DCDN: Distributed content delivery for the modern web

by

Nick J. Martindell

Supervised by Tom Anderson and Arvind Krishnamurthy

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering

University of Washington

June 5, 2014

Presentation of work given on _____

Thesis and presentation approved by _____

Date _____

Abstract

The current client server model used on the web is inefficient for the delivery of large static content. Content is transmitted independently to each consumer from the central infrastructure without respect to the copies that exist in the caches of other visitors. As demand grows, capacity remains fixed and all users experience delays. This problem has been exacerbated by the rise in popularity of large streaming content such as high definition web video. Desktop peer-to-peer (P2P) systems have shown great promise in alleviating these issues by leveraging the underutilized upstream bandwidth of each client to deliver content. Bringing these techniques to the web in a way that doesn't require user intervention would allow for a more efficient web.

This thesis explores the state of the art in browser-based P2P content delivery for the web and seeks to answer whether such systems can be used to efficiently and invisibly deliver content. It presents the DCDN (Distributed Content Delivery Network) research platform which serves content to the users of a website using only their HTML5 enabled web browser. It then uses the platform to explore several possible optimizations for this method of content delivery and evaluate their success. Through this investigation, it shows that while browser-based P2P systems can be implemented quite simply, at this time their performance characteristics limit them to certain content types. High definition web video and long-duration audio streaming are key examples. In order to expand the possible use cases, significant roadblocks will need to be overcome. A few emerging technologies which may provide solutions within the next year are discussed, as well as somewhat far-fetched concepts for future improvement. At present this technology has great value, especially when considering that its ideal content types make up a large portion of the bandwidth currently used on the Internet.

Contents

1 Introduction

As the load placed on web infrastructure grows with the ever-increasing demand for large multimedia content, it is increasingly important to deliver it in efficient ways. Characteristics of the Hyper-Text Transport Protocol (HTTP) used to serve almost all web content, make it somewhat inefficient as a means of delivering static content (images, videos, etc. that don't change often over time) to users. The most notable issue is its focus on a strictly client-server relationship. With this system, the cost of serving content increases with the number of users due to the need for more servers and greater bandwidth to handle peak load. Large Internet companies spend a considerable sum on this infrastructure, but this increase in capacity through capital expenditure fails to address the underlying inefficiency of serving content in this manner: all the load is placed on the central server even though copies of the content exist in the caches of every client on the site.

This practice leads to an underutilization of the client's Internet connection. In practice, the majority of content on the web flows from large corporate data centers to home and office consumers while comparatively little content moves in the opposite direction. Additionally, each new visitor to a web site downloads a full copy of all page content from the central server without regard to other equally viable copies on the network. This focus on central infrastructure causes problems in times of high load during which service quality for users drops since that have to share the host's finite capacity. Both of these problems are addressed by peer-to-peer (P2P) content delivery techniques.

If such a P2P content delivery system could be used by a web host in a way which would be effortless and invisible to users, it could enable a more efficient web. Until recently, this would have required the user to install 3rd party software and/or browser plug-ins. However, recent developments in the JavaSript application programming interface (API) exposed to web browsers have enabled a new class of P2P applications requiring nothing more than a modern web browser. While the possibility of creating such applications has been explored in recent commercial endeavors, little is known to the research community about how this class of applications performs compared to conventional content delivery systems.

My research shows that while the existing knowledge about P2P systems is sufficient to create a functional browser-based system, there are new obstacles to overcome in order to provide the high performance users expect. Web security requirements, aspects of JavaScript's binary APIs, interaction with the DOM (The Document Object Model used to layout the content of a web page) and the comparatively erratic behavior of the client cause the performance of browser-based P2P systems to differ in ways that merit new investigation. This thesis investigates several of those characteristics and suggests techniques to overcome them. It also briefly explores technologies still in development that may provide for a better system.

2 Dependencies

In order to discuss the performance characteristics of browser-based P2P systems it is first necessary to understand the characteristics of the APIs on which such systems are built. These P2P primitives resemble those used by desktop applications but often operate at a higher level of abstraction and with a different security model. The speed of establishing various connection types makes up a significant portion of the performance overhead associated with DCDN.

2.1 WebSockets

WebSockets are an implementation of full-duplex TCP (Transmission Control Protocol) sockets in the browser. Like standard sockets, they provide bidirectional push communication as well as efficient sending and receipt of binary data. Unlike desktop sockets however, they are message-oriented and by default use UTF-8 text rather than binary streams. A configuration option allows the use of raw binary, but all data is still framed in messages. This design is reflective of a central use case: enabling server-push messaging, or the sending of updates from the server without a request from the client [12].

Performance wise, their behavior is comparable to regular sockets once the connection has been set up and their round trip messaging latency is near that of a ping. On the other hand, their handshake is complex and takes quite a while to set up. WebSockets first establish a TCP connection, then use HTTP to request an 'upgrade', before handing off the socket to the WebSocket Server. This requires several round trips before data can be sent or received causing a significant amount of startup latency even when connecting to a nearby server.

The following data shows the results of request latency tests for 2 types of connections: a WebSocket and an HTTP HEAD Request. The latter is identical to the HTTP GET request used to 'get' most web content, except that it only returns the response headers rather than the content itself. It is thus the smallest and fastest type of HTTP request typically used on the web.

Connection Type	Time from request to receipt of data (microseconds, avg. of 3 trials)
WebSocket	205.11
HTTP HEAD	45.64
WebSocket (2 sec. after page has loaded)	3.44
HTTP HEAD (2 sec. after page has loaded)	4.28

Table 1: Comparison of startup latency for WebSocket and HTTP HEAD requests to localhost

The 3rd and 4th row of data point to another factor affecting startup latency: the amount of other network activity occurring at the same time. In these cases, the connections are made after the page has loaded which allows the requests

to work without contention from the network activity associated with loading the page. In both cases, the request latency is reduced to a few microseconds rather than tens or hundreds of them. Additionally the 3rd row demonstrates how, once connected, a WebSocket can outperform even small HTTP requests.

2.2 WebRTC Peer Connections

WebRTC (Web Real Time Communication) Peer Connections are the fundamental P2P primitive for the browser. They provide nearly all the functionality needed to establish useful connections to peers and are thus both complicated and performance critical. Peer Connections act as a handle to another browser-based peer but don't provide any communication by themselves. Instead, they allow the establishment of video, audio or data channels that then provide the communication APIs relevant to the given media. This reflects one of the design goals of WebRTC which was to provide for audio and video chat without the use of browser plug-ins [17].

In order to establish connections, WebRTC requires a communication channel with which to send and receive essential setup information. In addition to the required communications channel, RTC can use one or more middle-man servers to aid in making direct connections for users behind Network Address Translators (NAT) [17]. The first kind is a STUN (Session Traversal Utilities for NAT) server which provides information about the external IP address and port of the clients so they can attempt a NAT (Network Address Translator) traversal. In the event that the NATs are not traversable and a direct connection cannot be formed, WebRTC can use a TURN (Traversal Using Relays for NAT) server as a simple relay. Using a TURN server guarantees that the WebRTC connection will succeed, but undermines the goals of a distributed content delivery system since all traffic must be routed through the central TURN infrastructure. TURN servers will not be used for this investigation. Instead, a failure to establish a direct connection to a peer using WebRTC with STUN will result in the peer not being used for content delivery.

2.3 WebRTC Data Channels

WebRTC Data Channels allow for socket-like communication over a Peer Connection. Their API is meant to resemble that of WebSockets and the previous notes on WebSocket's use of UTF-8 text and message-based orientation apply. On the other hand, Data Channels utilize a different underlying protocol for communication, SCTP (Stream Control Transmission Protocol), which can emulate various types of traditional socket behavior. SCTP is a transport level protocol that functions as a replacement for UDP although most current implementations tunnel SCTP over traditional UDP (User Datagram Protocol). This is due to the lack of hardware support for SCTP by most NAT boxes which reply on the TCP/UDP port numbers to route packets. Even when configured to be fully reliable, SCTP uses UDP for tunneling rather than TCP since NAT traversals for TCP are far more complex and less likely to succeed [18].

In either the tunneled or regular mode, SCTP implements most of the congestion control and reliability features expected from TCP, but allows each to be toggled by the programmer. The latest Data Channel implementation in Google Chrome allows for all of the Data Channel reliability modes specified by the W3C (World Wide Web Consortium) [2]. This includes the choice of reliable/unreliable/partially-reliable delivery modes as well as in-order or unordered message ordering [6]. For the purposes of P2P content delivery, the most applicable mode is the reliable, unordered mode which ensures messages that are sent are delivered, without the overhead of buffering and ordering incoming messages.

Data Channels, as currently implemented, have some limitations that are more restrictive than those of WebSockets. The most notable is a message size limit of around 16 kilobytes, although this is said to be temporary [6]. Additionally, all WebRTC connections are required to be encrypted. While this protects the private video and audio streams of users, it also enforces a modest performance hit compared to open communication over Data Channels. In the case of content distribution it is unlikely that this encryption is truly desirable since the content is already public.

With regards to performance, WebRTC connections require the exchange of many session descriptions, NAT traversal candidates and an offer or acceptance over the message-passing channel before useful data can be sent. This means a lengthy setup period and one that can only begin after the message-passing channel is operational.

3 Measurement Platform Design and Architecture

DCDN, which stands for Distributed Content Delivery Network, implements a simple but functional P2P content distribution system. It operates next to the standard HTTP content delivery method and maintains full URL compatibility. If it is unable to retrieve content, it falls back to standard HTTP delivery if anything goes wrong. As such, I describe it as a supplementary content delivery protocol since it doesn't supplant the exiting method so much as supplement it. Unlike previous efforts at P2P content delivery for the web, DCDN is not intended for commercial use but as a simple and standardized platform for research into web P2P systems. While the 'basic' configuration implements a quite naive content delivery system, the intention is for DCDN to be easily extensible and enable the implementation and study of several types of optimizations for web based content delivery.

Web based P2P content delivery is not entirely novel, similar systems exist such as PeerCDN [9] and Swarmify [16], but it has not been satisfactorily explored and no such system is in widespread use. As such, there is little knowledge in the research community about the performance of such systems. Functionally, the goal of these systems is to shift the burden of content delivery

away from centralized infrastructure and onto the active users of the content. Due to the similarity in purpose, the protocol used by DCDN, and presumably the related systems, closely resembles that used by desktop P2P content delivery applications such as BitTorrent [3]. While most of the knowledge of these desktop systems likely translates to the new medium, it is unclear from previous research what new challenges these new systems face. To aid in understanding the adaptations described later in this investigation, the following description covers the basic implementation of DCDN. The remainder of the paper will explore variations on the protocol that address the challenges faced by browser-based P2P systems.

3.1 Protocol Summary

In order to serve data via the swarm of active site visitors, DCDN relies on a coordination server in addition to the standard HTTP server or traditional CDN (Content Delivery Network) used to fulfill requests for the content. No modifications are necessary to the existing infrastructure except for the inclusion of a single script and a minor modification to the HTML tags for content served over DCDN. The HTTP service remains the authoritative source of content for a given URL while the coordination server takes care of P2P specific tasks such as distributing the meta data and coordinating peer connections. The client script is responsible for obtaining the content from other peers or falling back to traditional HTTP retrieval of a chunk or the entire file.

DCDN's protocol is a distillation of primitives from existing P2P systems, most notably BitTorrent. For each file, 2 primitives exist: the file data itself and a hash of meta-data that facilitates it being served efficiently in the swarm. Each file is identified by its URL and is always assumed to be the latest version if multiple have existed at the same URL. To ease efficient caching and distribution, each file is treated as a series of chunks numbered from 0 and each is equal in size except for the last which is equal or smaller than the others.

All DCDN transfers begin by obtaining meta-data (Step 1 in fig. 1) about the file from the coordination server. This meta-data store describes basic attributes about the file that are required in order to begin obtaining data chunks. This includes the content-length, the size of each chunk in the file and the mime-type required to interpret the file. Its good to note that except for the chunk size, this information can also be obtained using an HTTP Head request on the URL. While it may not be strictly necessary to obtain meta-data before retrieving data from the HTTP server, it is required by the basic form of the DCDN protocol.

The client must also discover peers from which it can retrieve the file. This is aided by the coordination server which keeps track of peers likely to have chunks of the file in their caches. The peer list can be requested in parallel with the meta-data and once returned can be used to connect to one or more peers. The coordination server in the basic protocol returns a random set of 5 peers from the available peers that requested that resource. In the basic protocol, no information about which chunks a peer has is exchanged. Instead, the client

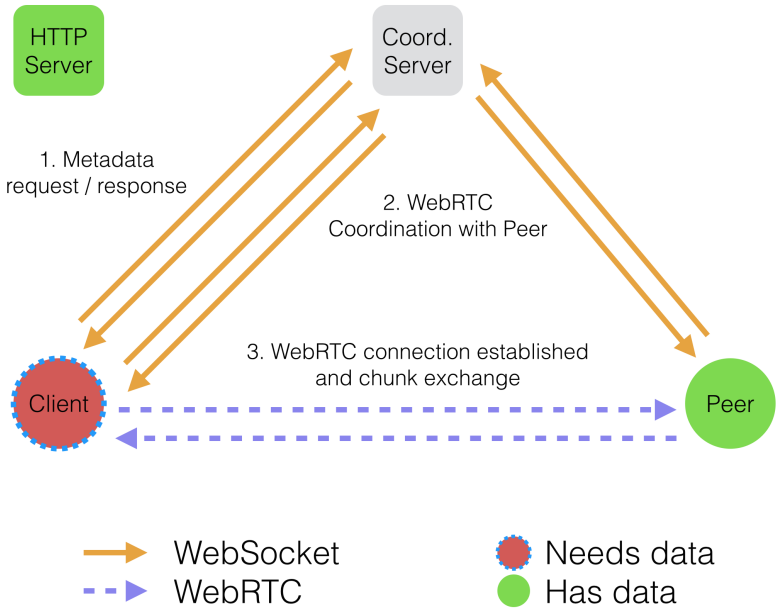

Figure 1: DCDN Protocol Flow

connects to all of them (Step 2 in fig. 1) and randomly requests chunks. In response, the client receives either the requested chunk or an error if the peer can't provide it.

Throughout the download, the DCDN client may request chunks from peers or from the HTTP server in any combination or order it chooses. The basic configuration gets all chunks via HTTP when no peers are available, or uses only peers when possible (Step 3 in fig. 1). This reduces the bandwidth demands on the server as much as possible, but likely reduces the user experience and perception of speed due to the long delay before peers are ready to provide content. A chunk will be obtained via HTTP in the latter case only if it is determined to be unavailable in the swarm. The basic implementation assumes a chunk to be unavailable if the first peer it is requested from replies with a chunkFail message.

The download is complete when all chunks have been retrieved. Upon completion, the basic implementation of DCDN returns a Blob URL to the assembled data which can be used in any context that the originally requested URL would have been valid. Clients can optionally register an onprogress handler which gets access to any sequential chunks which are available. The on progress handler is triggered any time a chunk is obtained which is higher in index than the last one yielded via the onprogress handler. This behavior augments the oncomplete handler, and can be used in addition or in replacement. The main use

of this onprogress handler is to enable media streaming before content finishes loading or the partial display of images as data becomes available.

3.2 Message format

To enable extensibility while preserving some measure of efficiency, DCDN uses a custom wire format for all of the previously described communications. The simple 2-part format described below allows for space and time-efficient encoding, decoding and transmittal of DCDNs messages using only the APIs available to the standard web browser. Additionally, the format has the pleasant property of being human readable while maintaining the efficient transmittal of arbitrary and large binary content.

The format consists of a JSON (JavaScript Object Notation) header followed by a binary body of 0 or more bytes. The first 2 bytes of each message are an unsigned short integer representing the length of the JSON-encoded header string. Following this value is the header itself, and the remainder of the message is arbitrary binary. There is only one binary section per message, although it is perfectly possible to specify how it should be split in the message header. This message format allows for complete flexibility of protocol while still being reasonable size-efficient.

DCDN's simple protocol is designed to be extended in order to test various optimizations and provide representative data about the performance of browser-based P2P systems. New message types can be added easily, and existing types can be extended with new data to enable experimentation. The next section will cover several variations and evaluate their effects.

4 Evaluation

The purpose of DCDN is to provide a simple platform for developing and evaluation web based P2P systems. In this section, I will describe several of the systems studied during the course of my research as well as evaluate their benefit in the context of P2P content delivery for the web. The purpose of this research is to provide a statistically substantiated idea of best practices for such systems. Most of the systems evaluated here rise from observations about the differences between desktop peer-to-peer systems and those in the web browser. The following is not a comprehensive list of optimizations but rather a small subset which attempts to characterize the key differences between desktop and web P2P systems.

4.1 Assessment Metrics

Each system will be assessed based on the time to first byte, time to last byte, effective bandwidth (content size divided by the time interval between first and last byte), the percentage of content served by peers and the effective bandwidth of the system. All times are measured as offsets from the navigationStart event.

Each of the following systems will be evaluated in ideal circumstances to provide a view of their best possible results. These circumstances specify that all parties communicate on localhost, that several (5) peers are available, and that each peer has the full contents of the file in question (in BitTorrent parlance, they are 'seeds'). The test page used for these trials is the 'Large Image' example available in DCDN's GitHub repository [15]. The page contains 2 HTML img tags, one that uses regular HTTP and one that uses DCDN, to obtain and display a 3.2 megabyte JPEG image. Each system will be tested according to the following procedure:

Trial Procedure:

1. The relevant code will be activated in the DCDN source
2. The coordination server will be restarted with the new code
3. The Chrome browser cache will be cleared completely
4. The test page will be opened in 5 tabs on the same computer to serve as 'peers'
5. In a new window, the 'client' will open the test page
6. The client will be refreshed 3 times to 'warm up' itself and the peers
7. The client will be used to make 10 trial visits to the test page (by refreshing 10 more times)
8. DCDN's built in statistics object will be dumped to a text file for each trial

In addition to the data highlighted in the evaluation text of each system, a complete table of data obtained for all the systems described is available at the end of the Evaluation section.

4.2 The basic system

The basic system is the one described in the 'Protocol Summary' section. This is the simplest working form of P2P content delivery explored in this thesis. In summary, the client connects to the coordination server, obtains metadata for the file, connects to peers, then obtains chunks of the file from those peers. The coordination server recommends all of the currently connected peers, the client connects to and uses them all, and the client does not obtain any chunks via HTTP unless it fails to get it from the first peer asked.

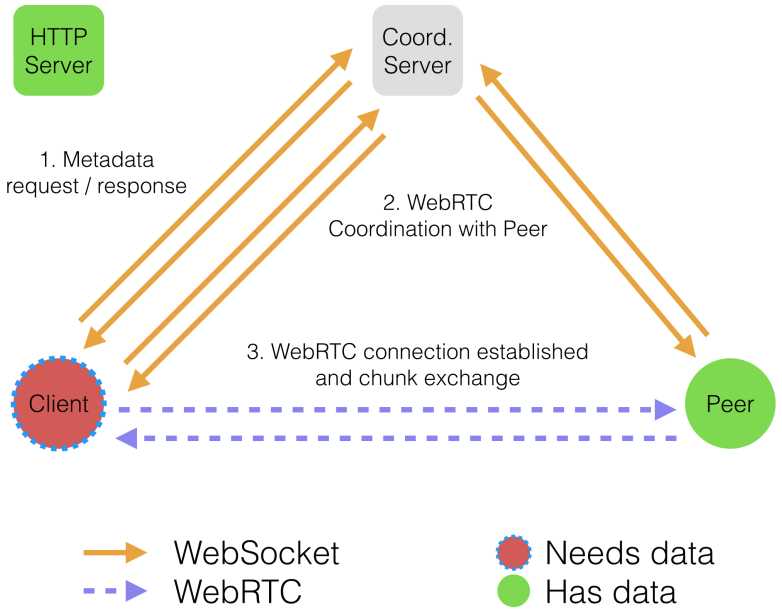

Figure 2: Basic DCDN Protocol

Results: The basic system showed a lengthy startup time as expected. Nearly half a second passed before the first byte was received. However, once data began to flow, the system delivered about 3.5 MBps which would be more than sufficient to stream audio or video. On the other hand, the high latency to first byte makes this system untenable for delivering image content as it was noticeably slower than the HTTP image download alongside it on the test page. Regarding the user experience, the HTTP image painted in as data was received but the DCDN image was entirely blank until the last byte was received at around 1.5 seconds.

4.3 HTTP Pre-fetch

Browser-based P2P content delivery systems such as DCDN have to allow graceful fullback to standard HTTP or risk of alienating a large portion of users with incompatible browsers. Since all URLs served through them must also be accessible via HTTP, it is quite possible it could be used to augment the performance of the client's download. This is a distinct advantage over desktop P2P systems in which the P2P protocol is the only way of accessing the content. BitTorrent's Web Seed is a notable exception, but due to competing and incompatible protocols, it is not in very widespread use [10] [7].

Because of the comparatively complicated setup that P2P systems must

perform before useful data can be transmitted, HTTP can be used to obtain chunks much sooner in the page-load time-line than DCDN's P2P. Much of this is due to the lengthy setup period of WebRTC Data Channels which require the transmission of several session descriptions before the connection is opened. Fetching chunks before peers become available could greatly improve the time to first byte. For streaming audio or video, this can make the difference between playback starting nearly immediately or after a long period.

On the other hand, using HTTP for content retrieval reduces the load balancing effect of using DCDN to distribute content since the HTTP server is once again involved in at least part of every download. However, this technique allows the content distributor to selectively prioritize their customer experience or their needs for bandwidth reduction. In this evaluation the client will obtain the first 20% of the file's chunks via HTTP as soon as it has received the file's meta data. All remaining chunks will be retrieved from peers.

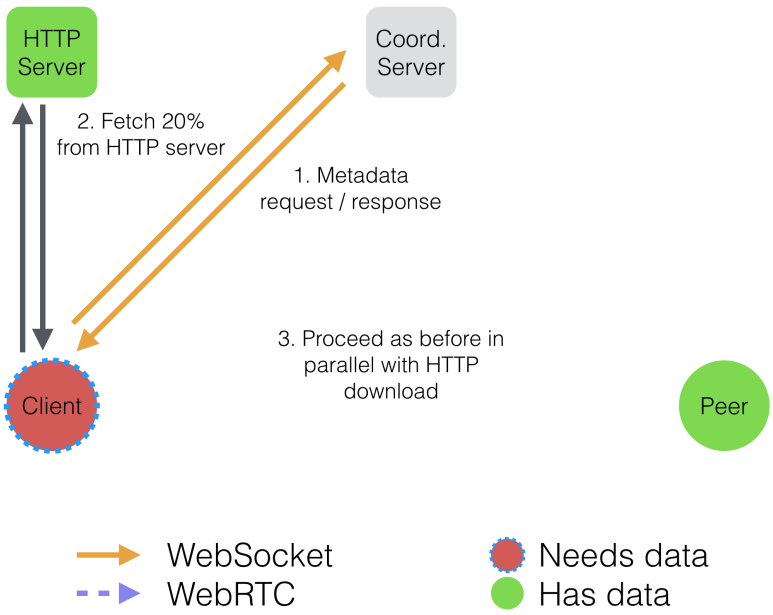

Figure 3: Changes for HTTP Pre-fetch

Results: HTTP Pre-fetch was successful in cutting the time-to-first byte by about 100ms redcucing the total latency to about 360ms. This reduction is quite significant but not enough to compete with HTTPs time to first byte of 7 ms on localhost. Even on the Internet, HTTP often delivers the first byte in as little as 100ms, or 1/3 the time of this DCDN variation.

This variation exhibited an unexpected result. The time to last byte was longer and thus the effective bandwidth of the system was actually decreased. The cause of this anomaly merits further investigation, but I suspect it is due to contention for the network stack between the HTTP pre-fetch and WebRTC / WebSocket communication for peer connection.

4.4 HTTP HEAD-start

The benefit of the HTTP pre-fetch optimization described above is somewhat mitigated by the need to obtain the file's metadata first. This is necessary in the basic DCDN protocol since the chunk size and content-length (which determines the number of chunks) are unavailable before the metadata arrives. This would not be an issue except that the connection to the coordination server relies on a WebSocket. WebSockets provide low-latency 2 way communication once established, but the process of opening one can take several hundred milliseconds. If the file in question is small, it could have been downloaded in entirety before the WebSocket has opened.

While communication with the coordination server is the only way to obtain a file's chunk size, it is quite easy to obtain its content-length via an HTTP HEAD request. An HTTP HEAD request returns only the HTTP headers, but not the content making it a quick way to get this data about the file. With this information and a prearranged chunk-size, the client could begin obtaining chunks via HTTP as soon as the HEAD request returns. The advantage of doing so is that chunks can be pre-fetched from HTTP before the Web Socket has even opened, giving the client a 'HEAD-start'. At this point no peers are available so this optimization operates similarly to the HTTP pre-fetch described above, only sooner in the time-line.

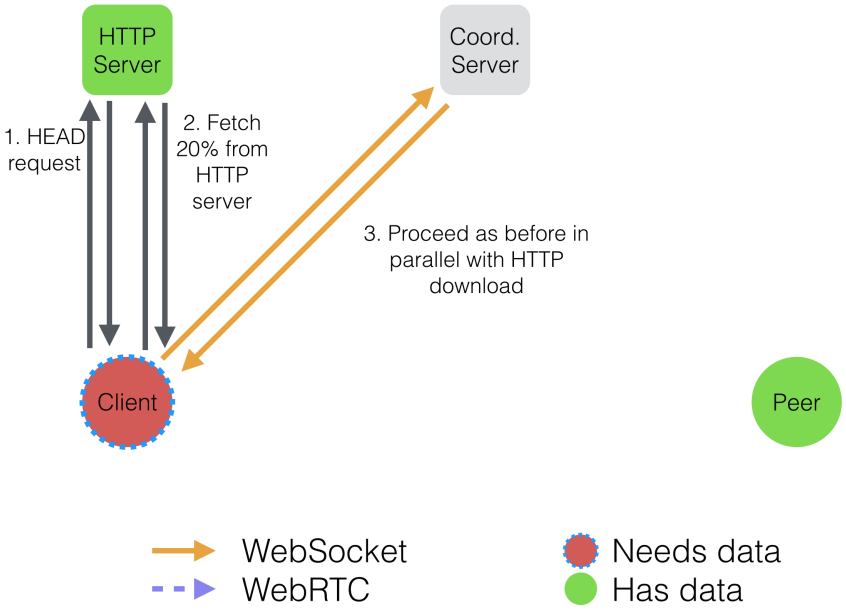

Figure 4: Changes for HTTP HEAD-start

Results: This change resulted in the best performance out of all the variations tried. It maintained the lowered time-to-first byte seen in the HTTP Pre-fetch configuration while also allowing the highest effective bandwidth of all variations. Compared to the basic system's bandwidth of 3.41 MBps, this system's increase to 3.60 MBps is significant. I suspect that allowing the HTTP activity to occur before the metadata was fetched allowed the traffic to complete before peer coordination began. In this way, the time to last byte was not adversely impacted and was actually lowered by about 160ms compared to the basic system since fewer chunks had to be fetched from peers.

4.5 Mixed HTTP and P2P

Similarly to the previous systems explored, this change combines traditional HTTP content download with P2P delivery. The main change in this case is that HTTP will not only be used for the pre-fetch period, but will be used throughout the download to obtain chunks. The HTTP server will be treated as any other peer in the round-robin peer selection system. If this enables faster content delivery, this could enable a web master to 'tune' performance based on the amount of load-reduction needed and their needs for good customer experience. This would be accomplished by adjusting the portion of content

served via each protocol.

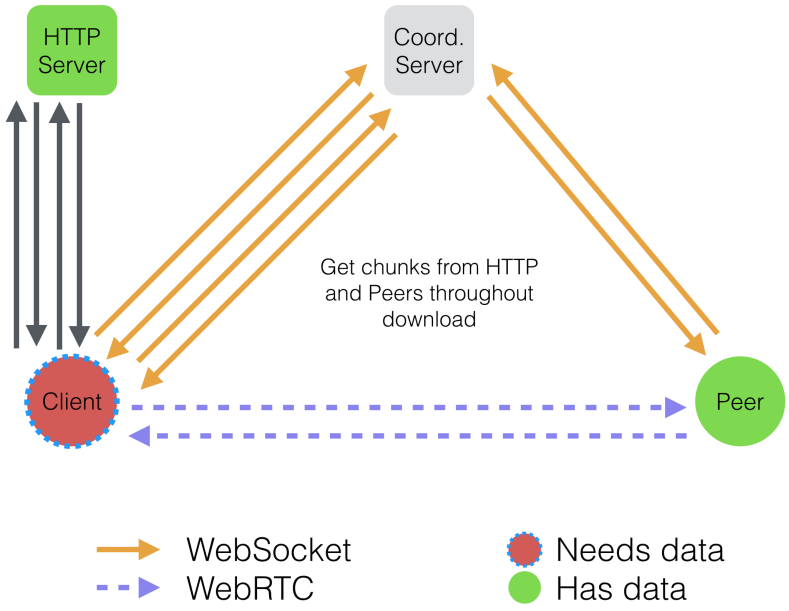

Figure 5: Changes for Mixed HTTP and P2P

Results: This system did not live up to expectations and instead resulted in speeds nearly equal to the basic system but without as dramatic load-balancing effects. The only significant statistic change was in the percentage of content served over P2P which dropped to nearly 50%. In other words, the HTTP server took half the load of serving the content but without noticeable gains in performance.

4.6 Comparison of configurations

Overall, the performance of the system saw only minor changes by mixing HTTP and P2P techniques. All forms of DCDN showed about 3.5 MBps transfer rate in the test setup with the exception of the HTTP Pre-fetch system which showed slower average speeds. Regardless, the transfer rate shown by DCDN is quite sufficient to enable streaming of web video. In fact, the rate is higher than Netflix reports its fastest consumer Internet Service Provider (ISP) in the U.S., Google Fiber, provides them for video streaming [14]. This implies that while DCDN has significant startup latency and serves small files poorly, that it is capable of streaming HD video very effectively.

	Time to first byte (ms)	Time to last byte (ms)	Percent served over P2P	Effective bandwidth (MBps)
Basic system	464.71	1411.92	100.0	3.41
HTTP Pre-fetch	369.11	1536.02	79.63	2.77
HTTP 'HEAD start'	353.77	1251.81	79.63	3.60
Mixed HTTP and P2P	386.13	1312.77	54.89	3.49

Table 2: Results from all DCDN variations served from localhost

In order to show that DCDN is capable of serving content effectively on the Internet as well as on the localhost test-bed, the following data shows the basic system's performance when the HTTP and coordination servers are relocated to a Virtual Private Server (VPS) in Los Angles, CA. Performance is not drastically altered by this change.

	Time to first byte (ms)	Time to last byte (ms)	Percent served over P2P	Effective bandwidth (MBps)
Localhost	464.71	1411.92	100.0	3.41
Los Angeles, CA	470.16	1335.47	100.0	3.74

Table 3: Comparison of basic system served from localhost or Los Angles, CA

5 Discussion

The HTML5 P2P framework is still in development and has yet to reach adoption outside of the Chrome and Firefox browsers. Due to its developmental nature, I encountered a number of roadblocks while working on DCDN that could be alleviated by improvements to web technology soon-to-be released. Lastly, Ill discuss the possibility of 2 moonshot improvements that could enable new user experiences or simply provide better performance.

5.1 Roadblocks

In its current form, DCDN is limited by both the technologies it relies on and the support of various web browsers for them. In this section, I will discuss some of the lessons learned while building this system as well as comment on the work-in-progress nature of the technologies I've built upon.

5.1.1 JavaScript binary API

While the additions of WebSockets and WebRTC to JavaScript provide surprisingly effective control over networking in the browser, they are recent additions to a language designed for far simpler tasks. This is most apparent when working with the binary versions of WebSockets or RTC Data Channels since JavaScripts support for raw binary is somewhat patchwork. While excellent primitives like Typed Arrays exist which allow for efficient view, modification, and creation of raw binary buffers they can only be used with APIs which explicitly support them.

Binary mode WebSockets, WebRTC Data Channels and XML HTTP Requests are examples of APIs that utilize Typed Arrays. On the other hand, the persistence infrastructure such as HTML5 local storage and IndexDB only allow the storage of UTF-8 text (Note: IndexDB in Firefox supports Blob storage at the time of writing) (Note: Hours before the publication of this thesis, it was announced that Chrome will get Blob support in the next few days [1]). While it may be possible to serialize arbitrary binary into and out of text strings this is not a very reliable or efficient solution. Chrome's File System API provides an effective solution that supports all of the binary types, but the API has failed to be adopted by other browsers and is considered dead by many [13].

So far we have only covered static arrays of binary data, but there is a second type of binary primitive which is necessary to efficient content delivery: the stream. A stream is the most natural representation of partially complete binary content and yet JavaScript does not support a Stream primitive. A simple stream would allow chunks to be obtained by DCDN and consumed by the DOM easily and efficiently via a straightforward and familiar interface. If it were possible to create a URL to a stream, in the same way one can be made for a Blob, this would solve the currently difficult problem of inserting content into DOM tags. This kind of DOM interaction brings about the next roadblock to seamless content delivery.

5.1.2 DOM interaction

Once data has been obtained by DCDN it is surprisingly difficult to get it from JavaScript into the tag it fulfills. For some reason, JavaScript has no ability to push content directly into the contents of a media tag in the way it can fill a text or HTML element. The only way to fill the content with arbitrary data is to change the tag's source element to a URL pointing at the data. The browser then follows its normal procedure for obtaining content from the URL.

Thus, the accepted method of filling DOM content with raw data from JavaScript is to turn the data into a Blob (a rather expensive process for large data), generate a Blob URL to the data which is now in the 'Blobstore', and finally set the DOM elements 'src' attribute to the URL. This works well enough for small files where the entire contents can be downloaded quickly, but poorly for large content since no image, audio or video data will appear until the entire content is available, Blob'ed and yielded.

One solution is to generate Blob URLs to partial content as it is downloaded and then repeatedly switch out the source URL of the tag. This works as expected and shows a partially loaded image, or partially watchable video, but is a hack at best. The expense of creating multiple blobs increases as the available content grows to the point where a large part of the browser's processing time is spent Blob'ing new data. While acceptable for small to medium images, this solution is entirely untenable for video content.

Luckily there is a kind of stream API for audio and video tags appropriately called the Media Stream API. This API allows exactly the behavior we desire: the ability to hand off binary buffers to a stream that is then used to fill the contents of the media tag. That said, the system as implemented is nowhere near the simplicity of a binary stream. Aside from flaky implementation details I'll describe next, the Media Stream API only works for audio and video tags making it fairly narrow in applicability. Its further limited to a subset of the supported audio and video codecs supported by HTML5. Even then, it is picky about the encoding details of the source material. Of the three WebM videos I attempted to use with it, one worked. I was unable to find an explanatory difference between the 3 videos. Other quirks include seemingly random freezes or hesitations in the streaming media even when the stream has been passed the full content. As previously mentioned, this system does not provide a solution for filling img tags, which are still the most common media element on the web.

Even when coerced into functioning, the Media Stream API as currently implemented is not east to work with. The API is quirky and has strange (undocumented) restrictions such as allowing only one buffer to be appended in a row without events being handled in-between. A functional hack arround this issue involves using 0-millisecond timeout callbacks to append buffers. This causes each buffer to be appended inside a separate event handler which allows the Media Source API to do its work between each event. This seems to imply that the Media Source API's buffer management is done on the main JavaScript thread. This is supported by my observation that the second of two buffer appends will fail unless other events are allowed to process even if a significant (several second) delay is added between each append.

5.1.3 User behavior

Above and beyond the challenges of implementing an efficient content distribution system in a language designed for web scripting is the challenge of creating an efficient P2P network using short-lived browser-based clients. Unlike typical P2P clients which tend to operate in the background for the duration of the user's computer session, browser-based clients run a page's JavaScript for only a few seconds to a minute before the user navigates to another page. Even for transfers of small files, this means that all clients must be treated as unreliable and very temporary.

This short client longevity is a challenge for a low-latency P2P systems since the coordination server's peer recommendations are based on data that is valid for only a very short time. It is very likely that a peer could disconnect from

the swarm between the time the server recommends them to a client and the time the client receives that message. More importantly, it means that any models of file chunk availability maintained by the coordination server must be refreshed much more frequently than in a typical long-lived P2P system. Peers obtain chunks very quickly, seed them for a matter of seconds and then disappear just as quickly. Luckily, this effect is somewhat mitigated by the fact that the largest content also takes the longest time to consume. Videos, which make up for much of the bandwidth usage in the US, are easily served by P2P systems and encourage a client to stay online for the duration of the video.

5.2 Improvements in the pipeline

Throughout the course of my research it has become clear that the JavaScript browser API is undergoing substantial change to enable desktop-class web applications. While WebSockets and WebRTC have been largely completed and standardized, there is further progress on the horizon for the performance of browser-based content distribution. This section will cover some improvements on the way in the form of new web technology and my ongoing work on DCDN.

5.2.1 Service Workers

Service Workers are a concept currently in development and partially released in the canary builds of Google Chrome. Their developers describe them as a solution the the issues of the HTML5 Application Cache, an oft ridiculed [8] solution for offline caching of content. Service Workers provide programmatic control over how an HTTP request is handled before the request even reaches the browsers cache. In its simplest form, the technology allows a web page to register a Service Worker which will be stored in the browser and used to fulfill all subsequent requests for URLs at the same origin. The interesting part is that after being installed, the service worker is activated before even the HTML of the page is requested on subsequent visits and can intercept and control the retrieval of all sub-resources.

This concept is one of the first that allows JavaScript to run before its host page has been loaded and also gives unparalleled ability to intercept requests. Currently, DCDN requires that content tags use special 'data-src' attributes rather than the typical 'src' attribute to prevent the browser from retrieving content via normal HTTP. Until JavaScript loads, then interprets this custom attribute, and finally requests its content through DCDN, no effort is made to obtain the content. In some tests, DCDN doesn't even see the request for the URL until 100 milliseconds into the page load time-line; by contrast a typical image request starts in as little as 25 milliseconds. In addition, since the browser is unaware that content will ever be filled in, it typically displays some form of visual feedback indicating failure to the user until DCDN is able to obtain some content.

Service Workers will allow for more intelligent interception of requests but also allow for programmatic control of the browsers cache. As discussed in the

section regarding JavaScript's binary APIs, there is no easy and efficient way to persist chunks which have already been obtained via P2P in a way comparable to the browsers large, low-latency HTTP cache. Service Workers enable the worker to add items to a cache specific to the worker, thus solving this problem. Once a URL has been obtained via DCDN, the service worker could then store its contents in the cache preventing the need to re-download it the next time. Because the data in the cache is programmatically accessible to the worker, this solution even preserves the ability to serve the content even if it has not been downloaded by DCDN on that particular page-load.

At the moment, there is no indication that Service Workers will be able to yield partial content to the user agent in the way that an HTTP server can using chunked encoding. This is necessary to enable the ideal user experience in which content loaded using P2P would yield partial content as it becomes available. Users who have watched an image 'paint in' as the HTTP server returns data will be familiar with this technique. In response to a ticket I filed on the subject, a developer confirmed that it is not possible to yield partial content to a request, but that this ability is being strongly considered [19].

Service Workers are only partially specified at the time of writing and the implementation in Chrome Canary is far from complete. Further information can be obtained at the projects GitHub page [4] at `https://github.com/slightlyoff/ServiceWorker/`.

5.2.2 Heuristic peer recommendation

In addition to the system configurations evaluated above, a more significant change is in development. This alteration would exploit the predictability of video chunk ordering to heuristically predict the state of each clients cache. This should allow for the coordination server to provide better peer recommendations.

Unlike traditional desktop P2P systems which typically retrieve the full file before yielding it to the user, web based content delivery is almost entirely for immediate use. Whether the user is streaming audio, video or simply viewing images, they want the content as soon as it is available. In desktop systems, chunks are retrieved in order of convenience according to some heuristic such as 'rare chunks first'. Generally, the goal is to increase availability of the file in the swarm while still completing the entire download as fast as possible. By contrast, the web user's demand for near-instant use of the content necessitates an entirely different strategy: in-order retrieval.

This style of content retrieval has only just begun to be popular in desktop P2P applications, such as Popcorn Time, which stream content from a BitTorrent swarm. While in-order delivery has been classically seen as a strength of client-server schemes and a weakness for P2P, there are certain adaptations which can close the gap. One example are optimizations based on the comparatively deterministic behavior of the client in obtaining chunks. The in-order nature of streaming media allows for very accurate prediction of a peer's chunk cache based only on the time they initially connected.

In DCDN and other web P2P systems, this property can be exploited by

the coordination server when recommending peers to the client. Peers who have the required chunks in their caches can provide better service to the client and would be better candidates for peer recommendation. In the case of streaming media, peers that started shortly before the client are very likely to have already obtained the chunks the client now needs. The client can then request these chunks with a very high success rate without the need for time-consuming negotiation between peers such as the exchange of cache-status bitmaps.

5.3 Moonshot Improvements

In addition to the improvements currently under development, there are a few other areas of exploration that may yield advances for P2P content delivery. I term these 'moonshot' improvements because they are unlikely to be the focus of much attention in the near future, but may become points of high interest if P2P content delivery takes off.

5.3.1 Donation of idle resources

One such moonshot concept would be the establishment of a system whereby a user could donate their web browser's idle time to serving content in exchange for incentive from the content provider. This resembles the SETI@home project [5] which uses a computational screen saver to analyze satellite data, aiding in the search for extra-terrestrial life. As applied to content delivery, this concept would take load off of the commercial server infrastructure of content providers in exchange for a small discount on a streaming subscription or an ad free experience. Until large content companies choose to offer incentives for opt-in P2P contribution this type of system is unlikely to be adopted.

From a technical perspective, donated idle resources would mean a more stable platform from which to serve content since the P2P client would no longer be tied to the lifespan of a single web-page. Implementing this could be as simple as allowing DCDN to run in a browser extension in addition to its current form as a page script. Users who have been incentivized to install the extension would permanently cache and serve selected resources to other users on the site. This would require user intervention to install the extension, but that is justified by the explicit incentive structure.

5.3.2 Heuristic optimizations for P2P networking

The second moonshot is less likely to be explored until the use of WebRTC PeerConnections becomes widespread. These P2P connections take a very long time to establish and require a custom message-passing channel to be established beforehand. It seems that a browser could heuristically predict that a connection is likely to be established by a certain web-page and begin the connection process speculatively, later handing the result to the page script if the connection is actually requested. This sort of speculative execution is a common optimization in microprocessors, but has even been used in the realm of network connections

before. The Systemd init service on Linux uses a similar technique to create network connections before the network card is fully online [11].

6 Conclusion

Peer-to-peer content delivery would restore some balance to the Internet which is currently dominated by companies with large financial resources. By dramatically reducing the capital investment required to share audio and video it would allow anyone with a computer and an HTTP server to once again share their content on a level playing field; much as was the case with original text based Internet. It would also begin to restore the balance of upload and download traffic to end users which has the potential to make more effective use of the Internet capacity we already have.

In its current form DCDN proves that this kind of invisible P2P content delivery is possible and even viable for certain types of media. Compared to HTTP it exhibits a lengthy time to first byte, but this is not a bar to its use for streaming audio and video. Additionally, some of the less-than-perfect aspects of its performance may be remedied with coming advances in web technology. In its current form, this project could be used in limited commercial application, although it needs a bit more work to be fully stable. Regardless, even at this early stage, it is clear that the technology is viable, that its use is motivated and that it has the potential to solve some of the capacity issues faced by the web as its use continues to scale beyond its original conception.

It should be clear by now that the technologies involved in creating web based P2P applications have not yet reached maturity. Many are still being standardized, many are incomplete and many more are just ideas at this point. For this reason, the work described here on DCDN and the results demonstrated should be taken as a work in progress. As standards are completed and new technologies invented, I hope that this research will serve as a building block on the road to a more distributed web.

References

[1] Anonymous Google Developer jsb[OBSCURED]@google.com. (2014, Jun 5). Issue 108012: IndexedDB should support storing File/Blob objects [Online]. Available: https://code.google.com/p/chromium/issues/detail?id=108012#c153

[2] A. Bergkvist et al. (2013, September 10). WebRTC 1.0: Real-time Communication Between Browsers (W3C Working Draft) [Online]. Available: http://www.w3.org/TR/webrtc/

[3] B. Cohen. (2012, Oct 20). The BitTorrent Protocol Specification (3rd revision) [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html

[4] A. Russell (2014, June 5). The Service Worker Specification [Online]. Available: https://github.com/slightlyoff/ServiceWorker

[5] D. P. Anderson et al. (2011). About SETI@home [Online]. Available: http://setiathome.ssl.berkeley.edu

[6] D. Ristic. (2014, February 4). WebRTC data channels [Online]. Available: http://www.html5rocks.com/en/tutorials/webrtc/datachannels/

[7] Headlight Software, Inc. (2006, March 13). HTTP/FTP Seeding for BitTorrent [Online]. http://www.getright.com/seedtorrent.html

[8] J. Archibald. (2012, May 08). Application Cache is a Douchebag [Online]. Available: http://alistapart.com/article/application-cache-is-a-douchebag

[9] J. Hiesey, F. Aboukhadijeh and A. Raja. (2013). PeerCDN [Online]. Available: https://peercdn.com

[10] J. Hoffman. (2011, August 22). HTTP-Based Seeding Specification [Online]. Available: http://www.webcitation.org/6184q7Pjn

[11] L. Poettering. (2011, May 18). Systemd for Developers: Socket Activation [Online]. Available: http://0pointer.de/blog/projects/socket-activation.html

[12] Mozilla Developer Community. (2014, March 18). WebSockets [Online]. Available: https://developer.mozilla.org/en-US/docs/WebSockets

[13] Mozilla Developer Community. (2013, Sept 6). File System API guide [Online]. Available: https://developer.mozilla.org/en-US/docs/WebGuide/API/File_System

[14] Netflix Inc. (2014, April). USA ISP Speed Index [Online]. Available: http://ispspeedindex.netflix.com/usa

[15] N. Martindell. (2014, May 18). DCDN: A Distributed CDN based on modern web technologies [Online]. Available: https://github.com/DaemonF/dcdn

[16] Swarm Labs, LLC. (2014). What Is Swarmify? [Online]. Available: http://swarmify.com

[17] S. Dutton. (2012, July 23). Getting Started with WebRTC [Online]. Available: http://www.html5rocks.com/en/tutorials/webrtc/basics/

[18] S. Guha and P. Francis. "Characterization and Measurement of TCP Traversal through NATs and Firewalls," in Proceedings of Interet Measurement Conference (IMC), Berkeley, CA, Oct 2005.

[19] tabatkins (pseudonym). (2014, May 17). Partial data / Range requests - Issue #280 [Online]. Available: https://github.com/slightlyoff/ServiceWorker/issues/280