WebTorrent: a BitTorrent Extension for High Availability Servers

Gary Sivek gsivek@mit.edu
Steven Sivek ssivek@mit.edu
Jonathan Wolfe jwolfe@mit.edu
Michael Zhivich mzhivich@mit.edu

May 6, 2004

Abstract

Achieving content high-availability is one of the most important goals of a webserver system. In order to achieve high-availability in the traditional client-server setting, the server must have the bandwidth and the hardware needed to handle any peak load that might occur. However, this is a very costly and rarely practical solution, especially for most non-commercial servers subjected to the Slashdotting effect. We propose a WebTorrent system based on BitTorrent that will leverage the resources of the clients to help the server make the content more available. Such a system will help alleviate the load on the server and reduce client download times.

1 Vision

1.1 WebTorrent Overview

In this paper we describe WebTorrent, our system to improve delivery of web sites in high demand. It leverages the BitTorrent infrastructure in order to provide faster download speeds for clients requesting very popular web sites and to alleviate the heavy load on these web servers. The system consists of plugins for both the client and the server – in our implementation we use Mozilla as the client browser of choice and Apache as the server of choice.

The success of the system relies on the willingness of clients to maintain small caches of bundled web sites for other clients to retrieve using BitTorrent. However, the overhead associated with the added infrastructure should not be too significant to reduce server performance during periods of low load.

1.2 Design Choice

We considered all three approaches to improving the flow of web traffic: client-side software only, server-side software only, and coordinated client-side and server-side software. The first two approaches have the advantage of being easy to deploy, since they require work either on the part of a single server administrator or a handful of web surfers, but not both. However, server-side-only approaches tend to involve increasing hardware, bandwidth, or otherwise spending money, and none of these options are desirable or immediately available to the average home broadband server. Client-only approaches tend to involve distributed caching, which has legal implications, makes it harder for a server administrator to change the content reliably, and requires extra work for the client to determine where the cached content exists. We feel that implementing software on both ends is the most elegant solution, allowing easy administration on the server end and no more than a one-time install on the part of the willing clients. Moreover, as our tests demonstrate, performance benefits are seen even if a small fraction of clients use WebTorrent-enabled browsers; thus, it is to the advantage of both the server administrator and the clients to use our system.

1.3 Key Challenges for WebTorrent

The BitTorrent infrastructure is currently designed to handle large files – a single transfer chunk is 256KB. The majority of web site requests involve HTML files and images that are much smaller than this minimum BitTorrent chunk. Because of the new proposed use of BitTorrent, the WebTorrent system must appropriately adapt the BitTorrent protocol to be useful for small requests while maintaining performance acceptable for browsing the web. We propose a scheme for bundling appropriate HTML files and images together in order to reduce the combined overhead of invoking BitTorrent for many small files. In addition, we configure BitTorrent to use smaller chunks of 16KB when transferring webpage bundles. While doing so adds to the total overhead, this strategy ensures a reasonable number of chunks in a bundle, such that BitTorrent benefits are not lost.

Backwards compatibility must be maintained such that clients without the WebTorrent plugin can still be served by a WebTorrent-enabled server. In order to achieve this goal, WebTorrent-enabled servers simply continue to serve content in traditional fashion for such clients. In fact, WebTorrent strives to reduce the load even further by making use of clients who volunteer to also act as backup servers. Clients that do not support WebTorrent can then be redirected to those alternative servers (within reason) to shed more load.

2 Motivation

2.1 The Slashdot Effect

Our motivation for this system is the common "Slashdot effect." This effect occurs when a server that cannot handle large amounts of traffic is bombarded with visitors because it is hosting a website that has been linked from an immensely popular site like Slashdot [11]. Our goal is to create a system that uses BitTorrent to deliver such high demand web sites both to increase performance for clients who sometimes cannot even retrieve (at any speed!) a copy of the desired web site and to reduce load on such Slashdotted servers to keep them from shutting down.

Slashdot, itself, should not solve this problem on its own by caching such web content. Sites that generate revenue from advertisements would prefer that clients load their site directly rather than a cached copy from Slashdot. More importantly, Slashdot would then have to ensure that it does not hold a stale copy of the site in its cache [12].

2.2 BitTorrent and Its Limitations

BitTorrent works by decentralizing the download process for clients. Instead of fetching an entire large and popular file (Linux kernel source, for example) from a single server, BitTorrent clients download just a small torrent file which contains information about the tracker and the pieces of the desired file. The clients then retrieve a list of peers from the tracker, which acts as a coordinator and keeps information about which pieces of the file each peer has. Once the client has a list of peers, it can exchange file pieces with these peers without communicating to the original server. The client also keeps the tracker informed about the pieces that it already downloaded, and retrieves a new list of peers after some time interval. Thus, BitTorrent is spreading out the load among clients, effectively taking the load away from the original webserver and providing clients with a potentially faster way to download that large file.

While BitTorrent succeeds in its goal distributing the server load among the clients, it suffers from several robustness problems. First, all clients download the torrent information file from the original server, called the directory server. While the torrent file is small, it is possible that the server's link will be saturated by many clients trying to obtain this file. This can perhaps be mitigated by mirroring the torrent file on different servers. Second, any particular torrent is assigned to only one tracker; thus, all clients must communicate with the machine that runs the

tracker. While the messages in the client to tracker communication are short, a malicious client could bring down the tracker (or the directory server) and effectively terminate the torrent distribution. These problems are mentioned to show that BitTorrent can be improved, but solutions are beyond the scope of this paper.

3 Design

The design of WebTorrent builds on the existing BitTorrent system to allow webpage distribution via torrent files. There are two parts to the system – a server-side module and a client-side browser plug-in. These components are designed to augment currently popular software – Apache on the server side and Mozilla on the client side. The server-side module makes it possible for Apache to switch from distributing HTML pages to distributing the same content via torrent files if the load rises above a configurable threshold. Likewise, the client plug-in allows the client to transparently view webpages downloaded as torrent files alongside the ones downloaded in a traditional fashion. Both are entirely backwards-compatible; the headers added by a WebTorrent client plugin will not interfere with a server that does not have WebTorrent module, and the WebTorrent-enabled server can tell from the initial request whether the client has the plugin and act accordingly. This section will discuss the details of both the server-side and client-side components and how they will fit into the existing software to enable high availability of webpage content.

3.1 Server-side Module

The server-side implementation consists of a single Apache webserver module called `mod_webtorrent`. Apache must be configured for WebTorrent by associating one or more file types with the WebTorrent module via an entry in the server configuration file. For example, *httpd.conf* might contain the line "`AddType application/x-webtorrent .whtml .whtm`" to register those two extensions; for the sake of this description, assume that `.whtml` is a registered WebTorrent extension. Now, Apache will use `mod_webtorrent` to handle all such files and treat them as static content. Note that WebTorrent cannot handle all files because dynamic content must be produced by the server anew for each request, rather than pieced together from cached copies on clients. The server administrator will also have the option of specifying a load threshold via "`WTLowLoad n`" command in the WebTorrent section of *httpd.conf*. If the number of requests per minute is smaller than `n`, then the server will switch from serving content via WebTorrent to normal operation.

The BitTorrent technology does not become efficient until files reach a reasonably large size, so it would very inefficient to send a small HTML file by itself, especially if other data, such as images, associated with the page require more bandwidth. Thus, `mod_webtorrent` must do some preprocessing to maximize the size of the data sent as a single torrent bundle and minimize the number of connections made. The logical bundling strategy is to group all content on a given page into a single bundle. The first time a client requests a specific `whtml` file, the module will parse the HTML file for the necessary resources (embedded JPEGs, GIFs, JavaScript source files, etc.), group the desired files into a tarball and generate the appropriate torrent file to serve to the client. This tarball serves as the bundle, which the module stores with a list of its contents; every thousand requests or five minutes, whichever happens first, the module checks the `mtimes` of these files, and if any have changed then the bundle expires and must be created anew. Alternatively, the administrator can create the website bundle and the appropriate torrent by running a deployment script when adding the content to the webserver. Then, the module implementation is less complicated, and the administrator has control over the files included in the bundle. Once the bundle and the appropriate torrent file are created, `mod_webtorrent` starts a tracker and a completed BitTorrent downloader on the server. The downloader is going

to perform the first few uploads to transfer the website bundle to several clients; afterward, it can be taken down to save bandwidth on the server. Ideally, the completed downloader and the tracker would run on a separate server; however, our design is aimed at users who are running smaller servers and do not have extra machines for these tasks.

Now, suppose a client requests a `whtml` file. There are two possibilities: either the client is WebTorrent-enabled or it is a standard "vanilla" browser. If the client is WebTorrent-enabled, it will specify "`Accept: application/x-webtorrent`" as part of its HTTP request headers. If the number of open connections does not exceed the `WTLowLoad` parameter, the server acts as a normal webserver and simply serves the requested file as HTML; otherwise, it acts as a BitTorrent directory server and send a torrent file. If the client further indicates "`X-WTBackupServer: n`" and "`X-WTBackupServer-Port: p`" (where n and p are positive integers) as part of its HTTP headers, `mod_webtorrent` will measure the time required to complete the entire HTTP transaction and use the result to estimate the client's connection speed. The IP address and port number (p), connection speed, maximum number of connections (n), and the current timestamp are then stored in a priority queue of backup servers associated with this particular file. The queue is sorted in descending order by connection speed, breaking ties with timestamps in ascending order. A client will expire from the queue when either n becomes 0 or a minute has passed since the timestamp value, which is updated when the server passes the client a new incoming connection (and decrements n).

Alternatively, if the client does not specify "`Accept: application/x-webtorrent`," then the server is required to give regular HTTP responses. Under low load conditions, the server can perform the standard HTTP communication; under high load, though, the server will instead return a very short reply with status code 302 (Found) and a `Location` field corresponding to the top volunteer server in the priority queue. Since, as discussed in the next section, client-side backup servers are entirely voluntary, we are assuming that a large enough fraction of them is reliable. Then, this strategy adds robustness to the system – the Slashdotted server might drop some requests because its link is getting saturated. If the queue is empty, the server should be idle enough to handle requests itself. In either case, a reasonably-sized pool of WebTorrent clients will, through the combination of BitTorrent and HTTP redirecting, sufficiently lower the bandwidth requirements on the server to ensure solid performance even under unusually high loads.

3.2 Client-side Plugin

A WebTorrent client consists of a single plugin written for Mozilla, with no extra software required beyond the LibBT library designed to provide a language-independent implementation of the BitTorrent protocol [6]. The plugin announces its availability by appending the flag "`application/x-webtorrent`" to its `Accept` header in HTTP requests; if a server does not support WebTorrent, the plugin will not perform any further work because the server will return a file type other than `application/x-webtorrent`.

The WebTorrent plugin is registered with the `application/x-webtorrent` type, so it is called whenever the browser sees a file of this type. If the server returns a file of this type, the browser must be prepared to download the associated web page via BitTorrent. The plugin can then download the bundled page using LibBT, unpackage it to a local directory, and then redirect the browser to the resulting local cache of the page. The client keeps a copy of the downloaded page available for sharing by keeping its BitTorrent client open for an hour; after this time has expired, the BitTorrent client may be closed because it is assumed that either enough other clients are open to share the load or the load spike placed on the server has diminished.

The client plugin also contains a tiny web server which the user may choose to enable; if used, it will provide cached copies of the page for redirected clients which do not support

WebTorrent. It informs the server of this ability by adding "`X-WTBackupServer: n`" and "`X-WTBackupServer-Port: p`" flags to its initial HTTP request, announcing the number of connections it is willing to serve and the port on which it is listening. These values are configured by the user, as is the decision of whether to act as a backup server.

The tiny web server simply caches the loaded pages in memory, storing the domain name and request URI in a lookup table that allows it to find the contents of each requested file. Since only static content can be served, only the HTTP `GET` method is supported. This web server tracks the number of requests it has received for cached copies of a site and terminates either when it has served the promised number of pages or when a minute passes with no requests arriving. Thus, the backup server does not use excessive memory by caching too many sites for too long. The server that originally provided the content must also track the number of redirects it has made to this backup, along with the time since the last redirect, in order to avoid sending unaware clients to a backup server that is no longer accepting connections.

4 Implementation

Since both Mozilla and Apache provide C APIs for developer use, our testing implementation is written in C. Unfortunately, we ran into several problems while implementing our system. First, both Apache and Mozilla are in the middle of further developing their APIs, so many of the old documented functions no longer exist, while many of the new existing functions are not yet documented. While the source code is, of course, the only "required" documentation, the resulting implementation process was much more difficult and time-consuming than we anticipated. In addition, a C port of the BitTorrent library [6] seems to have a several issues that have not been entirely fixed yet, so we had to resort to calling the Python implementation of BitTorrent [5] from our test code. Thus, our test implementation is far from perfect, and can be further optimized and extended in functionality when LibBT is ready. In this section we will discuss both our planned implementation and the changes we were forced to make due to lack of a working C BitTorrent client.

4.1 LibBT - a BitTorrent library

To create a C implementation of the Apache and Mozilla modules, we found a C port of BitTorrent called LibBT [6]. This library implements the necessary functions associated with the BitTorrent protocol, thus allowing C applications to utilize the BitTorrent framework. LibBT uses `libcurl` [7] and `e2fsprogs` in providing the necessary functionality. Libcurl is a C library that provides client-side URL transfer functionality. It supports a variety of protocols, including HTTP and HTTPS, and provides both downloading and uploading capabilities. It is thread-safe, IPv6 compatible and builds on a variety of operating systems. The other package used by LibBT, e2fsprogs, is a suite of filesystem utilities for the Linux ext2 and ext3 filesystems. Thus, LibBT seemed like a good choice for a BitTorrent implementation, since both Mozilla and Apache run on Linux, which is a convenient development platform. Ideally, we would like to extend the project to include a BitTorrent implementation that would run on Windows, which would allow Windows Mozilla and Apache users to enjoy the benefits of WebTorrent.

4.2 Apache Module Implementation

The original design called for Apache module to use the server-side part of the API provided by `libbt` to create a directory server, set up a tracker and an "origin" downloader. Due to our troubles with LibBT, we resorted to using the Python implementation of the BitTorrent tracker and downloader, and ran them manually on the server during the testing period.

The Apache module has a very simple structure. The module consists of handler methods that are called by Apache at different stages of execution. At startup, Apache reads the global server configuration files and calls any appropriate handler methods in its modules. Once Apache is serving requests, it calls appropriate matching handlers at each stage of request processing [1]. Our Apache module defines handlers only for reading configuration files, as it needs to extract `WTLowLoad` settings from `httpd.conf`, and for sending the request back to the client, as it needs to have control over whether the client is served an appropriate torrent file or request is delegated to the standard HTML handler.

During startup, Apache reads its configuration files and calls the `set_lowload` handler in `mod_webtorrent` if the "`WTLowLoad`" keyword is encountered. The handler then sets the `lowload` variable accordingly. Once the configuration files are read, Apache will execute the post-configuration handler – `wt_register_hooks`, in the case of our module. This method creates the circular linked list of structures that allow our module to track the server load. Each time the request handler is called, a timestamp is recorded in the current linked list entry, and the current index is advanced. Since the circular linked list wraps around every `lowload` requests, the module can calculate how long the server took to handle the last `lowload` requests.

During request processing, the module's main handler, `wt_handler`, is called. After checking that the filetype is appropriate, the module uses the `lltbl` circular linked list to determine whether the server is under low load and sets `wtstatus` flag accordingly. At this point, we have committed to handling the request, so it is logged in `lltbl`. The module then checks the "`Accept`" field of the client request and decides whether to pass the request on to the standard HTML handler by setting the appropriate `content-type` and `handler` fields in the request object or to handle the request by sending the torrent. In the latter case, the appropriate torrent file is located and transmitted to the client; if the file is not found, the appropriate HTTP error code is returned. The actual transfer of the tarball that contains the webpage bundle is left to the BitTorrent downloader that is running on the server machine. It coordinates with the tracker and the client without any intervention from the server module.

4.3 Mozilla Plugin Implementation

The Mozilla plugin uses the Netscape Plugin API, which requires that the plugin provides a set of functions to be called by the browser. One method, `NPP_GetMIMEDescription`, provides the browser with the string `application/x-webtorrent:.whtml:WebTorrent page`; this is used to register the plugin with the browser so that it may be invoked when the browser is about to receive a file of the `application/x-webtorrent` MIME type. This seems to work on many, but not all, versions of Mozilla, as explained in Mozilla bug #241780 (discovered by one of the authors while writing the plugin); if the browser does not automatically register the plugin, visiting the address `about:plugins` will force it to call this method and thus discover the plugin.

Since browsers that do not have the plugin are unable to handle WebTorrent downloads, the plugin must make itself visible by inserting `application/x-webtorrent` into the list of types in the `Accept:` field of the browser's HTTP headers, as mentioned above. This is accomplished by calling a function commonly used by the browser to read the browser variable `network.http.accept.default`, which contains the contents of the `Accept:` header field. If the value does not already include `application/x-webtorrent`, the string is prepended to it.

Once a WebTorrent download begins and the plugin is initialized, the browser informs the plugin of an input stream waiting to be processed. The plugin requests the stream – that is, the torrent of the requested web page – to be saved to a file in the browser's local cache. Once the torrent has been saved, the plugin spawns a BitTorrent downloader, which fetches the bundle and saves it into `/tmp`. The main thread occasionally checks the downloaded bundle to determine when the download is complete; once it has finished, it extracts the individual files to create a local copy of the requested page but leaves BitTorrent running to provide the bundle

to interested clients elsewhere.

The plugin runs as a full-window plugin, meaning it is expected to occupy and draw itself in the entire browser window, rather than be embedded in it like an applet. In order to leave the rendering of the desired web page up to the browser, the plugin simply redirects its window to the local copy of the page in `/tmp` once the archive has been expanded. Currently, the BitTorrent downloader is left running indefinitely in a separate thread; once LibBT code becomes a stable component of the WebTorrent system, it would be possible to gracefully shutdown the BitTorrent downloader after some fixed time.

5 Testing and Performance

The ideal way to test WebTorrent is to subject a server with `mod_webtorrent` to the Slashdotting effect by having many clients simultaneously attempt to retrieve a website from the server. In this situation, we would be able to measure and compare server performance under loads from both normal and WebTorrent-enabled clients. However, such a test requires control over multiple machines so that the server's network link is saturated, and thus is rather impractical for lack of such resources. The alternative solution is to simulate link saturation on the server by rate-limiting the standard HTTP port (80) together with BitTorrent ports (6981-6999). Then, the Slashdotting effect can be simulated using just a few of our home machines as clients.

In our test we use a FreeBSD server running Apache 2.0.49 with `mod_webtorrent`. We have used the default Apache configuration, with the following modifications to *httpd.conf*:

LoadModule	webtorrent_module modules/mod_webtorrent.so
AddType	application/x-webtorrent .whtml
WTLowLoad	0

Figure 1: Apache2 configuration changes

Since both the server and our client machines are all running on the same high-speed local network, we use FreeBSD's included `ipfw` [8] to restrict server bandwidth to 256 Kbps, which approximates a residential broadband connection and allows us to use fewer machines during the test, while still simulating link saturation. We also restrict per-torrent bandwidth to 128 Kbps using the `--max_upload_rate` option of `btdownloadheadless.py` to prevent BitTorrent from completely blocking Apache connections.

We used Mozilla to save some typical webpages for our tests. The webpages we chose are Slashdot, Google News, and CNN.com. These websites are visited quite often and present what we feel is a typical size and structure of a site that is likely to be Slashdotted. We then created the appropriate webpage bundles by making a tarball of the .whtml page together with a folder containing the necessary images, style sheets, etc. The appropriate torrent files were then created for the tarball using `btmakemetafile.py`, which comes with the Python distribution of BitTorrent. When making the torrent file, we set the chunk size to 16KB, which allowed for both a small torrent file and a reasonable number of pieces to take advantages of multiple clients in BitTorrent. Both the resulting website bundle and its corresponding source files were then placed on our test server. The clients could then retrieve either version by getting `slashdot.whtml`, `google.whtml` or `cnn.whtml`, and the server would reply with the appropriate content, depending on whether the client is WebTorrent-enabled. Thus, we could compare the time it takes to download whtml and appropriate images directly from the server to the time it takes to download the corresponding tarball using BitTorrent.

We decided against running Mozilla on the client side due to the resource drain multiple copies of Mozilla would impose on our test machines. Also, we found no easy way to control all the instances of Mozilla from a central location. Instead, we ran a much smaller `client`

driver program that essentially simulates Mozilla, either with or without the plugin, from the command-line. A `runcli` script then controlled how many downloads to start with `client` and the probability of such a download being WebTorrent-enabled.

For the duration of the test, we launched 80 simultaneous downloads of a single webpage (such as `slashdot.whtml`) using `runcli`. Our driver program was configured to run multiple downloads at once and acted both as a WebTorrent-enabled and normal client. Thus, we were able to run several different tests, varying the fraction of the clients that are WebTorrent-enabled. We ran the tests from four machines on four different networks at MIT: one in the Stata Center and one in each of the Alpha Epsilon Pi, East Campus, and Random Hall living groups. Each machine ran 20 downloads at a time.

The test, as performed, has a few minor advantages and disadvantages. First, it puts quite a heavy load on the machines running many copies of python, which may slow them down to some degree. However, this means that any improvements that we found are actually likely to be somewhat underestimated, so this is acceptable. Second, a machine running multiple copies of python can trade data between internal copies of BitTorrent much faster than it could over a network with another machine. However, this is also acceptable, since not only does it diminish the effect of the first problem, but it also makes our test less dependent on remote network speeds. Our goal was, after all, to increase server availability, and this therefore gives a more accurate representation of the time needed for the server itself to finish its work. Third, the runtimes found have large margins of error, as each network's speed depended on its usage by other students at the time of the tests, but this is true of any network, and our experimental data still exhibit the expected trend. The results of our tests are summarized in Figure 2, where each number is the time required for a particular host to finish all 20 of its requests to our test server.

Website	Tarball (bytes)	HTML (bytes)	WebTorrent Probability	Stata (sec)	AEΠ (sec)	EC (sec)	Random (sec)
Slashdot	92160	70445	0.0	128.5	130.5	130.7	130.1
			0.3	82.0	119.7	117.9	123.6
			0.7	63.4	79.4	66.5	94.3
			1.0	13.7	25.5	34.9	27.6
Google News	163840	133552	0.0	240.9	240.3	189.2	223.3
			0.3	195.0	181.0	166.5	189.6
			0.7	96.1	115.1	117.7	144.4
			1.0	22.6	24.0	46.2	34.8
CNN.com	296960	188765	0.0	203.8	238.9	189.2	214.4
			0.3	199.9	191.1	189.2	197.9
			0.7	83.0	139.3	119.6	143.5
			1.0	14.1	17.9	56.5	35.8

Figure 2: Times per set of 20 requests for normal and WebTorrent-enabled clients

The data we acquired demonstrates that the webserver spends much less time processing the WebTorrent-enabled requests than it does for normal HTML requests. We recognize that these measurements do not reflect the entire time the server takes to process a WebTorrent request – this is just the time to transfer the .torrent file to the client. More time and bandwidth is actually involved, since all the clients will also be talking to the tracker that is running on the server. However, the tarball is rather small, so the communication between the tracker and the clients is minimal. Also, getting the tarball out to the clients at first is not taken into account in this analysis. We believe that since the server will be serving the tarball via BitTorrent only for the first few minutes – enough time to get it out to several clients – the bandwidth and time consumed will be small, since it is being amortized over many clients. After the tarball has been

transferred to several clients, each additional client will put very little load on the server, since it will be getting all of its tarball pieces from peers.

Our data also shows that our system successfully diminishes the load on the server even if just a few clients use WebTorrent. The system performance increases with the number of WebTorrent-enabled clients; however the data shows that wide adoption of WebTorrent is not essential to improving the server performance and client download times. Thus, server administrators have an incentive to include `mod_webtorrent` into their Apache installation and clients have an incentive to use the Mozilla WebTorrent plug-in.

6 Future Work

Some features of our design were left unimplemented, partly due to LibBT not being ready and partly due to HTTP protocol being not as extensible as we hoped. Some of the things that we would like to see implemented are:

- A solid C implementation of the BitTorrent downloader and tracker. This would allow for a better interface with `mod_webtorrent`, resulting in higher degree of control, such as shutting down the client once the tarball has been transferred to several clients to lower the load on the server even more.
- A limited-functionality backup server running as part of the Mozilla plugin.
- Implementation of `X-WTBackupServer` and `X-WTBackupPort` flags. Our best working idea is to insert them into the "`User-Agent`" string.
- A filesystem-independent implementation of BitTorrent downloader that would allow users running Mozilla and Apache on Windows to enjoy the benefits of WebTorrent.
- A simple file archival utility to replace tar, which as seen in the CNN test in Figure 2 can create archives much larger than necessary and waste bandwidth.
- A deployment script that an administrator can run to package the website and its appropriate content into a bundle. This is an alternative to our proposed design that requires the Apache module to contain an HTML parser. Such a script would be easier to implement, and will reduce the amount of work the module has to perform.
- A configuration tool that allows the administrator to specify what files are to be packaged into a single website bundle. This tool would be very useful, since a separate downloader client needs to run on the server for each website bundle; thus, if the website consists of many small pages, it will save resources if the server packages several pages into a single bundle. Allowing for an administrator deployment script could solve this problem as well.

7 Related Work

High availability of web content has become a serious problem due to several different causes. Most of the problems are caused by hostile computers performing DDoS attacks against some server or by the Slashdotting effect. Different approaches have been presented in attempts to solve this problem, ranging from web caches on the client side to load-balancing a server farm on the server side.

Noble et al. [14] present a fluid replication system that tries to make the web content more available via client-side replicas when performance of the main server is becoming poor. Coral [15] uses proxies and DNS redirection together with a sloppy distributed hash table (DSHT) to redirect the client to a cached copy of the desired page, hosted on some volunteer node. Both of the above solutions are strictly client-based.

Some of the server-side solutions are Akamai [16], Digital Island [17] and Mirror Image [18]. These companies host web services and content on their large server farms and load-balance among the machines to keep any one server from going down. By using DNS redirection, these companies are serving the desired content from the server cluster that is closest to the client, thus minimizing latency. However, such a solution is expensive and probably out of reach for small websites that often fall prey to the Slashdot effect.

BitTorrent [5], a basis for our WebTorrent application, combines the server-only and client-only approaches. It uses the increase in traffic to make the content more available – by using client bandwidth and resources. BitTorrent sets up a peer-to-peer network to share the popular file by breaking it into pieces and ensuring that the least-available piece is the first one to be downloaded off the Slashdotted server. This approach maintains availability, while the hashes distributed in the torrent file ensure consistency of all pieces. Some cooperation on the part of clients is needed (the clients need to run even after their download has completed to allow other peers to get the file from them), but the benefits are evident.

References

[1] Robert S. Thau, "Apache API Notes." *http://modules.apache.org/doc/API.html*

[2] "Autogenerated Apache 2 Code Documentation." *http://docx.webperf.org/*

[3] Netscape Communications, "Netscape Gecko Plug-in API Reference" *http://devedge.netwscape.com/library/manuals/2002/plugin/1.0/*

[4] Fielding et al, "Hypertext Transfer Protocol – HTTP/1.1" *http://www.ietf.org/rfc/rfc2616.txt*

[5] Python Implementation of the BitTorrent Package *http://bitconjurer.org/BitTorrent/*

[6] BitTorrent C Library. *http://sourceforge.net/projects/libbt*

[7] D. Stenberg, "LibCURL" *http://curl.haxx.se/libcurl/*

[8] W. M. Shandruk, *http://www.freebsd-howto.com/HOWTO/Ipfw-HOWTO*

[9] Bram Cohen, Interview with Slashdot.org *http://interviews.slashdot.org/interviews/03/06/02/1216202.shtml?tid=126&tid=185&tid=95*

[10] S. Adler, "The Slashdot Effect: An Analysis of Three Internet Publications" *http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html*

[11] CmdrTaco, "FAQ: What is the 'Slashdot Effect'?" *http://slashdot.org/faq/slashmeta.shtml*

[12] CmdrTaco, "Suggestions: Slashdot should cache pages to prevent the Slashdot Effect" *http://slashdot.org/faq/suggestions.shtml#su900*

[13] Geek.com, "Slashdotted: Surviving the Slashdot Effect" *http://www.geek.com/features/slashdot*

[14] Brian Noble, et al., "Fluid Replication" *http://mobility.eecs.umich.edu/papers/netstore99.pdf*

[15] Michael Freeman, et al., "Democratizing content publication with Coral," In Proc. 1st USENIX/ACM Symposium on Networked Systems Design and Implementation, San Francisco, CA, March 2004. *http://www.scs.cs.nyu.edu/coral/*

[16] Akamai Technologies, Inc. *http://www.akamai.com/*

[17] Digital Island, Inc. *http://www.digitalisland.com/*

[18] Mirror Image Internet. *http://www.mirror-image.com/*