# Zauberflöte: A Transparent Peer-to-Peer CDN

Anish Athalye
*aathalye@mit.edu*

Ankush Gupta
*ankush@mit.edu*

Katie Siegel
*ksiegel@mit.edu*

## Abstract

Zauberflöte is a peer-to-peer content delivery network. The system uses WebRTC to cache and distribute content between peers. Zauberflöte provides an easily accessible way for developers to alleviate content delivery bandwidth concerns by marking DOM elements such that Zauberflöte will fetch these resources from peers with that resource. The Zauberflöte system has server-side tracker and WebRTC signaling channel components, as well as client-side scripting that requests and delivers content between peers. This version of the system implements basic chunking, parallelism, and fault tolerance. In this paper, we describe the motivation for Zauberflöte and the design and implementation of the different components that comprise the system. We then detail future improvements that could be made to increase the robustness and performance of the overall system.

## 1 Introduction

It is common for personal blogs and other small websites to gain sudden attention on large, high-traffic websites such as reddit. Since these websites usually experience much smaller traffic loads, the hardware devoted to hosting them is typically not powerful enough to cope with this large influx of content requests. These sudden, unexpected bursts of traffic often result in websites loading at extremely slow speeds, or not loading at all, and is colloquially known as the "reddit hug of death" or the "reddit effect."

Typical means for dealing with this situation involve utilizing Content Delivery Networks (CDNs), but these are expensive, especially for websites which only experience such high loads on an occasional basis. An ideal solution would be a lightweight means to mimic the benefits of an ordinary CDN, which is typically backed by a robust network of machines.

In this paper, we present Zauberflöte, a peer-to-peer CDN. Zauberflöte provides an easy way for developers to distribute content to clients via a peer-to-peer network, rather than directly from the central server that hosts the content. Zauberflöte is comprised of a server-side component and a client-side component. The centralized, server-side component both tracks resource propagation metadata and acts as an assist for opening peer-to-peer data channel connections. The lightweight client-side component requests resources from peers and distributes resources in response to peer requests.

Zauberflöte's design is based on the BitTorrent protocol [1], by which clients requesting content attempt to receive chunks of data from other clients, rather than retrieving the entire file from a central server. As with BitTorrent trackers, any developer could host their own tracker and service it to connect peers requesting a resource with peers distributing the resource.

The Zauberflöte client-side code runs purely in the browser and is completely transparent to the user. The user does not have to install any application on their computer to obtain desired resources peer-to-peer. As a result, the Zauberflöte client code only runs when a user is currently visiting the website, adding no burden to the user from constant background processes.

In designing Zauberflöte, we took into consideration the unique problems presented by browser-to-browser content sharing. Website visitor churn is often very high with any type of website; our system had to be robust under these common conditions. Zauberflöte forms peer-to-peer connections quickly and immediately streams requested resources through these data channel connections. Zauberflöte also recovers quickly from data request failures; the Zauberflöte data chunking component minimizes latency from these failures and quickly retries resource requests with other peers.

## 2 Zauberflöte API

Zauberflöte supports peer-to-peer loading for CSS (via `link` elements), Javascript (via `script` elements), and images (via `img` elements). For an existing website to utilize Zauberflöte, the developer simply needs to make two changes to the DOM elements.

The first change is renaming the `src` or `href` attribute to `data-zf-fallback`. Zauberflöte uses the `data-zf-fallback` attribute in order to (1) prevent the browser from automatically fetching the resources, and (2) provide a URL to be used in the event that the peer-to-peer network is unable to serve the content.

The second attribute that must be present on a DOM element that is fetched by Zauberflöte is the `data-zf-hash` attribute. The `data-zf-hash` attribute is the SHA1 hash of that resource, which can be computed by the developer, and is utilized by Zauberflöte to ensure integrity of data fetched via peer-to-peer interaction.

## 3 Implementation

The Zauberflöte architecture consists of several loosely-connected components (see Figure 1). At the highest level, there is a download manager that provides a reliable download service. It relies on the tracker interface and the connection manager. The tracker interface facilitates communication with the central tracker for information about peers and file sizes, and the connection manager provides an interface for peer-to-peer message transfer.

Our components rely on two standard APIs provided by modern browsers, WebSockets [2] for client-server communication and WebRTC [3] for peer-to-peer communication.

### 3.1 Connection manager

The connection manager is responsible for establishing connections between peers and facilitating peer-to-peer message transfers. It is a thin layer on top of WebRTC that handles maintaining connections with multiple peers, associating peers with peer IDs that are assigned by our tracker. It manages all signaling through the signaling channel, which implements signaling through our central server using WebSockets. At this time, this signaling server is the same server that is running the tracker, but they are logically separate components, so theoretically, they could be decoupled if the need arises for better load balancing.

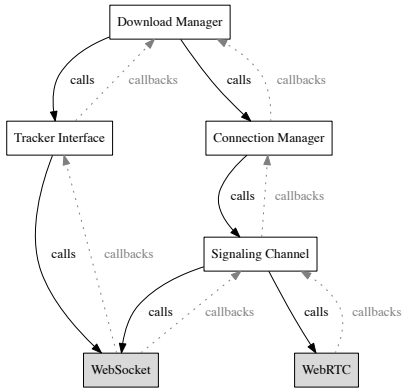


Figure 1: Relationship between different client-side components of the Zauberflöte peer-to-peer download service. Components in gray are standard browser APIs.

### 3.2 Download manager

The download manager is the top-level client interface for peer-to-peer downloads. It provides an API by which the client can provide a hash, and optionally, a fallback HTTP URL. The download manager will asynchronously download the content associated with the hash and call a user-provided callback function when finished. The download manager implements reliable download, preferring peer-to-peer over HTTP when possible.

#### 3.2.1 Chunking

Zauberflöte divides assets into chunks; chunks are individually requested and sent between peers. We use a standard chunk size of 25 KB in our implementation; preliminary tests showed this chunk size to be reasonable. When a client asks for a resource, the download manager for that client determines the number of chunks for that resource from the resource's total size. The download manager then evenly distributes requests for these chunks among the peers with the resource.

Requests for chunks time out on a per-chunk basis. We track the time Zauberflöte last sent a request for each chunk; if this time exceeds a cutoff, we randomly choose another peer with the resource and send the chunk request to that peer. We periodically retrieve an updated peers list for this resource from the tracker. Zauberflöte uses exponential backoff when sending chunk requests so we don't flood peers with requests as a result of slow network connectivity.

#### 3.2.2 Hash validation

When fetching content peer-to-peer, it is necessary to authenticate data that is downloaded. Zauberflöte achieves this by using a cryptographic hash function to authenticate data that is downloaded. Assets that are to be downloaded peer-to-peer are identified by their hash, and once the download manager finishes downloading the content from peers, it verifies that the hash of the contents matches what is expected. The download manager rejects content if there is an authentication error. In the current implementation, if this occurs, the download manager falls back to downloading the content over HTTP from the fallback URL.

#### 3.2.3 Parallelization

Zauberflöte divides assets into chunks primarily to enable parallelization. Because individual peers may not have high outbound bandwidth, it is beneficial to download in parallel from many peers. Our implementation parallelizes chunk downloads, with pending chunk requests balanced between available peers in the network. If certain peers are slow to respond, the download manager automatically requests chunks from faster peers to optimize download time. Parallel downloads help reduce download latency.

#### 3.2.4 Fault tolerance

Our download manager implements reliable download semantics. The implementation is resilient to peer failure, automatically re-requesting chunks from live peers when necessary. In the worst case, even if all peers go down, the implementation falls back to downloading over pure HTTP from the centralized server as a last resort. In any case, the download manager guarantees that the data will be downloaded and that the download will be authenticated.

### 3.3 Tracker interface

Via the tracker interface, the download manager can request a list of peers that have a certain resource, specifying the desired resource via that resource's unique SHA1 hash. After a client receives the full resource, the download manager publicizes that this client has the resource via this tracker interface, letting the central tracker know that it possesses the resource.

### 3.4 Signaling channel

The signaling channel facilitates opening the WebRTC data channel connection between peers. The WebRTC protocol requires one peer to send a WebRTC offer and ICE candidate to the other peer, and that other peer to send back a WebRTC answer, for a peer-to-peer data channel to be opened. The server-side signaling channel facilitates this information transfer between peers by directing offers, ICE candidates, and answers to their specified peer recipient.

### 3.5 DOM injection layer

The DOM injection layer facilitates the link between the DOM as displayed in the browser and the entire peer-to-peer system. The DOM injection layer creates a WebSocket, and then instantiates a Tracker and Signaling Channel. It uses these to then instantiate a Connection Manager and in turn, uses this to instantiate a Download Manager.

The DOM injection layer then iterates through all the DOM elements with the `data-zf-hash` and `data-zf-fallback` attributes. It the values of these attributes to fetch the data via the Download Manager. Upon receiving the data for each web element, the DOM injection layer creates an in-memory Blob and generates URL for this object. It then obeys the following rules, which can easily be expanded to support other DOM elements:

**Script elements** have the `type` of their Blob set to `text/javascript`. They then have their `src` attribute set to the URL of the in-memory Blob.

**Link elements** have the `type` of their Blob set to `text/css`. They then have their `href` attribute set to the URL of the in-memory Blob, and `rel` attribute set to `stylesheet`.

**All other elements** have the `type` of their Blob set to `application/octet-stream`. They then have their `src` attribute set to the URL of the in-memory Blob. As most browsers can intelligently determine the content of `application/octet-stream` data, this functions properly for Image nodes, and serves as the generic template for any other node.

### 3.6 Server implementation

There are two main server-side components to Zauberflöte. First is the tracker, which maintains a mapping from resource hashes to the peers that have that resource. Second is the WebRTC handshake channel, which passes information between peers to enable them to open WebRTC data channels.

#### 3.6.1 Tracker

The tracker is hosted on a remote server and tracks client connections. The client-side tracker interface opens a

WebSocket connection with the remote tracker, which registers the existence of this client. The remote tracker responds to requests from the client for lists of peers with a given resource. When a client publicizes that it has a resource through the tracker interface, the remote tracker updates the list of peers with the resource to include this client, as long as its WebSocket connection remains open. Upon WebSocket disconnection, the tracker "forgets" about the client and erases all data about the client's held resources.

#### 3.6.2 WebRTC handshake channel

WebRTC handshake information–the offers, ICE candidates, and answers–are sent through the client-side signaling channel through a websocket to the server. The client-side signaling channel specifies a peer to which this WebRTC handshake information should be sent; if the server has an open websocket connection with that peer, the server passes that message to the peer.

## 4 Evaluation

We tested Zauberflöte on a simple site that loads one large 768 KB image from a remote server. In our tests, we compare page load times using both plain HTTP and Zauberflöte in a range of scenarios. Specifically, we varied two parameters: the number of seeding peers and the number of leeching peers. A seeding peer is a peer that has the desired resource, which in our tests, was the single image on the site. A leeching peer is a peer that desires the resource, so must request it either from the central site host or from peers with the resource.

Our tests launched a certain number of seeding peers, then launched a number of leeching peers, measuring the average load latency for the leeching peers. We compare this latency data with the page load latency when the leeching peers must load the content from the central site host.

### 4.1 Simulating the "reddit effect"

Our tests involved programmatically opening browser windows and measuring load times. Due to limitations of the number of browser windows that can be reasonably opened and the memory overhead of Chrome, our browser testing environment of choice, we could not reasonably open enough browser windows to exactly imitate millions of simultaneous visitors to a site. Instead, we simulated the "reddit effect" by throttling the content delivery bandwidth of the central server hosting the desired resource. We limited the content delivery rate to 64 KB/s and performed benchmark tests that compared the performance of Zauberflöte against the HTTP content delivery latency under the throttled bandwidth conditions.

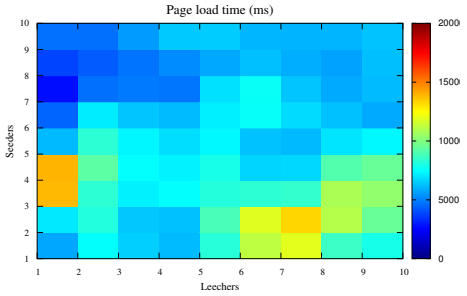


Figure 2: Average page load time (lower is better) over P2P for seeders vs leechers.

### 4.2 Testing infrastructure

Our testing infrastructure was made up of a virtual machine hosting our website, a tracker running on the host machine, and a collection of web browsers acting as clients. All test code ran on the same physical machine. We used Vagrant to provision an Ubuntu 14.04 LTS virtual machine running nginx 1.4.6 to serve a test page designed for benchmarking. To simulate a low-resource server / heavy loads, we used kernel traffic shaping to limit the outbound bandwidth of the virtual machine. To simulate browser clients, we wrote a test driver in Scala that made use of Selenium WebDriver and ChromeDriver to launch and control Google Chrome browser windows. The test driver spawned and communicated with browser windows and benchmark code through Javascript so it could measure page load times.

### 4.3 Results

#### 4.3.1 Scalability of Zauberflöte

The first empirical analysis that we performed was determining how page load time over Zauberflöte scales as we vary the number of users who have the page loaded (we call these users *seeders*) and the number of users attempting to concurrently download the page (we call these users *leechers*).

As seen in Figure 2, the performance of Zauberflöte depended on the seeder-to-leecher ratio. In general, the lowest latency in load time was achieved when the seeder-to-leecher ratio was highest. However, the system performed well in any condition with many seeder peers, reflecting the performance benefits provided by chunking of resources and parallelization of sending chunks. The system performed worst when there were many leechers and only a few seeders.

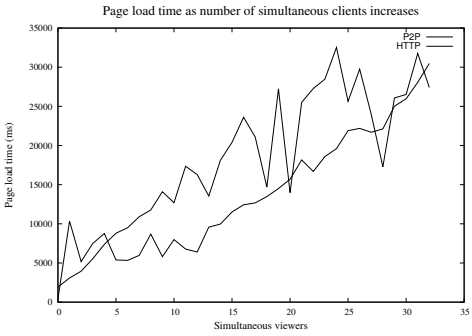


Figure 3: Average page load time (lower is better) over P2P compared to HTTP on throttled server assuming constant ratio of 1 seeder per leecher.

#### 4.3.2 Zauberflöte vs HTTP

Our next empirical analysis was to keep the ratio of seeders to leechers constant at 1, and compare the page load times for a throttled HTTP server vs Zauberflöte as we increase the number of users attempting to concurrently load the page.

As seen in Figure 3, Zauberflöte performed well in comparison to loading content over HTTP. While both experienced increases in content delivery latency with increasing numbers of peers requesting content, in general Zauberflöte had lower latency than HTTP. Note that our simulated "reddit effect" implies hundreds of thousands of other simultaneous connection; however, since we are simulating the effect, these imaginary peers cannot distribute content to new peers. So, Zauberflöte could potentially provide an even greater latency drop with more active seeding peers, which would be the case in a real-world scenario.

## 5 Future work

There remains numerous optimizations that would make Zauberflöte more performant when used on a deployed, high-traffic site. We detail these potential optimizations in this section, along with features which could allow Zauberflöte to handle more use cases.

### 5.1 Per-chunk hash validation

As of right now, Zauberflöte verifies the integrity of the downloaded resource once all chunks have been retrieved and joined. In normal circumstances where all chunks successfully download, this may not prove to be an issue. However, waiting until all chunks are downloaded and combined to verify the file means that if a single chunk had been corrupted or compromised, all chunks for the file would need to be downloaded, combined, and verified again.

An improvement to Zauberflöte that would mitigate this issue would be to have the Tracker module provide checksums for each chunk. From the user's perspective, the `data-zf-hash` attribute currently used to store the hash of the file would be replaced with the hash of the JSON object storing a list of the hashes for each chunk. Upon connection to the tracker, each client would receive that list and verify its integrity, and then be able to verify and redownload each chunk independently in the event of failure.

### 5.2 Hybrid downloads over HTTP and P2P

Another improvement to Zauberflöte would be to allow it to parallelize downloads via HTTP and P2P transfers simultaneously. By using the `Range` HTTP header attribute, Zauberflöte could fetch the same data as contained in P2P chunks from the HTTP fallback server. By utilizing both HTTP and P2P, Zauberflöte may be able to provide service speed that is expected in all instances to be faster than simply using HTTP alone.

### 5.3 Variable request sizes

Implementing variation in requested chunk sizes could allow Zauberflöte to independently determine a more optimal chunk size for each asset. For instance, we hypothesize that different chunk sizes are optimal for differently sized assets, and potentially for different media types. If Zauberflöte more intelligently determined the optimal size of chunks for each resource, overall download latency would decrease.

### 5.4 Bandwidth detection and prioritization

If Zauberflöte could detect which peers have higher bandwidth connections, and thus are more capable of swiftly delivering data to a client, Zauberflöte could prioritize requesting data from these peers. Perhaps Zauberflöte could request larger chunks from peers with higher bandwidth connections. Zauberflöte could also dynamically react to poor connections with peers during the chunk requesting process; chunk requests that subsequently time out could be redistributed more intelligently among peers with higher bandwidth connections.

### 5.5 Support for HTML5 media

It is possible to add functionality such that Zauberflöte can construct `ReadableStreams` (as specified by the Streams standard [4]) from the chunks of data as it receives them. Using this stream, the DOM injection layer

could feasibly replace HTML5 media elements such as embedded Video and Audio. Since these types of media are particularly large in size, being able to distribute them in a decentralized, peer-to-peer manner could provide substantial load reduction for the central server.

### 5.6 Automatic conversion to Zauberflöte

Developers can integrate their sites to use Zauberflöte by adding attribute tags to the site resources they wish to be distributed using the system. However, sites feature dozens of resources; as a result, adapting a site to use Zauberflöte could present a burden on the developer. If we were to provide a script that developers could use to change HTML such that all resources were loaded using Zauberflöte, this would ease integration pains and increase adoption.

## 6 Related work

Currently, alternative peer-to-peer content delivery networks do exist. The three prominent peer-to-peer CDNs are PeerJS, PeerCDN, and Peer5. PeerJS is open-sourced and similarly transparent, but only encapsulates the functionality of our connection manager module. PeerCDN [5] is a closed-source alternative providing similar functionality to Zauberflöte; it offloads site content delivery to users of a site, significantly lowering strain on the central source. However, PeerCDN was acquired by Yahoo! and no longer exists. Finally, Peer5 is a service used by companies today. Unfortunately, Peer5 is a paid service that is entirely closed-source; the Peer5 code is not transparent to the clients on which it runs.

## 7 Conclusions

Zauberflöte is an open-sourced system that will allow developers to easily utilize a peer-to-peer CDN to increase the content delivery bandwidth of their sites, even when operating under heavy request loads. Preliminary data shows that the system performs well under medium and high workloads, and when the seeder-to-leecher ratio is high. Future work could refine the system such that it works well in a wide variety of scenarios. Our results show that peer-to-peer CDNs are a promising low-cost alternative to CDNs that require an extensive network of server and server infrastructure.

## References


[1] B. Cohen, "The bittorrent protocol specification," tech. rep., Jan. 2008. http://www.bittorrent.org/beps/bep_0003.html.

[2] W3C Web Applications Working Group, "The websocket api," tech. rep., Sept. 2012. http://www.w3.org/TR/2012/CR-websockets-20120920/.

[3] W3C WebRTC Working Group, "Webrtc 1.0: Real-time communication between browsers," tech. rep., Feb. 2015. http://www.w3.org/TR/2015/WD-webrtc-20150210/.

[4] Web Hypertext Application Technology Working Group, "Streams standard." https://streams.spec.whatwg.org.

[5] PeerCDN, "Peercdn." https://peercdn.com.