Anying Li, Lara Araujo, Vinay Mayar

6.824 Final Project Writeup

Maygh: A Peer-to-Peer CDN

Github link: https://github.com/vinaymayar/maygh

1. Introduction

A large percentage of content loaded by websites is static content like images, videos, and scripts. However, distributing that content to many different clients is expensive because it requires a lot of bandwidth and server infrastructure. As a result, large websites use third-party CDNs, like Akamai, to distribute static content for them. This also turns to be expensive.

Maygh allows clients that have loaded static content to distribute it directly to other clients, thus reducing bandwidth on the server. Maygh is included by the developer in the client-side code of a website and requires no additional plug-ins to be downloaded by the end-users.

2. Design

On a high level, Maygh comprises a centralized coordinator and a set of clients, each of which is a user accessing the site and running the Maygh Javascript client code in his or her browser. The coordinator is responsible for establishing connections between clients who wish to share content, at which point the clients are responsible for actually transferring the content.

Maygh Coordinator:

The Maygh coordinator is a centralized server that is run by the website operator. It has two main purposes: serving as a directory for clients and their content and serving as a protocol server (also known as a signaling server) for client-to-client connections.

It fulfills its first purpose by maintaining state about which clients have what content through two maps. The first is the *content location map*, which maps each piece of content (identified by its hash of its data URI) to a list of clients (identified by their IDs) that have that content, and the second is the *client content map* which maps each client to a list of content that they have. When a client makes a request for a piece of content to the coordinator, the coordinator looks up which clients have that content in the *content location map*, randomly chooses one of them in order to load balance the requests, and returns that client to the requester. The coordinator keeps these two maps up-to-date by periodically sending heartbeats to all clients and removing dead ones from the maps.

It fulfills its second purpose by acting as a forwarding service for client-to-client specific messages when two clients wish to establish a connection. These include offer, answer, and ICE candidate messages.

Maygh Client:

The Maygh client code is a Javascript library to which the site operator must include a reference in their web page. When a user accesses the site, the Maygh client code runs in the browser, and the user becomes a Maygh client.

The Maygh client API supports two main functions: `connect` and `load`.

`connect()` – Starts a websocket connection with the coordinator on the port on which the coordinator is running.

`load(content_hash, id, src)` – loads the content with hash `content_hash` into the dom element with `id`. If the Maygh client is unable or fails to load the content from another peer, it loads the content from the source `src` on the web server. After the client has loaded the content from some location, it stores that content in persistent storage for transferring to other peers.

The Maygh client also includes functions that are specific to setting up and receiving connections from other peers as well as transferring and receiving content.

Maygh protocol:

The `load` function is where the bulk of the messages are sent in the system. It it called when the user loads the web page on every piece of content which is tagged by `Maygh.load()`. Figure 1 below shows a timeline of the messages that are sent when content is successfully delivered from another peer.

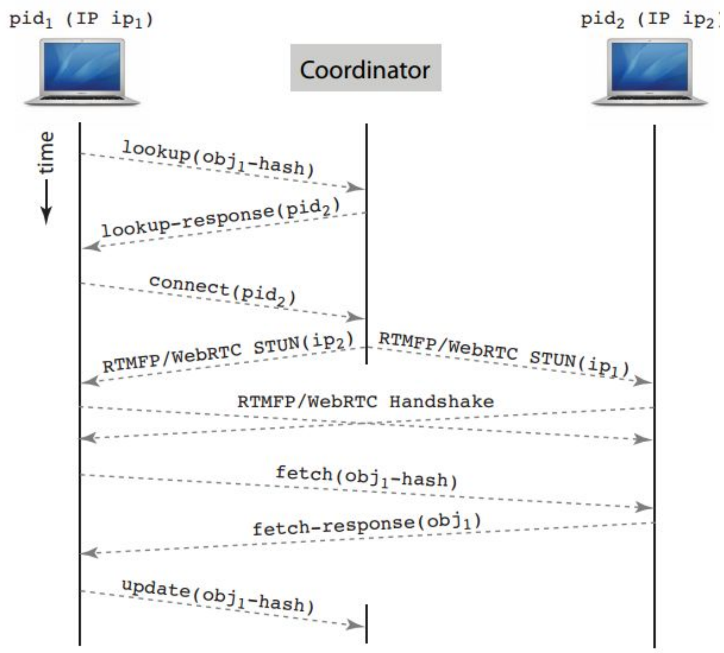

Figure 1: Maygh messages that are sent when a client (with pid_1) requests content (with hash obj-hash) successfully from another client (with pid_2).

On a load for a piece of content with the content hash `obj-hash`, the client (with ID pid_1) first makes a request to the coordinator with that content hash. The coordinator then looks up that obj-hash in the *content location map*, and responds with the ID of a peer that has that content (say client with ID pid_2). Client 1 then sends an offer to Client 2 through the signaling server (the coordinator in this case), and the two clients exchange a handshake and establish a peer connection. After the connection is established, Client 2 sends the content to Client 1, which then sends a message back to the coordinator to notify it that Client 1 has successfully received the content. The coordinator then updates its maps accordingly, and Client 1 stores the new content in persistent storage.

Note that should any of the steps above fail in any way, we default to loading the content from source. This is because we wish to keep latency as low as possible, and we know that loading from source is always a safe option. Failures that can occur include but are not limited to: the hash of the content sent from Client 2 doesn't match what Client 1 is expecting, the coordinator goes down during any part of the exchange, Client 2 is unresponsive during the handshake, Client 2 stops responding while sending a resource.

3. Implementation

Both the Maygh client and coordinator were implemented in Javascript; the client is a simple Javascript library, and the coordinator is a Node.js server.

The communication between the coordinator and the clients was done using the Websocket library `socket.io`. The advantages of `socket.io` over other websocket implementations are its ease to use and the fact that it came with its own heartbeat implementation, which we leveraged to implement Maygh heartbeats.

For the browser-to-browser communication required between peers the Maygh paper suggested the use of either WebRTC or RTMFP. We decided to use WebRTC because it was better documented and open-sourced. In order to use WebRTC to communicate between clients, we needed to write code to support the handshake process. The first client uses the coordinator as a signaling server to forward the handshake offer and the second client replies to that offer with an answer. Other than just sending the offer to establish the connection, the clients must also transmit ICE (Interactive Connectivity Establishment) candidates to their peer in order to establish a connection. This was also done using the coordinator as an intermediary.

Once a connection is created, clients can communicate with each other using the established data channel. Because WebRTC limits the size of each message transmitted, in order to transfer a large file, we must chunk it on the sender and then reassemble it on the receiver (for example, a 1.6MB file was chunked into 42 messages).

In order to cache the files in each client, we used the Javascript local storage as a persistent storage. This was enough for our simple website with a medium number of clients. However, if Maygh was being used by many different websites with a lot of static content, we would have to use a bigger storage, such as Firebase. We used SHA1 to hash the content, but this could be changed to a longer hash function like SHA256 to avoid collisions in larger websites.

To test our implementation, we made a test suite using Nightwatch.js. Nightwatch.js uses a tool called Selenium to run browser processes. Our tests run two browser processes that connect to an example website with four Maygh-loaded static resources. By timing when each browser loads the website, we can make assertions about whether a browser should load resources from the web server, from a peer, or from local storage. By repeatedly connecting and disconnecting browsers, we stress test the system and assure that clients handle errors at any stage while loading a resource.

4. Future Work

Other than using Firebase as storage and using SHA256 to avoid collisions when dealing with larger amounts of content, future work on this project could involve supporting multiple coordinators to increase load balancing and assigning client-to-client connections based on clients' geographic location to decrease latency.

In our design, we did not account for all ways in which clients could behave maliciously. In future iterations of Maygh, we wish to implement more security features such as client blacklisting and traffic covering.

5. Citations

Zhang, Liang, Fangfei Zhou, Alan Mislove, and Ravi Sundaram. "Maygh."*Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13* (2013).