# FairTorrent: Bringing Fairness to Peer-to-Peer Systems

Alex Sherman
Dept. of Computer Science
Columbia University
1214 Amsterdam Ave MC0401
New York, NY 10027
asherman@cs.columbia.edu

Jason Nieh
Dept. of Computer Science
Columbia University
1214 Amsterdam Ave MC0401
New York, NY 10027
nieh@cs.columbia.edu

Clifford Stein
Dept. of IEOR
Columbia University
500 West 120th St. MC4704
New York, NY 10027
cliff@ieor.columbia.edu

## ABSTRACT

Peer-to-Peer file-sharing applications suffer from a fundamental problem of unfairness. Free-riders cause slower download times for others by contributing little or no upload bandwidth while consuming much download bandwidth. Previous attempts to address this fair bandwidth allocation problem suffer from slow peer discovery, inaccurate predictions of neighboring peers' bandwidth allocations, underutilization of bandwidth, and complex parameter tuning. We present FairTorrent, a new deficit-based distributed algorithm that accurately rewards peers in accordance with their contribution. A FairTorrent peer simply uploads the next data block to a peer to whom it owes the most data as measured by a deficit counter. FairTorrent is resilient to exploitation by free-riders and strategic peers, is simple to implement, requires no bandwidth over-allocation, no prediction of peers' rates, no centralized control, and no parameter tuning. We implemented FairTorrent in a BitTorrent client without modifications to the BitTorrent protocol, and evaluated its performance against other widely-used BitTorrent clients. Our results show that FairTorrent provides up to two orders of magnitude better fairness, up to five times better download times for contributing peers, and 60% to 100% better performance on average in live BitTorrent swarms.

## Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols-Applications; C.2.4 [**Computer-Communication Networks**]: Distributed Systems-Distributed Applications

## General Terms

Algorithms, Design, Experimentation, Measurement, Performance

## Keywords

Peer-to-Peer, Fairness, BitTorrent, File-sharing, Free-riding


*CoNEXT'09,* December 1–4, 2009, Rome, Italy.


## 1. INTRODUCTION

Peer-to-Peer (P2P) file-sharing is a popular, cheap and effective way to distribute content. However, P2P file-sharing applications suffer from a fundamental problem of unfairness. Many users free-ride by contributing little or no upload bandwidth while consuming much download bandwidth. By taking an unfair share of resources, free-riders cause slower download times for contributing peers. If a P2P system could guarantee fair bandwidth allocation, where by fairness we mean that a peer receives bandwidth equal to what it contributes, the system would be able to guarantee a certain level of performance for contributing peers.

Fair bandwidth allocation in P2P systems can be difficult to achieve for several reasons. First, no central entity controls and arbitrates access to all resources, unlike scheduling fair allocation of bandwidth for a router [7, 3, 26]. Second, the amount of bandwidth resources available is not known in advance and peers cannot be relied upon to specify their own resources honestly. Finally, strategic peers and free-riders may try to take advantage of the system by contributing little or no bandwidth, while consuming others' resources.

Many approaches have attempted to address the problem of fair bandwidth exchange. The most common approach is the tit-for-tat (TFT) heuristic employed by the popular file-sharing system BitTorrent [6]. TFT splits a peer's upload bandwidth equally among a subset of neighboring peers for a time duration called a round, then adjusts this subset each round based on estimates of their bandwidth rates from the previous round. Block-based TFT [5, 29, 11], used by BitTyrant [21] peers among one another, attempts to improve on TFT by augmenting TFT with hard limits on the amount of data one peer can owe another. PropShare [14] also attempts to improve on TFT by using the Proportional Response algorithm [32] to split the peer's upload bandwidth in proportion to the contribution received from its neighbors in the previous round. All of these approaches are rate-based and suffer from a fundamental flaw. First, they require long round durations to estimate bandwidth contribution of the neighboring peers, and waste much bandwidth each round before discovering other contributing peers. Second, they assume that a peer's allocation measured in a given round is an accurate estimate of the future contribution from that peer. In practice, this assumption is problematic as each peer can change its allocation or even stop uploading to a given peer. This results in significant problems with existing approaches, including unfairness, exploitation by strategic clients [18, 21, 27], bandwidth underutilization [5, 11], ad-hoc parameter tuning requirements, and poor performance.

We present FairTorrent, a new deficit-based distributed P2P algorithm that solves the problem of fair bandwidth exchange in the presence of free-riders and strategic peers. FairTorrent accurately rewards peers in accordance with their contribution. It runs locally at each peer and maintains a deficit counter for each neighbor which represents the difference between bytes sent and bytes received from that neighbor. When it is ready to upload a data block, it sends the block to the peer with the lowest deficit. Unlike other approaches, FairTorrent uses a completely different mechanism that does not require an estimate of neighboring peers' rate allocation. Therefore, it does not require rounds for discovering favorable peer sets, and does not waste bandwidth by over-allocating its bandwidth in a round. Intuitively, by selecting the destination of the next block to go to the neighbor with the smallest deficit, FairTorrent always rewards the peer to whom it "owes" the most data, or a peer who reciprocates at the highest rate. FairTorrent results in very fast rate convergence, a high degree of fairness, and thus better performance for high-contributing peers. Our analysis for various scenarios shows that FairTorrent peers in a $n$-node network contribute at most $O(\log n)$ more data blocks than what they receive from other peers.

FairTorrent provides several key advantages over other approaches: (1) It provides fair bandwidth allocation, operating only at individual peers, in a distributed manner that does not require any centralized control of peers or other P2P resources. (2) It is provably resilient to free riders and other strategic clients. (3) It allows a peer to maximize its upload capacity utilization. (4) It avoids long peer discovery and reaches a fast rate convergence; i.e. it quickly obtains a bandwidth reciprocation rate from its neighbors equal to its own contribution. (5) It does not need to estimate and predict peers' allocations, allocate precise upload or download rates for any peers, or rely on advanced knowledge of available bandwidth of other peers. (6) It has no magic parameters, requires no tuning, and is simple to implement. A FairTorrent client requires no changes to the BitTorrent protocol, making it compatible and easy to use with existing BitTorrent clients.

We implemented FairTorrent inside a BitTorrent client and compared its fairness and performance against four other widely used open-source BitTorrent implementations: the original BitTorrent Python client by Bram Cohen [6], the popular Azureus Java BitTorrent client [2], the PropShare [14] client, and the strategic BitTyrant [21] client. Our results show that FairTorrent outperforms all other clients across a wide range of different client bandwidth distributions, static and dynamic scenarios, as well as in live swarms. For peers with widely different bandwidths across a uniform distribution, FairTorrent provides more than an order of magnitude better fairness and up to 50% faster download performance. For a high bandwidth uploader in the presence of many low contributors, FairTorrent can provide two orders of magnitude better fairness and up to five times faster download performance. For a distribution based on live BitTorrent networks, FairTorrent can provide two orders of magnitude better fairness and more than two times faster download performance. In a network dominated by Azureus clients, the most popular client today, FairTorrent can improve contributing peers' download times 50% more than BitTyrant or PropShare. In live swarms, FairTorrent outperforms all other clients by 60% to 100% on average. We further demonstrate that performance in live swarms is highly dependent on the limit on the number of connections for a peer. Previous work on BitTyrant [21] and PropShare [14] do not account for this properly by using an order of magnitude higher connection limit when comparing against Azureus, resulting in overstated performance improvements.

## 2. BACKGROUND AND RELATED WORK

BitTorrent [6] employs a rate-based tit-for-tat (TFT) heuristic to incentivize peers to upload and attempt to provide fair exchange of bandwidth between peers. Peers participating in the download of the same *target file* form a *swarm*. The target file is conceptually broken up into pieces, typically 256 KB. Peers tell one another which pieces of the target file they already have and request missing pieces from one another. Requests are typically made for 16 KB sub-pieces. Peers that already have the entire file are *seeds*. Peers that are still downloading pieces of the file are *leechers*. TFT is used in a swarm to enable fair bandwidth exchange during the current download of a file. It operates by having each BitTorrent client upload to $N$ other peers in a round-robin fashion, where $N - k$ of the peers have provided the best download rate during the most recent 10 to 20 second time period, and $k$ peers are randomly selected to help discover other peers with similar upload rates. $N$ is typically between 4 and 10, and $k$ is typically 1 or 2. Thus, the *active set* of peers that a client uploads to is updated each round based on the measurements of their download rates. BitTorrent refers to the selection and deselection of a peer for uploading as *unchoking* and *choking*, respectively.

Because of its popularity, much work has been done in studying BitTorrent's behavior. BitTorrent peers tend to exchange data with other peers with similar upload rates over a large file download [13]. Under some bandwidth distributions, the system has been shown to eventually converge to a Nash equilibrium [24]. However, there is no evidence that this behavior extends to shorter file downloads, dynamic environments, skewed distributions of users, or modified but compatible BitTorrent clients. In fact, several modified BitTorrent clients [18, 27, 21] have been developed which exploit different strategies to achieve better performance at the expense of users running unmodified BitTorrent. For example, BitTyrant [21] exploits the fact that BitTorrent will reciprocate at a higher rate even when receiving much smaller bandwidth from BitTyrant in return.

The previous studies demonstrate that BitTorrent's TFT heuristic does not result in fair bandwidth exchange. Because TFT only identifies and exchanges data with a small number of peers at a time, a BitTorrent client may waste much time and bandwidth while discovering peers with similar upload rates in a large network [21]. Further waste occurs because relationships with discovered peers may be unstable, as the other peers are also always searching for better peers. Even after discovering peers with good upload rates, BitTorrent continues to blindly donate a portion of bandwidth by randomly uploading to other peers in hopes of reciprocation.

Block-based TFT [5] (BB-TFT) attempts to improve on BitTorrent by changing TFT to allow a client to upload to a larger set of neighbors at the same time, but stopping the upload to a peer once its deficit, the difference between what it uploaded to the peer and what it downloads from that peer, exceeds a certain threshold. While FairTorrent and BB-TFT appear similar in that both use peerwise deficits, their us-

age of deficits is fundamentally different. FairTorrent uses deficits to decide deterministically where to send the next data block. By sending the next block to a peer to whom FairTorrent "owes" the most data, it always acts to minimize *unfairness* and thereby converges quickly to fair bandwidth exchange with its peers. BB-TFT uses deficits only to impose a threshold, but still uses TFT for scheduling. Merely stopping one peer from uploading to another when a threshold is reached does not by itself help peers reach fair rate reciprocation. The hard threshold causes underutilization of the peers' upload capacities [5], as high-uploading peers may stop uploading to other peers once they reach the limit. To compensate, a tracker could attempt to match peers with similar bandwidth by trusting peers to report their capacities honestly [5]. This is problematic in practice as peers could game the system by lying about their bandwidth. While a limited simuluation study shows that BB-TFT can improve the fairness of BitTorrent [5], another study shows that BB-TFT has poor performance and bandwidth utilization compared to BitTorrent [11]. Our experiments with BitTyrant [21] peers, which use BB-TFT when exchanging data with each other, also show that BitTyrant clients do not receive a fair exchange rate and suffer from underutilization. SWIFT [29] augments BB-TFT with a heuristic to have peers donate a small fraction of their bandwidth, but it is not clear how to tune their highly-parametrized algorithm in a realistic deployment scenario.

PropShare [14] attempts to achieve fair bandwidth allocation by implementing the Proportional Response algorithm [32] on top of the Azureus client. Using the algorithm, a peer splits its upload rate to its neighbors for a given round in proportion to the contributions received from them in the previous round. Under simplistic assumptions, the theoretical work in [32] shows that the algorithm converges to a *market equilibrium*, and a peer receives fair reciprocation after $O(\log n)$ rounds, in a $n$-node network. The algorithm assumes that each peer can accurately estimate all of its neighbors' upload rate allocations towards it in each round; peers require these estimates to readjust their own allocations correctly in the next round. This assumption is problematic in practice. Very long rounds would be required to allow each peer to exchange enough data with all of its neighbors to obtain estimates. However, estimates over long rounds would be of limited usefulness for determining allocations under dynamic conditions that occur in practice. Propshare attempts to address this problem by estimating allocations over four exponentially-weighted 10-second rounds. Because peers' allocations change and PropShare uploads to and estimates rates of only a small subset of neighbors in each round, PropShare fails to create an accurate view of the current rate allocations of a larger neighborhood. Moreover, the peerwise convergence process is often interrupted and reset because PropShare clients divert 20% of their bandwidth in each round to explore new peers, much like BitTorrent. Our experimental results show that PropShare demonstrates poor rate convergence and does not reward contributing peers fairly. In contrast, FairTorrent's new deficit-based approach does not need to explicitly set upload rates and avoids the pitfalls associated with having to estimate peers' changing rate allocations, dealing with insufficient and ad-hoc round durations, and needing to explore a peer's entire neighborhood. It may be possible to tune PropShare to use very small blocks in order to help it explore a larger neighborhood faster, as required by [32]. However, using small blocks would significantly increase the overhead associated with BitTorrent and IP protocols.

Some work has explored tradeoffs between performance and fairness in BitTorrent. Based on the assumption that leechers leave the system upon completion of download, one model proposes to optimize average performance by lowering the download rate for high uploaders to keep them in the system longer [9]. However, in practice, many leechers already remain in BitTorrent systems as seeds [22]. Furthermore, other work suggests that fairness does not need to come at the expense of performance [30].

Other approaches have explored different aspects of improving fairness in non-BitTorrent P2P systems. Reputation-based systems [12, 17, 4] maintain reputation metrics for each peer to make it easier for reputable peers to find one another. Such systems suffer from problems with bootstrapping and collusion [25], where malicious peers can hype one another's reputation. Even if a perfect reputation metric can be established, such systems do not provide a mechanism for translating reputation into a highly-fair bandwidth-sharing service. BAR [1], BAR Gossip [16], and FlightPath [15] propose to stem out selfish users by relying on signed proof-of-misbehavior messages. BAR relies on agreement by a quorum of nodes to punish a misbehaving peer, and thus suffers from collusion attacks. BAR Gossip and FlightPath are only applicable when the data broadcaster has the authority to evict a node, typically not the case in P2P networks.

Credit-based systems [31, 19, 10, 28] use virtual credit to incentivize fair exchange of services among peers in P2P systems. They typically require significant overhead as well as trusted third party agents to maintain credit values and verify the services provided. Unile FairTorrent, credit-based systems, including eMule [8], which maintains peerwise credits, are not designed to provide finer granularity fairness during the current download of a file. Credit is typically maintained over many file downloads over many days. Performance for the current download of a file depends more on prior credit accumulation by all of the participants, and not the current willingness of a peer to contribute bandwidth.

The problem of fair bandwidth allocation has perhaps been most extensively studied in the context of scheduling packets through a router [7, 20, 3, 26]. Given a set of flows with associated service weights, the problem is how to schedule packets to allocate bandwidth in proportion to the respective weights. While there are some similarities between the packet scheduling problem and the problem that FairTorrent addresses, the key difference is that in the latter case, peers have no explicit or assigned weights. Weights in packet scheduling correspond roughly to upload rates in P2P systems, but these rates are not assigned or known in advance. The resulting challenge in P2P systems is how to provide fair bandwidth exchange given that the upload rates are not known, change dynamically, and can be difficult to estimate. While deficits have been used for packet scheduling [26], FairTorrent uses a completely different algorithm to solve a fundamentally different distributed problem.

# 3. FAIRTORRENT ALGORITHM

FairTorrent implements a fully distributed algorithm that provides fair data exchange, without any global allocation or management service beyond what is already provided by BitTorrent. A FairTorrent client uploads a data block to the

peer it "owes" the most data and automatically converges to the individual reciprocation rates of its peers, without measuring or predicting these rates. For compatibility with BitTorrent, FairTorrent uses the same BitTorrent protocol, torrent files, and tracker service. We also borrow the terminology from BitTorrent, including *seeds* and *leechers*.

## 3.1 Leecher Behavior

We first describe the basic algorithm run by the leechers. Let $Sent_{ij}$ be the total number of bytes that a leecher $L_i$ has sent to a leecher $L_j$, and $Recv_{ij}$ be the total number of bytes that $L_i$ has received from $L_j$. Each $L_i$ maintains a *deficit variable* $DF_{ij}$ for each $L_j$, where $DF_{ij} = Sent_{ij} - Recv_{ij}$. Thus, a positive (negative) deficit implies that $L_i$ uploaded more (fewer) bytes to $L_j$ than it downloaded from $L_j$. Initially, each $DF_{ij} = 0$. The values $DF_{ij}$ are maintained in sorted order by $L_i$ in a list called *SortedPeerList*. When $L_i$ is ready to send the next data block, it chooses to send that block to the peer with the smallest $DF_{ij}$.

Procedure 1 shows the FairTorrent operation RecvBlock executed by $L_i$ when it receives a block from some peer $L_j$. RecvBlock checks that peer $L_j$ is a leecher. Subroutine ValidBlock is used to check the MD5 hash of the block. Just like BitTorrent, FairTorrent obtains the list of the MD5 hashes of all the blocks from the torrent metafile that it downloads prior to the download of the actual content file. If peer $L_j$ is a leecher, and the block is valid, FairTorrent increments $Recv_{ij}$ and decrements $DF_{ij}$ by the number of bytes received from $L_j$, and re-inserts $L_j$ into the *SortedPeerList* sorted from lowest to highest deficit values $DF_{ij}$.

**Procedure 1** RecvBlock(*peer j, data_block p*)

```
if IsLeecher(j) and ValidBlock(p) then
  Recv_ij ← Recv_ij + size(p)
  DF_ij ← DF_ij − size(p)
  SortedPeerList.ReInsert(j)
end if
```

Procedure 2 shows the FairTorrent operation SendBlock executed by $L_i$ when it is ready to send some data to its peers. Just like BitTorrent, up to $B$ bytes are sent, where $B$ is based on the peer's upload capacity $\mu_i$ bytes/s. $\mu_i$ is typically set by the user to limit consumption of bandwidth capacity. SendBlock tries to pick a leecher with the the lowest possible value of $DF_{ij}$ and send a block, or finish sending a block, to that leecher. It examines the *SortedPeerList* starting at the lowest index, which contains the peer $L_j$ with the lowest $DF_{ij}$. If currently $L_i$ has no pending request from $L_j$, (i.e. it has no useful data to send to $L_j$), or the connection is not writable, (i.e. there is no room in the TCP socket buffer), then the procedure examines the next peer on the list, with the next lowest deficit. SendBlock assumes the existence of several subroutines. HPRF($j$), or HavePendingRequestFrom(j), returns true if there is a pending request from peer $L_j$. CWT($j$), or CanWriteTo($j$), returns true if there is room in $L_j$'s buffer to send a block. BytesRemainsToSend(j) returns the number of bytes remaining to send $L_j$ from the current block being sent to $L_j$.

Once it selects a peer $L_j$ to whom it can send data, FairTorrent tries to send a block, or finish sending remaining bytes of a block to $L_j$. The procedure can send a maximum of $B$ bytes and the actual number of bytes remaining of the current block. FairTorrent uses blocks of size *block_size* = 16 $KB$ for simplicity, given the default 16 KB request size used by almost all BitTorrent-compatible clients. The subroutine SEND is used to write the actual bytes. SEND returns the number of actual *bytes_written*, and decrements $B$, the bytes still available for further writes. If a block is fully sent to $L_j$, i.e. BytesRemainsToSend(j)== 0, then FairTorrent increments $DF_{ij}$ with the size of the block sent, *block_size*, and re-inserts $L_j$ into the *SortedPeerList*. *SortedPeerList* maintains peers in the order of non-decreasing deficit values, and breaks ties based on the peer's local random ordering of neighbors.

**Procedure 2** SendBlock(*allowed_bytes B*)

```
n ← 0;  sz ← Size(SortedPeerList)
while (B > 0) and (n < sz) do
  j ← SortedPeerList[n]
  while !(HPRF(j) and CWT(j)) do
    n ← n + 1;  j ← SortedPeerList[n]
  end while
  if (n < sz) then
    use_bytes = max(B, BytesRemainsToSend(j))
    bytes_written ←Send(j, use_bytes)
    B ← B − bytes_written
    if BytesRemainsToSend(j) == 0 then
      Sent_ij ← Sent_ij + block_size;
      DF_ij ← DF_ij + block_size;
      SortedPeerList.ReInsert(j)
    end if
    n ← n + 1;
  end if
end while
```

As SendBlock is the most important procedure in FairTorrent, we illustrate its behavior with an example of three leechers in Figure 1. For simplicity, the example assumes leechers always have data to send to one another, expresses values in blocks rather than bytes, uses equal-size blocks, and breaks deficit ties by having each peer order its neighbors from lowest to highest peer ID. Leechers $L_1$, $L_2$ and $L_3$ have upload capacities of $\mu_1 = 3$, $\mu_2 = 2$ and $\mu_3 = 2$ blocks/s, respectively. Thus, $L_1$ sends a block every 1/3 of a second, $L_2$ and $L_3$ send blocks every 1/2 of a second. All peers send their first block at time 0.000. Figure 1 shows all the clock times at which at least one of the peers sends a block. Each peer keeps track of its deficit variables, $DF_{ij}$, shown underneath each peer at each clock time. Arrows show the source and the destination of each block. At time 0.000, all the $DF_{ij} = 0$. Thus, $L_1$ sends to $L_2$. $L_2$ and $L_3$ each send a block to $L_1$. When $L_1$ sends a block to $L_2$, it sets $DF_{12} = 1$. Before time 0.333, it will have received a block from $L_2$ and will have decremented $DF_{12}$ back to 0. Thus, at time 0.333 $DF_{12} = 0$. Since it receives a block from $L_3$ before time 0.333, it will set $DF_{13} = -1$. At time 0.333, $L_1$ will send its next block. Using procedure SendBlock, $L_1$ will pick $L_3$ as the peer to send the block to, because it has lower deficit: $DF_{13} < DF_{12}$. $L_1$ sends a block to $L_3$ and increments $DF_{13}$ to 0. Figure 1 shows the process until time 2.000. In these 2 seconds, each peer sent the maximum number of blocks limited by its capacity and received the same number of blocks from its neighbors. At time 2.000, the system reverts to the same state as at time 0.000. Over a long time period, the same behavior will repeat every 2 seconds and all deficit values will remain between $-1$ and 1.

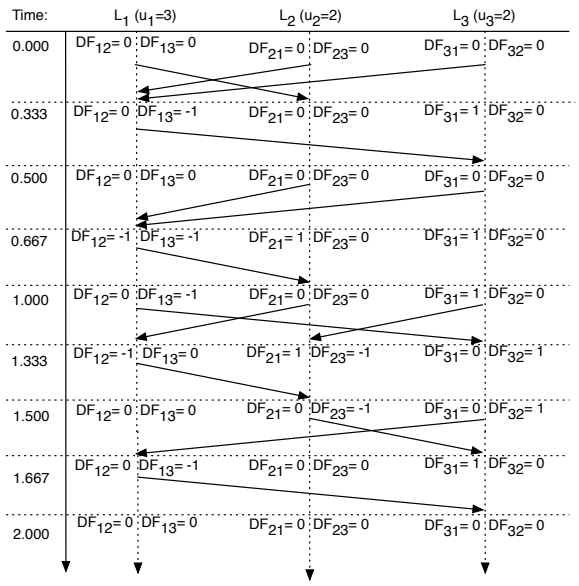


**Figure 1: FairTorrent algorithm for leechers $L_1$, $L_2$ and $L_3$ with upload capacities of 3, 2 and 2.**


**Figure 2: Peers $L_1$, $L_2$ and $L_3$ with upload capacities 3, 2, and 2. Bandwidth allocated under FairTorrent (left) vs Equal-Split (right).**

In a distributed fashion without knowing the neighbors' capacities, FairTorrent achieved a convergence between each pair of nodes, and convergence between the total upload versus download rate for each peer. Counting the arrows between each pair of peers that indicate the number of blocks sent in the two second cycle, we see that $L_3$ and $L_2$ exchanged 3 blocks each with $L_1$ (or 1.5 blocks/s) and they exchanged 1 block with each other (or 0.5 blocks/s). Figure 2 (*Left*) shows the resulting bandwidth allocation provided by FairTorrent for this example.

Consider the same example using the equal-split rate of the original BitTorrent. The equal-split heuristic results in each peer splitting its capacity evenly among its neighbors, as in Figure 2 (*Right*). Unlike FairTorrent which quickly reaches convergence, the rates diverge when using the equal-split heuristic. $L_1$ pushes 1.5 blocks/s to each neighbor, but receives only 1 block/s in return from each, resulting in unfair service as the deficits with each peer will grow.

Consider the same example using PropShare. Each peer begins in round one by sending at equal-split rate, then adjusts its rate proportionally to the exponentially weighted rate received in the recent four rounds. $L_2$ and $L_3$ send to $L_1$ at a rate of 1, 1.2, 1.28, and 1.314, in each of the first four ten-second rounds, respectively. While FairTorrent reaches convergence and 0 deficit in 2 seconds, after 40 seconds using PropShare, $L_1$ will have accumulated a deficit of at least 12 blocks and will still not have converged even in this miniature example. In a larger network, as the *active sets*, the neighbors each peer uploads to, are changing, PropShare may take longer to converge or even diverge.

## 3.2 Seed Behavior

Since seeds in a swarm do not download from peers in that swarm, using deficits to allocate bandwidth from a seed among leechers is of limited utility. FairTorrent allocates seed bandwidth to be split evenly among leechers by simply sending blocks in a round-robin fashion.

In other clients, such as Azureus a seed typically uploads only to a few peers in each round, resulting in very skewed allocation of seed bandwidth. In addition to splitting the seed bandwidth more evenly, the round-robin approach allows FairTorrent seeds to disseminate blocks to more leechers, and thus increase the likelihood that a leecher has blocks to trade with its neighbors.

## 3.3 Exchanging Data

**Unchoking Peers.** By default, FairTorrent establishes TCP connections with 50 neighbors. We chose this default to be the same as Azureus for a more fair comparison with the most popular client. According to the BitTorrent protocol, $L_i$ must *unchoke* $L_j$ to upload to $L_j$. An *unchoke* message lets $L_j$ know that it can send requests for data blocks until a *choke* message is received. Azureus and BitTorrent unchoke only a few peers at a time. The reason for this is that it forces higher uploading peers to push at a higher rate to each unchoked peer. This method allows higher uploading peers to measure each other's rates and discover one another in a swarm over time. Unfortunately, even if this discovery eventually succeeds, low uploading peers can also detect high uploading peers and leech off their bandwidth.

In contrast, a FairTorrent leecher unchokes each neighbor interested in its data. FairTorrent does not rely on BitTorrent's discovery method via selective unchoking to find "like-uploading" peers. Instead, a FairTorrent leecher $L_i$ is able to exchange data at diverse rates with different peers. Therefore, $L_i$ only needs to explore a small subset of peers as long as their combined reciprocation to $L_i$ equals $\mu_i$.

**Reaching Rate Convergence.** $L_i$ begins by moving along its randomly ordered list of peers and sends a block to each $L_j$ that requests data, and increments $DF_{ij}$. If some $L_j$ reciprocates, $DF_{ij}$ is reset, and $L_i$ will send the next block to $L_j$ before moving further down its list. As soon as it receives reciprocations at a rate of $\mu_i$, $L_i$ does not need to send blocks further down the list as it has reached rate convergence, or, in other words, it obtained a download rate from other leechers that equals its own upload rate. Our evaluation and analytical results show that FairTorrent is able to obtain a very fast rate convergence, since it only needs to explore a small subset of its neighbors.

PropShare is also able to split its upload rate unevenly to reward its neighbors according to their contributions. However, there are two main distinguishing characteristics that allow FairTorrent to reach a much faster rate convergence than PropShare. First, FairTorrent does not rely on estimating its peers' rate allocations towards it, as it does not need to explicitly set upload rates. By sending a block to the peer with the smallest deficit, or to whom it owes the most data, FairTorrent implicitly sends blocks faster to a peer from which it observes a faster rate. Because rate allocations change dynamically, and PropShare can only accurately estimate rates of several peers in each round, it fails to create an up to date view of its neighbors' rates, and ends up splitting its own rate incorrectly as a result. Second, the Proportional Response algorithm that PropShare

tries to emulate requires that a peer estimate the rates of all of its neighbors in each round. This requires PropShare to explore a large neighborhood of peers. FairTorrent, on the other hand, only needs to explore a small subset of its neighbors. As long as it is sending blocks to a set of neighbors whose reciprocation rate to $L_i$ equals $\mu_i$, it does not need to explore peers beyond this subset.

Thus, FairTorrent avoids the pitfalls associated with having to estimate peers' changing rate allocations, dealing with insufficient round durations, and having to explore a large neighborhood. We show in our evaluation that FairTorrent peers reach rate convergence much faster and more precisely than other clients.

**Data Availability.** FairTorrent leverages BitTorrent's *rarest-first* policy when requesting a block from a leecher or a seed. It always asks for a least commonly available block among its neighbors. This policy makes it highly likely that a peer always obtains a block missing among some of its neighbors, and thus, it is able to trade [24]. The intuition, is that once a peer obtains a rare block it can send it to many neighbors in exchange for distinct blocks. Also, procedure SENDBLOCK does not require a peer to always have a block to send to each neighbor. If $L_i$ temporarily has nothing to send to a peer with the smallest deficit, it can skip that peer and send to the next peer on the list.

**Bootstrapping.** When a new peer joins the system it needs to obtain a block to start trading. The fact that FairTorrent seeds send blocks in a round-robin to their 50 leechers, rather than sending to only several leechers over a 10 second round as does BitTorrent, allows a new peer to obtain a block faster. It may be useful to extend FairTorrent to allow a seed to maintain even more than 50 connections. It may also be useful to allow a leecher to send a block to a new peer with a very small probability. These changes would increase the probability that a new peer obtains a block quickly. In our experiments under various scenarios, we did not find these extensions necessary.

# 4. FAIRTORRENT PROPERTIES

FairTorrent guarantees a high degree of fairness, fast convergence between a leecher's upload rate and its download rate from other leechers, and resilience to strategic peers. The theoretical analysis of these properties is beyond the scope of this paper, but we give a brief theoretical intuition behind these properties. In addition, FairTorrent incurs low overhead. The most expensive step in a FairTorrent implementation is the re-insertion of a peer inside the *SortedPeerList*, a step that runs in $O(\log n)$.

## 4.1 Fairness and Service Error

To measure fairness, we borrow the terminology of *service error* from the scheduling literature [3, 26, 7]. Service error measures the difference between the optimal fair share and the actual service received by a schedulable entity. In the context of P2P, we use service error to measure the maximum difference between the number of bytes a peer has contributed and received from other leechers at any time during a download session. The smaller the absolute maximum service error experienced by any leecher, the higher the fairness of the algorithm. We separate out the maximum *positive* service error: $E^+_{\max} = \max_i \sum_j DF_{ij}$, and the maximum *negative* service error: $E^-_{\max} = \max_i \sum_j -DF_{ij}$. For regular users, whose objective is to download as fast as possible, it is more interesting to consider the bound on $E^+_{\max}$, which bounds the bytes unreciprocated by other peers. The bound on $E^-_{\max}$ is also interesting as it bounds the maximum number of bytes that a free-rider can obtain.

For $n$-node network, for a wide range of upload rate distributions, and under a simplifying assumption that users always have data to exchange, we can prove that a peer in FairTorrent never incurs $E^+_{\max}$ or $E^-_{\max}$ of more than $O(\log n)$ data blocks with high probability. In this paper, we focus on empirical results and give only a brief proof sketch of one theorem which bounds the service error.

THEOREM 1. *In an n-node FairTorrent network, where upload rates are selected uniformly at random from the range* $[0, r]$*, and for an experiment lasting up to* $O(n^{k_1})$ *rounds, for some positive constants* $k_1, k_2, c$*,*

$$\Pr[E^+_{\max}, E^-_{\max} > c \log n] < n^{k_1} e^{-k_2 n},$$

*where a round constitutes a time interval long enough for at least one peer to send a packet.*

*Proof Sketch:* We first show that in just $O(1)$ rounds the system reaches a convergent state, where half the total upload capacity is used to reciprocate "owed" blocks, and thus, the total number of "owed" blocks, $L$, does not grow. Moreover, we show that $L = O(n)$ with probability at least $1 - e^{-k_2 n}$. In such a a state, on expectation, a single node owes just $O(1)$ blocks. Applying Chernoff Bounds, we show that no peer owes or is owed more than $O(\log n)$ blocks with high probability. Applying a simple union bound we show that the state persists over $O(n^{k_1})$ rounds with probability at least $1 - n^{k_1} e^{-k_2 n}$ □

## 4.2 Fast Convergence and High Utilization

Fast convergence follows directly from the bounds on the service error. Since a leecher $L_i$ never contributes more than $O(\log n)$ blocks than what it receives, as soon as the peer sends a block to $O(\log n)$ peers, it will start to obtain full reciprocation. Thus, the maximum time to reach full reciprocation is the time it takes a peer to send $O(\log n)$ blocks.

FairTorrent, by definition, has high utilization because the algorithm deterministically tells the peer to send a block as long as it has a peer to send the block to. Because of the bounds on $E^+_{\max}$, a peer that wants to download faster has an incentive to upload up to capacity and be assured that its downloaded data will lag by at most $O(\log n)$ blocks.

## 4.3 Resilience to Strategic Peers

Resilience to strategic peers can also be deduced from the bounds on the service error. Regardless of the strategy, even a peer that uploads nothing will receive at most $O(\log n)$ free blocks without contribution.

**Generic Strategy.** There is an additional intuition for the resilience to a generic strategy that comes from the definition of FairTorrent. In FairTorrent each strategic peer $L_i$ competes with every peer $L_j$ for the bandwidth of a peer $L_k$. More formally, as long as $DF_{kj} < DF_{ki}$, $L_k$ will always prefer to send a block to $L_j$ before it sends a block to $L_i$. Thus, regardless of any generic upload behavior chosen by $L_i$, as long as it has reciprocated less to $L_k$ than $L_j$ has, it will always lose the contest for $L'_k s$ bandwidth to $L_j$.

**Whitewashing.** A *whitewasher* is a peer that free-rides, then leaves and reenters the system with a new identity and

zero deficit to try to clear its debt. Whitewashing is a difficult problem for any P2P system. FairTorrent does offer some protection against whitewashing, as each time a free-riding peer enters the system, it will be limited in the number of free blocks that it can get from other leechers, $O(1)$ on expectation and $O(\log n)$ in the worst-case for many capacity distributions. One way to discourage multiple reentries in FairTorrent would be for the seeds to allocate less bandwidth to new peers, making it uneconomical for the leechers to leave and reenter.

# 5. EXPERIMENTAL RESULTS

We implemented FairTorrent (FT) in both the original BitTorrent Python client, which implements the documented version of the BitTorrent protocol, and in the popular Azureus Java client. Each implementation was only a couple hundred lines of code, demonstrating that FairTorrent is simple to implement. We ran experiments on PlanetLab [23] to compare FairTorrent in a realistic wide-area network environment against four other BitTorrent implementations: (1) original BitTorrent 3.9.1 (BT), the code base used for our FairTorrent Python implementation, (2) Azureus 3.0.4.2 (AZ), the code base used for our FairTorrent Java implementation, (3) PropShare 1.1.1 (PS), the latest PropShare version available, and (4) BitTyrant 1.1.1 (TY) which acts strategically towards non-BitTyrant clients and uses block-based TFT with other BitTyrant peers. To measure fairness and performance, each client logged its instantaneous maximum positive and negative service error ($E^+_{\max}$ , $E^-_{\max}$ ), bytes uploaded and download from each peer, and its download completion time.

We present results for four sets of experiments: (1) clients with upload capacities across a uniform distribution which begin downloading simultaneously and remain as seeds in the system upon download completions, (2) a high contributing peer participating in a swarm with low contributors and free-riders, all of which begin downloading simultaneously and remain as seeds in the system upon download completions, (3) clients with upload capacities based on a distribution from live BitTorrent networks which join the system asynchronously and leave upon download completion, and (4) live BitTorrent swarms. A 32 MB target file was used for all experiments except for the live BitTorrent swarms. The file size corresponds roughly to a short movie/music video, but was also small enough to allow us to run a large number of experiments on a consistent set of PlanetLab nodes despite PlanetLab instabilities. Unless otherwise indicated, all clients were configured with their respective default number of simultaneous connections, 50 for Azureus and FairTorrent, up to 80 for BitTorrent, and 500 for PropShare and BitTyrant. Except for the live BitTorrent swarms, FairTorrent Java client and Python client results were similar, so only the Python client results are shown due to space constraints.

## 5.1 Uniform Distribution

The uniform distribution represents a wide range of peers with diverse upload capacities participating in a swarm. We used a network of 50 leechers and 10 seeds. A small seed to leecher ratio, 1:5, was used to show that the results do not depend on high data availability; the average ratio in live BitTorrent swarms is 1:1 [22]. All nodes were configured with download bandwidth capacities of 100 KB/s and upload bandwidth capacities of no more than 50 KB/s, reflecting a typical scenario where most ISPs allow users a download rate at least twice their allowed upload rate. Each leecher received an upload capacity randomly selected between 1 and 50 KB/s. Seeds were configured with upload bandwidth capacities of 25 KB/s each, chosen to match the average upload bandwidth capacity of a leecher.

We ran five experiments with five different sets of upload capacities generated randomly from the respective distribution, resulting in 250 leecher measurements. To first show the fairness and performance of each client in isolation, we ran the same set of experiments for a homogeneous network of each client. Since PropShare turns off "seeding" capability by default, we changed this default to allow its clients to seed to enable PropShare to work in a homogeneous network.

Figures 3 to 12 captioned with a prefix U show the uniform distribution results. Figures 3 to 7 show the average upload rate versus the average download rate from other leechers experienced by each leecher during its download. A reference line $y = x$ is also shown. for the ideal case, in which the download rate should equal the upload rate. Figure 3 shows that FairTorrent closely matches each peer's upload rate with the average download rate from other leechers. Figures 4 to 7 show that BitTorrent, Azureus, PropShare, and BitTyrant are unable to match each peer's upload rate with the average download rate from other leechers. Higher contributing peers in Azureus deviate more than lower contributing peers from their ideal fair allocation. Even worse, higher contributing peers in BitTorrent, and even more so in PropShare and BitTyrant, are likely to receive a download rate far below their contribution, while lower contributing peers receive a higher level of service than their contribution. Unlike FairTorrent, these systems are limited by their round-based rate allocation, where it may take many rounds for a high-contributing peer to find a more favorable set of peers to exchange data with. Previous claims that block-based TFT improves fairness versus BitTorrent were based on an unrealistic simulation-only study limited to 3 classes of upload capacities, where even high-uploaders can quickly discover like-uploading peers [5]. Using real peers in a real network, Figure 7 instead shows that block-based TFT as used in BitTyrant results in the worst overall fairness and worst performance for higher contributing peers, which suffer from underutilization due to hard limits on peerwise deficits. In contrast, Figure 3 shows that all higher contributing peers in FairTorrent are resilient to low contributing peers and are able to obtain a matching leecher download rate.

Figures 8 to 9 show how quickly peers obtain rate convergence. They plot a CDF for each system which demonstrates how quickly a fraction of peers converges to a download rate of at least 90% of their upload capacity, e.g., how quickly does a 50 KB/s peer begins to get at least 45 KB/s rate from other leechers. Figure 8 shows rate convergence for high-uploading peers, those with capacities of 40 to 50 KB/s. 100% of high-uploading peers in FairTorrent reach this convergence, as compared to 59%, 78%, 16% , and 62% for BitTorrent, Azureus, PropShare, and BitTyrant, respectively. Moreover, all the high-uploading peers in FairTorrent reach convergence within 15 seconds from the start of the experiments, demonstrating FairTorrent's fast convergence property. In other systems, it takes at least 800 seconds for the highest fraction of peers to reach convergence. PropShare has the worst rate convergence of all systems because

it fails to estimate neighbors' rate allocations accurately as it exchanges data only with a subset of peers in each round. In addition, as each peer in Azureus updates its active set of peers to which it uploads, the convergence process between peers is often interrupted and reset.

Figure 9 shows rate convergence for all peers. Including free-riders and low contributors, 85%, 73%, 76%, 72%, and 71% of peers reach rate convergence in FairTorrent, BitTorrent, Azureus, PropShare, and BitTyrant, respectively. For some low-contributing peers, the BitTorrent protocol overhead represents more than 10%, reducing the effective data upload rate to be below 90% of capacity, and thus they do not reach the 90% convergence even in FairTorrent. However, Figure 3 shows that even low contributing peers in FairTorrent obtain a matching effective download rate. FairTorrent not only has the highest fraction of peers that reach convergence, but also has an order of magnitude faster convergence than other systems, which take over 900 seconds for the highest fraction of leechers to converge.

Figure 10 shows the maximum positive service error $E^+_{\max}$ and maximum negative service error $E^-_{\max}$ for each of the four networks. For example, $FT+$ and $FT-$ denote $E^+_{\max}$ and $E^-_{\max}$ for FairTorrent. A range of percentiles is shown for each network, starting with the 50th percentile, the median maximum service error across all leechers for a network, and up to the 100th percentile, the worst maximum service error. FairTorrent has one to two orders of magnitude smaller error than all other systems. The median FairTorrent $E^+_{\max}$ was just 79 KB. The maximum FairTorrent $E^+_{\max}$ was 436 KB, meaning that during the entire download of the file, no FT leecher gives more than 436 KB of service than what it receives from other leechers. This is 18 to 73 times smaller than the maximum $E^+_{\max}$ of other networks. The maximum $E^+_{\max}$ for Azureus and BT was over 8 MB, more than 25% of the entire 32 MB file. $E^+_{\max}$ for PropShare reached 19 MB, or 60% of the file. $E^+_{\max}$ for TY reached 31 MB, or almost 100% of the file. PropShare, which tries to provide proportional allocation using rate estimates, has the worst fairness of all systems other than BitTyrant. In contrast, FairTorrent provides significantly better fairness than all other clients, which results in fast rate convergence and performance more correlated with a peer's contribution.

Figure 11 shows the average and maximum download times for the peers in each network. For each network, the left bar, labeled with letter "H", shows download times for the high-uploading leechers, those with 40 to 50 KB/s capacities, and the right bar shows download times for all leechers. For the high-uploading leechers, the average download times were 690, 733, 745, 1027 and 963 seconds for FairTorrent, BitTorrent, Azureus, PropShare, and BitTyrant, respectively. The worst download times for the high-uploaders were 756, 876, 980, 1200, and 1298 seconds for FairTorrent, BitTorrent, Azureus, PropShare, and BitTyrant, respectively. Because of faster rate convergence and tighter maximum deficit, the FairTorrent high-uploading peers receive a more fair reciprocation for their contribution.

One might think that the better performance for high-contributing FairTorrent peers comes at the expense of the low-contributing ones. This is not the case. Figure 11 shows the worst download times were 1347, 1892, 1849, 1758, and 2266 seconds for FairTorrent, BitTorrent, Azureus, PropShare, and BitTyrant, respectively, with FairTorrent 31% to 68% faster. There are several reasons that FairTorrent performance for lower contributors does not suffer. First, FairTorrent guarantees a certain level of performance to each leecher. Each leecher receives a matching leecher download rate plus a constant fraction of the seed bandwidth. Thus, a leecher does not get stuck because of unlucky initial peer selection, slow download times, or poor seed allocation. Second, FairTorrent has higher bandwidth utilization.

The aggregate bandwidth utilization was 95.3%, 93.7%, 89.8%, 88.7% and 82.8% for FairTorrent, BitTorrent, Azureus, PropShare and BitTyrant, respectively. Part of the under-utilization was the BitTorrent protocol overhead which accounts for roughly 3% for each system. FairTorrent obtains better utilization for two reasons: (1) FairTorrent does not set an arbitrary constraints such as BitTyrant's block-based TFT. (2) As long as there is a request from any neighbor FairTorrent will upload a block, making it less likely that a leecher runs out of blocks to send to its *active set* of peers.

Figure 12 shows the standard deviation in download rate observed by each leecher, as measured over consecutive 15 second intervals. Data points representing FairTorrent are captured by the lowest layer on the graph, clearly separated from the rest. The average standard deviations in leecher download rates were 1.8 KB/s, 6.0 KB/s, 8.0 KB/s, 6.7 KB/s and 12.3 KB/s for FairTorrent, BitTorrent, Azureus, PropShare, and BitTyrant, respectively, with FairTorrent's standard deviation more than 3 times smaller than the next closest system. Because of its fast and more precise rate converegence, FairTorrent is able to maintain a download rate close to its upload contribution throughout the download. The low standard deviation makes FairTorrent more amenable for use in P2P live streaming systems, where achieving and maintaing a steady download rate of stream updates is critical for good performance.

## 5.2 Skewed Distribution

To show what happens when a peer is surrounded by others that do not contribute much, we ran a single high uploader in a swarm of many low-contributing peers. We ran the same experiments with the same network of leechers and seeds as in Section 5.1, but with a *skewed* distribution of upload capacities. The high uploader received an upload capacity of 50 KB/s, and the 49 low contributors received randomly selected upload capacities between 1 and 5 KB/s. In addition to running experiments for homogeneous networks, we ran the same experiments for non-FairTorrent networks in which the high uploader was replaced by a FairTorrent client to show how FairTorrent performs in the presence of low contributors that are not FairTorrent clients. These results are denoted by FT/BT, FT/AZ, FT/PS, and FT/TY for the respective BitTorrent, Azureus, PropShare, and BitTyrant networks.

Figures 13 to 14 captioned with a prefix S show the skewed distribution results. Figure 13 shows the maximum and the average download times for each network, with the left bar, labeled with letter "H", for the high-uploader, and the right bar for the entire set of peers. The high uploader in FairTorrent completed its download in 644 seconds on average, 3 to 5 times faster than the high uploader in BitTorrent, Azureus, PropShare or BitTyrant. This is because the FairTorrent high uploader achieved an averaged download rate of 47.9 KB/s that closely matched its average upload rate of 48.3 KB/s. When FairTorrent replaced the high uploader in other systems, it significantly improves download times.

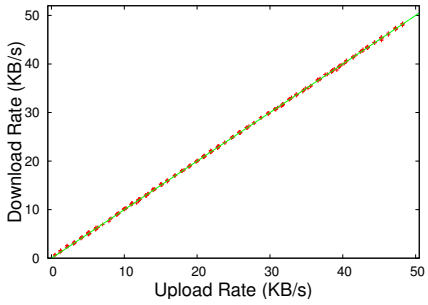


**Figure 3: U: FairTorrent fairness**

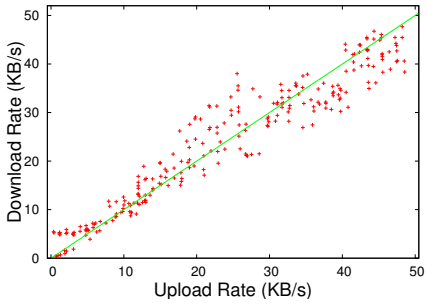


**Figure 4: U: BitTorrent fairness**

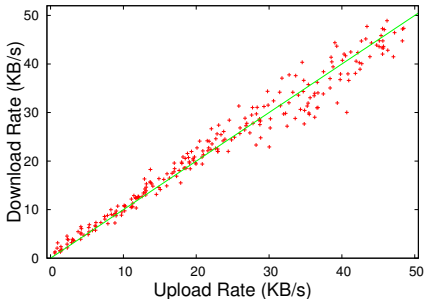


**Figure 5: U: Azureus fairness**

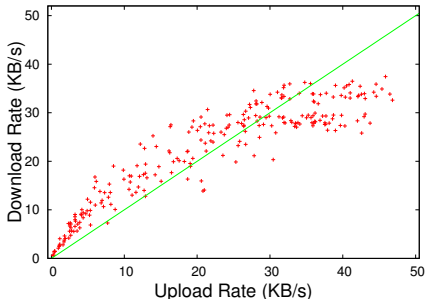


**Figure 6: U: PropShare fairness**

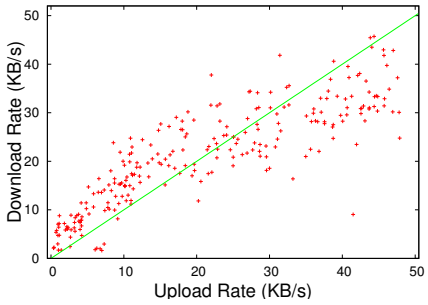


**Figure 7: U: BitTyrant fairness**

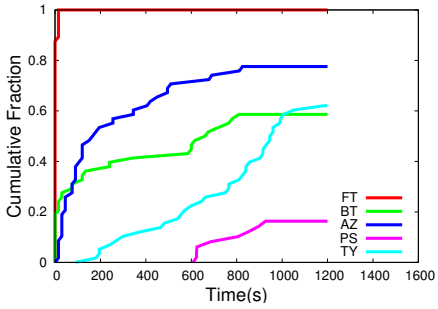


**Figure 8: U: High uploader rate convergence**

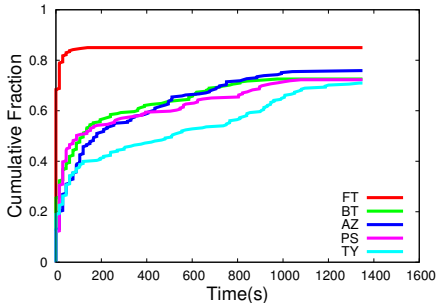


**Figure 9: U: Rate convergence**

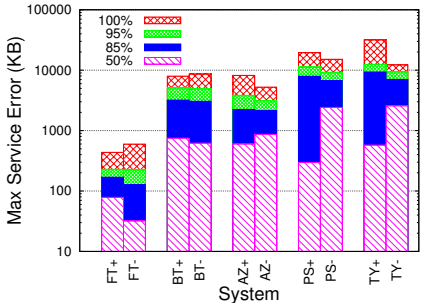


**Figure 10: U: $E^+_{\max}$ and $E^-_{\max}$**

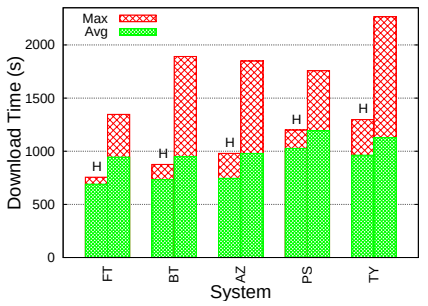


**Figure 11: U: Download time**

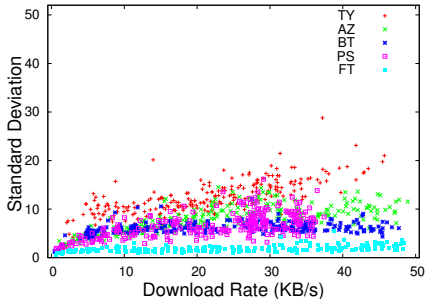


**Figure 12: U: Standard deviation of the download rate**

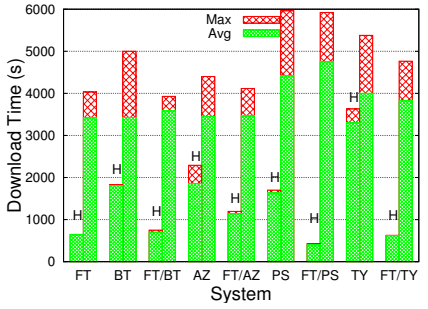


**Figure 13: S: Download time**

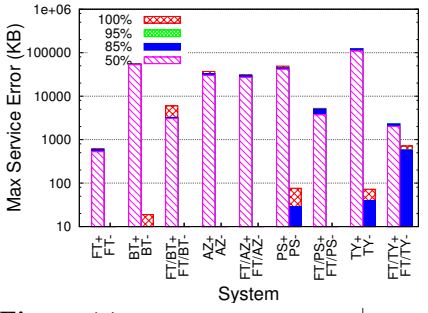


**Figure 14: S: High uploader $E^+_{\max}$ and $E^-_{\max}$**

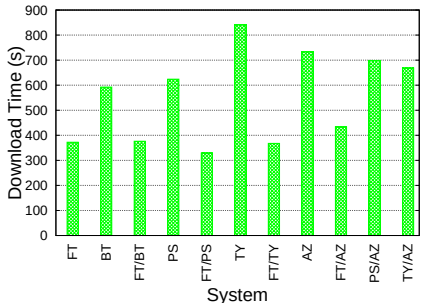


**Figure 15: D: High uploader download time**

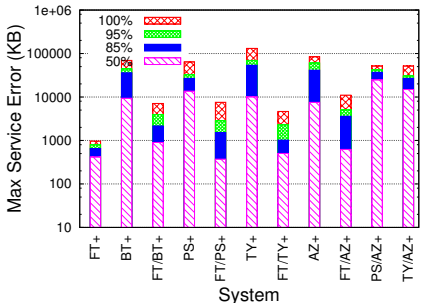


**Figure 16: D: High uploader $E^+_{\max}$**

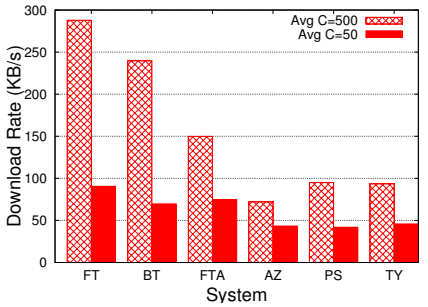


**Figure 17: Live download rate**

FairTorrent reduces download times from 1804 to 703 seconds comparing BitTorrent to FT/BT, from 1859 to 1138 seconds comparing Azureus to FT/AZ, from 1633 to 402 seconds comparing PropShare to FT/PS, and from 3305 to 615 seconds comparing BitTyrant to FT/TY. The improvement is possible because FairTorrent adapts to the upload rates of the surrounding low uploaders and is therefore able to get a high leecher download rate, even though the low uploaders do not run FairTorrent. These results motivate the adoption of FairTorrent, especially for high-uploading users.

Figure 14 shows the $E^+_{\max}$ and $E^-_{\max}$ of the high uploader in each system. $E^+_{\max}$ always dominates $E^-_{\max}$ as the high uploader typically serves more data than it receives in the skewed distribution. However, $E^+_{\max}$ for FairTorrent is 60 to 200 times, roughly two orders of magnitude, smaller than other systems. $E^+_{\max}$ of the high uploader was measured to be 555 KB for FairTorrent, as compared to 51 MB, 31 MB, 49 MB, and 113 MB for BitTorrent, Azureus, PropShare and BitTyrant, respectively. When FairTorrent replaced the high uploader in other systems, it still reduces $E^+_{\max}$. FairTorrent reduces $E^+_{\max}$ by a factor of 15 comparing BitTorrent to FT/BT, by 10% comparing Azureus to FT/AZ, by a factor of 10 comparing PropShare to FT/PS, and by a factor of 50 comparing BitTyrant to FT/TY.

The smaller fairness improvement for the high uploader in FT/AZ only tells part of the story, as FairTorrent improves the high-uploader capacity utilization from 55% to 97%, and its leecher download rate from 9.8 to 23 KB/s. The poor behavior of the high-uploading Azureus client is due to a slightly different unchoking behavior of Azureus, which is less aggressive than BitTorrent, and often sticks with other low-contributors rather than switching to the high-uploader. When FairTorrent replaces the high-uploader, it is able to improve the overall utilization by unchoking and interacting with more peers at a time.

## 5.3 Dynamic Live Distribution

The dynamic live distribution represents peers with upload capacities taken from live torrents, which asynchronously join and leave, which do not seed upon completion if they are leechers, and participate in a larger, non-mesh network. We used a larger network of 100 leechers and 20 seeds, and configured the upload capacities of the leechers based on a distribution of the upload capacities in live BitTorrent networks [21]. Capacities were also scaled down by a factor of 10 to allow PlanetLab to handle the capacity of the highest uploaders [21]. The distribution resulted in upload rates in the range of 4 to 197 KB/s with a mean of 17 KB/s; live capacities represent a very skewed distribution. Each of the 20 seeds maintained a 25 KB/s upload rate as in earlier experiments. We increase the leechers download bandwidth to 220 KB/s to accommodate peers with higher upload rates. To create dynamic conditions, leechers entered the swarm with randomly picked time offsets such that one leecher would enter every 5 seconds. Upon completion of the download, the leecher would leave the system and re-enter immediately with a clean cache and a new client ID. Each experiment ran for 2000 seconds, allowing clients on some machines to re-enter and complete the download multiple times. We began taking measurements 500 seconds into the experiment, once the last of the 100 leechers joined the swarm.

In this live distribution, peers with the top 10% of the capacities accounted for 50% of the leecher bandwidth. We focus on the fairness and performance of these top 10% of contributing peers in our measurements to show why they have a strong incentive to run FairTorrent. As in Section 5.2, we ran experiments for homogeneous networks and non-FairTorrent networks in which the contributing peers used FairTorrent to show how they perform in the presence of low contributors that are not FairTorrent clients. As a counterpoint to previous work [21, 14], we also ran experiments with the contributing peers using PropShare or BitTyrant inside Azureus, denoted by PS/AZ and TY/AZ, respectively.

Figures 15 to 16 captioned with a prefix D show the dynamic live distribution results. Figure 15 shows the average download times for high contributors for the homogeneous FairTorrent, BitTorrent, Azureus, PropShare, and BitTyrant networks were 372, 593, 733, 624, and 842 seconds, respectively. Because of more optimal fairness and fast rate convergence, FairTorrent outperformed BitTorrent, Azureus, PropShare, and BitTyrant by 37%, 49%, 40%, and 56%, respectively. BitTorrent performed somewhat better than Azureus because its higher number of connections made it more likely to find other higher uploading peers among its neighbors. However, even though BitTyrant and PropShare use 500 connections creating a mesh, they were not able to outperform other clients due to poor rate convergence.

When FairTorrent replaces the top 10 leechers in each network, it significantly reduce the download times to 376, 434, 330, and 368 seconds for FT/BT, FT/AZ, FT/PS, and FT/TY, respectively. It successfully reduces download times by 37% to 56% because FairTorrent leechers are able to quickly adjust to the reciprocation rates of both FairTorrent and non-FairTorrent leechers. For FT/TY, FairTorrent achieves download times similar to those in a homogeneous FairTorrent network despite being surrounded by 90% of strategic BitTyrant clients. FairTorrent is resilient against strategic peers because it only rewards its peers fairly for their contributions. Furthermore, FairTorrent avoids the long discovery times of BitTorrent and slow and imperfect convergence of PropShare. This provides great incentive for high contributors in live networks to adopt FairTorrent. Since Azureus is the most common P2P client, we compared the effect of FairTorrent (FT/AZ), PropShare (PS/AZ), and BitTyrant (TY/AZ) clients that replace high contributors inside Azureus. While FairTorrent reduces download times by 40%, Figure 15 shows that PropShare and BitTyrant reduce the download times by only 5% and 9%, respectively, leaving users with much stronger incentive to adopt FairTorrent. Comparing PS/AZ with the all-PropShare network shows that increasing the number of PropShare peers does not improve performance. Even worse comparing TY/AZ with the all-BitTyrant network shows that increasing the number of BitTyrant peers degrades performance. In contrast, comparing results for FT/AZ with the all-FairTorrent network shows that increasing the number of FairTorrent peers only further improves performance.

Figure 16 shows that FairTorrent achieves much better fairness which explains its better performance for high contributors. Only $E^+_{\max}$ is shown as $E^-_{\max}$ is significantly smaller for the high uploaders in all cases. FairTorrent maintains its maximum service error under 1 MB. In contrast, the maximum service error for BitTorrent, Azureus, PropShare, and BitTyrant was 53 MB, 102 MB, 64 MB, and 72 MB, respectively. The reason that $E^+_{\max}$ for these clients increases dramatically in the dynamic scenario is due to joins and

leaves that affect the rate convergence. FairTorrent clients that replace high contributors in other systems also reduce $E_{\max}$ by an order of magnitude due to fast convergence.

## 5.4 Live Swarms

Live swarms consisted of forty popular live BitTorrent swarms that distributed large files of 1 to 10 GB to thousands of users. For each swarm, we had all clients join it at the same time, but configured our clients not to talk to one another. We capped the upload and download rates of each client at 300 KB/s and 600 KB/s respectively. To avoid overusing bandwidth and memory resources on PlanetLab and avoid potential copyright infringement issues with live swarms, we did not download entire files but instead joined each swarm for 1500 seconds and measured the download rate obtained by the clients. For live swarms, we observed that for each type of client, performance was highly correlated with the number of configured connections. As a result, we ran two sets of tests, first configuring each client with a 500 connection limit, the default for PropShare and BitTyrant, and then, configuring each client with 50 connections, the default for Azureus.

Figure 17 shows the average download rate obtained by each client when configured with 500 and 50 connections. FTA denotes our FairTorrent Java client implementation in Azureus. With the 500 connection limit, the average download rates were 288 KB/s, 240 KB/s, 150 KB/s, 72 KB/s, 95 KB/s, and 94 KB/s for FairTorrent, BitTorrent, FTA, Azureus, PropShare, and BitTyrant, respectively. FairTorrent outperforms all other clients by 20% to 300%. With the 50 connection limit, the average download rates were 91 KB/s, 70 KB/s, 75 KB/s, 43 KB/s, 42 KB/s, and 46 KB/s for FairTorrent, BitTorrent, FTA, Azureus, PropShare, and BitTyrant, respectively. FairTorrent outperforms all other other clients by 21% to 116%. FairTorrent and BitTorrent significantly outperformed other clients. The main reason for this was that the BitTorrent Python code implemented better timeout behavior with peers with which it could not complete a handshake, thus obtaining more active connections over time. This behavior was not a factor in the non-live tests where all connections were completed.

To avoid comparing connection management behavior, we compare clients based only on the Azureus implementation. FairTorrent as denoted by FTA still outperforms Azureus, PropShare and BitTyrant by 58% to 108% with the 500 connection limit, and by 63% to 79% with the 50 connection limit. In live networks, FairTorrent again quickly adopts to its neighbors and is more successful at retaining both low and high uploaders as data suppliers. Furthermore, the plain Azureus client had very similar performance to both PropShare and BitTyrant when all clients had the same 50 connection limit, suggesting that PropShare and BitTyrant may not provide improvement in smaller swarms. When all clients use 500 connections, Azureus is only 30% worse than PropShare and BitTyrant. It appears that previously reported results [14] comparing PropShare and BitTyrant to Azureus did not normalize for the number of connections, but instead used a 500 connection limit default for PropShare and BitTyrant while comparing against Azureus with its default 50 connection limit.

# 6. CONCLUSIONS AND FUTURE WORK

FairTorrent is a new fully distributed P2P algorithm that accurately provides each peer with fair service commensurate with its bandwidth contribution. FairTorrent's deficit-based algorithm avoids the pitfalls of previous approaches that suffer from slow peer discovery, inaccurate bandwidth estimates, bandwidth under-utilization and complex tuning of parameters. FairTorrent does not require bandwidth estimates, a centralized system, peer reputation, or third-party credit-keeping services. We compared FairTorrent against BitTorrent, Azureus, PropShare and BitTyrant. We have shown that due to its high-degree of fairness, compared with other P2P systems, FairTorrent is able to provide much better performance for contributing peers in a number of scenarios: 31% to 68% better performance in the uniform distribution, a 3 to 5 times improvement for a high uploader in a skewed distribution, 37% to 56% better performance for high contributors in a dynamic scenario with live capacities, and 60% to 100% better performance in live swarms. FairTorrent is resilient to free-riders, low contributors and strategic peers in both FairTorrent and non-FairTorrent networks. When replacing high contributors in the most popular Azureus network, FairTorrent improves the performance of both high contributors as well as the entire system, suggesting that FairTorrent is amenable to gradual adoption by users. We believe that the high fairness and performance guarantees of FairTorrent provide a strong foundation for developing more reliable and robust P2P services.

One important and increasingly popular P2P service is P2P streaming. For good performance, a streaming client must achieve and maintain a steady download rate of stream updates, a problem that is especially difficult in the presence of free-riders. Previous approaches for P2P file-sharing incur long peer discovery times and imprecise and slow rate convergence, making them poor candidates for supporting streaming. The fast rate convergence of FairTorrent enables it to achieve and maintain fair bandwidth exchange even over short time intervals, characteristics that would be very beneficial for P2P streaming. Extending FairTorrent to support streaming is a promising direction for future work.

# 7. ACKNOWLEDGMENTS

Nikolaos Laoutaris provided helpful comments on earlier drafts of this paper. PlanetLab [23] was used for all of our experiments. This work was supported in part by NSF grants CNS-0426623, CNS-0905246, CCF-0728733 and AFOSR MURI grant FA9550-07-1-0527.

# 8. REFERENCES

[1] A. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J. Martin, and C. Porth. BAR Fault Tolerance for Cooperative Services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Oct. 2005.

[2] Azureus. http://www.azureus.com/.

[3] J. C. R. Bennett and H. Zhang. WF2Q: Worst-case Fair Weighted Fair Queuing. In *Proceedings of the 15th IEEE International Conference on Computer Communications (INFOCOM '96)*, Mar. 1996.

[4] K. Berer and Z. Despotovic. Managing Trust in a Peer-to-Peer Information System. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management*, Nov. 2001.

[5] A. Bharambe, C. Herley, and V. Padmanabhan. Analyzing and Improving a BitTorrent Network's Performance Mechanisms. In *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM '06)*, Apr. 2006.

[6] B. Cohen. Incentives Build Robustness in BitTorrent. In *Proceedings of the 1st Workshop on the Economics of Peer-to-Peer Systems*, June 2003.

[7] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM '89)*, Sept. 1989.

[8] eMule. http://www.emule-project.net/.

[9] B. Fan, D. Chiu, and J.Lui. The Delicate Tradeoffs in BitTorrent-like File Sharing Protocol Design. In *Proceedings of the 14th IEEE International Conference on Network Protocols (ICNP '06)*, Nov. 2006.

[10] M. Ham and G. Agha. ARA: A Robust Audit to Prevent Free-Riding in P2P Networks. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing (P2P '05)*, Aug. 2005.

[11] S. Jun and M. Ahamad. Incentives in BitTorrent Induce Free-riding. In *Proceedings of the 3nd Workshop on the Economics of Peer-to-Peer Systems*, Aug. 2005.

[12] S. Kamvar, M. Schlosser, and H. Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *Proceedings of the 12th International World Wide Web Conference (WWW '03)*, May 2003.

[13] A. Legout, N. Liogkas, E. Kohler, and L. Zhnag. Clustering and Sharing Incentives in BitTorrent Systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, June 2007.

[14] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. BitTorrent is an Auction: Analyzing and Improving BitTorrent's Incentives. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, (SIGCOMM '08)*, Aug. 2008.

[15] H. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, and M. Dahlin. FlightPath: Obedience vs Choice in Cooperative Services. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, Dec. 2008.

[16] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR Gossip. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Nov. 2006.

[17] Q. Lian, Y. Peng, M. Yang, Z. Zhang, Y. Dai, and X. Li. Robust Incentives via Multi-level Tit-for-tat. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS '06)*, Feb. 2006.

[18] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *Proceedings of the 5th Workshop on Hot Topics in Networks (HotNets '06)*, Nov. 2006.

[19] T. Ngan, D. Wallach, and P. Druschel. Enforcing Fair Sharing of Peer-to-Peer Resources. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[20] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[21] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do Incentives Build Robustness in BitTorrent. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, Apr. 2007.

[22] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. One Hop Reputations for Peer to Peer File Sharing Workloads. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, Apr. 2008.

[23] Planetlab. http://www.planetlab.org/.

[24] D. Qiu and R.Srikant. Modeling and Performance Analysis of BitTorrent-Like Peet-to-Peer Networks. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, (SIGCOMM '04)*, Sept. 2004.

[25] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation Systems. *Communications of the ACM*, 43(12):45–48, Dec. 2000.

[26] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round-Robin. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, (SIGCOMM '95)*, Aug. 1995.

[27] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free Riding in BitTorrent Networks with the Large View Exploit. In *Proceedings of the 6th International Workshop on Peer-to-Peer Systems (IPTPS '07)*, Feb. 2007.

[28] M. Sirivianos, X. Yang, and S. Jarecski. Dandelion: Cooperative Content Distribution with Robust Incentives. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX '07)*, June 2007.

[29] K. Tamilmani, V. Pai, and A. Mohr. SWIFT: A System with Incentives for Trading. In *Proceedings of the 2nd Workshop on the Economics of Peer-to-Peer Systems*, June 2004.

[30] S. Tewari and L. Kleinrock. On Fairness, Optimal Download Performance and Proportional Replication in Peer-to-Peer Networks. In *Proceedings of the 4th International IFIP-TC6 Networking Conference (NETWORKING '05)*, May 2005.

[31] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. Karma: A Secure Economic Framework for Peer-To-Peer Resource Sharing. In *Proceedings of the 1st Workshop on the Economics of Peer-to-Peer Systems*, June 2003.

[32] F. Wu and L. Zhang. Proportional Response Dynamics Leads to Market Equilibrium. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC '07)*, June 2007.