LeechLock: Preventing Selfish Clients in the BitTorrent Protocol

John Locke
john.b.locke@gmail.com
University of Maryland, College Park

Russell Moriarty
rmoriarty@gmail.com
University of Maryland, College Park

May 1, 2011

Abstract

Although the BitTorrent protocol incentivizes sharing by increasing a peer's download rate, tools have demonstrated that the tit-for-tat mechanism alone is insufficient to prevent peers from downloading content without reciprocation. We propose an enhancement to the BitTorrent protocol, called Leechlock, which uses globally shared tables to track peers' ratings over time within the swarm. The selfish peers who contribute sufficiently little to the swarm will be marked as non-contributors, and denied further requests for pieces. We show that with LeechLock enabled in a BitTorrent swarm, selfish peers are able to download only a small amount of data before they are locked out of the swarm with 100% accuracy. Finally, we run simulation tests to explore the effects of additional swarm parameters in a controlled fashion.

1 Introduction

The rise in popularity of the BitTorrent protocol in recent years has led to further advances in the optimization of content distribution. However, its popularity has also attracted selfish users interested in downloading as much as possible while uploading as little as possible. By leeching content without reciprocating with their own upload bandwidth, these users violate one of the original goals of the protocol. Such tools as BitTyrant [1] and BitThief [2] have been successful in strategically manipulating BitTorrent.

One issue that must be addressed in any peer-to-peer file sharing protocol is the development of a bootstrapping mechanism that simultaneously encourages new members to join a swarm and prevents newcomers from freeloading. Obviously, when a leech first joins a swarm, he has no data to share. It is inevitable that a peer will have to supply this user with some small amount of data without hope of reciprocation, otherwise the user has no means to prove he is well-intentioned. This is the reason that this vulnerability has been so tricky to solve [3] [12]. If a user must be allowed to receive data without uploading, then he is always able to receive some amount of data without uploading a single block [3].

One proposed solution to the bootstrapping problem is to encrypt a block of data to ensure a newcomer forwards the data before gaining access to it [4]. Unfortunately, this approach demands increased overhead and a imposes a requirement that all members of a swarm have the same client software implementing the encrypt-and-forward scheme.

Levin et al. [5] explore the feasibility of using an alternative incentive mechanism, called PropShare, whose intent is to ensure that users are motivated to upload their own data in order to receive more. But PropShare is plagued by a similar problem to the original BitTorrent protocol; its bootstrapping mechanism is susceptible to *freeloaders*, or peers who leech data with no intent to provide upload bandwidth. As much as 20% of PropShare's bandwidth is dedicated to researching new neighbors by optimistically uploading blocks to peers. A knowledgeable user could exploit this vulnerability by designing a client that uses only this research bandwidth, hopping from peer to peer until its download is complete. However, PropShare does make it significantly more difficult for BitTyrant-type clients to siphon bandwidth, and is certainly an improvement to the BitTorrent protocol. We propose a BitTorrent enhancement that goes further to lock out selfish peers.

We propose a wire-protocol compatible modification of BitTorrent, called LeechLock, to further encourage sharing without the downsides of an encryption mechanism via a peer review mechanism by which clients share information about potential freeloaders. Each client maintains a table of peers with whom it has participated in uploading and/or downloading. For each peer, the client maintains a rating of that peer based on its return on investment (i.e., how much content the client has downloaded from the peer vs. how much it has uploaded). Once the rating of a peer drops below a threshold, peers will refuse to provide data to the leech. By sharing these rating tables with other clients, members of the swarm can learn about freeloaders, including ones that they personally haven't dealt with before, and avoid uploading blocks to them until they start cooperating. While the ability of a client to share its ratings table requires a modification of the protocol, our approach does not strictly require all swarm members to share, and hav-

ing at least a fraction of the swarm implementing the table sharing mechanism boosts the overall swarm's resistance to freeloading.

We evaluate the performance of LeechLock both in terms of its efficiency and in terms of its susceptibility to freeloading. First, we modify an existing popular BitTorrent implementation, the Rasterbar libtorrent library [6], to explore the feasibility of such a modification in a real-world application. Further, we develop an easily modified protocol similar in function to BitTorrent, BitSim, which optionally includes a separate prototype of LeechLock to further analyze LeechLock behavior in an environment free from common unpredictable variables. The libtorrent implementation determines the ability for a LeechLock-modified libtorrent client to properly weed out the selfish peers. Our control BitSim experiment consists of simulating a fixed number of malicious leeches among a swarm of standard leeches, and then measuring the time required for each peer to obtain the entire file, then the same experiment is then performed in the BitSim simulation with LeechLock turned on. We show that when clients do not share their tables, as is the case with standard BitTorrent, freeloaders are able to download the entire file from the swarm at a rate faster than the honest clients. However, when these malicious leeches operate within the LeechLock prototype, they are only able to download the entire file when the file size is sufficiently small, and at a slightly slower rate. This shows that the table sharing mechanism of LeechLock is effective in locking out freeloading clients, and will likely be a valuable addition to BitTorrent.

In this paper, the following definitions are used for the purposes of introducing LeechLock. A *swarm* is the set of all peers sharing the same file. *Seeders* or *seeds* are peers that join the swarm with all pieces. *Leeches*, *leechers*, *peers*, *nodes*, and *clients* are used interchangeably to refer to any type of participant in the swarm. *Freeloaders* and *selfish* peers are peers who do not contribute to the swarm. A *tracker* is an HTTP agent that tracks all peers currently in the swarm and responds to requests for peers.

The rest of the paper is organized as follows: In Section 2 we give a more detailed account of attempts to game the BitTorrent protocol, and other proposals for preventing them. In Section 3 we present the motivation for developing LeechLock, and contrast it with some of the proposals given in Section 2. In Section 4 we present the basics of the LeechLock enhancement, and in Section 5 demonstrate why it should be resistant to certain types of strategic gaming. Section 6 details our libtorrent LeechLock implementation as well as the results of the experiment. Section 7 details our BitSim experimental simulation and its results. Potential avenues for future work are given in Section 8, and concluding remarks are given in Section 9.

2 Related Work

Some peers, particularly those that are new to a swarm, necessarily must upload a block to a node with which it does not have a previous relationship. Traditional BitTorrent refers to this as *optimistic unchoking* [7], and PropShare refers to this as *research bandwidth* [5]. In both cases, the sending node has no way of knowing whether the receiving node plans to play by the rules and perform uploads or if it is attempting to freeload.

Freeloading clients such as BitThief [2] have been successful in downloading files from swarms without providing any upload bandwidth in return. Such clients do so by requesting to be optimistically unchoked by many members of the swarm. Once a client uploads a block to the newcomer and then learns that it is not cooperating, it can shun the freeloader and never upload to him again. However, if the number of clients in the swarm is large enough, a freeloader may request only a limited number of blocks from each client (depending on the swarm size and total file size) and it will be able to download the complete file without ever uploading a block.

2.1 Super-seeding

Super-seeding, first implemented in the BitTornado client [8], appears to solve the bootstrapping problem by forcing the newcomer to prove its worth before it receives anything more than the original block. A newcomer is required to prove it is not a freeloader by uploading the block it just received from the unchoker to some other member of the swarm. Once the original unchoker observes that some other client has that block, it will then allow itself to upload another block to the newcomer.

However super-seeding wasn't actually designed to solve the bootstrapping problem. It is meant as a performance optimization for the initial seeder, particularly when there is a single seed, allowing the initial seeder to upload fewer blocks before another client obtains the entire file and thus is able to act as another seeder. The lead developer of BitTornado has even stated that super-seeding mode is "not recommended for general use" [8], and studies have shown that the benefits of super-seeding vary widely based on swarm characteristics [9].

2.2 Encrypt-and-forward

Another approach for solving the bootstrapping problem is to have the optimistic unchoker encrypt the block before uploading it to the newcomer. After confirming that the newcomer has uploaded the encrypted block to one or more clients of its choosing, the unchoker sends out the key to decrypt the block. This idea was briefly sketched in a section of PropShare [5]. While the theory behind this idea shows some promise, this mechanism has not

yet been implemented or tested on BitTorrent and therefore the real-world feasibility cannot yet be confirmed.

2.3 PeerReview

PeerReview [10] is a system designed to provide accountability in distributed systems. By maintaining a secure, tamper-evident log of all inter-node messages, PeerReview is able to identify faulty or misbehaving nodes. It does so by comparing the observed message sequence with a reference implementation of the protocol. If the logs show that a node produced a message that is not generated by the reference implementation when the same sequence of messages are sent to that node, the node is exposed and marked as faulty.

PeerReview has been applied to three problems [10]: an overlay multicast system, a network file system, and a peer-to-peer email system. While Haeberlen et al. proved PeerReview to be effective in enforcing protocol conformance in these three applications, it seems less likely to be useful for the BitTorrent application. The quality required by the above three applications in order to apply PeerReview is the ability to model the protocol as a deterministic finite state machine. This quality is not possessed by a BitTorrent swarm of clients with different software implementations. This would also stifle innovation in designing new choking/unchoking algorithms since any modification of the standard would result in a client being labeled as faulty. Further, any algorithm that relies on randomness would be unacceptable for PeerReview. Even in regards to the network file system application, the authors state that the kernel had to be patched in order to remove some small amount of randomness in the block allocation policy.

While the idea of PeerReview is interesting, it is clear that the exact mechanism of protocol enforcement is not practical for preventing selfish clients in BitTorrent. But rather than judging clients based on their strict adherence to a protocol, it is desirable to judge clients based on their upload bandwidth contribution.

2.4 *k*-tit-for-tat

Jun and Ahamad [12] propose eliminating optimistic unchoking completely, and replacing it with a *k*-tit-for-tat mechanism. Under this scheme, a client continues uploading to a peer until the deficit with that peer (i.e., number of blocks it has uploaded to that peer minus the number it has downloaded) exceeds k, where k is some fixed number, called the niceness constant.

While this scheme made it tougher for freeloaders to download the complete file from the swarm, removing the optimistic unchoke in this manner also increased download times. Clearly it would be more desirable to keep the optimistic unchoke for the sake of faster download times, while still being tough on freeloaders.

3 Motivation

As shown by the multitude of proposals, ensuring that peers within a swarm are cooperating can be difficult. Specifically, how can one design a scheme that encourages clients to upload blocks to newcomers while simultaneously preventing newcomers from freeloading?

As Cohen [11] points out, the choking/unchoking algorithm is not part of the BitTorrent protocol, but it is important for good performance. However Cohen goes on to present a tit-for-tat model of BitTorrent, and fails to address the problem of freeloaders that rely solely on peers' optimistic unchoke slots.

The encrypt-and-forward scheme [4] seems like an excellent scheme for simultaneously preventing freeloaders and putting newcomers to work immediately. However, block encryption is CPU-intensive in nature and the necessitates key management. It also requires a vast majority of the swarm to have it implemented, otherwise the newcomer might not know enough other clients to which it could forward the encrypted block, and thus prove its good intent. It would be preferable to have a scheme that would at least be somewhat effective at preventing selfish leeches, even if less than a vast majority of users have it implemented.

Even the *k*-tit-for-tat scheme, which proved to be tough on freeloaders, does not take advantage of the collective knowledge of the swarm: after client x wastes his upload bandwidth uploading k blocks to a freeloader, client y is not informed of the freeloader's ill-intent, and may upload another k blocks to it.

One could imagine the swarm maintaining a blacklist of the non-reciprocating clients so that each client knows those who do not reciprocate, and therefore do not deserve any data. The problem with this approach is that any node can blacklist any other node unfairly. Any scheme that involves blacklisting must ensure that there is no incentive for rogue clients to spread false negative ratings of other peers, and that any one individual client cannot blacklist a peer by itself. Rather, an approach whereby information about a peer is built-up over an appropriate duration of time and is based on information from multiple clients.

Any solution to the bootstrapping problem should be resistant to a Sybil attack, unlike super-seeding and *k*-tit-for-tat. It is also desirable to keep the added overhead to a minimum, implying that the encrypt-and-forward scheme is insufficient.

On the other hand, LeechLock will still function if other members of the swarm are running standard BitTorrent. Further, the higher the number of LeechLock clients present in a swarm, the more resistant the swarm will be to uncooperative peers.

4 LeechLock

The fundamental idea behind LeechLock is that members of a swarm should share information with each other in order to identify and shun freeloaders. We propose a peer rating mechanism, inspired by popular community-driven websites.

First we describe how an individual client maintains its peer ratings, and then we explain how nodes collaborate to share their ratings with other peers. The third subsection describes the bootstrapping mechanism, which allows newcomers to build up their ratings quickly and thus become integrated into the swarm.

4.1 The ratings table

As previously mentioned, Jun and Ahamad [12] propose requiring each client to maintain a table of peers with which it has dealt, along with each peer's deficit to the client: number of blocks uploaded to that peer minus the number downloaded from that peer. LeechLock's tables may be viewed as a generalization of these deficit tables.

One problem with the deficit tables is that they do not take into account the long history a client might have with a given peer. In other words, if a client has downloaded a million blocks from a peer but has uploaded a million plus k, that peer will be cut-off from future dealings until it makes up the deficit. This is clearly undesirable, as a rating for a given peer should take into account the client's history with that peer.

Under Jun and Ahamad's scheme, client i maintains, for every other peer j, the number of blocks uploaded to peer j, b_{ij}, and the number of blocks downloaded from peer j, b_{ji}. Rather than maintaining these running totals for all time, we propose that client i compute these values over the course of a 5 second interval, and then update its rating of j as follows:

$$rating[j] \leftarrow \alpha \times rating[j] + (1-\alpha)\frac{b_{ji}}{b_{ji}+b_{ij}}$$

$$timestamp[j] \leftarrow time$$

The low-pass filter constant α is proposed to be 0.9. If no blocks were sent to or received from j over the course of the 5-second interval, then no update occurs (neither $rating[j]$ nor $timestamp[j]$ is updated). After a specified *maxtime* elapses during which $rating[j]$ is never updated, j's rating is considered stale and removed from the ratings table.

Each client maintains a table of other nodes of the swarm that it is aware of, along with a rating indicating how cooperative each one has been in the past. It also maintains a timestamp for each node's rating so that it can be aged off or updated appropriately. When the client uploads to a node, it lowers that node's rating, and when the client successfully downloads from a node, it raises that node's rating. The ratings are updated with new measurements using a low-pass filter, allowing newer information about uploading/downloading to be given more weight than older information.

We had considered the possibility of factoring incoming peer requests into the calculation of ratings, for example deducting points from a peer each time it requests data. This seems reasonable, since cooperative peer requests would presumably be outweighed by reciprocated data, and selfish peers would solely request data and become locked out more quickly. However, in practice we found that this process led to false positive locks on cooperative leeches due to some quirks in the BitTorrent protocol. In particular, we found that some peers would only request pieces from a subset of peers, never giving the cooperative peers a chance to prove their positive intentions. This would lead to a gradual decline in the peer's rating, as it continually requested data from peers, and never provided any data to achieve positive points. Fortunately, we found that solely deducting points from outgoing peer requests to be adequate in real-world tests.

4.2 Sharing the tables

Up to this point, each individual client is maintaining its table of peer ratings based solely on its own interactions with other peers. While this might be somewhat beneficial on its own, it doesn't solve the problem of a malicious peer exhausting the generosity of a leech, then quickly moving on to another. The real advantage of LeechLock is revealed when clients collaborate and share their ratings tables with each other. This allows members of the swarm to learn from each others' experiences. We envision two possible ways to implement the table sharing mechanism: tracker-based and peer-to-peer.

4.2.1 Tracker-based table sharing

This method of table sharing is the simplest and most natural way for users to share their ratings of other peers. A central authority (the tracker) maintains a universal ratings table that averages over the inputs from all peers that have dealt with that client before, and maintains this composite rating for each client.

Whenever a client wants to know the rating of another client, it simply queries the tracker, and then it can make a decision about whether to upload to that client or not. An alternative implementation, which is implemented in our LeechLock prototype, would allow each client download the entire table from the tracker at regular intervals so that the query could be performed locally. In either case, at each receipt of a peer i's rating sent by peer j, peer i will be updated as follows:

$$rating[i] \leftarrow rating[i] + \frac{(rating[i] - rating_j[i]) \times rating[j]}{max(rating)}$$

Here an update to a given peer's rating is only given as much weight as that peer's own rating within the swarm.

Peers with the highest ratings have the most influence on other peers, while those who have low ratings are prevented from "badmouthing" potentially good peers.

The main advantage of tracker-based table sharing is that the central location ensures that every member of the swarm has the most complete and up-to-date information available. Peers may simply check-in with the tracker whenever they desire the most up-to-date ratings for all peers in the swarm. From a simplicity standpoint, tracker-based sharing is also easier to implement, and less susceptible to bugs.

The main disadvantage of this method is that it requires new tracker software to be deployed, although the tracker responses delivered by such an updated tracker would be compatible with existing BitTorrent clients. Additionally, a tracker-based table sharing implementation requires clients to place their trust in the tracker. We contend that a certain degree of trust is already required, as the tracker is responsible for maintaining member nodes in the swarm. However we understand that it may be considered a step in the wrong direction for a peer-to-peer system to move added responsibility toward a central location such as the tracker.

4.2.2 Peer-to-peer table sharing

This method of table sharing does not rely on a central authority to maintain the ratings table. Rather, each client will have a slightly different ratings table, which is formed by taking into account its own experiences with other peers as well as the ratings tables it receives from other clients. In this way, each client averages its own experience with that of others to form its own individual view of reality.

This has the obvious disadvantage of possibly missing the complete picture, since each client forms its ratings based solely on the information from clients from which it has received tables. Information can still propagate indirectly through the swarm, but it is not as efficient as having a central authority maintaining the universal ratings table.

The advantage of this method is that it does not require putting any extra trust into the trackers and it does not require any modification of the tracker software. Even though it is not completely wire protocol compatible, it does allow a heterogeneous swarm of clients. Clients that have the peer-to-peer table sharing implemented will collaborate amongst themselves to share their ratings, and depending on the size of the swarm this might still be enough to deter freeloaders.

The algorithm for updating a client's table based on receiving a table from one of its peers is based on a few principles. First, an individual client's own experience with a particular node should be weighed more heavily than any information it receives from other peers' tables. Second, a client should give higher precedence to a table it receives from a peer from which it has downloaded a substantial number of blocks.

Based on these principles, we arrive at the following table updating scheme. When client i receives a table from peer j, for each node k in that table, client i updates its rating of k by making use of a weight w computed as follows:

$$w \leftarrow \frac{time - timestamp[k]}{maxtime} \times \frac{b_{ji}}{\sum_m b_{mi}}$$

$$rating[k] \leftarrow (1-w)rating[k] + w \times rating_j[k]$$

$$timestamp[k] \leftarrow time$$

This gives greater weight to newer information about client k, and also ensures that j's rating is given weight proportional to the amount that i has downloaded from j. Since j cannot negatively report k for free, this prevents a node from giving negative reviews (or undeserved positive reviews in the case of a Sybil attack) of other clients without contributing any upload bandwidth of its own.

Recent improvements in the BitTorrent protocol have led to such popular implementations as DHT [13] for decentralizing BitTorrent traffic further, and might be useful for piggybacking such peer-to-peer trackerless implementations of LeechLock.

4.3 Bootstrapping

The LeechLock implementation does not outright solve the bootstrapping problem, but in practice is quite effective at weeding out non-cooperative peers very early. When a peer first enters the swarm, it is allowed to freely leech data, per the standard BitTorrent implementation. However, depending on the threshold that has been set for what's considered a selfish leech, these peers can be very quickly identified and denied further requests for data. In some initial testing, these leeches have been locked out of the swarm with as little as 2MB of data being downloaded. In our final implementation, which was tweaked to ensure no cooperative peers were locked out, this number was as low as 8MB. These results are explored further in section 6.

5 Potential Attacks

The most obvious type of attack LeechLock is designed to counter is the ability of clients to rely on the optimistic unchoke to obtain an entire file without contributing any upload bandwidth to the swarm. However, the introduction of a peer ratings table introduces some of its own potential for abuse.

Let us consider the possibility of a node that wants to maliciously give another user poor ratings in an attempt to prevent the swarm from uploading to that user. First, this would only be possible if the user was a newcomer who has not yet built up good ratings with other clients. Otherwise, the malicious node's poor ratings would be

drowned out by all the other users that have positive dealings with that user. However, because the first client that finds out about the newcomer announces a message when it optimistically uploads a block to him, this will allow other users to "test" the newcomer by trying to download the block from him. Thus, the malicious peer would not have time to submit his negative ratings before other users have already started downloading from him, thereby building up its positive ratings.

Another possibility is that a malicious group of users, or Sybils, could conspire to give a user poor ratings in an attempt to shun that user (and effectively kick him out of the swarm). However, in order for this to work, all conspiring users would need to be uploading a substantial number of blocks to many of the other users in order for their negative reviews to have enough effect to bring down the victim. If this is the case then the Sybils must all be performing a very large number of uploads to many users in the swarm, and this would be very good for the swarm as a whole, and the Sybils are not actually performing a bad deed. All the extra work of performing the additional uploads would not benefit the malicious user.

6 Libtorrent Implementation

In order to test a LeechLock prototype in a real-world swarm, we have modified the popular Rasterbar libtorrent library to include a basic implementation of the modifications listed in section 4. Libtorrent proved to be a perfect platform for LeechLock because it is open source, portable, and used in a variety of BitTorrent clients. The libtorrent-enabled client that were used for the majority of tests was hrktorrent [14] due to the simplicity of its code base, its CLI, and strict adherence to the BitTorrent protocol, while qBittorrent [15] was used for the modified tracker implementation.

6.1 Implementation Details

We modified libtorrent as follows: we enhanced each torrent object with two ratings tables, one called "transient" and the other called "steady". The transient table contains the client's personal view of each peer it has interacted with since the last report to the tracker, and is used to report negative or positive interaction that has recently occurred. This table is cleared on each submission to the tracker, so each report contains only recent interactions. The steady table is populated directly with reports gathered from the tracker, and is not modified by the client itself.

As mentioned in section 4, the transient table is updated by deducting points from a client when requesting data from it, and adding points after receiving valid data from it. At the end of a 5 second interval, the client sends the current contents of its transient table to the tracker by appending an additional request parameter for each

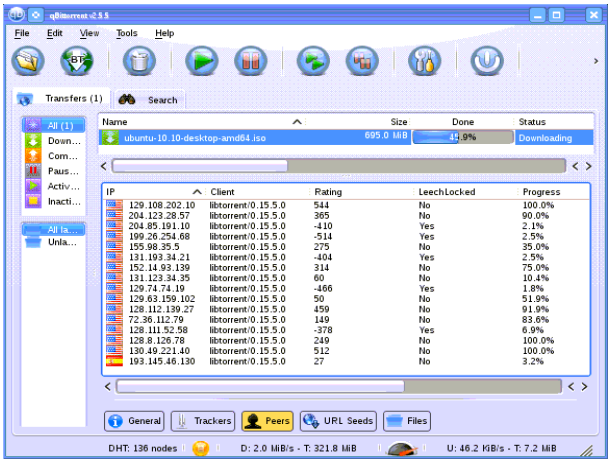

Figure 1: *Our modified qBittorrent tracker. This execution includes a LeechLock-enabled peer and shows the divergence between the cooperative peers and the selfish ones. There are 5 properly identified selfish peers in this case.*

peer with whom it has interacted since the last tracker report. In return, the peer receives an updated report of the global ratings for each peer (stored in the existing bencoded peer dictionary). This table is stored directly in the peer's steady ratings table, and is ultimately the set of ratings that is considered in the decision to lock a peer from the swarm. Once a peer's steady rating value has dropped below a threshold (which is scaled to roughly the size of the swarm), the peer will refuse to provide any data to the locked peer.

The tracker maintains only a set of steady ratings, which is updated based on input from all peers. For each peer report, a positive rating results in a slight increase in the steady rating, which is weighted based on the size of the swarm. The tracker assumes the same threshold in consideration of LeechLocked peers, and peers that have already been LeechLocked are not taken into account in the ratings. Therefore, a locked peer cannot attempt to retaliate against a cooperative peer.

The ability for the tracker to piggyback these ratings values in the response dictionary is part of the basis for compatibility with existing BitTorrent clients. Since existing clients are unaware of the "rating" dictionary value, it is simply disregarded and unused, while LeechLock clients are able to use this data to shun the right peers. Further, since reports from peers are done via request parameters (one per contacted peer), the modified tracker merely treats a standard BitTorrent client identically to an LeechLock client that has no peers to report.

6.2 Tests and Results

All tests were executed on a *PlanetLab* [16] slice consisting of 20 nodes. For the main set of tests, ten LeechLock-enabled peers, six selfish leeches, and two seeds were allocated to these nodes (two nodes remained idle for these

tests). During these tests, the clients shared a 650MB file, and the average download time for each peer was captured. After executing three tests with this configuration, all six selfish leeches were locked out of the swarm within one minute, and managed to download an average of 22MB before being locked out of the swarm. The highest amount of data a selfish peer managed the download was 48MB, while the lowest was only 8MB. Therefore, in order to reap the benefits of LeechLock, a download size of 50MB or more would be recommended, although there is no harm in using it for files of smaller size aside from a minor performance penalty, discussed in the simulation results.

All of the cooperative LeechLock-enabled peers downloaded the file in its entirety at varying completion times, ranging from 4 minutes and 38 seconds up to about 30 minutes. This disparity in download times is attributed to the varying CPU usage and bandwidth availability on various PlanetLab nodes, and is the main reason that download times are not included in our analysis. For this reason, we used controlled simulations outlined in section 7 below to determine time-sensitive LeechLock optimizations. Running a more controlled BitTorrent swarm with large bandwidth would likely shed more light on the real-world performance of the LeechLock clients.

To ensure compatibility with the existing BitTorrent protocol, the above setup was executed with an additional two nodes hosting an unmodified hrktorrent/libtorrent implementation. Since these clients do not consider ratings in their decision to share with a peer, this resulted in the selfish clients all successfully downloading the file, albeit at an average of 26% slower than the cooperative clients. It follows that the higher the percentage of LeechLock-enabled clients in the swarm, the less successful these selfish clients will be in downloading a file in a reasonable time frame.

7 Simulation Setup

The libtorrent LeechLock prototype was useful in demonstrating that using the existing BitTorrent protocol, such a modification is achievable. However, we found the variable download rates and unpredictable CPU spikes of the PlanetLab cluster to be less than optimal for determining the efficiency and immediate benefit for a LeechLock-enabled client. Therefore, we further explore the advantages of LeechLock by simulating a swarm of standard BitTorrent clients alongside several freeloading clients, who do not upload any data. Our Java-based simulation, BitSim, simulates the main characteristics of the BitTorrent protocol while also lending itself to simple modifications for the addition of such features as a LeechLock prototype.

7.1 BitSim

The BitSim protocol behaves as follows: a tracker is launched, which provides a list of currently connected peers to a new leech. Peers are then launched, and may be initialized as a seeder (a peer that starts with all data pieces), a leech (a peer which starts with no data pieces and always shares when requested), or a freeloader (a peer that starts with no data pieces and never shares data). Each peer keeps an array of its currently held data, as well as its personal ratings of all the other peers. Peers check in with the tracker every 2.5 seconds to share their personal ratings of other peers, to inform the tracker that they are still available for data sharing, and to retrieve the current global ratings of other peers. Data exchange is simulated through the transmission of XML messages which indicate to a peer whether or not it was provided the data it asked for. XML messages are also used to convey peer information to and from the tracker, such as the peers' locations and ratings. Depending on whether or not LeechLock is enabled, a peer may use its current rating of a peer to make a decision about sending a requested piece.

7.2 Execution

We executed each trial run as follows: execution was synchronized such that all peers would begin requesting data simultaneously to ensure that results were not skewed by any peers starting early. Further, each trial was executed with exactly 5 seeders, and run on swarms containing 3 freeloaders and 27 leeches (10% freeloader simulation), and 12 freeloaders and 18 leeches (40% freeloader simulation). These two swarm simulations were run using various file sizes ranging from 16 to 256 blocks. We ran the swarm simulation until all peers have attained all data pieces[1]. We measure the effectiveness of the freeloaders versus standard leeches by charting the amount of time required for each of the two peer types to procure all data pieces.

7.3 Results

A surprising note about the results of the LeechLock simulation was the profound effect that the larger percentage of freeloaders had on the health of the swarm. In the control experiment (which did have have LeechLock enabled), the average download times for a swarm of 40% freeloaders were 35% slower than a swarm of 10% freeloaders. With so few peers willing to share their data, much overhead is wasted futilely attempting to solicit data from unwilling freeloaders, causing the entire swarm to suffer. This is the main motivation behind the LeechLock enhancement; to improve the overall quality

[1] In the demonstration trials of LeechLock, it became clear in some cases that certain peers would never attain all pieces. In these cases, the trial was run for a reasonable amount of time to conclude that termination would never occur, and the behavior was verified through code traces.

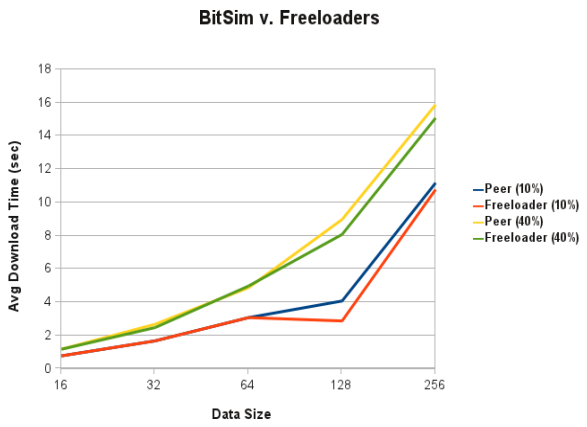

Figure 2: *Our control simulation setup, using peers without the LeechLock enhancement*

of the swarm and to ensure that cooperative peers achieve lower download times. Finally, the control experiment shows that freeloaders finish their downloads slightly faster than the standard leeches, since a freeloader has one more peer to download data from. This behavior would diminish with the size of the swarm.

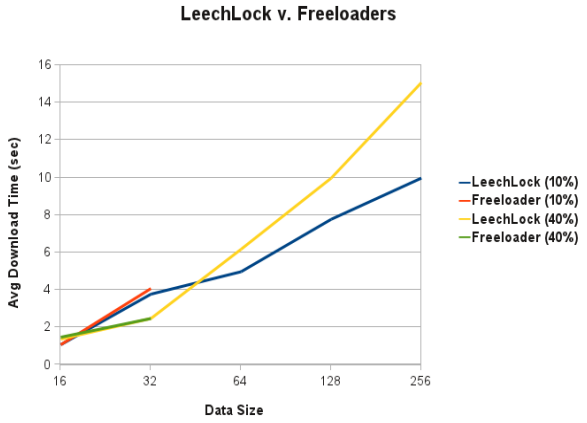

Figure 3: *Simulation results using the LeechLock-enabled clients*

Next, we deployed the LeechLock prototype, which pits the LeechLock-enabled clients against the freeloaders. The most obvious deviation from standard BitSim trial is selfish clients were consistently locked at a data size of 32 blocks. In the 16-32 block range, there is simply not enough information for the other peers to make a reliable opinion about the non-sharing peer. While the peer's rating suffers somewhat as a result of its stinginess (resulting in a slightly longer download time), its rating has not dropped below the threshold required to entirely blacklist a peer from the swarm. However, reduction of this threshold would create too many freeloader false positives, and considering a 16-32 block file is atypical for a BitTorrent distribution, the ability for a freeloader to obtain these files is acceptable by our standards. At 64 blocks and above, the freeloader has turned down too many peers who have all uploaded their negative feedback about this peer to the tracker, which has distributed this negative rating data to all the other peers. The peers, in turn, refuse to upload any data whatsoever to these selfish leeches, essentially locking them out of the swarm.

The final download results showed that LeechLock download rates were slightly slower for smaller data sizes, and improved with the size of the data. For tiny sets of data, the overhead of running the LeechLock and determining which peers to lock is the cause of this penalty. The cutoff for improvements in download times was at 256 blocks, which is a very low download size. Most BitTorrent downloads would be much larger than this [17], and as such in the vast majority of cases, the LeechLock clients would offer substantial improvement.

7.4 Additional Discoveries

In implementing the libtorrent LeechLock prototype, there was some debate as to whether it would be a good idea to permanently lock freeloaders from further participation in the swarm. By permanently locking peers who are considered selfish, immediate purification of the swarm occurs, since no piece requests are made to clients that are known to not share. This allows each peer to fill their request pipeline with requests only to known cooperative peers and not waste these slots on peers that will not reciprocate. The downside is that peers that may have been locked out, perhaps simply due to ill-intentioned peers or a multitude of requests for a piece it simply does not have, are given no chance to prove their good intentions and will never be able to continue to download a file.

A permanent lock trial was executed in the libtorrent implementation to determine if this would result in any benefit in practice. After several trials, we determined that the difference in download rates when permanently locking peers was insignificant when compared to imposing a temporary lock. This was due to the variability of the various nodes' network connections as well as the small impact expected from this optimization. As a result, we retained the soft-lock mechanism and continued to request pieces from locked peers, as the benefit to minimizing false-positive reports for freeloaders outweighs the minor performance improvement of locking them out.

However, BitSim provided an excellent test for a more widespread scenario free from the variable and unpredictable download rates of the PlanetLab setup. We decided to test permanently locking selfish clients by removing them from the list of peers after a "lock" determination is made to see if it makes a difference in down-

load times. The simulation results are shown in figure 4. Not surprisingly, larger data sizes yield a greater increase

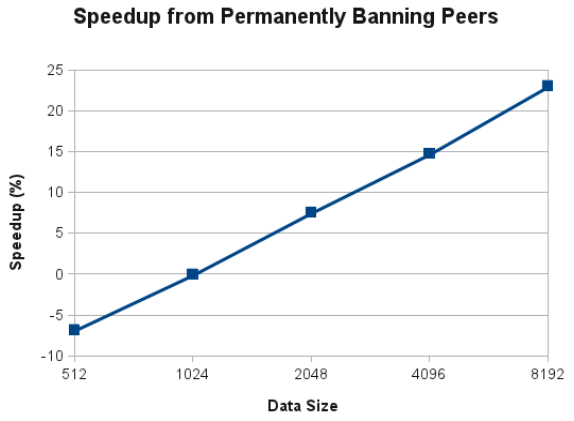

Figure 4: *The increase in simulation speed from permanently banning leechlocked peers, instead of considering them for future requests.*

in speedup, as the clients have a longer period of time to exchange with only known cooperative peers. The magnitude of the increase is interesting, however, in that the speedup appears to double each time the data size is doubled. The crossover point was at 1024 blocks, and we determined data sizes below that point to be less effective in such a scenario. Since BitTorrent downloads tend to consist of more than 1024 blocks of data, we intend to further investigate permanent banning as a viable option for the LeechLock implementation.

While we executed this simulation to determine whether temporarily or permanently banning peers was the best option for LeechLock, the results also provided some insight as to how these selfish peers cause harm in the swarm, and the benefit that LeechLock can provide. In completely weeding out the freeloaders from the swarm, large shared files will likely benefit from a great decrease in the expected download time.

8 Future Work

The current LeechLock implementation proves that the global sharing of peer ratings can benefit the swarm. However, further refinement of the implementation can provide several benefits. For example, for the purposes of our experiments we deemed it sufficient to add peers to the ratings table based on their BitTorrent peer ID. Clearly, this is not secure, as a selfish client could simply declare a new peer ID (or restart their client) in order to continue downloading. We suggest updating the implementation to track peers by IP address instead of ID would prevent these types of manipulations.

Also, the current LeechLock implementation is intended to solve the BitThief problem, but not the BitTyrant one. That is, a peer that shares no data at all will be successfully shut out of the swarm, but a peer that attempts to contribute as little as possible to the swarm has not been tested thoroughly. For this reason, we suggest further exploring the potential for combining both the LeechLock and PropShare concepts into the same client. Other options for solving the "barely enough" peer issue include reconsidering the permanent locking functionality described in the simulation results section. In this case, a peer would likely think twice about contributing as little as possible, since crossing the lock threshold would disallow them from downloading a file at all. Additionally, continued refinement of the exact point values assigned for receiving a piece (or requesting one) is likely to minimize this type of behavior.

Automated pattern detection is also an important feature to be built into any community-driven rating system. For example, community-driven websites are susceptible to groups of users who always vote in each others' interests, resulting in artificially high ratings for some content. In LeechLock, if a user managed to spoof a number of clients comparable to the size of the swarm, they could ensure that their ratings are always very high. While detecting clients originating from the same IP address can be effective, in order to properly tackle this issue we propose that an algorithm to detect common patterns indicative of gaming the protocol be developed. In general, it would be interesting to see if a few modified LeechLock client what always reports negative values for all peers (or positive values for preferred peers) could cause some chaos in the swarm.

Further, with LeechLock enabled on a BitTorrent client, we could also expand these experiments to a large computer network to ensure that LeechLock is scalable and valuable for thousands of peers instead of just tens. Alternatively, a beta public release of the LeechLock implementation could allow us to monitor the true benefit of this BitTorrent enhancement.

Finally, despite our exploration and refutation of potential attacks on LeechLock, as others have proven the vulnerability of BitTorrent to freeloaders, there may also be additional attacks and exploits that may be made on LeechLock. Discovery and resolution of such vulnerabilities would be mandatory for the success of LeechLock, as even one successful circumvention would render its main purpose, fairness within the BitTorrent protocol, ineffectual.

9 Conclusion

We have described LeechLock, a modified BitTorrent implementation that is more resistant to leeches who do not share. After describing the modifications and explaining how it should be resistant to certain types of attacks, we demonstrated its utility by running a

LeechLock-enabled swarm on a set of PlanetLab machines, and showed that it is possible to use a tracker-based ratings system to successfully weed out and deny content to all peers who do not contribute to a swarm. The results showed that selfish peers would be locked out of the swarm within a minute, after downloading 8-48 MB of data, with an average of 22MB being successfully downloaded. The highest amount of data a selfish peer managed the download was 48MB, while the lowest was only 8MB, therefore LeechLock is particularly recommended for download sizes of 50MB or more. We also demonstrated that standard BitTorrent clients *are* capable of cooperating with LeechLock-enabled peers, and the download rates for selfish peers will be a function of the number of these standard BitTorrent peers in the swarm.

Further, by simulating a BitTorrent swarm using BitSim, we were able to explore the details of what happens to selfish peers in a LeechLock swarm compared to a normal one, and demonstrated the value of a our swarm improvement algorithm. The most interesting outcome of the BitSim test was that above 1024 blocks of data, the speedup achieved by permanently locking selfish peers increased exponentially, effectively doubling every time the data size doubled. Future work should include brainstorming new types of attacks and attempts to game LeechLock, solutions for those attacks, as well as enhancements to the existing prototype to provide faster download rates and ensure that peers who share very little are locked out in the same way as peers who share nothing.

10 References

1. M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In NSDI, 2007.
2. T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap. In HotNets, 2006.
3. M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding in BitTorrent networks with the large view exploit. In IPTPS, 2007.
4. D. Levin. Incentive-compatible Bootstrapping.
5. D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. BitTorrent is an auction: Analyzing and improving BitTorrent's incentives. In ACM SIGCOMM, 2008.
6. Rasterbar Software libtorrent. http://www.rasterbar.com/products/libtorrent.
7. BitTorrent Protocol Specification v1.0. http://wiki.theory.org/BitTorrentSpecification.
8. BitTornado. http://www.bittornado.com.
9. Z. Chen, Y. Chen, C. Lin, V. Nivargi, and P. Cao. Experimental Analysis of Super-Seeding in BitTorrent. In IEEE ICC, 2008.
10. A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In SOSP, 2007.
11. B. Cohen. Incentives build robustness in BitTorrent. In P2PEcon, 2003.
12. S. Jun and M. Ahamad. Incentives in BitTorrent induce free riding. In P2PEcon, 2005.
13. Distributed hash table. http://en.wikipedia.org/wiki/Distributed_hash_table.
14. Hrktorrent - a light console torrent client using rb_libtorrent. http://50hz.ws/hrktorrent/.
15. qBittorrent official website http://qbittorrent.sourceforge.net/.
16. Planetlab — An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/.
17. Torrent Piece Size. http://wiki.vuze.com/w/Torrent_Piece_Size