to enforce fair sharing and reject some download requests.

Other approaches have explored allowing end users to participate in CDNs, including Akamai's NetSession [4], Flower-CDN [13], BuddyWeb [55], Squirrel [22], and Web2Peer [39] (client-side applications that assist in content distribution to other clients), as well as Firecoral [50] (a

browser plug-in that serves content to other Firecoral users). While the goals of these systems are similar to Maygh, all require the user to download and install a separate application or plug-in, significantly limiting their applicability and userbase in practice. WebCloud [60] allows users' browsers to participate in content distribution without requiring any client-side changes; however, it requires that CDN-server-like redirector proxies be deployed within each ISP region.

Additionally, recent work [44] has demonstrated that information flow patterns over social networks (called *social cascades*) can be leveraged to improve CDN caching policies. This work is complementary to ours, and suggests that Maygh will perform especially well in systems like online social networks. Finally, other recent work [35] has examined the benefits of allowing ISPs to assist CDNs in making content delivery decisions. This approach is also similar in spirit to Maygh, but focuses on optimizing the server selection strategies employed by ISPs today.

6.2 Alternate approaches

Maygh can be viewed as approximating p2p content exchange through web browsers. Much previous work has focused on building standalone p2p systems for content storage and exchange [11, 26, 29] or avoiding the impact of flash crowds [42, 46]. However, unlike Maygh, almost all work on p2p systems assumes a full networking stack and is incompatible with being run inside a browser. Recent work [10] has moved towards leveraging resources on web clients to provide better scalability for web-based services, but is primarily focused on the services themselves, rather than content distribution.

There has been a long history of work that examines corporative web proxy caches, where proxy caches on the same side of a bottleneck link corporate to serve each other's misses [15, 52]. While these systems have the same net effect of Maygh (the reduction of load on the operator), their motivation is usually to lower network usage at the edge of the network, rather than at the operator. As a result, unlike Maygh, cooperative caches must be deployed and configured by network administrators at the edge and are not under control of the operator.

7. Conclusion

Over the past two decades, the web has provided dramatic improvements in the ability and ease of sharing content. Unfortunately, today, web sites who wish to share popular content over the web are required to make substantial monetary investments in serving infrastructure or cloud computing resources, or pay organizations like CDNs to help serve content. As a result, only well-funded web sites can serve a large number of users.

We have presented the design of Maygh, a system that distributes the cost of serving content across the visitors to a web site. Maygh automatically recruits web visitors to help serve content to other visitors, thereby substantially reducing the costs for the web site. A thorough evaluation of Maygh using real-world traces from a large e-commerce web site demonstrated that Maygh is able to reduce the 95th-percentile bandwidth due to image content at the operator by over 75%, providing a substantial monetary savings, and a small-scale deployment demonstrated that Maygh imposes little additional cost on clients and is compatible with the web browsers and sites of today.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Katerina Argyraki, for their helpful comments. We also thank Avleen Vig for his assistance with the Etsy traces, David Blank-Edelman and Rajiv Shridhar for their assistance with the Northeastern traces, Bimal Viswanath for the use of MPI-SWS servers, and Michael Mislove for his assistance with the New Orleans latency experiments. Finally, we thank the developers of the open source ArcusNode project.

This research was supported by NSF grants IIS-0964465 and CNS-1054233, and an Amazon Web Services in Education Grant.

References

[1] P. Aditya, M. Zhao, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, and B. Wishon. Reliable Client Accounting for P2P-Infrastructure Hybrids. *NSDI*, 2012.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. *SOSP*, 2001.

[3] Adblock Plus : Statistics for Adblock Plus. `http://bit.ly/10WEytx`.

[4] Akamai NetSession. `http://www.akamai.com/client`.

[5] Alexa - Top Sites in the United States. `http://www.alexa.com/topsites/countries;1/US`.

[6] Alexa Top 500 Global Sites. `http://www.alexa.com/topsites`.

[7] ArcusNode. `https://github.com/OpenRTMFP/ArcusNode`.

[8] S. Buchholz and T. Buchholz. Replica placement in adaptive content distribution networks. *SIGACT*, 2004.

[9] BitTorrent. `http://www.bittorrent.com`.

[10] R. Cheng, W. Scott, A. Krishnamurthy, and T. Anderson. FreeDOM: a New Baseline for the Web. *HotNets*, 2012.

[11] L. P. Cox and B. D. Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. *SOSP*, 2003.

[12] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing Residential Broadband Networks. *IMC*, 2007.

[13] M. E. Dick, E. Pacitti, and B. Kemme. Flower-CDN: a hybrid P2P overlay for efficient query processing in CDN. *EDBT*, 2009.

[14] Etsy. `http://www.etsy.com`.

[15] L. Fan, P. Cao, and J. Almeida. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *SIGCOMM*, 1998.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, 1999.

[17] M. J. Freedman and R. Morris. Tarzan: A Peer-to-Peer Anonymizing Network Layer. *CCS*, 2002.
[18] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. *NSDI*, 2004.
[19] Adobe Flash Player Statistics. http://www.adobe.com/products/flashplatformruntimes/statistics.html.
[20] A. Gupta, B. Liskov, and R. Rodrigues. One Hop Lookups for Peer-to-Peer Overlays. *HotOS*, 2003.
[21] X. Hei, C. Liang, J. Liang, Y. Liu, and K. Ross. A Measurement Study of a Large-Scale P2P IPTV System. *IEEE Trans. Multimedia*, 9(8), 2007.
[22] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. *PODC*, 2002.
[23] S. Ihm. Understanding and Improving Modern Web Traffic Caching. Ph.D. Thesis, Princeton University, 2011.
[24] S. Ihm and V. S. Pai. Towards Understanding Modern Web Traffic. *IMC*, 2011.
[25] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocol for relieving hot spots on the World Wide Web. *STOC*, 1997.
[26] J. Kubiatowicz. OceanStore: an architecture for global-scale persistent storage. *ASPLOS*, 2000.
[27] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: surviving organized DDoS attacks that mimic flash crowds. *NSDI*, 2005.
[28] C. Liu, R. White, and S. Sumais. Understanding Web Browsing Behaviors through Weibull Analysis of Dwell Time. *SIGIR*, 2010.
[29] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A Cooperative Internet Backup Scheme. *USENIX ATC*, 2003.
[30] R. Lawler. NFL Pushes HD Video, with Help from Akamai. 2010. http://gigaom.com/2010/09/22/nfl-pushes-hd-video-with-help-from-akamai/.
[31] J. Mirkovic and P. Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *CCR*, 34(2), 2004.
[32] Most Popular Extensions :: Add-ons for Firefox. http://bit.ly/122o6ge.
[33] Mozilla Metrics Report, Q1 2010. http://mzl.la/aplroe.
[34] Node.js Framework. http://nodejs.org/.
[35] I. Poese, B. Frank, B. Ager, G. Smaragdakis, and A. Feldmann. Improving Content Delivery Using Provider-aided Distance Information. *IMC*, 2010.
[36] K. Park and V. S. Pai. Scale and Performance in the CoBlitz Large-File Distribution Service. *NSDI*, 2006.
[37] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, IETF, 2010.
[38] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), IETF, 2008.
[39] H. B. Ribeiro, L. C. Lung, A. O. Santin, and N. L. Brisola. Web2Peer: A Peer-to-Peer Infrastructure for Publishing/Locating/Replicating Web Pages on Internet. *ISADS*, 2007.
[40] Real-Time Media Flow Protocol (RTMFP) FAQ. http://www.adobe.com/products/flashmediaserver/rtmfp_faq/.
[41] A. Su, D. R. Choffnes, A. Kuzmanovic, and F. E. Bustamante. Drafting behind Akamai: Inferring network conditions based on CDN redirections. *ToN*, 2009.
[42] A. Stavrou, D. Rubenstein, and S. Sahu. A Lightweight, Robust P2P System to Handle Flash Crowds. *ICNP*, 2002.
[43] S. Spoto, R. Gaeta, M. Grangetto, and M. Sereno. Analysis of PPLive through active and passive measurements. *IPDPS*, 2009.
[44] S. Scellato, C. Mascolo, M. Musolesi, and J. Crowcroft. Track Globally, Deliver Locally: Improving Content Delivery Networks by Tracking Geographic Social Cascades. *WWW*, 2011.
[45] T. Stein, E. Chen, and K. Mangla. Facebook Immune System. *EuroSys*, 2011.
[46] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. *IPTPS*, 2002.
[47] D. K. Smetters and V. Jacobson. Securing network content. PARC, Technical Report TR-2009-1, 2009.
[48] Y. J. Song, V. Ramasubramanian, and E. G. Sirer. Optimal Resource Utilization in Content Distribution Networks. Cornell University, Technical Report TR2005-2004, 2005.
[49] Stanford Javascript Crypto Library. http://crypto.stanford.edu/sjcl/.
[50] 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. *IPTPS*, 2009.
[51] C. Vicari, C. Petrioli, and F. L. Presti. Dynamic replica placement and traffic redirection in content delivery networks. 2007.
[52] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative Web proxy caching. *SOSP*, 1999.
[53] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. *SIGCOMM*, 2005.
[54] L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson. Reliability and Security in the CoDeeN Content Distribution Network. *USENIX ATC*, 2004.
[55] X. Wang, W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. BuddyWeb: A P2P-Based Collaborative Web Caching System. *NETWORKING*, 2002.
[56] W3C Indexed Database API. http://www.w3.org/TR/IndexedDB.
[57] W3C Real-Time Communications Working Group. http://www.w3.org/2011/04/webrtc-charter.html.
[58] W3C Web Workers. http://www.w3.org/TR/workers/.
[59] W3C WebStorage. http://www.w3.org/TR/webstorage/.
[60] F. Zhou, L. Zhang, E. Franco, A. Mislove, R. Revis, and R. Sundaram. WebCloud: Recruiting social network users to assist in content distribution. *IEEE NCA*, 2012.
[61] N. C. Zakas. How many users have JavaScript disabled? 2010. http://developer.yahoo.com/blogs/ydn/posts/2010/10/how-many-users-have-javascript-disabled/.