# Automatic Transition To Peer-to-Peer Download

Roger Deloy Pack

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Daniel Zappala, Chair
Mark J. Clement
Christophe Giraud-Carrier

Department of Computer Science

Brigham Young University

August 2010

ABSTRACT

Automatic Transition To Peer-to-Peer Download

Roger Deloy Pack

Department of Computer Science

Master of Science

For traditional web servers, available bandwidth decreases as the number of clients increases. This can cause servers to serve files slowly or to become completely overwhelmed when load grows too high. BitTorrent is a peer-to-peer solution to this problem, but it requires manual configuration for each file to be delivered this way. We develop a new system that integrates peer-to-peer file delivery with traditional client-server downloads. Clients initially attempt to download a file from a web server; if this is too slow, they transition to peer-to-peer delivery. Experiments with a prototype system show that it serves up to 30x faster than traditional web servers.

Keywords: Peer-to-Peer Web, BitTorrent

# ACKNOWLEDGMENTS

Thank you to my wife, the most loving person I know, my Mom, the epitome of patience, Nikkala and my Dad, my educational role models, Ben, my work role-model, Jon, my cohort, and Carolyn, my best bud. You deserve to read this! Also thanks to the many faculty members and teachers who have helped me grow through the years.

BRIGHAM YOUNG UNIVERSITY

SIGNATURE PAGE

of a thesis submitted by

Roger Deloy Pack

The thesis of Roger Deloy Pack is acceptable in its final form including (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory and ready for submission.

| | |
|---|---|
| Date | Daniel Zappala, Chair |
| Date | Mark J. Clement |
| Date | Christophe Giraud-Carrier |
| Date | Kent E. Seamons, Graduate Coordinator |
| Date | Thomas W. Sederberg, Associate Dean, College of Physical and Mathematical Sciences |

## Contents

## List of Figures

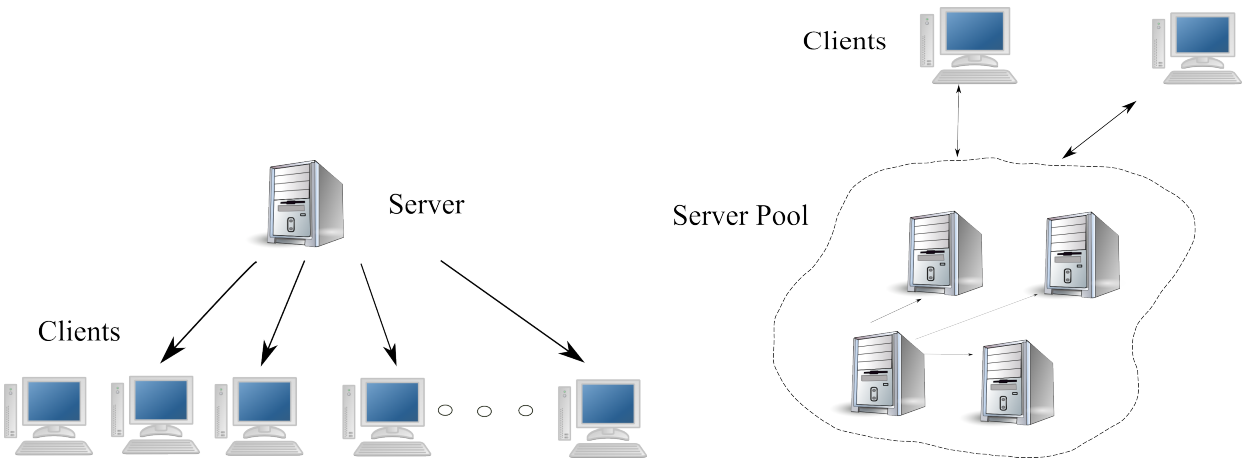
## List of Tables

# Chapter 1

## Introduction

In the traditional Internet architecture, a single server handles multiple simultaneous incoming client requests. The bottleneck in such systems under high load can become lack of bandwidth from the server [15], because bandwidth is split $N$ ways among the requesting clients (Fig. 1.1(a)). When an increasing number of clients request a web page, that page therefore loads more and more slowly for users. This problem occurs any time the ratio of bandwidth to clients is low, for example, when a server is poorly provisioned, it experiences a sudden spike in load, or a small number of incoming clients download very large files. An example of this problem is the "slashdot" effect, in which a sudden flash crowd of clients suddenly requests certain pages, overwhelming unexpectant servers. These situations cause web servers to server files slowly, become overwhelmed, or even crash.

This problem is commonly combated by distributing the requests among a set of servers that mirror the content of the original server (Fig. 1.1(b)). A company establishes a content distribution network (CDN) of mirrors by locating a set of servers in many different places on the Internet, then automatically redirecting clients to these servers, thus distributing the load to provide better performance. An example of this is the large commercial CDN run by Akamai [1], which performs distributed downloads for its clients. For example Apple uses Akamai's CDN to distribute their iTunes™program. This solution, however, requires a dedicated pool of servers to provide the extra bandwidth. Not all sites can afford a CDN, and mirrors require cooperation, configuration, and maintenance.




(a) Traditional HTTP download (b) Server CDN example

Figure 1.1: Traditional client server download

As a cheaper alternative, many web sites are beginning to use *peer-to-peer content distribution*, or *swarming*, with BitTorrent. In BitTorrent, clients download only some blocks of a file from a central server, and download the other blocks from their peers in the system. While a client participates, it is both downloading the blocks it needs and uploading the blocks it has to other peers (Fig. 1.2). Peer-to-peer content distribution enables a web server to serve a large file to a large numbers of users with better scalability [21]. Swarming protocols have been shown to actually decrease total download time per peer as the number of peers increases, which is the opposite of the traditional client-server paradigm [18].

Using BitTorrent currently requires some configuration for both servers and clients, and isn't well-integrated into normal HTTP downloads. A server wishing to use BitTorrent must first create a special file with a ".torrent" extension that lists a target file's size, the MD5 hashes for each block of the file, and the IP address of a tracker for that file. The tracker is a dedicated machine that helps connect the peers to each another, and must be administered by the site using BitTorrent. Users start a BitTorrent download by selecting the ".torrent" file on a web server, which launches a BitTorrent client program. The BitTorrent client will contact the tracker to receive a list of peers who are currently downloading the file. It then establishes connections and begins sharing and receiving blocks with its neighbors.

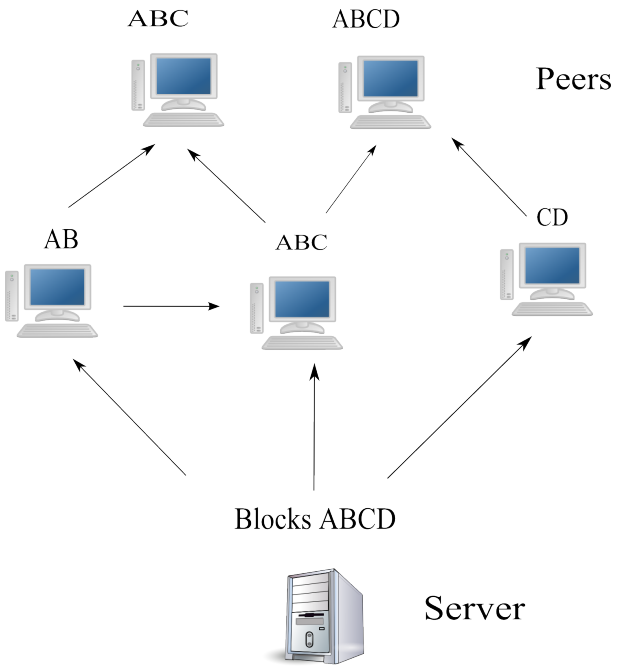



Figure 1.2: Swarming example. Arrows represent block exchanges.

BitTorrent is not commonly used for serving all the files of a web site since it requires per-file configuration, and a tracker must be established for the files. It is used mostly for large files where configuration time is not as much of an issue. Since BitTorrent uses a separate program, it will not work well for downloading web pages which must be viewed in a browser. BitTorrent also requires users to install client software and understand how and when to use it, creating a barrier to entry. For example, many download sites offer links to BitTorrent downloads, but users may ignore them.

In this thesis, we integrate peer-to-peer delivery into a standard web client, so that an unmodified web server can automatically offload traffic as load increases. This system provides for automatic integration of swarming downloads with HTTP, thus lessening the configuration and manual intervention needed for use. Using experiments on PlanetLab, we show that peers who request small files can receive them at a rate of up to 30x the speed of traditional file server download, and can decrease the load on busy servers.

# Chapter 2

# Related Work

Much work has been done to study distributed file downloading. Solutions fall into three basic categories: client-side protocols, cooperative (client and server-side) protocols, and server-side protocols.

## 2.1 Client-Side Protocols

Protocols that use only client-side interaction require no change in the server software, allowing them to be implemented in clients without requiring servers to be aware of the changes. However, this requires that clients must self-organize without the help of the centralized server. Our protocol is an example of such a system.

Shared web caches are an example of a client-side only protocol. They allow peers to download files from some well known cache – typically a specific set of computers, such as those run by PlanetLab. Coral [10] and CoDeen [22] are examples of such caches. Shared caches like these require a dedicated set of proxies, however. Squirrel [11] is another instance of a shared web cache. Squirrel clients join a local Distributed Hash Table (DHT) which is used to track which files have been downloaded by which peers. The DHT is then used to locate the peer owners of files, who then serve as proxies for that file. The advantage of this is it doesn't require dedicated infrastructure. The disadvantage is it does not offer an algorithm for transparent transition to peer-to-peer delivery, nor does it allow for partial file downloads. It was also built on the assumption of being run on a local (well-connected) network, not on the Internet at large.

Coral-style caches are useful, but have limitations. They are constrained to the number of participants, and they require an infrastructure of hosts who are willing to altruistically provide bandwidth, which is rare, because of expense. Also, when load surpasses the bandwidth of the combined sum of proxies, they will still become overburdened. For example, if Coral receives a request for a file too many times, it begins redirecting peers back to the origin server, thus not solving the original problem. Coral is also limited in the size of files it can cache by PlanetLab policies. With our system this should not be a problem, as more peers means that more are available to upload. These systems also lack an algorithm for transition to peer-to-peer delivery, and sharing of blocks among peers. They also add an extra hop in network latency per request, even if a request from the origin server would have been the fastest way to download the file.

### 2.2 Cooperative Protocols

With cooperative protocols, a server and clients work together to provide scalable downloads. Participation by the server makes it easier to build a system, but it will only work for those sites that adopt the new mechanism.

One style of cooperation for web servers uses redirection. For web redirection, servers typically redirect clients to former clients that have downloaded files previously [13, 15]. This allows a server to "meter" its upload speeds and redirect peers, when appropriate, thus providing a backup strategy for over-loaded servers. An example of this is the pseudo-server system [13]. This style of protocol typically requires changes to both the server and client software, however, and the server could still become overloaded in extreme cases.

A second style of cooperation uses swarming to download large files. This is best exemplified by BitTorrent. Swarming protocols are highly effective at serving loads orders of magnitude higher than an ordinary web server [21].

OnionNetworks has proposed an extension to HTTP to incorporate swarming. With this protocol, HTTP response messages include hashes of files and a list of peers that clients

may connect to, and download from, in a swarming fashion [6]. Unfortunately their proposal has not seen wide-spread adoptance nor research, and in extreme cases the origin server could still be overwhelmed.

The Shark [2] filesystem allows clients to download blocks of files from nearby neighbors who have blocks already on their local machines. Shark is similar to our proposed system, except applied to files in a file system, not web objects. It requires a custom DHT for peer localization and proximity estimation, and a central server for hash values of files. It also lacks HTTP integration.

Some recent proposals integrate HTTP clients with BitTorrent itself [19]. In these solutions a user is presented the option of downloading via HTTP or via BitTorrent (swarming), depending on which they think will give them the quicker download. This provides a backup for overloaded servers. To the users this requires a manual choice between normal and swarming downloads. The Osprey system [16] is an FTP server that automatically generates .torrent files for all files in an FTP sub-directory, and integrates a BitTorrent tracker into the server software to handle the different types of requests. These systems allow peers to manually switch to swarming if the server load becomes too high. There has also been development in plugins for web browsers to ease downloading of BitTorrent files [14, 12], and java applets to ease the cost of client installation [5]. Web seeding integrates HTTP servers into BitTorrent downloads [23]. It currently requires extra configuration by the server, and suffers from the problems above. There has also been the creation of "trackerless" torrents, which allow peers themselves to act as trackers for the peer lists [23], but still no integration of such with normal HTTP downloads. There has also been the introduction of "Metalink" files which list both HTTP and BitTorrent sources for files [24], allowing for combined HTTP and BitTorrent downloads, but use of these files requires their creation and that both servers and clients to understand and use the protocol.

Dijjer [7] is a tool that automatically performs swarming downloads of any web file. If passed a URL like http://dijjer.org/get/http://mysite.com/video.mov, the request is inter-

cepted by the locally running Dijjer software, which performs a distributed download of the file. One can also right-click on any arbitrary link and select "download via Dijjer" for the same effect. Dijjer contacts a quasi-DHT (similar to Freenet [8]) for block hashes and downloads the blocks from random peers, also caching blocks from random peers that contact it and request blocks. Unfortunately, Dijjer lacks automation in the transition to download, and, as the Dijjer software is currently implemented, requires peers to cache material in which they were never interested. This may be instrusive, for example by clients potentially caching illegal or unethical web objects. Also the DHT they use is non-deterministic, which might be less than optimal, and their protocol has little published research.

Overall, client and server side cooperation protocols work well at alleviating flash crowds, and a few provide manual transition to peer-to-peer delivery. However, they require both the server and the client to understand the protocol, which hinders their adoptability, and lack an automatic transition.

### 2.3 Server-side Protocols

Server-side only solutions typically create a pool of servers that mirror each others' content, helping to alleviate the impact of flash crowds [20, 25]. In these systems, servers will cover for each other, so that if one becomes very crowded then the others will share the load with it until load decreases. Backslash [20] is one example. In Backslash, when one server becomes overloaded it redirects requests and uploads necessary files to other servers for them to act as temporary mirrors, therefore relying on the volunteer hosts in the system.

Although server-side systems can help, they require cooperation among web sites, which typically doesn't happen. Client-side solutions have the potential to provide good performance, regardless of whether the server has taken steps to deal with high load.

# Chapter 3

## Automated Swarming

Automated swarming is a client-side system that automatically switches from client-server to peer-to-peer content delivery when a server becomes overloaded.

The goals for this system include:

1. It should be transparent to servers. This allows the origin server to remain unchanged, so end users can benefit from the system despite using servers who know nothing about it.
2. It should be easy to use by end-users by automatically transitioning to peer-to-peer content delivery when the server is slow.
3. It should not require dedicated, special-purpose infrastructure. Using a general-purpose infrastructure, coupled with a dependence on the clients for transferring blocks, makes this system easier to adopt.
4. It should be non-intrusive. Peers should not be required to cache blocks of files in which they were never interested. Peers will avoid caching files they never downloaded, and thus will not be responsible for content they don't anticipate. Users would also only be using their bandwidth for files they previously downloaded, encouraging adoption.
5. It should be fast for small files.

To accomplish this, in automated swarming clients monitor downloads and transition to peer-to-peer delivery when a download becomes slow. Interested peers then find each

other using a Distributed Hash Table (DHT). They download the file from peers, then add themselves to a list of peers willing to share the file.

### 3.1 Transition to Peer-to-Peer Delivery

A peer wishing to download a file using automated swarming first tries to download it from the origin web server using a traditional client-server download. If at some point one of the following conditions occurs, the client switches to peer-to-peer delivery:

1. First, the client waits a maximum number of seconds $T$ after the start of a download, to receive the first byte of data from the origin server. The first data byte means any byte received from the origin–a header or content byte. TCP connection handshake packets do not count as a data byte. If $T$ seconds pass without the client receiving any data, it transitions to a peer-to-peer delivery. This allows the system to decide quickly whether the origin server is over-burdened.
2. After the client begins receiving data from the server, it monitors whether the download rate falls below a certain threshold of $R$ bytes per second over the last $W$ seconds. It starts measuring this after the first byte is received. This means that it will only ever decide that the server is slow after $W$ + the time it took to receive its first data byte, or at least $W$, and at most $T + W$. If the receive rate ever drops below $R$, the client transitions to peer-to-peer delivery. This is to accomodate servers with slow connections or servers that become overloaded during a download.

### 3.2 Finding Peers

Peers interested in downloading the same file find each other using a Distributed Hash Table (DHT). A DHT is essentially a distributed peer-to-peer hash table that maps a key to a value. A client can, for example, query the DHT with a key of "file x block y" and get a list of peers with block y of file x.

We run a DHT using the Bamboo OpenDHT code on available PlanetLab peers– typically about 300 are active. Three peers are designated as "gateways" that the others contact to bootstrap themselves into the DHT. After servers join the DHT, each creates a web service that accepts queries from peers outside the DHT. When a server receives a query, it forwards it through the DHT, retrieves the answer, and returns the answer to the requestor. Members of the DHT run continously, servicing requests. Data is stored as key, value maps, with many values storable per key. If there are more than ten values stored, only the first 10 values are returned for the first request. If a peer wants more than the first 10 values, it must make a repeat query and request values 11-20, etc. We use the DHT to store peer information, to help peers find each other. An alternative design would be to store blocks of the file in the DHT. However, this would put more stress on the DHT, due to bandwidth demands. This design would also violate design goal #4.

In order for our clients to use this DHT, each is given a list of all known DHT members out-of-band. When they are instructed to download a file, they "port sniff" on the web service ports of DHT members, to create a list of live DHT members. They query 10 DHT servers at a time, until they have found at least 5 that are active. They then round robin across the list of known active members for queries they later perform.

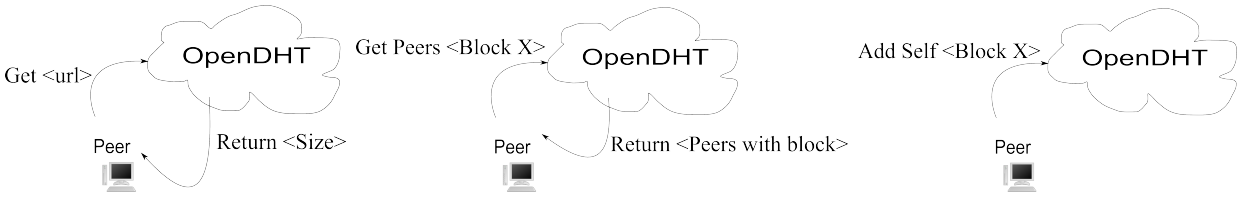
When a peer performs a query, for example for a key "http://host/filename_peers_for_block_num_y", it requests this key and also a "redundant key" for the same, like "http://host/filename_peers_for_block_num_y_key_redundancy_1". Note that Bamboo actually queries a hash of the query string, not the string itself. The peer requests both keys from two different gateways (using the round robin described above), thus the total number of queries per request is four. The reason for this is that sometimes keys are hosted on poorly connected members of the DHT, making response time slow. Sometimes gateways themselves are temporarily slow, so using multiple gateways per request avoids this slowdown, as recommended by the authors of OpenDHT [17]. The client times out DHT requests after 60 seconds if no response is received, as a reasonable value to

keep from overloading the DHT. When a request times out, the client will automatically retry it, for up to three times, before giving up on the request.

When a client transitions to peer-to-peer delivery, it first checks to see if it knows enough meta-data about the file it is downloading. It needs to know the size in order to look up peer lists. If the client has previously received an HTTP header with size information from the origin, it has the information it needs. If it doesn't, it queries the DHT, using a key of the URL being downloaded, for known meta-data. Other peers store the meta-data here when they have previously downloaded the file (Fig. 3.1(a)). While performing this meta-data lookup, a client also does an HTTP HEAD request for the file to the origin server, and uses the first successful response from either of these two to determine file size. If there is no meta-data listed in the DHT, a client continues to poll the DHT every 5 seconds to see if meta-data for the file has appeared. At some point either the meta-data will appear in the DHT or the HTTP HEAD request will return (with size information) and the peer will then have the meta-data it needs and will start peer-to-peer delivery.

### 3.3 Peer-to-Peer Delivery

To accomplish peer-to-peer delivery, the client randomly chooses $b$ blocks of the file to download at a time (Fig. 3.1(b)). It retrieves the list of peers who have each block by querying the DHT for a key composed of the URL concatenated with a given block number (like "http://host/filename_peers_for_block_num_y"). If there are no peers listed yet for that block, it polls the DHT every second to see if new peers have arrived. While it does not yet have a list of any peers, it also contacts the origin and starts a download of the block, so as to not waste time in the case that no peers are ever available. This is useful for some peers that have DHT connections that are too slow. When it receives a peer list from the DHT, the client shuffles the list randomly, then connects to each peer one at a time, and requests the block. If the peer begins to serve the block, the client closes its connections to the origin, if open, and downloads the rest of the block from that peer. If the peer interrupts




(a) Peer downloads meta-data

(b) Peer downloads a list of peers that have a block

(c) Peer adds itself to list of peers who have the block

Figure 3.1: Steps to accomplish a peer-to-peer-web download

the connection or is not online, the client chooses the next peer until it runs out of peers to use. Then it repeats the steps for that block, but requesting the next 10 peers listed in the DHT.

When a peer only has a few blocks remaining, it tries to download the last few blocks from several peers (redundantly), in an effort to avoid the last block problem [4]. It downloads all remaining blocks from a total of $b$ peers, so if $b$ is set to 6 and 3 blocks are left, it downloads each remaining block from two redundant peers, with the last block being downloaded from all six peers redundantly.

### 3.4 Sharing

Clients must share information using the DHT to make automatic swarming work. First, when a peer learns the size of the file from its original HTTP request or its later HTTP HEAD request, it stores this "meta-data" about the file in the DHT. It does this by querying to see if this data has already been stored there. If it hasn't, it stores it by setting the value of the URL in the DHT to the file's size. Later peers then use this to retrieve the meta-data for the file if necessary. Second, whenever a client downloads a block, it adds itself to the list of peers willing to share that block (Fig. 3.1(c)). Finally, after completing a file download, the client lingers a certain number of seconds. After its linger time is up, it drops all current connections to peers and performs DHT remove requests to remove itself from the peer lists in the DHT.

# Chapter 4

## Experiment Results

To test our system, we implemented a prototype client in the Ruby programming language and ran experiments using the client on the Planetlab system. Typically about 300 clients participated in each experiment. To run each experiment, a central "driver" program randomly selects members of the pool and instructs them to download a specific URL. To choose members, the driver first pings known members of PlanetLab to create a list of currently active clients. It then randomizes this list and uses it in round robin fashion to launch clients. If more peers download the file than there are clients available, clients are re-used using the same list in a round-robin fashion, though peer-to-peer connections to other peers on the same host are dis-allowed. Client launch times are generated serially using a Poisson distribution with an inter-arrival time that is the total startup time divided by the number of peers. For example if 1000 peers are expected to enter the system within 100 seconds, it uses a Poisson distribution with an average inter-arrival time of 0.1 seconds to start peers, and finishes starting them in about 100 seconds. Peers are told the system time when they start a download, and use an offset from this time to log events. Because of TCP latency in transmitting the time, and system clock skew over time, log times can be up to a few seconds off.

Files are downloaded from a well provisioned server at BYU running an Apache web server (version 2.2.4) that is bandwidth limited to 256 KB/s using the mod_bw module [3], and it has a 256 connection limit. The 256 KB/s (2 Mb/s) upload limit is similar to a cable modem server, or two T1's [9], and can serve approximately 2.5 100 KB files per second,

which is reasonable for a low traffic web site. If it had a limit of 50 Mb/s, not uncommon in large datacenters, it could serve about 60 files per second, so we would expect the client-server speeds to be about 60 times as fast as those in our tests. The 256 connection limit is Apache's default server configuration.

Experiments run until all peers finish downloading the file. Each experiment is repeated three times and results are averaged. A different filename is used for each run, so as to not re-use the same DHT keys.

When peers are done downloading a file, the driver collects their logs and we calculate statistics. We measure the following:

1. *Download times–*

   The time for a client or peer to download the file, in seconds.

2. *Server upload speeds–*

   Average bytes uploaded over the time of the experiment, from the origin. This is measured by summing the number of bytes received from the origin during each second. This only measures bytes received from the origin, not necessarily all bytes sent, as peers sometimes close their socket connections to the host early, and any data received after the socket has already been closed is dropped by the kernel. We disregard the first and last 20% of entries for this, as the system hasn't reached steady state during those times.

3. *Percent received from peers–*

   We also calculate the sum percent of the file received from peers versus from the origin. This is the number of data bytes that were retrieved from the origin for any block of the file downloaded, compared to the number of data bytes received from peers during the download.

4. *DHT get, put, and remove times–*

We track the time it takes to do a DHT operation by measuring time elapsed from initiating a TCP connection with the DHT proxy until an answer is received, for peers during the course of an experiment.

5. *Type of peer downloads–*

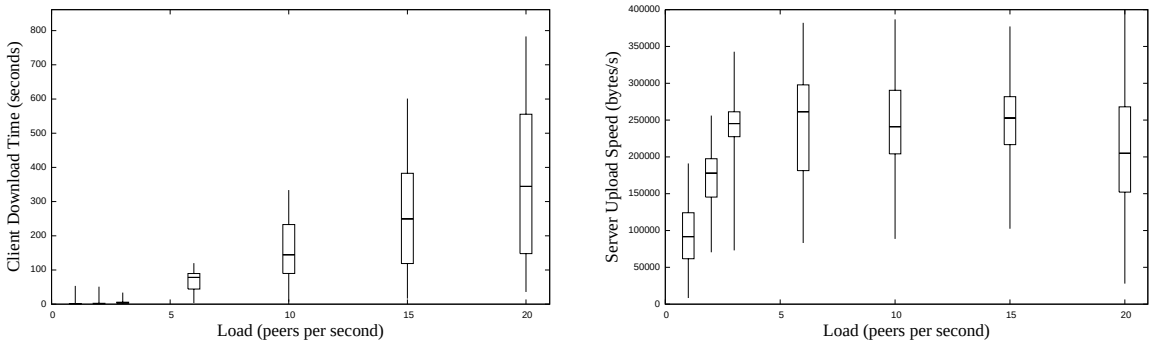
   We also keep track of the way that peers download a given file. Some peers encounter a fast server and download the entire file from the origin. Some peers transition to peer-to-peer delivery and complete the download that way, either because of a slow first byte ($T$), or a slow origin server speed ($R$). Note that if a peer transitions to peer-to-peer delivery, we count this as a peer-to-peer download, regardless of whether they actually receive bytes from their peers–in a few cases they may end up still downloading it all from the origin. Some peers fail to complete the download, typically because of network problems on their host.

All box plots show 25th and 75th percentiles, with outlying lines showing 1st and 99th percentiles, and mid-lines showing the median. For default system parameters, we choose reasonable values of block size of 100 KB, origin minimum speed ($R$) of 128 KB/s, $R$'s calculation window ($W$) of 2 seconds, and first-byte timeout ($T$) of 1 second. Peers download the file from up to 20 other peers at a time ($b$). Peers linger, serving the file, for 60 seconds after completing a download.

### 4.1 Scaling with load

For our experiments we first test how well our system performs under different loads. We vary load from 1 to 20 peers/second. Note that a semi-popular website might serve half a million hits a day, which averages to 6 per second, so our limit of 20 per second is similar, if you take into consideration spikes in load.

We run a traditional client-server transfer, to establish a base line from which to compare our system. Figure 4.1(a) shows the download times for client-server downloads, as




(a) Download times

(b) Load on the origin server

Figure 4.1: Traditional client server download under varied load.

load increases. Median download times start at 1.33 seconds with a load of 1 peer/second, and quickly grow to a high of 344 seconds with a load of 20 peers/second. Download times increase linearly instead of exponentially because we start a fixed number of peers and let all downloads complete, without more peers entering the system.

The top outliers at a load of 20 peers/second typically wait in line around 800 seconds before finally receiving the file from the origin. Figure 4.1(b) shows that load on the origin server grows quickly to its theoretical cap of 250 KB/s under a load of 3 peers/second. Under a load of 20 peers/second, server speed decreases to 203 KB/s, which shows the limitations of the rate limiter in that it becomes bursty and less reliable with higher connection rates.

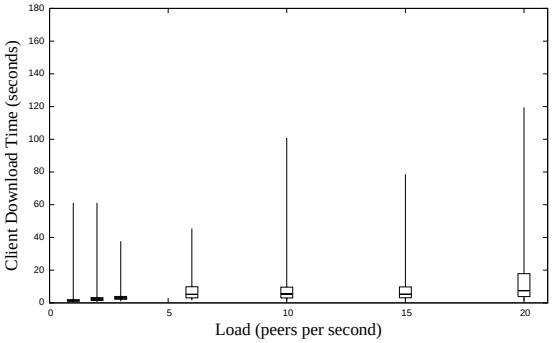
We next test automatic swarming under the same loads, using the default parameters. Median download times using automated swarming are more than 30 times faster than client-server times (Fig. 4.2(a) versus 4.1(a)). Times start at 1.4 seconds with a load of 1 peer/second, and grow to 5.23 seconds with a load of 6 peers/s and 7.46 seconds with 20 peers/s, compared to 344 seconds for client-server. The Y axis for Fig. 4.2(a) is different than that for client-server in Fig. 4.1(a) because the difference is so great that it would obscure the details.

We examined our data to determine how clients transition to swarming. Figure 4.2(b) shows which clients transition due to $R$ (slow download), which due to $T$ (slow first byte),

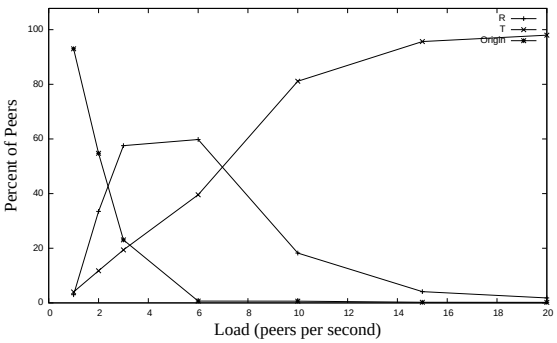
and which stay with the origin server. At a load of 20 peers/second, 99.5% of peers give up on the origin server because of a slow first byte after one second ($T$). They then query the DHT for a peer list, and receive a response with a median latency of 5.2 seconds. They download the file almost immediately once they receive the peer list, hence the median download time of about 7 seconds.

The amount of peer-to-peer transfer, as expected, increases with higher loads. Under low load clients tend to download the file only from the origin. After a load of about 10 peers/second almost 100% of the transfer is from peers. At the same time, load drops dramatically on the server, as shown in Fig. 4.2(c). For smaller loads, peers start downloading the file from the origin, receive some portion of it within the first $W$ (2) seconds, and then transition to peer-to-peer transfer because the rate of download is less than $R$. Thus for the first 2 seconds they are downloading from a slow origin, then they switch to fast peer-to-peer delivery. The first percentile of download times almost always are the first few clients, who encounter a still fast origin server and download the file directly from it. Later clients almost always download the file from peers.

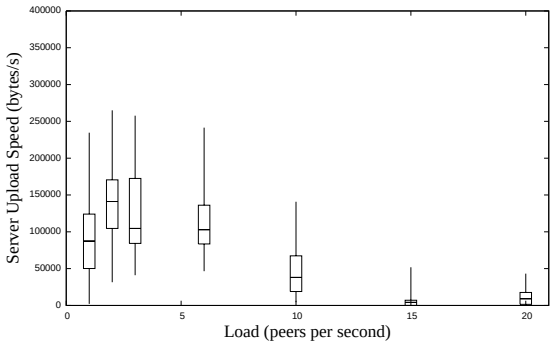
There are two basic causes for why some outlying clients take a long time to download the file. One is that a few poorly connected clients take so long to download blocks from peers that their peers finish lingering and drop the existing connections. These clients then have to request a new peer list from the DHT, causing an increase in latency. This problem is exacerbated by the fact that clients request the last block from multiple peers. This causes some redundancy in bytes received, which slows down incoming data, causing slow peers to expire their connections' linger time even more often. The other factor causing slowdown is a sometimes unresponsive DHT. Typically requests come back quickly, however a few requests timeout after 60 seconds, and peers either end up downloading the entire file from the original server within that 60 seconds, or have to perform a new query, and then download the file normally from their peers after that. We're not entirely sure why this behavior occurs with Bamboo.




(a) Download times




(b) Cause of transition to P2P download




(c) Load on the origin server

Figure 4.2: Automatic Swarming under varied load

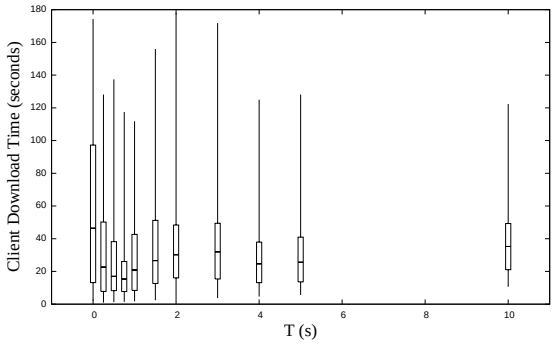
### 4.2 Impact of system parameters

If a peer can download a file quickly from the origin, it does no help to also download the file from its peers. Our system accomodates for this by specifying parameters for the conditions under which it will transition to a peer-to-peer delivery. We test the impact of varying these and other system parameters, in order to explore the dynamics of our system. In each test we start 1000 peers, with an average of 15 peers entering the system per second, using a Poisson distribution with an interarrival time of 0.066 seconds. Thus all peers enter the system within the first 67 seconds. Peers download a 100 KB file, using a 100 KB block size, $b$ (connection limit) set to 5, $R$ set to 128 KB/second and $W$ set to 2 seconds. Peers' linger time is set to 20 seconds. This load (15 clients per second) is similar to a cable modem serving 6x its capacity.

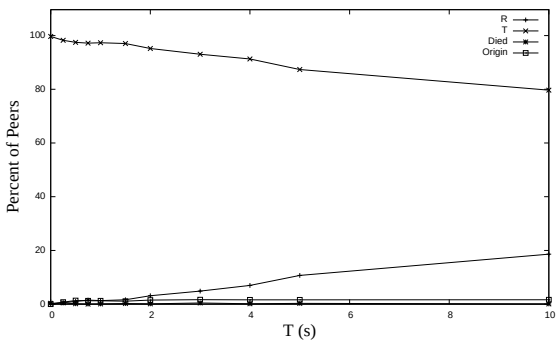
We first vary only $T$, the timeout for a response byte from the origin server. We expect that an extremely low value will cause transitioning too early and an extremely high value will cause transitioning too late, both resulting in slow download times. Figure 4.3(a) shows that the results hold to our hypothesis. With $T$ set to 0 seconds, median download time is 46.43 seconds. It then drops to 15.35 seconds with $T$ set at 0.75 seconds, but rises to 35 seconds with $T$ set to 10 seconds.

Figure 4.3(b) shows a breakdown of the reasons peers transition to peer-to-peer delivery. $T$ is the number of peers who transition to peer-to-peer delivery because of a slow first byte, $R$ is the number of peers that transition because of a too low value of $R$, *died* is the number of peers that never complete the download and *origin* is the number of peers that download the entire file from the origin server. Higher values of $T$ cause $R$ to become more important. With low values of $T$, almost 100% of peers transition to peer-to-peer delivery because of $T$. With $T$ set to 10 seconds about 80% of peers transition because of $T$, and 20% because of $R$, so far fewer.

We next examine the effect of varying $R$, the server's minimum calculated speed. We vary $R$ from 32 KB/s to 1 MB/s, and set $T$ to a reasonably high value of 10 seconds, so

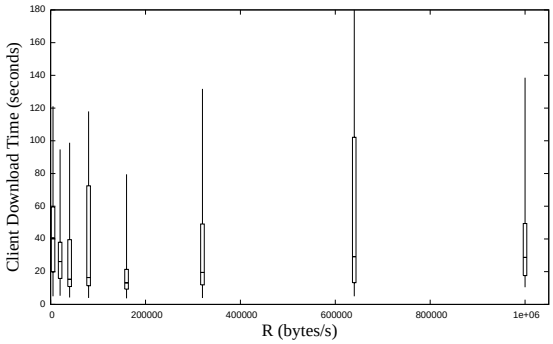



(a) Download times




(b) Cause of transition to P2P download

Figure 4.3: Varying T

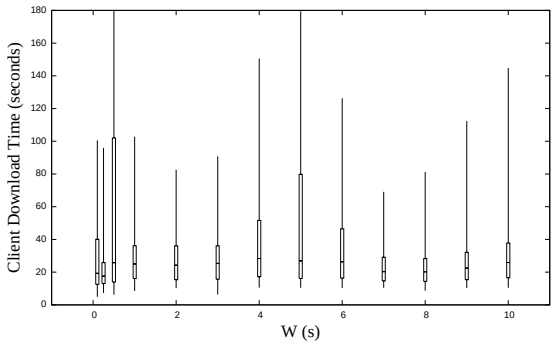



(a) Download times

Figure 4.4: Varying R

that $R$ will be more of a factor in causing peer-to-peer transition. The expected result us that too high of values of $R$ will cause peers to transition too early, and that values too low will cause peers to transition too late, both resulting in poor performance. Figure 4.4 shows that this is the case. With $R$ at 5 KBps, median download times are 40 seconds, with $R$ at 40 KBps, download times drop to 15 seconds, and reach a low of 13 seconds with $R$ set to 160 KBps. Higher values than 160 KBps, however, result in slower downloads. $R$ set to 1 MBps has a median download time of 30 seconds.

We next vary $W$, the amount of recent download history used to calculate $R$. Our hypothesis is that a small value for $W$ will cause the calculation to be too sensitive and thus slow download times because it will be inaccurate. We vary $W$ from 0.1 seconds to 10 seconds



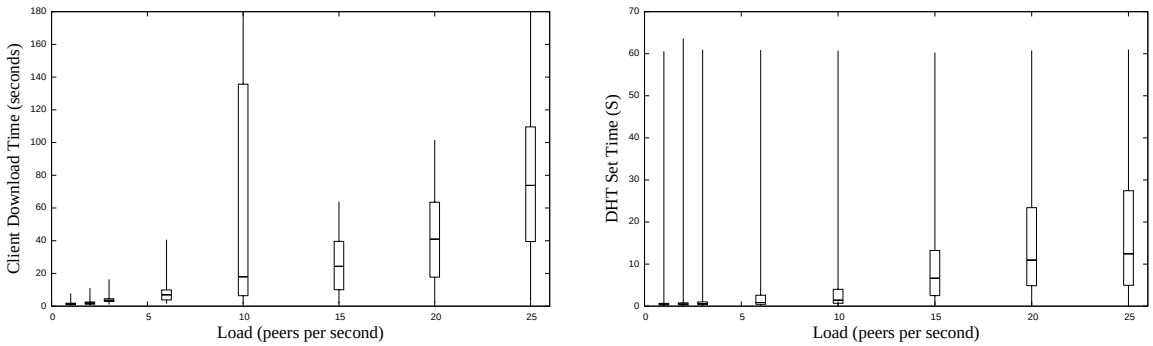

(a) Download times

Figure 4.5: Varying W

and run the same experiment described above, but with $T$ set to 10 seconds and $R$ set to 128 KB/s. Figure 4.5 shows the results are slightly different than our hypothesis. Varying $W$ does make some difference, but only when set to a very low value. Median download times are 17 seconds with $W$ set to 0.25 seconds. For $W > 0.25$ seconds median download times stay at approximately 25 seconds. Surprisingly, about 80% peers still transition to peer-to-peer delivery because of a slow first byte ($T$), even with $T$ set to its larger value of 10 seconds. The reason for this is that we don't start calculating $R$ until after the first byte is received, and in the majority of cases (800/1000) $T$ expires first, so $T$ still causes most of the transitioning, and appears to be more of a dominant factor overall. Lower value of $W$ only affects the first 200 peers, who receive bytes from the not-yet-overloaded origin. These 200 peers transition more quickly to peer-to-peer with a lower $W$, but there are relatively few of them.

### 4.3 Effect for a full web page

We next run an experiment where clients download a full web page. The normal pattern for a page view is to first access some root page which references several other objects, such as images, javascript, flash, css, etc. This is the case for a 2007 snapshot of the byu.edu web site, which had a medium sized main page that linked to over 10 smaller, objects. To




(a) Download times (b) DHT Put times

Figure 4.6: Downloading multiple simultaneous files

model this behavior, we run a test where each peer first downloads a 100 KB file; when that file download completes each peer then downloads ten other 10 KB files in parallel from the same origin server, and the total time to download all 11 files is logged. We set the parameters to be those mentioned above for testing system parameters, and vary the load from 1 peer/second to 25 peers/second.

The expected result for this experiment is that downloading multiple pages will be slower than one, since it will put more stress on the DHT. Our results in Figure 4.6 prove this hypothesis correct. Total download time starts at 1.3 seconds, and grows quickly with higher load, approaching 150 seconds at a load of 25 peers/second. DHT response times also grow similarly with load. Median DHT "put" times start at 0.41 seconds at a load of 1 peer/second, and grow to 12 seconds at a load of 25 peers/second. DHT get and remove times show a similar growth. This explains some of the download slowdown. Because linger time is set to 20 seconds, peer lists are outdated soon after they are created, because they typically take 12 seconds to set, then another 12 seconds to retrieve, and by that time peers' linger times have already expired.

As Figure 4.7 shows, a client-server system still fares much worse using the same load. The upper line is the same test using client-server download.

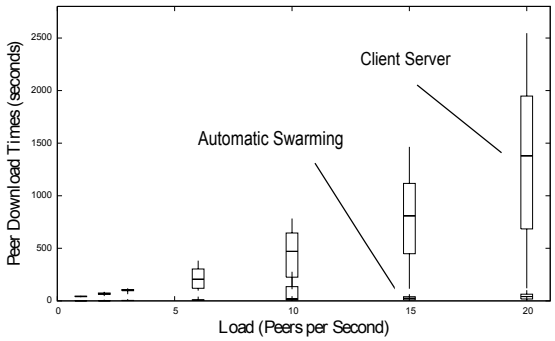
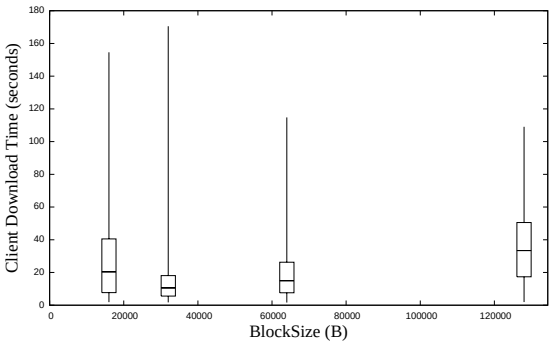



Figure 4.7: Comparison of P2P versus client server download times for multiple files

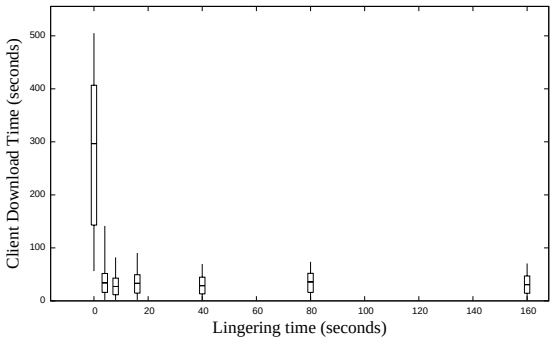



(a) Download times

Figure 4.8: Varying block size

## 4.4 Varying Block Size

We next measure the effect block size has on download time. We download a 100 KB file with block sizes ranging from 16 KB to 100 KB, using the same default parameters as specified above, and 100 peers entering the system within 100 seconds. Figure 4.8 shows that 32 KB blocks result in the quickest downloads. This is faster because downloading more blocks allow our system to download blocks from several peers simultaneously, without redundancy. In this case a block size of 32 KB results in a total 3 blocks, and since $b$ by default is 5, all 3 blocks are downloaded simultaneously. 32 KB blocks were also shown to be most effective in [21].




(a) Download times

Figure 4.9: Varying linger time

## 4.5 Varying linger time

We next vary the amount of time peers linger, our hypothesis being that a longer linger time will speed downloads. Figure 4.9 shows this to be correct. We download a 100 KB file using the above default parameters, varying linger time from 0 to 160 seconds, with 100 peers entering the system within 100 seconds. With a linger time of 0 seconds, median download time peaks at 300 seconds, with a linger time of 1 second, median download time drops to 127 seconds. With a linger of 2 seconds, download time further drops to 60 seconds, and with a linger time of 4 seconds, download time drops to 34 seconds. Download time stays at about that same level for linger times greater than 4 seconds.

## 4.6 Downloading Large Files

We next test our system's ability to download large files. We download a 30 MB file using our system and then again using BitTorrent. For both systems, block size is set to 256 KB, origin servers are rate limited to 256 KB/second, and linger time is set to 0 seconds, though peers still share the file while downloading it. 100 peers enter the system at an average of 1 per second.

The expected result is that the two fare similarly. Our results, in Table 4.1, show that a few peers download faster with our system, but most download faster with BitTorrent.

Ours has a slower median time than BitTorrent (847 to 148 seconds), though it has a faster 99th percentile (982 to 2786 seconds). We're not entirely sure why BitTorrent is faster. One factor affecting performance is that BitTorrent's seed limits outgoing connections to 7, whereas Apache's connection limit is 256. This may allow BitTorrent to propagate full blocks more quickly to peers. BitTorrent's seed also favors peers with higher download speeds, which may help propagate blocks. In this test, it uses a dedicated tracker, which makes peer rendezvous quicker than using a DHT. BitTorrent uses a "rarest block first" selection policy, enabling it to choose blocks more efficiently. These differences may allow it to respond to a flash crowd such as this one better. In future work our system can be optimized to handle loads such as this better.

One interesting statistic is that the relative download times for each system are similar. The difference between the 25th and 75th percentile download time in our system is 150 seconds, or 5% of the 75th percentile's time. The difference between the 25th and 75th percentiles in BitTorrent is 24 seconds, or 7% of the 75th percentile's time: a similar percentage. These numbers implies that once a few peers have downloaded a file in its entirely, it doesn't take relatively long for the rest of the peers to complete the download. Figure 4.10 shows that most of the delivery comes from peers.

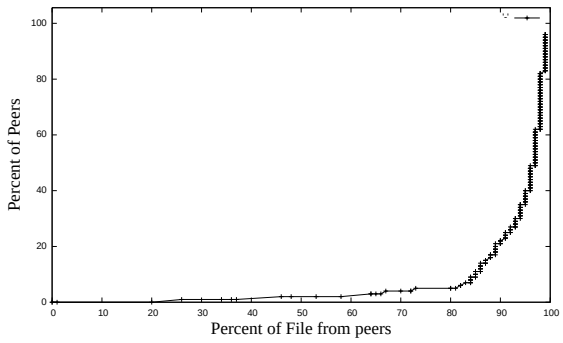
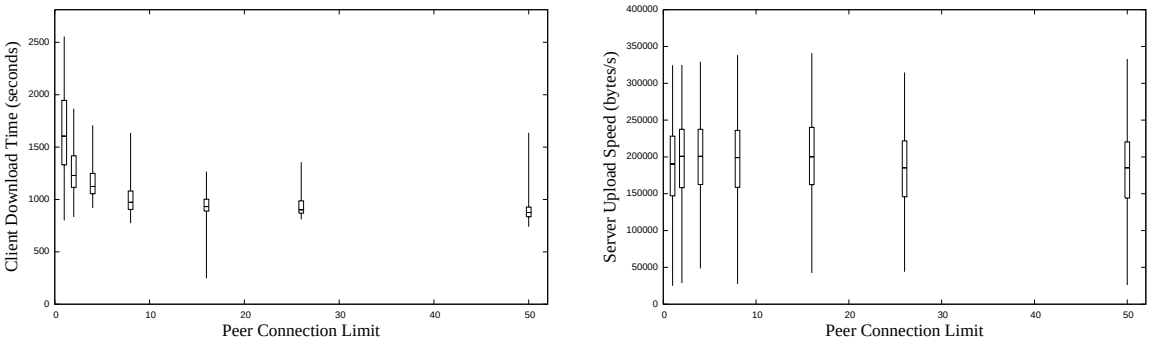



Figure 4.10: CDF of percent of file received from peers, 30MB file

Table 4.1: Download times of a 30MB file

| Percentiles | Our System (S) | BitTorrent (S) |
|---|---|---|
| 1 | 613 | 131 |
| 25 | 730 | 139 |
| 50 | 847 | 148 |
| 75 | 882 | 163 |
| 99 | 982 | 2786 |




(a) Download times (b) Load on the origin server

Figure 4.11: Varying number of concurrent peers

### 4.7 Varying peer connection limit

We next vary the maximum number of simultaneous download connections each peer is allowed ($b$). We repeat the above 30 MB download experiment, but vary the peer limit from 1 to 50. We expect that download times will suffer when the connection limit is too low. Figure 4.11 shows that, as expected, median download times suffer with a low limit. A limit of 1 connection yields a median download time of 1604 seconds. A limit of 16 has a median download time of 931 seconds. Limits greater than 16 yield slight improvements: a 50 peer limit results in a median download time of 875 seconds.

# Chapter 5

## Conclusion

We have shown that a system of cooperating web clients can reduce load on an origin server by automatically switching from client-server file transfer to peer-to-peer content delivery. Experiments show that our system is capable of serving up to 30x faster than a traditional web server. We have also found good settings for various system parameters. A server timeout ($T$) setting of 0.75 seconds resulted in best performance, as did a bandwidth limit ($R$) of 160 KBps. File block size of 32 KB was most effective, as was a linger time of 4 seconds or more and a peer connection count of 16 or more. The system is most effective for small files but does not yet compete well with BitTorrent for large files. The system met our goals of transparency to servers and ease of use, showing itself a viable option for users who wish to automate faster downloads.

Several aspects remain to be improved. All clients currently connect to the origin server, at times causing the same blocks to be served (redundantly) to various clients, slowing down transfer. We also do not validate the integrity of downloads. We assume peer trustworthiness and no file corruption. We also do not provide peer incentives for sharing. Peer lists could be optimized. Because they are returned in chronological order from Bamboo (oldest first), our system can experience problems if peers go offline without removing themselves from lists, or if linger times are close to expiring. This also causes peers to download blocks from the the oldest 10 peers listed, which can cause unfairness. In the case of a slow Internet connection, linger times may have already expired before peers even receive the list.

Under high loads DHT response times increase substantially. For example, if there is an extremely popular file, the requests for that peer list might overwhelm the DHT member responsible. There is also some redundancy in peer lists for large files. Clients request several different peer lists, one per block. These lists might be all the same currently, thus there is some avoidable redundancy and latency.

Currently this system is mostly useful for overloaded sites with static pages. There could be some way for peers to deal with dynamic pages, such as to store meta-data about which files are (or appear to be) static, and which are dynamic.

Currently we used fixed system parameters, such as $R$, $W$, and $T$. It may be useful to dynamically optimize these values.

## References


[1] Akamai. Akamai EdgeSuite Content Distribution Network, October 2006. http://www.akamai.com/en/html/services/edgesuite.html.

[2] S. Annapureddy, M. Freedman, and D. Mazieres. Shark: Scaling File Servers Via Cooperative Caching. *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation, Boston, USA, May*, 2005.

[3] I. Barrera. Summer Of Code Project : Bandwidth Limiter for Apache (mod_bw), 2006. http://bwmod.sourceforge.net.

[4] A. Bharambe, C. Herley, and V. Padmanabhan. Analyzing and Improving a BitTorrent Networks Performance Mechanisms. In *Proceedings of IEEE INFOCOM*, volume 378, pages 1–12, 2006.

[5] D. Castagna, A. Bahgat, and Shehata. Bitlet - The BitTorrent Applet, 2010. http://www.bitlet.org.

[6] J. Chapweske. HTTP Extensions for a Content-Addressable Web, May 2002. http://www.open-content.net/specs/draft-jchapweske-caw-03.html.

[7] I. Clarke. Dijjer Peer-To-Peer File Distribution, 2006. http://code.google.com/p/dijjer.

[8] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2009.

[9] Comcast. Comcast High Speed Internet: Speed Comparison, 2010. http://www.comcast.com/Corporate/Learn/HighSpeedInternet/speedcomparison.html.

[10] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing Content Publication With Coral. *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 239–252, 2004.

[11] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A Decentralized, Peer-To-Peer Web Cache. *Proceedings of the 21st Annual Principles of Distributed Computing*, pages 213–222, 2002.

[12] T. Jacobs. FoxTorrent FireFox BitTorrent Inline Plugin, 2008. http://code.google.com/p/foxtorrent.

[13] K. Kong and D. Ghosal. Pseudo-Serving: A User-Responsible Paradigm for Internet Access. *Computer Networks and ISDN Systems*, 29(8-13):1053–1064, 1997.

[14] Opera. Opera Integrates BitTorrent in Upcoming Browser, 2006. http://www.opera.com/press/releases/2006/02/06.

[15] V. Padmanabhan and K. Sripanidkulchai. The Case for Cooperative Networking. *Proceedings of the International Workshop on Peer-To-Peer Systems*, pages 178–190, 2002.

[16] J. Reuning and P. Jones. Osprey: Peer-To-Peer Enabled Content Distribution. *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries*, pages 396–396, 2005.

[17] S. Rhea, B. Chun, J. Kubiatowicz, and S. Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *Proc. of the Second USENIX Workshop on Real, Large Distributed Systems*, pages 25–30, 2005.

[18] R. Sherwood and R. Braud. Slurpie: A Cooperative Bulk Data Transfer Protocol. In *INFOCOM*, pages 941–951, 2004.

[19] G. Sivek, S. Sivek, J. Wolfe, and M. Zhivich. Webtorrent: A BitTorrent Extension For High Availability Servers, 2004. http://pdos.csail.mit.edu/6.824-2004/reports/jwolfe.pdf.

[20] T. Stading, P. Maniatis, and M. Baker. Peer-To-Peer Caching Schemes to Address Flash Crowds. *1st International Workshop on Peer-To-Peer Systems*, pages 203–213, 2002.

[21] D. Stutzbach, D. Zappala, and R. Rejaie. The Scalability of Swarming Peer-To-Peer Content Delivery. *Proc. of the 4th International International Federation for Information Processing-TC6 Networking Conference*, pages 15–26, 2004.

[22] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and Security in the CoDeeN Content Distribution Network. *Proceedings of the USENIX 2004 Annual Technical Conference*, 2004.

[23] Wikipedia. BitTorrent (protocol)— Wikipedia, The Free Encyclopedia, 2010. http://en.wikipedia.org/wiki/BitTorrent_(protocol).

[24] Wikipedia. Metalink — Wikipedia, The Free Encyclopedia, 2010. http://en.wikipedia.org/wiki/Metalink.

[25] W. Zhao and H. Schulzrinne. DotSlash: A Self-Configuring and Scalable Rescue System for Handling Web Hotspots Effectively. *International Workshop on Web Caching and Content Distribution*, pages 1–18, 2004.