Cooperative Browser Download Streams

CSC2209 Midterm Project Report

Robert Danek

Abstract

The Internet is host to many different types of media, some of which is very large in size and in high demand. Users would like to be able to download this content as fast as possible, and servers would like to serve it as quickly and efficiently as possible. However, as the demand for this type of content increases, it becomes more difficult to satisfy these goals. This project examines a potential solution: Cooperative Browser Download Streams (CBDS). The idea of CBDS is that users share the content that they are currently downloading with other browsers. With CBDS, geographically local browsers will have a closer node from which to download the content, and servers will experience reduced load.

1 Introduction

1.1 What is the problem?

The Internet is host to many different types of media. Some media tends to be fairly light-weight and can be downloaded quickly, such as plain text HTML pages, while other media, such as videos stored on YouTube, are much larger in size and consume more bandwidth on a network. Ideally users would like to be able to download content as fast as possible. Similarly, servers would like to be able to serve that content as quickly and efficiently as possible. Problems arise, however, if too many users are downloading large content from a server at the same time: the server inevitably becomes overloaded and the data starts being served at a slower rate. This could be because of resources on the server itself, or because of network congestion occurring on links near the server. As a result of this, the user also suffers by seeing extended download times. To improve both users' download experiences and servers' ability to provide content, this project explores the concept of Cooperative Browser Download Streams (CBDS).

The idea of CBDS is simply to have users serve the content that they are currently downloading. When a user is downloading large static content to their browser, if the user is willing to share that content, other users can connect to the first user's browser and download the content from him instead of going directly to the origin server. In this manner, the server will experience reduced load. Furthermore, any secondary users who are geographically closer to the first user than they are to the origin server will also likely experience a faster download.

1.2 Why is the problem interesting?

It is becoming more and more difficult for servers that provide content that ranks as the most popular on the Internet to serve that content. Many servers cannot keep up with the load, and need to use content distribution networks. Companies who want to serve this content are forced to invest more and more money to keep up with the popularity of their websites.

Companies would be interested in solutions to this problem if that solution did not require any capital investment on their part. And this is precisely one of the main selling features of CBDS: it does not require companies to buy more servers or pay for participating in a content distribution network. The onus is completely on the user for helping share the content that the server is providing.

1.3 Why is the problem hard?

Incentive to Participate

In order for cooperative download streams to work, a user needs incentive to participate in it. Without incentive, the user would have no reason for donating his bandwidth. One possible incentive is to have a server refuse to provide a client content unless the client is willing to participate in CBDS. This unfortunately would require the server be aware that the client is participating in CBDS. The work done in this project does not examine making changes to origin servers to enforce the use of CBDS.

Another incentive to participate is the potential gains in download speeds that the user can experience.

Transparency

Even if there is incentive, there also needs to be ease of use – participating in CBDS needs to be a transparent experience. If the user has to install several different pieces of software, run processes separate from his browser, or spend time changing his configuration, he may not want to bother. In that case he will go elsewhere for the content he was seeking, or abandon trying to download it all together.

The final implementation of CBDS in this project will be delivered as a Firefox extension. This will make it fairly easy for users to install the software and run it in their environment.

Protection of Privacy

If a user chooses to participate in CBDS, he will be essentially advertising to other users the content that he is downloading. This could be seen as an invasion of privacy; he may want to download the content, which requires him to participate in the scheme, but at the same time he may not want others to know that he has downloaded this content. How can a user's privacy be maintained while at the same time allowing him to participate in the scheme? This is a question that is not addressed in this project.

1.4 Overview of Solution

The goal of this project is to implement CBDS as a Firefox extension that allows a user to share static content as it is being downloaded. This requires intercepting HTTP requests and responses, and the ability to serve content. After the initial investigation of the Firefox extension mechanism, I decided to use a hybrid approach in my implementation. Rather than intercept requests and responses, and serve content, within a Firefox extension, I developed a proxy using Java, and then bootstrapped the Java code from within an extension. The extension also automatically changes the Firefox HTTP proxy configuration to point to the appropriate port. From an end-user standpoint, this solution is virtually indistinguishable from an implementation done purely in Firefox. It has the added advantage that the proxy can run in a separate process for debugging purposes. Also, this makes the solution portable to other browsers, albeit with slightly less transparency – users of other browsers will be required to manually run the Java process and modify their browser's configuration to point to the correct proxy port.

The rest of this paper is organized as follows. In section 2 I describe the related work. Following that, in section 3, I describe the accomplishments to date. Section 4 provides a detailed description of the implementation's design. Section 5 discusses of some of the limitations of my implementation and interesting problems I have encountered. Finally, section 6 lists the remaining items that need to be done along with a schedule as to when they will be complete.

2 Related Work

Extensive work has been done previously on cooperative web proxy caching, which involves caching proxies sharing the contents of their caches amongst themselves. However, Wolman et al in [1] demonstrated that cooperative proxy caching has only a minor benefit within limited population bounds.

In [2] Iyer et al examined Squirrel, a decentralized peer-to-peer web cache. This work is very similar to the work done in this project. Users running Squirrel basically agree to serve the contents of their cache to anyone else using Squirrel. In order to find where certain content should be downloaded from – i.e., a peer browser's cache, or the origin server – Squirrel makes use of Pastry, a peer-to-peer object location service. A problem that Squirrel does not address is that of incentive: what incentive does the user have to open his cache to anyone and allow people to download from him at any time? With CBDS, the idea is to only share content only for the time during which the user is downloading that content. This goes part way in addressing the question of incentive, since potentially improved download times provide at least some incentive. Furthermore, if servers were to implement policies that somehow enforced a client's use of CBDS, it is clear that enforcement would only be effective for the duration when the user is downloading content from that server. This is because as soon as a download is finished, there is nothing that prevents the user from turning off his internet connection.

There is also similar work done in CoopNet [3]. CoopNet examines the benefits that using peer-to-peer communication can have on improving the performance of client - server applications in a network. In particular, [4] examines the problem of alleviating flash crowds. A flash crowd on a server occurs when there is a dramatic surge in requests being made to the server. The proposed solution is for browsers to share content only for the duration of the flash crowd, and perhaps even for shorter durations – i.e., for the duration of time during which the browser is trying to retrieve the content. CBDS is very similar in this respect. However, the solution examined in [4] involves modifications to the origin server to allow for server redirects to occur during flash crowd scenarios; redirects are made to other browsers that have already started downloading content from the server. CBDS, on the other hand, makes use of OpenDHT to keep track of what content is currently being shared, and look ups must be explicitly directed to an OpenDHT node prior to going to the origin server or peer browser.

3 Items Accomplished So Far

The source code and binaries for all items listed in this section can be found at http://www.cs.toronto.edu/~rdanek/CSC2209.html. Instructions on using the binaries are also provided on the website. The design details of the following items are provided in the next section.

- Implemented a way for bootstrapping Java code from Javascript run within a Firefox extension.
- Developed a means for communicating with OpenDHT using Java and Apache XML-RPC.
- Developed a simple Java proxy that forwards HTTP requests from the browser to the origin server, and responses from the origin server back to the browser.
- Developed a means for sharing response streams of static content with other browsers.
- Developed a means for requesting resources from peer browsers instead of going to an origin server.
- Developed a means for requesting partial content (using Range requests) from origin servers when the peer browser stops sharing content in the middle of some other browser downloading it.
- Tested a simple scenario in which two browsers are running on 1 machine, each talking to separate CBDS instances. One browser initiates the download of a YouTube video, and then the second browser makes the request a few seconds later. The second browser successfully downloaded the stream from the first browser instead of going to the origin server.

4 Design

This section explains the design of the CBDS implementation.

4.1 Overview

The CBDS implementation consists of two main parts. The first is the Firefox extension, and the second is the Java proxy. The whole purpose of the Firefox extension is to: (1) start the Java proxy in-process with Firefox; and (2) modify the Firefox HTTP proxy configuration parameters to point to the localhost and port on which the proxy is listening. This allows for a clean user experience that does not require manually configuring the browser or starting external processes.

One purpose of the proxy is to intercept responses to HTTP requests, and if the response is for static content, make that response stream available for sharing with other browsers. OpenDHT, which is a service that provides a distributed hash table, is used to facilitate this. If the stream can be shared, a mapping is added to OpenDHT. This mapping consists of a key and a value, where the key is a concatenation of the origin server's host name and the name of the resource being returned (that is, the absolute path from the associated request header); and the value is the IP and port of the current browser that other browser's should connect to when requesting the resource.

The other purpose of the proxy is to intercept HTTP requests and decide if the request should be made directly to the origin server or if it should be made to another browser. This determination is made by seeing how many mappings exist in OpenDHT for the requested content. These mappings would have been placed there by other browsers who intercepted the response stream from a server and determined that the content is shareable, as described above. If the number of mappings is high enough, then the current load on the server that hosts the content is fairly heavy, and the browser will attempt to retrieve the data from a peer browser. Otherwise, the browser will request the resource directly from the origin server.

4.2 Initialization of CBDS using the Java Firefox Extension

LiveConnect is an API that allows users to call Java methods from within Javascript. The problem is that Java code that is executed as a result of a call from a Firefox extension (written in Javascript) runs with strict security permissions that prevent it from doing anything interesting, such as starting threads or opening server sockets. I made use of the Java Firefox Extension [5] that uses some tricks to set the security permissions to allow the Java code to do anything that it would be able to do if it was running as a standalone process. This extension was used for "bootstrapping" the Java proxy in my implementation.

On start up of Firefox, the extension is loaded and a call is made to edu.toronto.cs.cs2209.proxy.CBDS.initialize() to start the Java proxy. This method initializes the OpenDHT communication component and launches two threads: (1) a thread that opens a server socket for listening for connections from the local browser; and (2), a thread that opens a server socket for listening for connections from remote browsers.

4.3 OpenDHT Communication Component

The mechanism for communicating with OpenDHT [6] is contained entirely in the edu.toronto.cs.cs2209.opendht package. A user initializes this component by calling the OpenDHT.initialize() method. This will cause the list of OpenDHT servers to be retrieved from http://www.opendht.org/servers.txt. Ten servers from this list will be chosen at random, and then "pinged" to see which one responds the fastest. This is done to pick a node that will be able to provide the quickest service.

The "ping" used to determine which server is the best to use is not a real ping in the sense that ICMP messages are not used. (Java does not support sending ICMP messages.) Rather, a TCP socket connection is opened to the server, and the time is measured between when the connect is initiated and the connect completes. This is treated as the "ping" time. Details are in the edu.cs.toronto.cs.cs2209.util.FastestPing class.

The OpenDHT class provides three methods for communicating with the OpenDHT service. These are get(String key), put(String key, String value), and remove(String key). The get(String key) method is synchronous and will return only once the OpenDHT node has returned any mappings for the given key. The put() and remove() methods, however, are asynchronous; when calls are made to these methods, the request is posted to a separate thread and control immediately returns to the caller.

Requests are sent to the OpenDHT service using Apache XML-RPC. The details of this are contained in the XmlRpc class.

4.4 Handling Local Browser Connections

When a user makes a request in his browser, it will be intercepted by the Java proxy. In order to handle this connection, a call is made to edu.toronto.cs.cs2209.proxy.BrowserConnection.handle(Socket s). This will launch a new thread, BrowserConnection.BrowserConnectionThread, on which the request is processed.

The processing of requests from the local browser consists of:

1. Extracting the request header
2. Determining the origin server to which the request is destined
3. Determining the absolute path of resource being requested

Once these items are obtained, a look up operation is invoked on OpenDHT. If the lookup returns at least a certain amount of mappings (currently 1, to make testing the implementation simple), then a heavy load is likely being experienced on the origin server and a "server" connection will be made to a peer browser. The peer browser chosen will be one of the ones returned from the query to OpenDHT. If the number of mappings returned by OpenDHT is fewer than the minimum, a server connection is made directly to the origin server.

Server connections, whether they are to a remote browser or to the actual origin server, are represented by the class ServerConnection. After a ServerConnection object is created, the thread handling the local browser connection calls submitRequest(). This results in two things happening:

1. A new ResponseStream object will be created.
2. A new thread will be started on which the request will be submitted and the response received.

The ResponseStream object will be returned to the local browser connection thread that called submitRequest(). It is from this object that the browser thread extracts the response header and the response stream.

The new thread that is created will submit the request and parse the response from the server. Data coming back from the server will be fed into the ResponseStream object that was previously created for this request.

Also, if the response is deemed to be shareable, the ResponseStream will be stored in the SharedStreams singleton. Once there, it is available for look up by remote browser connections (see section 4.5 below).

What is contained in the ResponseStream depends on whether or not the response is shareable. If it is not, then the response stream is an identical copy of the data returned from the origin server. However, if it is shareable, then the data stream is transformed into a chunked transfer encoding as it arrives from the server (if it is not already using chunked transfer encoding).

4.5 Handling Remote Browser Connections

When a connection is made to the server socket listening for remote browser connections, the RemoteBrowserConnection.handle(Socket s) method is called. This launches a new thread for handling the remote browser connection.

What happens on this new thread is similar to what is done for a local browser connection. First the request header is extracted. Then, it is determined if the current browser is sharing the requested resource. This is done by attempting to retrieve a ResponseStream object from the SharedStreams singleton. If it is not there, the connection to the remote browser is immediately closed. Otherwise, the response stream is processed in a similar manner to how is done by local browser connections. Its contents is piped back to the remote browser chunk by chunk until the stream is closed or the end of the stream is reached. If the stream is closed, because the local browser finished downloading the resource, then the connection to the remote browser is terminated. This will mean the remote browser will have to send a range request to the origin server to retrieve the remaining data.

4.6 Shared Streams

Streams are shared via instances of the ResponseStream class. These objects are stored in a map in the SharedStreams singleton, where the key consists of the origin server name concatenated with the absolute path of the resource URI.

Data in a ResponseStream instance is stored in “chunks”. These chunks are currently byte arrays (byte[]) stored in a linked list in memory. Browser connections (remote or local) can read through this data sequentially. A future implementation of the ResponseStream class may want to write the chunks of data to disk and retrieve them only when necessary; this will reduce the memory requirements when sharing extremely large streams.

5 Limitations / Problems Encountered

- OpenDHT slows browsing down considerably.

  Each request that is made by a browser requires that a lookup be done in OpenDHT to see if any other browsers are sharing the response stream for that request. Unfortunately, the get operation can be very slow, despite the code that is used at start-up time to try and pick the fastest OpenDHT node available. The behaviour is inconsistent and depends on the actual node chosen. To confirm that the problem is OpenDHT (and not some poorly performing section of my own code), I commented out the calls OpenDHT.get(), and found that there was a noticeable speed-up in browsing.

- Sharing data from HTTP 1.0 servers is not supported

  To simplify the sharing code, all data that is to be shared is transferred using a chunked transfer encoding, even if the original stream from the origin server did not use that encoding. The way this is achieved is by taking the response header and adding the line “Transfer-Encoding: chunked” to it, and then formatting the message body accordingly. This technique cannot be used for responses from HTTP 1.0 servers since chunked transfer encoding was introduced in HTTP 1.1.

- Slashdot does not conform to the HTTP 1.1 RFC [7]

  Writing a proxy that conforms to the HTTP 1.1 RFC (RFC 2616) [7] is tricky. Some servers do not implement the RFC correctly and the proxy needs to be tolerant of this fact. For example, the Slashdot server responds that it supports HTTP 1.1, but when you make a request for a resource using an absolute URI (for example, GET http://www.slashdot.org/ HTTP/1.1), the server will respond with a 503 Service Unavailable Error. In order to get a proper response from slashdot, the absolute path has to be used instead (i.e., GET / HTTP/1.1). This is despite RFC 2616 explicitly stating that HTTP 1.1 servers must be able to handle requests made using the absolute URI. The solution in my implementation was to convert all absolute URI's to absolute paths in the request header before forwarding the request to the origin server.

6 Work Left To Do

- Handle NATs (Nov 18) – Need to discuss with Stefan

  Browsers may be hidden behind a firewall. The IP address that they think they have may be different from the IP address that others are aware of because of network address translation that occurs at the firewall. When a browser adds an entry to OpenDHT to let the world know that it is sharing some resource, it will be supplying the incorrect IP address.

  It is possible that this is a non-issue. If the user is behind a firewall, chances are the port on which the browser is listening for connections is blocked. This means that even if the issue of NATs could be handled, there would still have to be some way to poke a hole in the firewall so the browser could be reached.

- Answer the question: What are the potential bandwidth savings (from a server's perspective) when this tool is deployed? (Nov 25)

  Using Kazaa traces provided by Stefan, I will measure the potential bandwidth savings making the following assumptions:

  - Everyone downloads at 50kb/s
  - Whenever someone is downloading a file, it is available to be shared

- Answer the question: What is the impact of the CBDS tool on regular browsing? (Dec 4)

  I will write a Firefox extension to measure the latency introduced by the proxy. It will log the time at which a request is initially made, and then the time when the request is complete. This will be repeated for a number of different requests, and run in two different scenarios: one in which the CBDS tool is present and one in which it is not. The results will be summarized for inclusion in the final report.

  This experiment will not involve running multiple instances of the CBDS tool. It is only trying to answer the question of what impact the tool has on the user's browsing experience when the benefits of sharing download streams is not present.

- Code clean-up and improve comments (ongoing, Dec 20)
- Move report into a latex format for presentation in "conference proceedings" style. (Dec 20)

7 References

[1] Wolman, A., Voelker, G.M., Sharma, N., Cardwell, N., Karlin, A., Levy, H.M.: On the scale and performance of cooperative Web proxy caching. Proceedings of the 17th ACM Symposium on Operating Systems Principles (1999) 16-31

[2] Iyer, S., Rowstron, A., Druschel, P.: Squirrel: A decentralized peer-to-peer web cache. Principles of Distributed Computing (2002)

[3] CoopNet: Cooperative Networking. http://research.microsoft.com/projects/coopnet/

[4] Padmanabhan, V. N., Sripanidkulchai, K.: The Case for Cooperative Networking. Proceedings of the First International Workshop on Peer-to-Peer Systems (March 2002)

[5] Java Firefox Extension. http://simile.mit.edu/java-firefox-extension/

[6] OpenDHT. http://opendht.org/

[7] RFC 2616. Hypertext Transfer Protocol – HTTP/1.1. http://tools.ietf.org/html/rfc2616