2008-06-27

Reducing Seed Load in the BitTorrent File Sharing System

Brian T. Sanderson
Brigham Young University - Provo

REDUCING SEED LOAD IN THE BITTORRENT FILE SHARING SYSTEM

by

Brian Sanderson

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2008

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Brian Sanderson

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Daniel Zappala, Chair
Date	Quinn Snell
Date	Mike Jones

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Brian Sanderson in its final form and have found that (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 to the graduate committee and is ready for submission to the university library.

Date	Daniel Zappala Chair, Graduate Committee

Accepted for the Department

Date	Kent Seamons Graduate Coordinator

Accepted for the College

Date	Thomas W. Sederberg Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

REDUCING SEED LOAD IN THE BITTORRENT FILE SHARING SYSTEM

Brian Sanderson

Department of Computer Science

Master of Science

BitTorrent is an attractive peer-to-peer technology that attempts to reduce load on file sharers by allowing downloaders to share content between themselves. BitTorrent's current focus is to provide users with a fast download, which requires the file sharer to serve a disproportionate amount of the file. We present a modification to the BitTorrent seeding algorithm that reduces the load on BitTorrent file sharers. Essentially, if a block of a file is already available from a significant number of peers, the file sharer refuses to share that block, forcing peers to get it from each other. Using this modification, we show that there is a trade-off between the server's expended upload bandwidth and a longer peer download time. We also show some cases where we reduce the server's load as well as maintain a competitive peer download time by increasing the availability of rare blocks.

ACKNOWLEDGMENTS

To my wife Amy, who earned this degree as much as I did.

Contents

1 Introduction **1**

2 Related Work **7**

3 Implemented Solution **11**

3.1 Estimating Block Diversity 12

3.2 Reducing Seed Load 15

3.2.1 All blocks 15

3.2.2 Random selection 16

3.2.3 Less than k 16

3.2.4 Probabilistic s 16

3.2.5 Shuffle Rare 17

3.3 Choosing who to ping 17

3.4 Seed-seed interactions 18

4 Methodology **21**

4.1 Logging BitTorrent Messages and Events 23

4.2 Dealing with clock alignment 25

4.3 PlanetLab deployment 26

4.4 Practical Considerations 27

4.5 Computational tools 28

4.5.1 Log parser 28

4.5.2 Event log 28

4.5.3 Analyzer 28
4.6 Workload generator 29
4.7 Experiments 29

5 Results **31**

5.1 Overall performance 32
5.2 Reducing seed load 36
5.3 Comparing the download times 38
5.4 Increasing Block Diversity 41

6 Conclusion **45**

Bibliography **49**

Chapter 1

Introduction

An old adage states, "together we can accomplish more than we could alone." Sharing and cooperation are at the core of today's popular peer-to-peer (P2P) technology. P2P is appealing for several reasons. First, it provides a mechanism for Internet users to connect and share information with each other. This reduces network traffic on popular Internet servers and allows downloaders to achieve higher download rates for popular files. Second, many P2P technologies allow users to host their own content, thereby allowing an entire community to share personalized information with each other.

BitTorrent [4] is a popular and attractive P2P system because it allows users to cooperatively download content and to easily publish personalized content for others. CacheLogic, a British Internet traffic analysis company, showed that 30 percent of 2004 Internet traffic was from BitTorrent alone, making it by far the most popular P2P client available at the time [12]. Another study has shown that as much as 80% of the traffic on backbone Internet links is from P2P traffic [6].

A user who wishes to download a file with BitTorrent first locates and downloads a *torrent*. A torrent file contains the address of a *tracker*, the name of the actual file to be downloaded, and a list of the fixed sized pieces of the target file, called *blocks*. Torrents are usually located on the Internet page of a host who is providing a BitTorrent download (for instance, OpenOffice.org). The user then begins the download by opening the torrent with a BitTorrent download client. The user

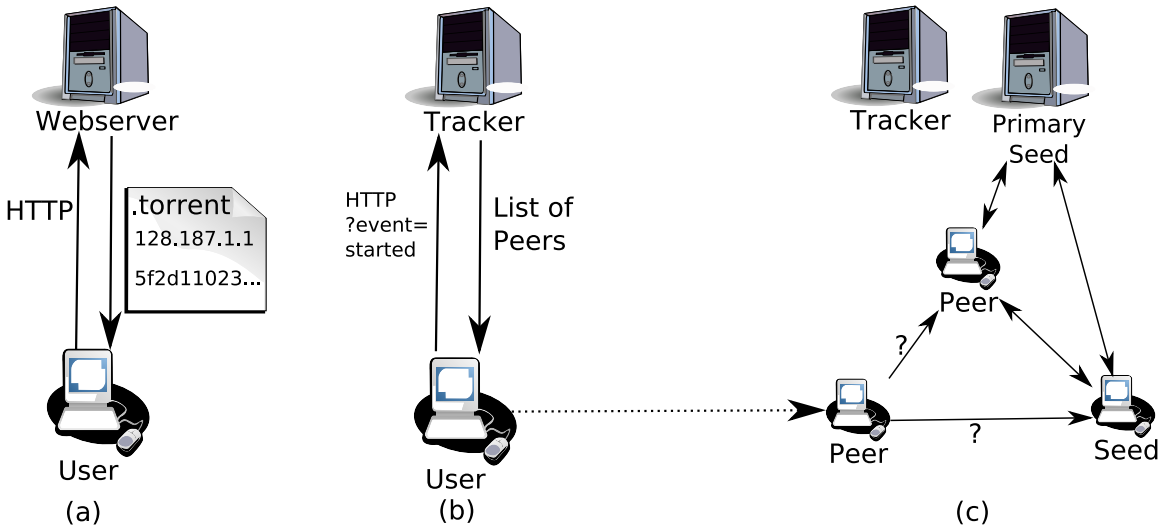


Figure 1.1: Overview of a user's interaction with BitTorrent. (a) The user locates and downloads a torrent file from a web server. (b) The user then contacts the tracker contained in the torrent and receives from it a list of peers currently downloading the file the user wants. (c) The user becomes a peer and begins receiving and exchanging blocks with others.

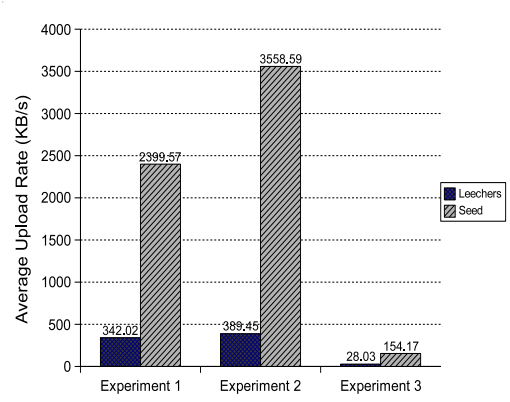
becomes a *peer* (or member) in the downloading community, called a *swarm*. Figure 1.1 illustrates this process.

Two entities, called the *tracker* and the *seed*, assist peers in downloading the file. Trackers maintain information about all peers in the swarm, such as IP addresses and completion percentages. They distribute this information to the peers, especially to ones who are recently arriving to bootstrap them into the downloading network. Seeds are peers that have the entire file and now altruistically provide the file to those that have not finished. A *leecher* is a peer who has not finished downloading and therefore relies on others for the file. Deceptively, a leecher and a *freeloader* are different terms in the BitTorrent literature. A leecher is an active participant who uploads portions of the file it has received to other peers, whereas a freeloader does not upload.

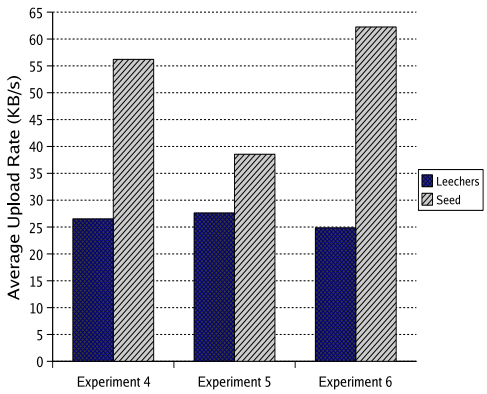
When the user starts a download with the BitTorrent software, the user's peer first contacts the tracker to get a list of other peers who are downloading the same file. The peer then connects to the peers in this list and requests data from them.

The contacted peers will place the new peer in their *unchoke list*, and send data once the peer surfaces to the top of the list. A peer will unchoke another if it is willing to upload data to that peer. A peer orders its unchoke list from the rate at which the peer is able to download from connected peers. Thus, the faster peers give data, the more likely they are to remain unchoked and receive more data. Each peer will pick the top few peers (depending on its bandwidth capacity – typically 2 to 4) and unchoke these peers. To get a starting peer bootstrapped, peers will periodically *optimistically unchoke* a connected peer. This means that one peer will unchoke another for no statistical reason other than to 'try out' an exchange with another. All peers remain in the queue until they are unchoked or find the data they need from another peer.

One of the open problems with BitTorrent is that seeds are commonly overburdened relative to the other peers in the swarm. In a traditional HTTP client-server download, the server can quickly expend all it's upload bandwidth, just given a few clients with a large download bandwidth. With BitTorrent, there is greater opportunity to distribute the load of providing a file among peers, especially seeds. Ideally, seeds should upload little to nothing if block diversity is high and shouldn't upload more than any other peer. However, Figures 1.2(a) and 1.2(b) suggest a large discrepancy between the seed's average upload rate and that of the peers. All of the experiments represented were run with one initial primary seed, which is the seed that is considered here. We see that in most cases, the seed is serving anywhere from 2 to 10 times as much as a peer in the swarm. This difference is most easily seen in Figure 1.2(a), which have the seed and peers configured with an unlimited upload rate and a 5 minute peer lingering time. The non-rate limited experiments have greater differences between upload rates because the seed has a much greater available upload bandwidth on average than the leechers.



(a) Shows three experiments run with standard BitTorrent prior to the time of this thesis. These experiments had the all peers configured with no rate-limit (e.g., no bandwidth capping), a 5 minute lingering time, and a constant peer inter-arrival time.



(b) Shows three experiments run with standard BitTorrent, whose data we present in this thesis. The seed and peers were configured with a 75 kilobyte bandwidth limit, no lingering time, and a constant peer inter-arrival time.

Figure 1.2: Comparisons between the seed's average upload rate and the peers' average upload rate.

A seed can become overburdened if there is poor *block diversity*. Blocks are fixed-sized chunks of the file the seed is providing. The problem is that if a peer runs out of unique blocks that it can request from its neighbors, it will be forced to locate and request needed blocks from a seed. If a peer's connected neighbors all have similar blocks, then that peer is said to have poor block diversity. A peer with poor block diversity might put undue load on the seed because of its inability to find the file elsewhere.

A second reason why the seed might become overburdened is if the seed has a large upload capacity. In this case, it will draw attention from leechers and serve many blocks of the file. When a peer wants to request a block that is held by two or more of it's connected neighbors, it will request that block from the fastest peer. As the main seed's upload capacity typically exceeds that of other peers, the seed will serve significantly more blocks.

The BitTorrent protocol is designed to give priority to minimizing download time for users, while providing incentive to share. Less attention has been paid to

content providers, who would rather spend the least amount of bandwidth to serve the largest audience possible. These two problems mentioned previously raise significant concern for content providers because of the idea that their seed is being used disproportionate to the other users in the system. Currently, the official command line BitTorrent implementation allows the user to specify a fixed upload rate at startup. The GUI version of the download client will allow the user to dynamically adjust the upload rate, albeit this change is user initiated. However, the seed software should be able to make much more informed choices about an ideal upload rate than a user ever could. The software knows which blocks are available to it, what rate the other peers are uploading, and potentially the block diversity of the system.

In an ideal BitTorrent implementation, an even distribution of all blocks would be given out to all peers in the system. The seed would be an equal candidate among all other neighbors because each peer would have maximally unique subset of blocks. Thus, there would never be a need to single out the seed because block diversity would be high. Seeds would also adjust their upload rate according to the diversity of blocks in the system. For example, if there was only one leecher in the system, the seed would upload at a maximum rate because each block it added to the system would help increase block diversity. If there were many leechers with different sections of the file, the seed would upload at a slow rate to encourage peers to share with each other.

In this thesis, we implement a modified BitTorrent seed that attempts to reduce the amount of bandwidth used by the seed while maintaining good performance for the leechers. This modification, called the *scout*, monitors block diversity and reduces load for the seed by only giving out blocks that it considers to be rare. The scout learns about block diversity by talking to its neighbors and by pinging other peers of the swarm to determine their current set of blocks. The scout is fully compatible with the BitTorrent protocol, so that it can be used with existing software.

We use a measurement study to show that the scout is able to decrease the seed's load by an average of 50%. However, we find that, in general, there is a trade-off between decreasing the seed's load and increasing peer download time. There are some notable exceptions to this finding, so that in several experiments we are able to both reduce seed load and keep download times approximately the same. We believe that with further refinement of our methods, and with different workload settings, that we may be able to improve this trade-off to have competitive upload times while still reducing seed bandwidth.

Chapter 2

Related Work

The work by Bharambe et al. [1] is most similar to the work proposed here. The authors create a simulation environment for BitTorrent and use it to compare current BitTorrent algorithms with alternate schemes. They implement an improvement to BitTorrent's local rarest first (LRF) block selection policy, which they call *smart seed*. Traditional LRF block selection works by having a leecher pick the block from a seed which is least replicated among its connected neighbors. The goal is to have peers select blocks which will increase the block diversity of the swarm. Smart seed changes the policy to have the seed decide which block to give out by examining the blocks needed by the leecher and choosing the one the seed has served the least. They show that a smart seed serves less numbers of redundant blocks (i.e., blocks that already exist in the swarm) and thereby makes better use of the seed's upload link.

What is not clear about the research done by Bharambe et al. is how to implement smart seed in the existing BitTorrent software. Since seeds know the subset of blocks that a leecher is in need of, it would be easy for the seed to decide which of those it has served the least. A simple way to implement their smart seed would be to have the leecher request a block as in traditional BitTorrent, but then to have the seed return the block that it has decided on. However, this approach violates the basic requirement that there be an implicit correlation between blocks requested and blocks received [5]. Since the authors did not specify how smart seeding was

implemented in the protocol, it is not obvious how their change could be compatible with live BitTorrent clients outside of a simulation environment.

Another BitTorrent modification seeks to reduce load on the seed by reducing cross-Internet service provider (ISP) bandwidth [2]. Bindal et al. present research showing that ISPs rate-limit P2P traffic because of the quantity and the expense of relaying it to other ISPs. In standard BitTorrent, clients that connect to the tracker are given a list of peers that are selected randomly from among all peers in the swarm. Bindal's tracker modification gives clients a list of other peers that use the same ISP, so that most of the P2P traffic stays within the ISP, which saves outgoing bandwidth for that ISP. Seeds also see a load reduction with this approach because they will serve content primarily to nearby neighbors on the same ISP. While this approach is novel, it doesn't tackle the more general problem of reducing load with the peers that are on the same ISP as the seed.

Slurpie [13] is a similar peer-to-peer bulk data transfer protocol. Slurpie has a policy called "backing off" which attempts to reduce load on the Slurpie server. Backing off works by having each eligible peer contact the server with some probability distribution. A peer becomes eligible once he has need of a piece which is not found among his neighbors. Once a peer is eligible, he then determines if he will contact the server by sampling from the ratio k/n where k is a small constant and n is a guess of the number of nodes in the system. By reducing k, there will be fewer connections to the server, encouraging more peer-to-peer exchanges, while increasing k may increase system performance. To implement such a system in BitTorrent would be quite a challenge. The client software would have to be modified to specially distinguish seeds from other peers and a sample of n would have to be estimated. The tracker would have to be changed to give out such information. These corrections involve major protocol changes and would require all users to upgrade. Also, this correction only limits the number of connections made to the server, not the bandwidth consumed.

Content distributors are most concerned with the amount of bandwidth consumed by the server.

Three studies argue that the rarest first block selection policy works well at promoting block entropy [7, 8, 9]. These studies involve deploying a modified BitTorrent client into a "live" torrent (such as a popular music download or a Linux distribution) and observing the interactions of this client with the other peers in the swarm. Admittedly, there is much to be learned from a single peer in the swarm. However, none of these experiments are in the position to conclude whether or not BitTorrent's rarest first policy really does a good job. Only a study with access to the states of all nodes at all times, such as the study by Bharambe et al., can reveal the answer to that question. The authors of [8] even state that they "guess" rarest first is performing optimally but without a "global" study, it is impossible to draw any conclusions.

This proposal differs from previous research in the the following ways: first, we offer an experimental evaluation of an entire BitTorrent swarm in our control, hosted on the PlanetLab network. Such control merits us the advantage of taking a realtime snapshot of the performace of the swarm and having more accurate measurements of block diversity and seed load. Second, we create a version of BitTorrent that is catered towards content distributors which, to our knowledge, has not been proposed yet. Ours is the first study to consider the amount of bandwidth used by seeds compared with other peers.

Chapter 3

Implemented Solution

The main goal of our solution is to reduce the load on the seed while having as little of an effect upon the leecher download time as possible. We do this by increasing the seed's knowledge about the distribution of blocks in the swarm, which allows it to make wiser decisions about which blocks to distribute. For example, if the seed knows a particular block is held by 20% of the peers, it might choose not to serve this block, in effect forcing peers to get it from each other.

The key to reducing load on a BitTorrent seed is to reduce the number of blocks served which are already available in the swarm. The *block diversity* of a single block is defined as the number of occurrences of that block in the swarm over the total number of peers. Thus, if block b has block diversity of 1.0, every peer is in possession of block b. It is desirable to prevent the seed from serving blocks which have high diversities and to have the seed readily serve blocks which are very rare.

The seed can prevent peers from requesting common blocks by advertising before-hand that it does not have those blocks. During a connection handshake, peers typically exchange lists of blocks which they currently hold. If, instead of advertising the possession of the entire file, the seed advertises only a subset of blocks, it may reduce its load. Ideally, we would like to select the rarest blocks of the swarm to be included in this subset. The selection and size of these subsets is the topic of the remainder of this section. By intelligently choosing the advertised subset, we can promote block diversity and reduce seed load.

3.1 Estimating Block Diversity

In the standard BitTorrent implementation, the seed only knows about the blocks that its directly connected neighbors have. To increase the seed's knowledge about the swarm's block diversity, we implemented an addition to the seed called a *scout*. The scout becomes active once a peer has obtained all the blocks of the file. It runs as a separate thread to periodically ping the peers in the swarm for the blocks they currently hold. With this collected information, the seed can make more intelligent decisions about the blocks it chooses to advertise. The scout's job is simplified since immediately following the connection handshake, peers exchange their list of blocks.

The scout must balance its desire to have an accurate estimate of block diversity with the amount of bandwidth it consumes. The more peers the scout can contact, the better the estimate the seed can make of the swarm's block diversity. However, if the scout runs unchecked, it can consume a significant portion of bandwidth, which would defeat the purpose of the solution. Accordingly, the scout is configured to contact a certain number of peers per minute, at rate r. Our results include comparisons of various values of r and their effectiveness at estimating the true block diversity, as well as the saved bandwidth as compared to the original seed implementation.

The scout keeps a count of how many times it sees a particular block among the peers it contacts. These counts are stored in a map m that maps block number to the occurrences seen. This map helps in computing the subset of blocks to advertise to new connections. We define $C(b)$ as the count of block b from the m map. Also, the scout keeps track of the number of unique peers it has seen so far, which we define as n. By normalizing m by n, we obtain an estimate of the actual distribution of blocks in the swarm. We define $f(b)$ as the normalized count of block b. Our results include comparisons of the scout's normalized m map to the true block diversity in

Item	Description
B	The set of all blocks in the file
i	The IP address of the leecher
p	The peer-id of the leecher
$C(b)$	The count of block b from m
$f(b)$	$C(b)/n$.
m_{inv}	$1 - m$

Table 3.1: Terms

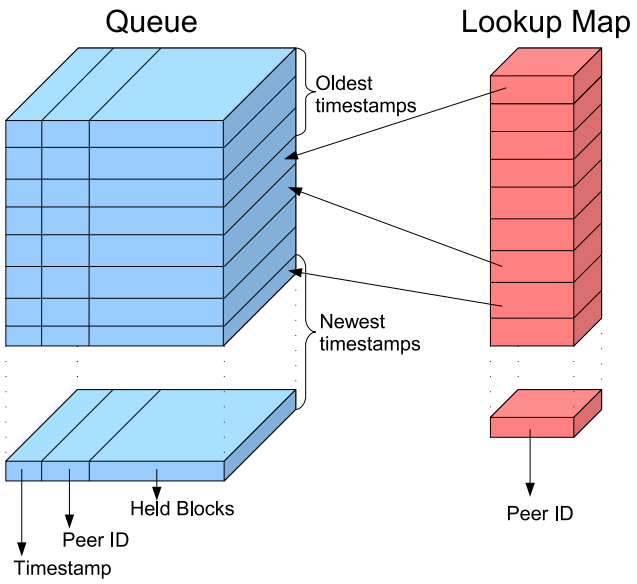


Figure 3.1: The data structure used to drive the *scout*.

the swarm at a given moment. Table 3.1 gives a summary of the terms described here and elsewhere in this section.

In order for the scout to do its job effectively, it needs to expire information from old peers that have left the swarm. To manage this, the scout maintains a data structure to timestamp and identify the blocks it received from other leechers in the swarm. The data structure supports fast lookups by peer id, which is the unique identifier generated by each peer. This structure is illustrated in Figure 3.1. The main component of the data structure is a queue which holds the oldest peers at the front of the queue, and the newest peers at the back. Whenever a new peer is

Item	Description
m	The map of blocks $\rightarrow$ occurrences
s	The subset of advertised blocks
n	The number of unique peers seen so far

Table 3.2: The scout's data structure

contacted, a ScoutPeer object is created and placed at the back (newest arriving) part of the queue. When a peer is contacted whose block list has now been updated, the ScoutPeer object is dequeued and then re-enqueued, so that it ends up at the back of the queue. This way, the queue remains in sorted order by timestamp of the latest block update. The underlying implementation of the queue is done using a linked-list. Table 3.2 summarizes the items used in the data structure.

Peers are dequeued (expired) from the front of the queue once the timestamp is x seconds old. It is conceivable that we would want to expire old data after a certain amount of time since a peer's list of blocks can grow very rapidly. We did not limit the size of the queue. In a real deployment with a large swarm, we might impose a limit to prevent the scout from using large portions of memory.

ScoutPeer objects have three main fields, which are the timestamp of the latest update from a peer, the peer ID of the peer, and the list of blocks that the peer holds. BitTorrent transfers a peer's list of blocks in a BITFIELD message; the list of blocks in the ScoutPeer object is simply the data from the latest BITFIELD message that the seed has received. The ScoutPeer has several other fields, namely the last advertisement made to the peer, the time of first contact with the peer, the time of last contact with the peer, the number of failed connection attempts, whether the peer is currently being contacted, and if the peer is locally connected.

A majority of the interactions with the scout's data structure involve looking up the peer id of a peer. For example, say we have received a list of blocks from a peer, we must now lookup and see if that peer already exists in the queue. With a

linked list this operation is a slow $O(n)$ traversal. To optimize this lookup, we create a hash table whose elements are key-value pairs, called a dictionary in Python or a HashMap in Java. The key is the peer-id and the value is a pointer to the ScoutPeer object with that peer id.

The data from the queue is pooled into the m map described earlier. Therefore, with each change to the queue, m is correspondingly updated.

3.2 Reducing Seed Load

Once the seed is able to estimate block diversity, the next step is to reduce seed load. We do this by modifying the seed to only advertise blocks held by relatively few peers. We define s to be the subset of blocks that the seed will advertise to a peer. There are various methods for computing s that we implement and compare to each other.

One fundamental requirement of these modified algorithms is that once the seed has advertised a certain block to a peer, it can never stop advertising that block to that peer. This prevents any possible incompatibilities with other BitTorrent clients. However, if the peer disconnects from the seed, and later reconnects, the seed is then able (and likely) to advertise a different s. This strategy makes sense because the BITFIELD message is typically only exchanged when peers connect. After connecting, peers use HAVE blocks to notify each other of new block acquisitions.

We describe below all the block advertisement strategies that are implemented in the scout:

3.2.1 All blocks

This selection policy ignores all the work done to maintain the data structure above, and advertises that all blocks are available to all peers. This is the standard BitTorrent seed implementation, and is included for comparison to the other methods. When we

run experiments with the All blocks strategy, we run the standard BitTorrent code, with no modifications besides to output message logs.

3.2.2 Random selection

This policy also ignores the data structure and chooses a random subset of blocks to advertise to each peer. Only the selection of blocks will be randomized. The size of the advertisement, z, is determined from the number of blocks that are held by less than 40% of peers. To build the advertisement we pick blocks randomly until we have selected z blocks. It is possible that this selection policy can cause members of the swarm to be unable to complete downloads, especially in smaller swarms. To make it more robust we include advertisement updates. These updates are found by re-running the algorithm periodically and advertising HAVE messages for any new blocks. Using this strategy we only had one Random experiment get into a state which it could not recover from.

3.2.3 Less than k

This policy takes all blocks b from the normalized m map where $f(b) < k$. We chose a k value of 0.4 for our experiments. This means that the scout will always advertise the rarest 40% of blocks to peers. If two peers connect at times when m is the same, they will be advertised the same s.

3.2.4 Probabilistic s

The probabilistic strategy assigns high probabilities to the rare blocks in the swarm and low probabilities to common blocks. We then choose a uniformly distributed random number for each block, and advertise that block if this number is greater than m. The size of this advertisement is dependent upon the numbers of rare blocks in the swarm.

3.2.5 Shuffle Rare

This strategy works by shuffling all the blocks in the file, B, using the peer id p and the IP address i. This can be done by concatenating p and i and taking the SHA-1 hash. The digest that results becomes the seed to the random number generator, which is used to shuffle all the blocks in the file, B. This process produces a shuffled set of blocks in an order that is consistent given p and i. Next, the shuffled set is truncated at $2z$ blocks and sorted by frequency. What results is an ordered list of blocks, with rarest first, and the size at most $2z$. From this list, the rarest z blocks are chosen for the s advertisement set. The benefits of this strategy are to provide a unique set of blocks to each peer while still giving priority to rare blocks.

3.3 Choosing who to ping

To maintain the scout's data structure and m, peers in the swarm are contacted periodically to obtain their current list of blocks. The scout will ping peers at rate r. The simplest strategy for deciding which peers to request updated blocks from is to always choose the peer which has the oldest data. However, this might not be optimal, for a few reasons. First, there will almost always be peers in the swarm that the scout does not know about. We will be more interested in obtaining a new peer's list of blocks than obtaining an updated list from a known peer, for this will have the greatest affect on our block diversity estimate. Second, some peers might be temporarily unavailable or have reached their maximum connection capacity. If the oldest peer does not respond, then it will be continuously pinged.

To get around the weakness of the simple algorithm and to give preference to new peers, we implement the following solution. Like the simple algorithm we always pull the oldest peer off of the queue to ping, though there are exceptions which will cause it to skip an older peer. These exceptions are if the peer is a locally connected

peer, it would not make sense to ping it. Directly connected peers send updates to each other when they obtain new blocks. Also, we implement a minimum recontact interval, which prevents a peer from being contacted more than once per 60 seconds. Every time a new peer arrives from a tracker advertisement, we place that peer on the oldest end of the queue. While this violates the sort order of the queue temporarily, this gives the desired preference to pinging new peers. After being pinged, it is placed in the proper position in the queue.

Internally, each element in the stack will have a counter. Each time a peer is not able to be contacted, the counter is incremented. Once the counter reaches three, the item is removed, and the peer is assumed to have left the system. If an ICMP host unreachable packet is ever received when trying to contact a peer, the peer's item in the queue is immediately removed.

3.4 Seed-seed interactions

When a modified seed contacts another modified seed in the same swarm, it will get misleading results which will cause it to underestimate the true block diversity of the swarm. In large swarms, it is doubtful that this mis-reporting will have any effect at all, especially when using one of the probabilistic s selection algorithms mentioned above. In small swarms with, for instance, two modified seeds and just a few leechers, the seeds may advertise blocks as available due to a low estimate of block diversity. Certainly the bandwidth required from these seeds would be no worse than than if they had advertised all blocks from the beginning, according to the default implementation.

Another possible side-affect occurs if a seed with the standard BitTorrent implementation resides in a swarm with another seed which uses the modified algorithm. The standard seed will receive more load than the modified seed because the modified seed is now refusing to host some of the blocks of the file. Blocks that may have been

otherwise requested from the modified seed are now seen as residing solely on the standard seed. We leave investigating seed-seed interactions for future work.

Chapter 4

Methodology

Most of the previous work done with BitTorrent observes a live swarm from the point of view of a single peer or tracker log. These approaches have the advantage of capturing the behavior of real user, yet are lacking in scope because they can only view the swarm from one angle. In our experiments, we launch a staged experiment with a tracker and peers we control, giving us the ability to take an omniscient view of the entire swarm. This allows us to view each peer's progress through the download in fine detail. Most importantly, we can evaluate the effectiveness of the scout modifications in comparison to standard BitTorrent.

We chose to run live experiments rather than a simulation of BitTorrent with our scout enhancements. This allows us to have greater confidence that our enhancements will work in practice, and not just in principle. Admittedly, this also introduces a greater amount of variance, and makes our results difficult to repeat. However, by running live experiments, we are able to use the official BitTorrent source code, with all of it's undocumented nuances, rather than a model of the BitTorrent protocol. Finally, BitTorrent runs over TCP which deals with real traffic and network topologies that are difficult to simulate accurately.

We run our experiments in a networking testbed called PlanetLab. PlanetLab [3] is a collection of several hundred computers (to date 693 nodes over 335 organizations) spread out across the world. In exchange for donating one or more

computers and bandwidth to link the machine, an institution is given privileges to run networking experiments. Currently, BYU hosts two such PlanetLab computers.

The computers on PlanetLab are very heterogeneous in terms of bandwidth, processor speed, Internet service providers, and traffic policies. All PlanetLab nodes communicate over the Internet, with no special VPN or encryption standard. Therefore, they are subject to the same routing and queueing algorithms found broadly in today's Internet. Also, several experiments might run simultaneously on a single PlanetLab computer, giving no guarantee for bandwidth, CPU time, or disk space. All these facts add to the reality of the experiments we performed.

To run a BitTorrent swarm, one needs a tracker, one seed, and a file to share. We run a tracker and a seed on a single computer hosted in the Internet Research lab. Given the minuscule amount of bandwidth required by the tracker, it is acceptable to run both services on the same computer.

For our experiments we share a 200 MB file of randomly generated data. We arrived at this size after conducting some preliminary experiments with PlanetLab's overall BitTorrent bandwidth. 200 MB equates roughly to a 6 to 20 hour experiment, depending on upload speeds, allowing ample time for messages to flow from one to another. A smaller file would be less common for BitTorrent and would restrict the amount of interactions between peers. A larger file would require a larger amount of bandwidth from each peer, which might attract negative attention from PlanetLab administrators.

We chose 75 kilobyte per second maximum upload rate for all peers in our swarms. This rate was encouraged by some negative feedback we received when we configured experiments to have no upload rate limit. Also, all leechers were configured to have no lingering time. This meant that immediately upon finishing the download of the file, the leecher left the swarm. We added a flag to the command line BitTorrent client to configure the lingering time.

Message	Parameters	Description
PIECE	*block-number offset length*	Sent to transfer a sub-block of the file from one peer to another.
HAVE	*block-number*	Sent to all connected peers when the sender obtains a block.
BITFIELD	*block-hexstring*	Informs the receiving peer the list of blocks the sender has.
INTERESTED		Sending peer wants one or more of the receiver's blocks.
NOT INTERESTED		Sending peer is not willing or able to upload blocks to the receiver.
REQUEST	*block-number offset length*	Sending peer would like a particular block from the receiver.
CANCEL	*block-number offset length*	Cancels a previous request.
CHOKE		Sending peer will no longer upload to the receiver.
UNCHOKE		Sending peer is now willing to upload blocks to the receiver.

Table 4.1: Brief descriptions of the standard BitTorrent messages

4.1 Logging BitTorrent Messages and Events

To measure the performance of the BitTorrent software and our modifications, we record all BitTorrent messages exchanged between peers in the swarm. The official BitTorrent source code (*www.bittorrent.com*) uses these messages for all peer communication in the swarm. The 10 common BitTorrent messages are the building blocks upon which all BitTorrent-compatible clients communicate. These are listed in Table 4.1 along with the message parameters and a brief description. For example, chunks of the file are transferred between peers using PIECE messages.

Since the BitTorrent client software does not provide a way to display these internal messages, we place modifications in the source code to record them. Each time a message is sent or received, we record the current timestamp, whether it was received or sent, the IP address of the remote peer, and the contents of the message.

In addition to the standard BitTorrent messages, we record several other events that occur during the life of the peer. These events are not additions to the BitTorrent protocol, and are never sent to other peers in the swarm. These additional events

Message	Parameters	Description
O	*clock-drift*	The local computer's offset from the actual time.
TRACKER	*peers-obtained*	The tracker was contacted.
LOCAL CONNECT		A peer connected to the local machine.
LOCAL DISCONNECT		A peer disconnected from the local machine.
SCOUT SUMMARY	*bytes-uploaded bytes-downloaded*	The scout's final bandwidth usage
SCOUT ADDPEER		A peer was added to the scout's queue.
SCOUT REMOVEPEER		A peer was removed from the scout's queue.
SCOUT UPDATEPEER		The scout received an update of a peer's blocks

Table 4.2: Brief descriptions of the additional event messages

help to get a better picture of the performance of the peer and the scout. Table 4.2 gives an overview of these events.

The TRACKER event occurs when the peer contacts the tracker. The reason why the new peer IP addresses don't appear in this event is because they can be found in the following SCOUT ADDPEER events. SCOUT-type events occur inside the scout when the peer queue is modified. The SCOUT SUMMARY is typically sent when the scout is terminating, indicating the amount of bandwidth the scout expended during it's entire lifetime.

The SCOUT ADDPEER and LOCAL CONNECT events are both important because they help differentiate when a peer has connected locally and when a peer was discovered via the tracker. In the former case, both events will occur together, while in the latter case only the SCOUT event will occur. SCOUT REMOVEPEER and LOCAL DISCONNECT events operate in the same way as the add events, occurring when peers leave the scout.

The purpose of the "O" event is to report how far the local computer's clock is from an NTP host's clock. This is always the first entry in the log, and the offset is factored in every instance a timestamp is recorded. The motivation for having this

event and the specifics of how this offset is obtained will be discussed in the following section.

From these message logs we calculate many swarm-based statistics such as the block diversity of the swarm versus time and the number of active peers versus time.

There are a great many other uses for the message logs beyond the scope of this project, including being able to:

- compute the number of active downloaders a peer has,
- determine the percentage of the bandwidth control messages consume,
- take a real-time snapshot of the entire swarm.
- determine the amount of time peers are starved, having no one to download needed blocks from.

Therefore, the message logs will be of value to future research.

4.2 Dealing with clock alignment

In order to do an accurate analysis of the message logs, we need to make sure the clocks are synchronized on all experiment machines. Although NTPd runs on each PlanetLab host, we have found that the clocks on some PlanetLab machines can be more than five minutes away from the actual time. Since normal users are not allowed to change the system clock, we use SNTP, or the Simple Network Time Protocol [11] described in RFC 2030.

SNTP is an interpretation of the NTP standard described in RFC 1305 [10]. SNTP uses the same packet format as NTP and so is able to communicate with NTP hosts. The difference lies in that SNTP uses NTP's stateless operation mode and typically only communicates with one host. SNTP is especially suited for obtaining accurate clock offsets in a simple exchange with an NTP host.

We implement a class which assembles and parses NTP packets. The class also negotiates clock offsets with NTP hosts using the SNTP standard. We probe the NTP host when the BitTorrent client is first run, and get an offset from the local clock to the NTP host's clock. This offset is then factored in when recording any timestamp in the message logs. This method for alignment is acceptable for our purposes since we only need a minute-level accuracy between the message logs.

4.3 PlanetLab deployment

One of the challenges of using PlanetLab is to upload and run software on so many PlanetLab machines. *vxargs* is a simple tool for running commands in parallel. While it could be used for any task that can be processed in parallel, it's main use is in executing commands on multiple remote hosts. We used vxargs to push new source distributions to clients, stop still-running hosts after an experiment, and download message logs.

While vxargs does make the job faster, it must still operate over PlanetLab's SSH connections, which can be slow and unreliable. To make nodes a bit more accessible, we developed a System V service that listens on a specified port for commands. This server, called the BitTorrent server, runs on every PlanetLab host and launches the BitTorrent client when the "start" command is received. The advantage to such a server is that each PlanetLab node's BitTorrent can be started simply and the server can monitor running BitTorrent clients and provide status. Besides start, the BitTorrent server has commands to petition the status of the node, shut down the BitTorrent client, and to stop the server.

4.4 Practical Considerations

The first step in preparing PlanetLab nodes for running BitTorrent experiments is to determine which PlanetLab nodes are in "boot" or production state. PlanetLab comes with a XML remote procedure call (RPC) interface to manage node login credentials, find nodes in boot state, and to add or remove nodes from from PlanetLab accounts. Using this API, we generate a list of the active PlanetLab hosts, which takes only a few seconds.

We then use vxargs to push the latest source distribution. This takes between 20 and 30 minutes with a 60% success rate. We do it a second time for all nodes that failed the first time, which has about a 5% success rate. We then run the series of commands that installs the distribution and sets up needed services, which takes about 45 minutes.

Finally, we send the "start" command to BitTorrent clients to begin the experiment. The amount of time this takes depends upon the workload distribution, though it took about 30 minutes on average. For the experiments that have every node start at the same time, we run the workload generator twice to get any nodes that timed out.

Nodes take from 2 to 24 hours to finish downloading once they have started. There are some that would have taken longer, but once 95% of nodes have downloaded the file, we terminate the experiment. Some nodes have extremely slow connections of less than 2 kilobytes per second. Since they are the exception, we are able to ignore them and have the experiment run in a reasonable amount of time.

Every few days, we copy the experiment logs from the PlanetLab nodes. This process takes from 12 to 18 hours, which includes two passes to retry failed nodes.

All in all, to prepare, run and collect data for an experiment takes about two full days. To save a little time, we are able to only perform some steps every other experiment. We were able to run over 85 experiments in about a one year time frame.

The experiment logs amounted to 381 gigabytes of disk space and consumed over 164 terabytes of BYU's upload bandwidth.

4.5 Computational tools

Once the experiment results have been collected, we use a series of tools to analyze and synchronize the data.

4.5.1 Log parser

This tool brings the data from the logs into an object oriented environment. Each message has an associated class that defines variables such as how many bytes of bandwidth the message consumed. These derived properties help for later computations.

4.5.2 Event log

The event log is a list that is populated while parsing each log for an experiment. The event log is derived from the message logs, recording the start and end time of each host, each time a block was received, and if the node became a seed. We can then sort the event log by time and come up with a picture of the block diversity for the entire swarm.

4.5.3 Analyzer

The experiment analyzer uses the parser and the event log to construct a view of the swarm over time. The analyzer does two important jobs. First, it populates the event log and outputs a file which contains the block diversity over time. Second, it parses the seed's log and outputs a file which contains the seed's view of the block diversity over time. Both of these files are on the same time scale and can easily be

compared and used in other applications. Third, we output peer statistics such as bytes downloaded and uploaded, time started, time ended.

4.6 Workload generator

The workload generator sends the "start" command to hosts according to a specified distribution. The ultimate goal for the workload generator is to provide some variance in the experiments by starting nodes at different times. The *inter-arrival time* t is defined as the time between the starting of one node to the starting of the next.

We use a value of $t = 0$ for all of our experiments, meaning that all nodes start at the same time. On a typical day with PlanetLab we could only obtain between 300 and 400 PlanetLab nodes to perform an experiment. This distribution allowed for maximum peer-peer interaction. We needed as many nodes present in the system at the same time as was possible to test how well the scout could perform. Some distributions, such as simulating a flash-crowd, would not have been realistic with this sample size.

We leave varying the workload distribution to future work. We believe that varying the workload will have more of an impact when there is a larger sample size.

4.7 Experiments

We were able to run 67 experiments that we used for our results. There were an additional 18 experiments that we ran for testing and troubleshooting. These test experiments helped us to determine that varying r, the number of peers pinged per minute, and the block selection algorithms have the greatest effect on our results. We ran experiments with with all combinations of the variables shown below.

- r: 0, 5, 10, 15, 20 peers per minute
- algorithm: Probabilistic, Less than K, Random, Shuffle Rare

We ran 2 to 4 iterations of each possible combination. We also ran 5 iterations using Standard Bittorrent for comparison to the scout-enabled experiments. These iterations were run spread out over the span of the experiments, so that the networking conditions matched the conditions of the scout-experiments.

The r parameter values were determined from our preliminary experiments. The preliminary results showed that having a value of $r > 20$ did not increase the number of peers the seed was able to contact per minute. As we shall discuss in the results, high values of r also increase the load on the seed's CPU, which can be detrimental to overall performance.

Chapter 5

Results

We found that our experiments varied greatly from run to run, even when identical settings were used. There were several reasons for this. First, over the year span of running experiments, networking conditions changed noticeably. It is likely that at several times during the year there were other experiments running simultaneously that also used significant portions of bandwidth on the PlanetLab nodes. Second, we notice that the scout enhancements introduce more variability into the final download time. Since the scout must choose which blocks to advertise, peers might get lucky or unlucky with the advertised set. Third, there is randomness associated with any BitTorrent download. One peer, for example, may be paired early with the seed and achieve high download rates. Another peer might be paired with high-latency node that is around the world or with a node that has smaller network capabilities. All of these facts add to the variability of the data shown in the graphs in this section.

In many ways we ran worst-case experiments. We chose to have no peer lingering time, which meant that we had only one seed at all times. Allowing peers to linger would have decreased the average leecher download time and may have decreased variability. However, our goal was to see how well the scout could function for the entire swarm. Having other seeds helping would have given the main seed fewer chances of generating advertisements, which is one metric we used to gauge its success. Introducing lingering time is an interesting topic for future investigation.

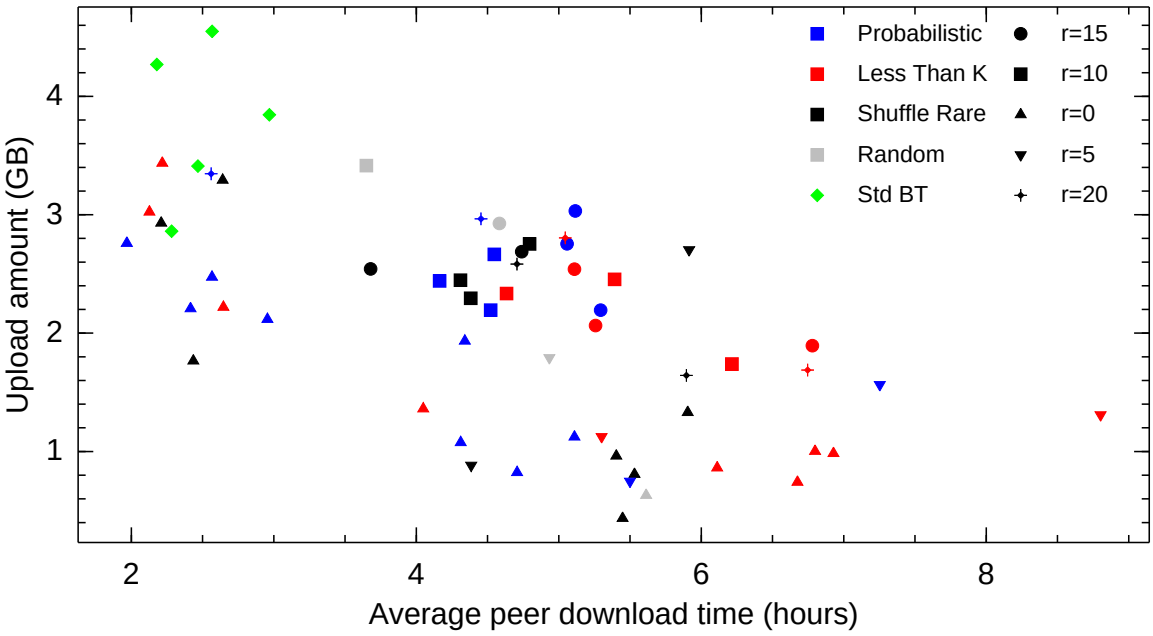


Figure 5.1: Shows the trade-off between the amount uploaded by the seed and the download time of the peers. Each point represents a single experiment. The shapes represent different values of r. The colors represent different block advertisement algorithms.

5.1 Overall performance

The principle goal of the scout modification was to reduce the upload bandwidth on the seed while not severely affecting peer download performance. Figure 5.1 shows the amount uploaded by the seed over the entire experiment versus the average peer download time for each experiment type. Each mark represents a single experiment. The colors represent the different algorithms that were run. The shapes represent the values of r, the number of peers pinged per minute. The standard BitTorrent implementation is shown in green, with r not applicable.

The standard BitTorrent experiments are shown in green in the upper-left hand corner. It is immediately apparent that the scout enhancements did reduce seed load. In most cases, they also increased peer download time, sometimes significantly. There are a number of cases where experiments with $r = 0$ did better than standard BitTorrent, both in terms of upload amount as well as download time. These include

Probabilistic (blue up-facing triangles), Less than K (red up-facing triangles), and Shuffle Rare (black up-facing triangles) all on the left-hand side of the graph. There were many experiments in which the seed uploaded less than 1 GB total, meaning it uploaded the file under 5 times to serve about 400 nodes.

We see that the general trend of Figure 5.1 is a trade-off between upload amount and download time. Typically, the seed must upload more to achieve a faster average download time, though there are some important exceptions. Given this trade-off, it is difficult to say which point on the spectrum is best. To a content-distributor the upload amount may be of first importance, while to the downloader a small download time is the most important. The slope of the trade-off line also depends on peer lingering time; in these results, peers do not linger at all once they finish the download. In this case, the penalty for cutting the seed's uploaded bytes in half is that we increase the average leecher upload time by about 2.5 to 3 times.

Many of the best experiments were configured with $r = 0$, meaning that the scout did not do any active probing of BitTorrent nodes. The scout, in this case, is still advertising only rare blocks, but is basing its understanding of rareness using only feedback from its directly connected neighbors. We noticed this difference midway through our experiments and ran several more $r = 0$ experiments.

We believe there are several reasons why $r = 0$ performed well with our experimental setup. First, the BitTorrent source code has a leecher connect to every peer that is given from the tracker. This means that if the tracker gives 100 peers, the client would attempt to make socket connections to each. The ramifications of this are that many leechers end up being directly connected to the seed. Although only a small handful of these are active connections that transfer data, all these connections send messages to the seed informing it when blocks have been obtained. In our experiments, we have found that as many as 100 leechers have been directly connected to the seed at one time. This means that the seed had a complete and up-to-date

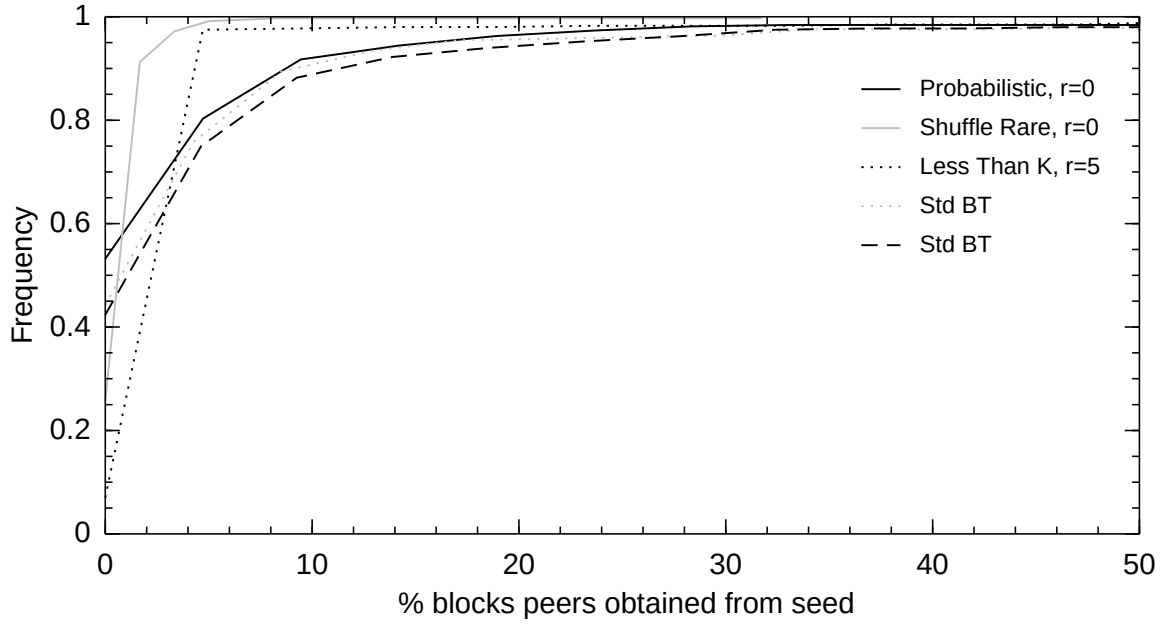


Figure 5.2: Shows the cumulative-density function of the percentage of blocks that were obtained from the seed. The five interesting experiments, as described previously, are shown here.

knowledge of the status of 25% of the 400 peers in our swarm. A probe from the scout is more uncertain than a direct connection, because with a probe the scout only sees a snap-shot of the peer's blocks. The second reason why $r = 0$ may have performed better is that by eliminating probing, we are eliminating some CPU and bandwidth requirements.

We believe that configuring the scout with $r > 0$ may perform better than $r = 0$ in swarms which have more than 2000 nodes. In this case the seed would be directly connected to less than 5% of the peers in the swarm. and would be less able to understand the overall status of the swarm. Probing would become more important to discover the large numbers of unknown peers.

We picked a few interesting experiments from Figure 5.1 to understand why some experiments did not follow the general trend, performing either exceptionally well or exceptionally bad. We chose five such experiments, which will be referenced in the following graphs. Probabilistic $r = 0$ is the left-most experiment on the figure.

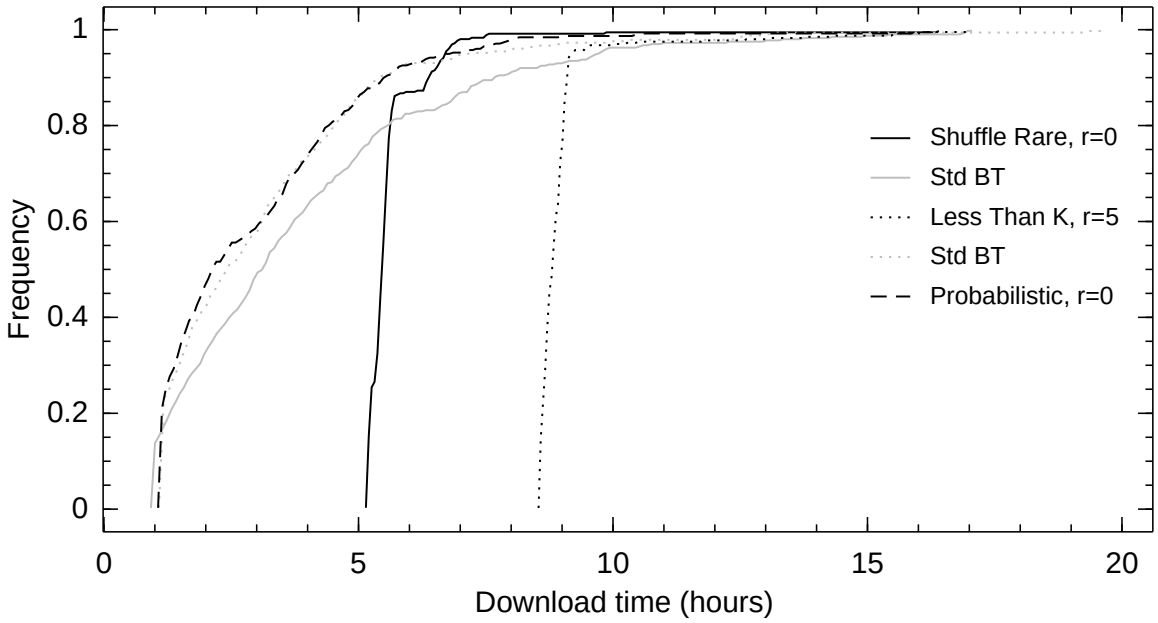


Figure 5.3: Shows the cumulative-density function of the leecher download times. The five interesting experiments, as described previously, are shown here.

Shuffle-rare $r = 0$ is the bottom-most experiment in the figure. Less-than-K $r = 5$ is the right-most experiment in the figure. The two standard BitTorrent experiments are the bottom-most and the right-most standard BitTorrent experiments.

Figure 5.2 shows a cumulative-density function (CDF) of the percentage blocks leechers obtained from the seed. Figure 5.3 shows the associated CDF of the leecher download time. In Shuffle-Rare $r = 0$, very few peers downloaded from the seed, with only 10% of the leechers downloading 5% of their blocks from the seed. This accounts for the small amount of uploaded bytes by the seed. The curves shown by Probabilistic $r = 0$ and the two standard BitTorrent seem to exemplify the ideal balance of seed vs. peer dependence. Experiments with curves that vary from this ideal uploaded less but increased the download time.

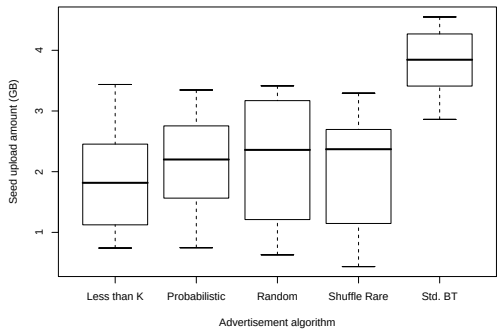
The Less-than-K $r = 5$ experiment shows a larger seed dependence than the others. This experiment is interesting because there were many leechers who were in endgame mode for hours, waiting for one last block to be made available to them.

The seed had failed to advertise this rare block to them, leaving them in an endless wait mode. As best as we can tell, the seed advertised this block to a few peers who were either very slow or had connection issues with other peers. These peers obtained nearly the entire file from the seed and were directly connected to it the entire experiment. Therefore, the seed thought that the block was somewhat available. Over time, more and more peers found themselves in endgame mode, lacking only that last block. Once the slow peers finally finished 8.5 hours later, the peers left, leaving the scout to realize that this rare block was now extremely rare. Fortunately, the scout re-runs the advertisement algorithm periodically and sends HAVE messages for any blocks that show up that were not advertised before. Once these slow peers left, then every peer was advertised the rare block and they finished in close succession. This failure is in part because the Less Than K algorithm advertises a very similar block set to peers that connect in close intervals. This means that even though the seed updates blocks periodically, unless the block-diversity has changed significantly, there might be very few blocks to update. Probabilistic and Random tend to do better than Less Than K because the periodic updates end up advertising blocks that the scout thinks have more diversity, allowing for more block availability.

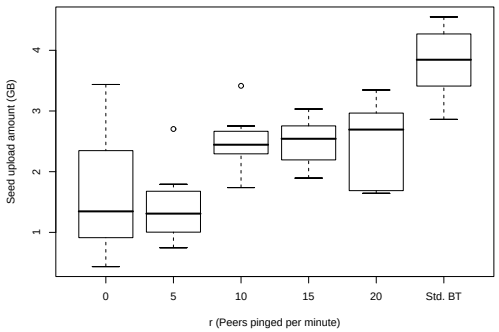
5.2 Reducing seed load

The first objective of the scout modification was to reduce the upload bandwidth on the seed. This was achieved by reducing the number of redundant blocks the seed had to serve. We found that by using the scout algorithms we were able to upload, on average, 50% less bytes to deliver the same content to peers as standard BitTorrent.

The following box plots, such as in Figure 5.4(a), use the standard 5-number summary to produce each box. The five numbers are the lower 25% quartile, median (the 50% quartile), upper 75% quartile, maximum, and minimum values. The 25% quartile is found by selecting all data below the median, and finding the median of



(a) Total seed uploaded bytes per advertisement algorithm. "Std. BT" is the standard BitTorrent algorithm, which advertises all blocks.



(b) Total seed uploaded bytes per setting of r. "Std. BT" represents standard BitTorrent, which has no applicable value of r. It is included for comparison.

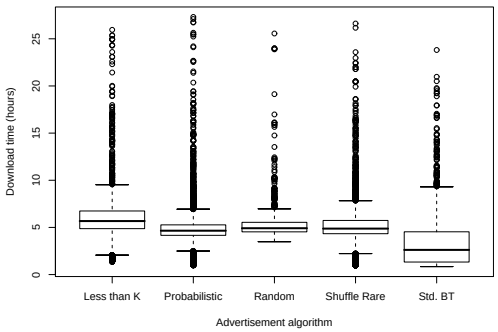
Figure 5.4: Shows the total number of bytes that were uploaded by the seed during an experiment. These are shown per algorithm and values of r.

that data. Similarly, the 75% quartile is found by selecting all data above the median, and finding the median of that data. The whiskers contain all data values within 1.5 box-lengths from either the 75% quartile or the 25% quartile box-ends. Any data values that are outside the whiskers are outliers and are marked with circles.

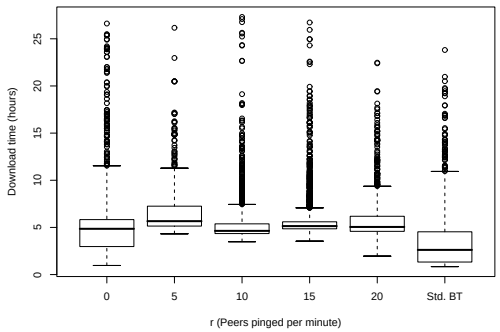
Figures 5.4(a) and 5.4(b) show numbers of uploaded bytes over the swarm for the different settings we varied. It is again confirmed from these plots that the scout reduced the seed load by 50%, on average.

The different scout algorithms were able to achieve about the same bandwidth savings on the seed. The seeds using the Less than K advertisement algorithm generally uploaded less than the others. As we discussed earlier, this is likely because of the lack of randomness in the Less than K algorithm, sometimes leaving leechers without needed blocks.

We see that setting r greater than 5 significantly increases the amount of bytes uploaded by the seed. This phenomena seems counter-intuitive at first, for we supposed that increasing the amount of uploaded bytes should decrease download time. However, since this isn't the case, the increased seed bandwidth must be explained by



(a) Average peer download time per advertisement algorithm. "Std. BT" is the standard BitTorrent algorithm, which advertises all blocks.



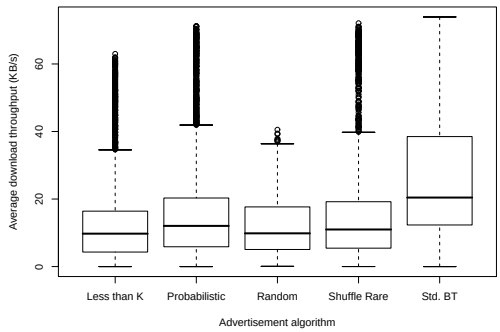
(b) Average peer download time per setting of r. "Std. BT" represents standard BitTorrent, which has no applicable value of r. It is included for comparison.

Figure 5.5: Shows the average time it took for peers to download the 200 MB file. These are shown per advertisement algorithm and value of r.

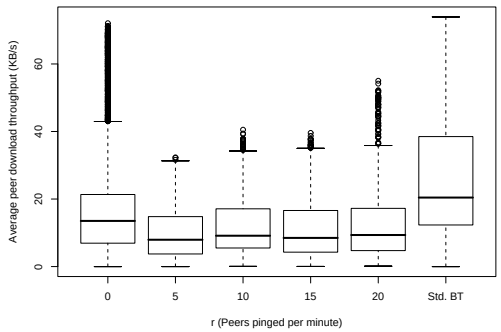
inefficiencies introduced with high values of r. We already mentioned that processor usage is an issue with high values of r. We notice that even for standard BitTorrent the seed's command line server would frequently go to 100% CPU usage during the beginning two hours of the download. The CPU in reference was a 3.2 GHz Pentium IV processor. This initial spike is due to the large number of peers attempting to request blocks from the only source of the file. Over time, the CPU usage decreases as peers are better able to rely upon each other. Placing another service on top of a loaded seed proved to be detrimental to performance.

5.3 Comparing the download times

Figures 5.5(a) and 5.5(b) are companion plots to figures 5.4(a) and 5.4(b). They show the leecher download time varied with the advertisement algorithm and chosen values of r. At first glance, it would appear from the graphs that there was a large amount of variance in the download times. However, considering that there are, on average, 400 downloaders per experiment and that there were about 10 iterations of each value of r, there were about 4,000 data points per box. The flat boxes in the



(a) Average peer download throughput per advertisement algorithm. "Std. BT" is the standard BitTorrent algorithm, which advertises all blocks.



(b) Average peer download throughput per setting of r. "Std. BT" represents standard BitTorrent, which has no applicable value of r. It is included for comparison.

Figure 5.6: Shows the average time it took for peers to download the 200 MB file. These are shown per advertisement algorithm and value of r.

scout experiments with $r > 0$ show that an overwhelming majority had download times right around the average. Although there were several outliers, these were only a very small fraction of the total number of peers.

Since $r > 0$ had so many downloaders close to 5 hours, we see that $r = 0$ performed significantly better in terms of attained download speed than the other scout experiments. 25% of all downloads that used $r = 0$ finished in the same average time as downloads that used standard BitTorrent and simultaneously saved 50% of the seed's bandwidth.

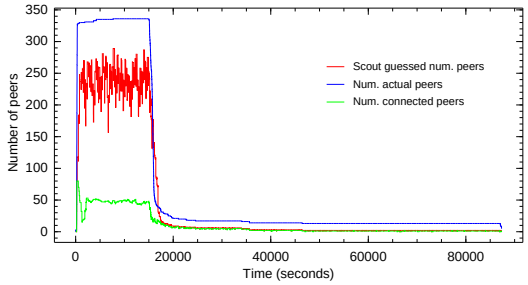
Another way to look at the download times are with the average download throughput per leecher. This is shown in Figures 5.6(a) and 5.6(b). This data is computed using a sliding-window throughput calculation for each peer. A sliding-window calculation has the advantage of eliminating small second-to-second fluctuations in throughput, thus highlighting the averages over time. To do this, we define a window size that is larger than the metric we are plotting, which is bytes per second. Here we picked the window size to be 60 seconds. To calculate the first second, we first fill up a one minute buffer with each message received in that minute and keep track of

how many downloaded bytes the messages represent. The first second's data point is then the total amount of downloaded bytes in the window over the window size. Then, we shift everything out of the window that occurred in the first second, and add any messages that occurred in the 61st second. With this window we compute the second point in the same way as the first, and so forth for the duration of the experiment. Using this method, each data point represents the average over a minute time. The box plots here contain the averages, per peer, of the sliding-window throughput calculation.

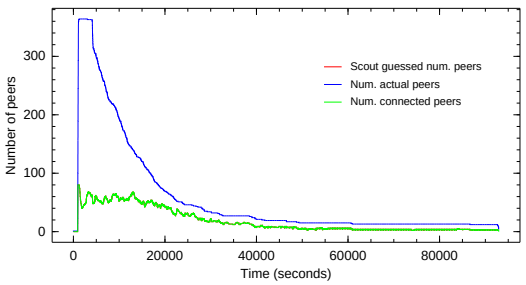
We set the maximum upload rate to 75 kilobytes per second for all peers in the swarm. Since downloaders can download no faster than is given them from others, we expect to see close to this limit in the download graphs. This limit makes sense even in the case where peer A is getting data from peer B and C both at 75 kilobytes per second. Peer A can only provide blocks to B and C at 37.5 kilobytes per second, assuming the load could be split evenly. Over time, either peer B or C will eventually throttle back and we shall see average download rates no faster than 75 kilobytes per second in peer A.

We see that peers in the swarm with $r = 0$ download the most out of any of the scout implementations. This higher download rate means that blocks were more available for peers when they needed them. It is also possible, as we have discussed, that since the seed was able to know about 25% of the swarm, and thus was able to give more accurate advertisements.

According to this graph there were no significant differences in the scout algorithms. However, there was some additional information that we were unable to represent with this graph. The Random and Less than K algorithms had one instance each in which the swarm was locked in search of a single block that the seed would not give out. Since the download times would have been infinite and these experiments were exceptions, they are not included on this graph. Shuffle Rare and Probabilistic



(a) Shows the number of peers in the swarm over time in the Probabilistic $r = 0$ experiment.



(b) Shows the number of peers in the swarm over time in the standard BitTorrent experiment.

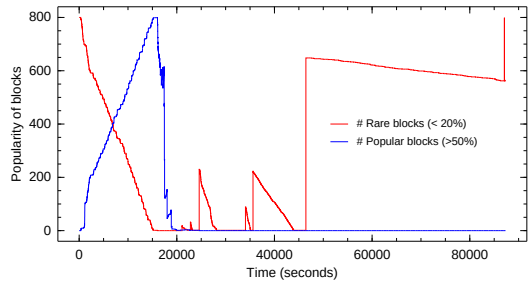
Figure 5.7: Shows the number of peers that were actually in the swarm, estimated by the scout, and directly connected to the seed.

never had these problems, and therefore might be considered less error-prone than the others.

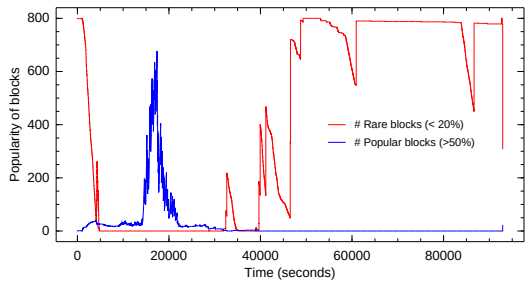
5.4 Increasing Block Diversity

Our third goal was to increase the swarm's overall block diversity. To show this, we iterated through each peer message log to produce a second-by-second breakdown of the blocks held by peers in the swarm. We then produced an identical breakdown from the perspective of the seed. From this generated data we prepared the plots from this section. We chose a standard BitTorrent experiment and a Probabilistic $r = 0$ experiment to compare results. These experiments were average-case experiments and were representative of other experiments. These experiments were run approximately a week apart.

To make good estimations of block diversity, the scout must make good estimations of the number of peers in the swarm. To increase the seed's awareness of other peers we probe peers in the swarm when $r > 0$. Figures 5.7(a) and 5.7(b) show the difference between the amount of peers the seed thought were in the swarm and the number of peers that were actually there. The first series shows the number of peers the seed was aware of, either from direct connections, probing, or from tracker



(a) Shows the numbers of rare and popular blocks in the swarm over time. From the Probabilistic $r = 0$ experiment.



(b) Shows the numbers of rare and popular blocks in the swarm over time. From the standard BitTorrent experiment.

Figure 5.8: Shows the numbers of rare (held by less than 20% of peers) and popular (held by more than 50% of peers) blocks over time.

advertisements. Figure 5.7(b) shows standard BitTorrent which did not do any probing. This means that the first and last series of the graph are equivalent, since it only has a knowledge of its directly connected neighbors.

Probing peers did increase the seed's knowledge of the peers in the swarm. We see that the scout knew about 62% of the peers in the swarm in the active period of the experiment.

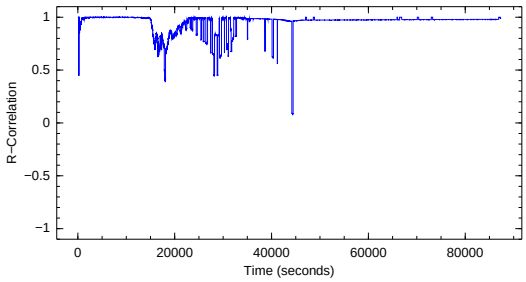
Figures 5.8(a) and 5.8(b) show the numbers of blocks that were held by peers that were rare or popular. Rare is defined as being held by less than 20% of the peers in the swarm at that time. Popular is defined as being held by more than 50% of the peers in the swarm. These graphs show the sum totals of the rare and popular blocks at each moment in time.

The tails of each experiment show that many blocks are rare during the long conclusion of the experiment. Once the majority of average-speed leechers had finished and left, there remained a few very slow or firewalled nodes. Many of these slow nodes had connection problems with our seed and never finished the download. This is why the peak-popularity times are seen when most of the peers are present in the system.

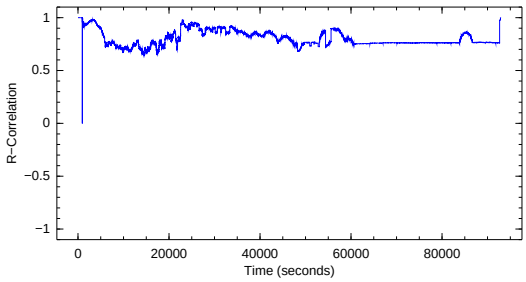
The standard BitTorrent experiment shows a spike of popular blocks during the average end of peer downloads. This is to be expected as the 400 peers approach completion. This spike is much more apparent in the scout experiment and nearly all blocks become common before the sharp termination of most peers. This curve suggests that most peers were waiting on one or two final blocks, and thus were stalled in their finish times. Once those very-rare blocks were available, peers were able to finish quickly. With our experimental setup, it is healthy to see the number of popular blocks stay below a certain threshold. In an ideal swarm, as more and more blocks become common, peers should start finishing the download. After finishing, they leave the swarm which should drop the numbers of common blocks.

This phenomena with too many popular blocks is observed in many scout experiments, and was the reason why many experiments performed slower than standard BitTorrent. If we could identify these few rare blocks before they became an issue of stalling the download, we could decrease the leecher download time significantly. One approach to accomplish this is simply to increase the number of blocks in the advertisement. For Random, Less than K, and Shuffle Rare, we advertise the number of blocks which are held by less than 40% of the peers. To advertise more with these, we can increase the number to 50% or 60%. For Probabilistic, we generate a random number for each block and see if that number is greater than the percentage of peers that hold that block. If we add .10 or .20 to the random number, we could advertise more. Another approach is to more aggressively expire old peer information from the scout queue. With these experiments, we expire peers 20 minutes after their last update to the scout. Perhaps by shrinking this expiration time we could cause the scout to create advertisements from newer information.

To gauge the similarity between the scout's estimate of block diversity and the actual block diversity, we computed the R-correlation between the two data sets. Figures 5.9(a) and 5.9(b) show the computed R-Correlation over time. R-Correlation



(a) R-Correlation data from the Probabilistic $r = 0$ experiment.



(b) R-Correlation data from the standard BitTorrent experiment.

Figure 5.9: Shows the R-Correlation between the estimated block diversity by the scout and the actual block diversity over time.

values range from 1.0 to -1.0, where 1.0 indicates exact correlation, 0.0 indicates no correlation, and -1.0 indicates an inverted correlation between two data sets.

Both experiments show a correlation to the actual block diversity found in the swarm. We see that 5.9(a) stays near to perfect correlation during the active period of the experiment, indicating that by doing extra probing we are increasing our knowledge of the swarm. This finding also gives us confidence that the scout is functioning properly according to our proposed design. This finding also reinforces the idea that the more peers there are in the swarm, the more beneficial probing will be. It is surprising to see that standard BitTorrent did as well as it did without any probing. We see that for a swarm of about 350 leechers, we can do well enough without probing, both with the $r = 0$ scout and standard BitTorrent.

Chapter 6

Conclusion

This work shows that it is possible to reduce the seed's upload bandwidth by modifying the seed to selectively advertise blocks based on their popularity. In our experiments we are able to reduce seed bandwidth anywhere from 0 to 90%, but this reduction comes at a trade-off to the final leecher download time. On average, our experiments save about 50% of the seed's upload bandwidth and extend the leecher download time by 2.5 to 3 times. The Probabilistic block advertisement algorithm performs best, even outperforming standard BitTorrent download times in some cases. Because the size of our swarms is relatively small, there is no apparent benefit to pinging additional peers to learn more about block diversity. Using the seed's current neighbors provides a good enough view of the swarm.

It is worth focusing on the several cases which uploaded less and kept leecher download times very close to standard BitTorrent. These experiments show that in some fortunate cases the scout was able to refuse advertisement of blocks that did not affect download times. The difficulty is in deciding which blocks to advertise without impeding peer download time. The fact that many of our experiments increased download time illustrates that this is a difficult problem.

There are a number of areas for future work. First, it is apparent that we chose settings that were overly conservative of seed bandwidth. We suppose that no leecher in this swarm would feel that doubling its download time would be acceptable for halving the seed's expended bandwidth. It might have been better for us to have

focused on reducing bandwidth 10 to 30%. This could have been achieved in the Probabilistic algorithm by assigning higher probabilities of selection to each block. In the Less than K, we would have chosen a k greater than .4. Random and Shuffle Rare could have advertised more by increasing z, the size of advertisement.

Another problem with the current design is that the scout algorithms ignore the blocks a peer holds currently. For example, if a peer connects and has 95% of the file, the same algorithm would be run for a peer that connected and only had 5% of the file. The chances are high that the peer that holds 95% is going to be advertised no new blocks that it doesn't already hold. By contributing a few extra blocks the seed could prevent this peer from waiting, perhaps several minutes, for the last remaining blocks to come available from a leecher. To get around this, the scout algorithms could be modified to only consider advertising the blocks that the peer does not have. For example, the Less than K algorithm would choose the rarest K blocks among those that the peer does not currently have. The Probabilistic algorithm would run with higher probabilities for the blocks the peers doesn't have, and so forth. We believe that this approach would prevent peers from starving as often as they did in our swarms.

We have shown that while the seed can predict the block diversity of the swarm with great accuracy, it is still not allowing the seed to identify blocks which do not affect download times. One reason this might be is that we are looking at *global* block diversity instead of *local* block diversity. Local block diversity is the distribution of blocks that are held by the peer and its directly connected neighbors. To understand local block diversity, the seed would have to be made aware of the connection tree of the swarm. We have not been able to come up with a way for the scout to obtain this information without a change to the BitTorrent protocol. Assuming that a change is possible without breaking existing clients, we could introduce a message that is exchanged on handshake. This message would contain a list of connected peers

ordered by the connection's activity. Such a change is relatively simple in terms of development, but it is questionable in terms of compatibility. With this new message, the seed could then construct connection trees and advertise blocks that were rare to that peer.

Running a simulation of BitTorrent with the scout would get around many of the limitations we encountered in Planet-Lab. Planet-Lab has the disadvantage of only allowing swarms of 300 to 400 peers, which is insufficient to simulate the true-to-life topologies of a popular BitTorrent swarm. We were unable to test workloads such as a flash-crowd. Also, we were very limited by the amount of experiments we could run and the variables we could vary because of the long turn-around time of Planet-Lab. A simulation could allow us to examine whether pinging additional peers is useful for larger swarms, and would also allow us to test advertisement algorithms under more controlled conditions.

Finally, there is an enormous amount of information contained within the many message logs we collected. Further analysis of these logs could give us insight into local block diversity of individual peers over time. We could determine how many peers starved during the experiment, and how long they starved. It would be interesting to see if any of their directly connected neighbors had blocks they needed. If the blocks these peers were waiting for were not being advertised from the seed then we could explore why and what went wrong. In the cases of the experiments that reduced seed load without increased leecher download time, it would be interesting to construct a full chronology for the experiment. All this analysis is complicated, for it involves aligning all the messages in the log (i.e., pairing a sent message with the received message in another log). With 400 message logs we ran into memory problems of trying to store enough logs in memory to do the alignment properly. We would need to devote resources into solving the alignment problems before we could approach these issues.

Bibliography

[1] A. R. Bharambe, C. Herley, and V. N. Padmanabhan. Analyzing and improving BitTorrent performance. Technical Report MSR-TR-2005-03, Microsoft Research, March 2005.

[2] R. Bindal, P. Cao, W. Chan, J. Medval, G. Suwala, T. Bates, and A. Zhang. Improving traffic locality in BitTorrent via biased neighbor selection. In *International Conference on Distributed Computing Systems (ICDCS)*, February 2006.

[3] B. N. Chun, D. E. Culler, T. Roscoe, A. C. Bavier, L. L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *Computer Communication Review*, 33(3):3–12, 2003.

[4] B. Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, 2003.

[5] B. Cohen. *The BitTorrent Protocol*. bittorrent.org, 2006.

[6] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 17:6–16, November 2003.

[7] M. Izal, G. Urvoy-Keller, E. W. Biersack, P. Felber, A. A. Hamra, and L. Garcés-Erice. Dissecting bittorrent: Five months in a torrent's lifetime. In *Passive and Active Network Measurement (PAM)*, volume 3015, pages 1–11. Springer, April 2004.

[8] A. Legout, G. Urvoy-Keller, and P. Michiardi. Understanding Bittorrent: An experimental perspective. Technical Report INRIA-00000156, INRIA, November 2005.

[9] A. Legout, G. Urvoy-Keller, and P. Michiardi. Rarest first and choke algorithms are enough. Technical Report INRIA-00001111, INRIA, February 2006.

[10] D. Mills. RFC1305: Network Time Protocol (Version 3) Specification, Implementation. *Internet RFCs*, 1992.

[11] D. Mills. RFC2030: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI. *Internet RFCs*, 1996.

[12] A. Parker. The true picture of peer-to-peer filesharing. In *Panel presentation, IEEE 10th Int'l. Workshop on Web Content Caching and Distribution*. CacheLogic, July 2005.

[13] R. Sherwood and R. Braud. Slurpie: A cooperative bulk data transfer protocol. In *INFOCOM*, 2004.