Cooperative Browser Download Streams (CSC2209 Fall '06 Project)

Robert Danek
University of Toronto
rdanek@cs.toronto.edu

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 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 content that they are currently downloading with other browsers. This paper shows through a number of simulations that for long-lived download streams, which can occur at low transfer rates or for extremely large files, this tool has the potential to moderately reduce the load on origin servers. Furthermore, it discusses the design of a proof-of-concept implementation of the tool that was developed.

1. Introduction

1.1 What is the problem?

The Internet is host to many different types of media. Some media tends to be fairly lightweight 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?

There are three main reasons that make the problem hard: incentive to participate, transparency of the solution, and privacy of user data.

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.

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 last issue to be discussed is that 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, and is left for future work.

1.4 Overview of results

As part of this project, several simulations were run in order to determine the practicality of the CBDS solution. The simulation aimed to answer two specific questions: (1) What fraction of the total bytes downloaded can be served by peer nodes instead of the origin server? (2) What is the number of concurrent uploads that a client has to potentially serve? The first question's goal is to determine what the possible bandwidth savings are from the server's perspective. Even if the savings are large, it may be the case that the number of peers that a client has to serve is exorbitantly large, negating the practicality of the solution; this is what the second question seeks to determine.

The data obtained from the simulations show that at low transfer rates, a moderate fraction of the total bytes transferred can be shared. However, the data also show that this sharing comes at the cost of having some clients being required to serve many concurrent uploads.

The other item that this project delivered was a proof-of-concept implementation of the CBDS solution. The implementation is provided as a Firefox extension. This requires that the extension be able to intercept HTTP requests and responses, and have the ability to serve content. After the initial investigation of the Firefox extension mechanism, it was decided to use a hybrid approach in the implementation. Rather than intercept requests and responses, and serve content, within a Firefox extension, a proxy was developed using Java. The proxy is bootstrapped from within the extension whenever the extension is loaded. 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 the main results of this project are described. This section is split up into several subsections, including: a description of how the simulator works, a summary of the data obtained from the simulator, and a description of the design of the CBDS Firefox extension. Following that, in section 3, the related work that has been done is described. Section 4 describes future work to be done, and the paper concludes in Section 5.

2 Results

2.1 How the Simulator Works

The simulations were done using a trace containing a list of 1640179 Kazaa requests made by over 24K clients during a period of 200 days. Prior to running the simulation on the trace, the trace was pruned so that it only contained requests for files

larger than 100M bytes. The rationale for this is that the CBDS solution is intended for sharing large files.

The Simulator works as follows. It maintains a priority queue containing events of type START_DOWNLOAD or END_DOWNLOAD that need to be processed. The priority queue is ordered based on the time of the event. In order to populate events into the queue, the simulation has to start by reading entries from the trace file. The main loop of the simulation basically reads an entry from the trace file, notes the time associated with that entry, and processes all events in the event queue up until the time of the current entry read from the file. It then adds the entry read from the file into the event queue, reads the next entry from the file, and continues processing events in the queue until the time associated with the latest entry read.

Besides the event queue, the Simulator also maintains a mapping of the clients actively downloading a given file.

Due to space limitations, all the details of the simulator's operation cannot be included here. A more complete description of the Simulator, along with its source code, are available at the CSC 2209 Project Web site [8].

2.2 Simulation Results

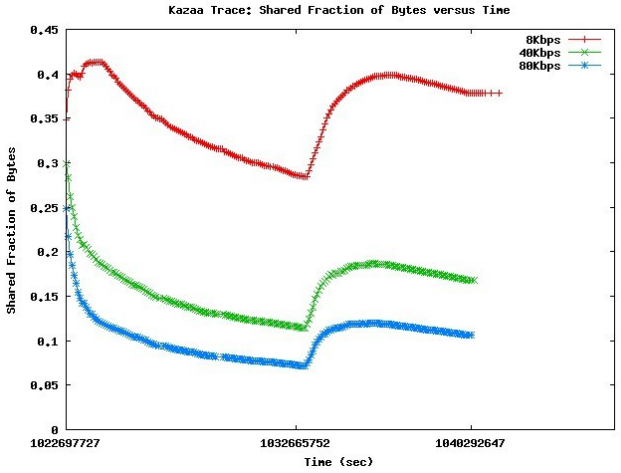

Figure 1

The first question that the simulation attempted to answer was: What fraction of the total bytes downloaded can be shared between peers instead of being downloaded from the origin server?

The results are illustrated in Figure 1. The figure shows the fraction of bytes that can be shared as time progresses in the simulation. For a very low transfer rate between nodes of 8Kbps, the final fraction of bytes that can be served by peers is 37.8%. As the transfer rate is increased, this value naturally decreases. This is because the size of the files in the trace we use remain constant, and hence the time it takes to download a file decreases. In turn, this means the amount of time a client can share its download stream is decreased, and so the fraction of bytes that can be uploaded to peers also decreases. For a transfer rate of 40Kbps, the fraction of bytes that can be served by peers is 16.8%; and for 80Kbps the fraction is 10.6%.

Data was collected for faster transfer rates, but is not illustrated in the graph, since the fraction of sharing dips below 4%. For a transfer rate of 400Kbps the fraction of bytes that can be served by peers is 3.5%; for a transfer rate of 800Kbps, it is 2.0%; and for a transfer rate of 4Mbps, the fraction is 0.55%.

One thing to note in Figure 1 is the kink in the graph that occurs midway. This kink is due to the source of the trace data used in the simulation. The data was collected over a 200 day period from a university campus, and part of it spans the summer. Since most students are not on campus during the summer, the amount of data being transferred naturally dips. When the Fall term starts again, there is an expected spike, as seen in the graph.

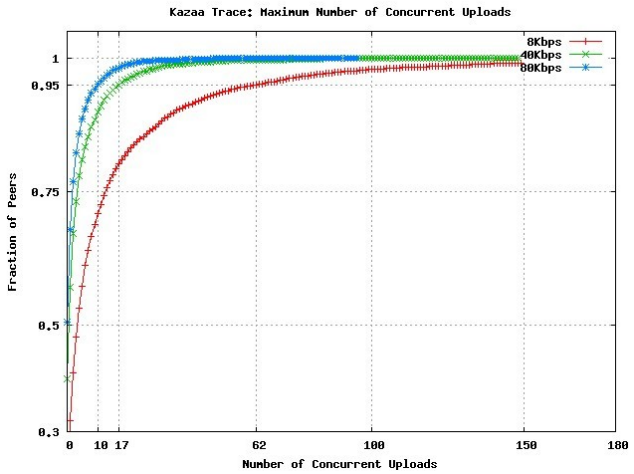

Figure 2

The other question that the simulation attempted to answer was: What is the number of concurrent uploads that a client has to potentially serve?

The results to this question are shown in

Figure 2. The figure illustrates the cumulative fraction of peers having to concurrently upload to at most *x* other peers. In particular, at a transfer rate of 8Kbps, 95% of the clients have to serve at most 62 concurrent uploads. However, there is some client that has to serve 532 concurrent uploads. (This last data point is not illustrated on the graph since it would not have been informative to see the progression from 99th percentile of clients to the 100th percentile occurring between *x=150* and *x=532*.) At a transfer rate of 40Kbps, 95% of the clients have to serve at most 17 concurrent uploads, while some client exists that has to serve 148 concurrent uploads. Finally, at a transfer rate of 80Kbps, 95% of the clients have to serve at most 10 concurrent uploads, yet there is at least 1 client that has to serve 95 concurrent uploads.

Data was collected for faster transfer rates, but is not discussed here since the fraction of sharing possible at those transfer rates is negligible.

The results from the first question indicate that the CBDS solution has some promise. However, the results obtained for the second question are discouraging. If a client has to be encumbered by serving too many concurrent uploads, the CBDS solution becomes impractical. There is hope for this problem, however, as discussed in the section on future work.

2.3 Design of the CBDS Implementation

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 local host 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.

The implementation of the Firefox extension in this project is based on the Java Firefox Extension [5]. Without the functionality this extension provides, the code would have to use LiveConnect. 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. The Java Firefox Extension sets security permissions in such a way to allow the Java code to do anything that it would be able to do if it was running as a standalone process.

The Java proxy that was implemented has two main purposes. One purpose 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 [6], 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.

An extended description of the design of the CBDS implementation is available in the Project Mid-term Report available at the CSC 2209 Project Web site [8]. The CBDS Firefox implementation is also available here.

3 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.

4 Future Work

One of the items not studied in this project was how geographic locality of peers could be exploited to increase download speeds. If two peers are physically close to each other, and one is downloading a file that the other wants, then the second client should be able to download the file faster than would be possible by going to the origin server (assuming the origin server is not local).

Another issue worth exploring is that of maintaining the privacy of CBDS participants. When a user shares his download streams with others, he is advertising his surfing habits to the world. Is there a way to share the download stream of a browser while at the same time hide from the peer downloading that data who the client is sharing that stream?

Providing stronger incentive for users to participate in CBDS is another problem that deserves to be studied. One proposal is to enlist the help of origin servers to provide that incentive – for example, by preventing users from downloading certain data unless they are using the CBDS tool.

The implementation discussed in this paper has a number of limitations, including its inability to handle firewalls and Network Address Translators (NATs), and its lack of support for HTTP 1.0. When a client is behind a NAT or a firewall, the IP address that it registers in OpenDHT when it's willing to share a download stream may be incorrect. Furthermore, the port on which the client is listening for peer connections may be blocked by the firewall. HTTP 1.0 is “tolerated” by the CBDS implementation, but static content from HTTP 1.0 servers cannot be shared. This is due to the way the implementation uses chunked transfer encoding for transmitting data between peers. (Chunked transfer encoding was introduced in HTTP 1.1 [7].)

One further thing worth studying is a more intelligent means for choosing peers from which to download a file. The simulations done show that randomly choosing a peer when there is more than one to choose from can cause a heavy load to be experienced by a number of 'unlucky' peers. One way to improve on this would be by ordering the peers in a chain based on the time they started downloading the file, and whenever a peer starts downloading, it starts downloading as much of the file as it can from the peer that precedes it in the chain. In this manner, a client should only ever have to serve at most one other peer.

5 Conclusions

This project was able to show through the execution of a number of simulations that Cooperative Browser Download Streams has the potential of saving a moderate amount of bandwidth on an origin server when extremely large files are being downloaded and the transfer rates between nodes is reasonably slow.

In particular, at a transfer rate of 8Kbps, 37.8% of the bytes transferred can be obtained from peer clients instead of the origin server. This value decreases as the transfer speed between nodes increases. At an extremely fast speed of 4Mbps, the shared fraction of bytes is only 0.55%.

The downside of these results was that the number of concurrent uploads that a client has to potentially serve can be extremely large. At a transfer rate of 8Kbps, 95% of the clients have to serve at most 62 concurrent uploads, and there is at least one client that has to serve 148 concurrent uploads. Even at moderately fast speeds of 400Kbps, 95% of the clients have to serve at most 3 concurrent uploads, but there is at least one client that has to serve 25 concurrent uploads.

The other outcome of this project was a rudimentary proof-of-concept implementation of CBDS. This implementation was a Firefox extension wrapping an HTTP proxy written in Java. Being able to deploy this tool as a Firefox extension goes a long way to solving the problem of transparency. Firefox users will only have to install the extension, which is done using an extremely simple mechanism within Firefox, and they will be up and running with CBDS.

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

[8] Robert Danek's CSC 2209 Project Page. http://www.cs.toronto.edu/~rdanek/CSC2209.html