FairTorrent: Bringing Fairness to Peer-to-Peer Systems

Alex Sherman, Jason Nieh, and Cliff Stein
Department of Computer Science
Columbia University

Technical Report CUCS-029-08
May, 2008
{asherman, nieh, cliff}@cs.columbia.edu

Abstract

The lack of fair bandwidth allocation in Peer-to-Peer systems causes many performance problems, including users being disincentivized from contributing upload bandwidth, free riders taking as much from the system as possible while contributing as little as possible, and a lack of quality-of-service guarantees to support streaming applications. We present FairTorrent, a simple distributed scheduling algorithm for Peer-to-Peer systems that fosters fair bandwidth allocation among peers. For each peer, FairTorrent maintains a deficit counter which represents the number of bytes uploaded to a peer minus the number of bytes downloaded from it. It then uploads to the peer with the lowest deficit counter. FairTorrent automatically adjusts to variations in bandwidth among peers and is resilient to exploitation by free-riding peers. We have implemented FairTorrent inside a BitTorrent client without modifications to the BitTorrent protocol, and compared its performance on PlanetLab against other widely-used BitTorrent clients. Our results show that FairTorrent can provide up to two orders of magnitude better fairness and up to five times better download performance for high contributing peers. It thereby gives users an incentive to contribute more bandwidth, and improve overall system performance.

1 Introduction

In the past decade, the usage of Peer-to-Peer (P2P) file-sharing applications on the Internet has experienced explosive growth. Many users and businesses now rely on P2P file-sharing for distributing videos, software, and documents. Although P2P file-sharing is now an integral part of our overall computing experience, these applications are plagued by a fundamental problem of unfairness in how bandwidth among peers is used and allocated. Unfairness causes many performance problems, including users being disincentivized from contributing upload bandwidth, a growing number of *free riders*, users who cap their upload bandwidth to zero or a small value to take as much as possible from the system while contributing little resources, and a lack of quality-of-service guarantees to support streaming applications.

Fair bandwidth allocation in P2P systems can be difficult to achieve for several reasons. First, bandwidth resources belong to and are controlled by individual peers, not by a single party. Unlike a router or a server, there is no central entity that controls and arbitrates access to all resources. Second, bandwidth resources are distributed all over the Internet, and therefore vary geographically and also are limited by the various Internet Service Providers. Third, the amount of bandwidth resources available is not known in advance and peers cannot be relied upon to specify their own resources honestly. Fourth, bandwidth resources may vary over time for several reasons, including network conditions, mobile peers connecting at different access points, and a user using the available bandwidth for other activities. Finally, any fair allocation mechanism must be strong enough to withstand attempts by free riders to manipulate the system.

Of course, fairness is not the only desirable property in a peer-to-peer system, users also desire good performance, measured by download time. We show that improving fairness leads to improved performance, sometimes, with substantial gains.

Many approaches have attempted to address the problem of fair bandwidth exchange. One approach, which is employed by the popular file-sharing system BitTorrent, is to use a tit-for-tat (TFT) heuristic. However, recent studies [27, 21, 15] have demonstrated weaknesses in the BitTorrent TFT mechanism that can be exploited by free-riding clients to take advantage of high-bandwidth contributors. For example, the LargeView exploit client [27] takes advantage of the fact that BitTorrent uses a fraction of its bandwidth to upload optimistically to randomly chosen peers. By opening connections to many peers, the LargeView client allows a large number of its neighbors to pick it as the target for their optimistically-spent band-

width. As another example, the BitTyrant client [21] is based on the observation that even though BitTorrent attempts to find reciprocating neighbors, it is often willing to upload to peers who reciprocate at a much lower rate. BitTyrant attempts to find peers where it can maximize the difference between what it receives from them versus what it uploads to them.

To address these problems, we present FairTorrent, a new P2P scheduling algorithm that fosters fair bandwidth allocation among peers. In its simplest form, at each peer, FairTorrent maintains a *deficit* counter for each neighboring peer which represents the number of bytes it uploaded to this neighbor minus the number of bytes it downloaded from it. When it is ready to send a packet of data, FairTorrent identifies the peer with the lowest deficit counter and sends data to that peer.

Our main result is that this surprisingly simple approach provides much better fairness while simultaneously decreasing most download times, often by a significant amount. Download times are also more correlated with the peers' upload rates, thereby incentivizing peers to upload more. FairTorrent has a number of additional useful properties: (1) FairTorrent 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) FairTorrent does not need to measure available download bandwidth, allocate precise upload or download rates for any peers, or rely on estimates or advanced knowledge of available bandwidth from users or other peers. (3) FairTorrent allows a peer to automatically adjust to dynamic network conditions and any changes in the rate at which peers contribute bandwidth, avoiding long discovery times of like-uploading peers as evidenced in BitTorrent [21]. (4) FairTorrent provides low variance in bandwidth allocation, enabling it to be useful for streaming applications. (5) FairTorrent has no magic parameters and requires no tuning, is simple to implement, and requires no changes to the BitTorrent protocol, making it easy to deploy with existing P2P systems.

We have analyzed, implemented, and measured FairTorrent to evaluate its effectiveness. Our analysis results show that FairTorrent runs efficiently, using $O(\log k)$ time per transmission in a network with k peers. We have proved for a small network that the algorithm has fairness bounds independent of the amount of data sent, and conjecture that a similar result holds for any network. We have implemented FairTorrent inside a BitTorrent client without changing the BitTorrent protocol, making it compatible with existing BitTorrent implementations. We evaluated its performance on PlanetLab against three other BitTorrent implementations, the original BitTorrent Python client by Bram Cohen [7], the latest version of the popular Azureus Java BitTorrent client [3], and BitTyrant [21], a modified Azureus 2.5 client that was optimized for improved download performance.

For peers with widely different bandwidths across a uniform distribution, our measurements demonstrate that FairTorrent provides more than an order of magnitude better fairness and up to 50% faster download performance compared to other BitTorrent implementations. Furthermore, FairTorrent makes more efficient use of available upload bandwidth and provides download times that are strongly correlated with the peers' upload capacities, incentivizing peers to upload more. For a high bandwidth uploader in the presence of many low contributors, our measurements demonstrate that FairTorrent can provide two orders of magnitude better fairness and up to five times faster download performance compared to other BitTorrent implementations. We also show that FairTorrent provides improved fairness and performance even if low contributors run other BitTorrent clients, including clients designed to exploit unmodified BitTorrent for better performance.

This paper describes the design, implementation, and evaluation of FairTorrent. Section 2 discusses related work. Section 3 presents FairTorrent, which is analyzed in Section 4. Section 5 presents experimental results on PlanetLab comparing FairTorrent performance to other BitTorrent implementations. Finally, we present some concluding remarks.

2 Background and Related Work

BitTorrent [7] employs a rate-based tit-for-tat (TFT) heuristic to incentivize peers to upload and attempts 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 in size. 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 round-robin fashion, where $N-k$ of the peers have provided the best download rate during the most recent time period, and k peers are randomly selected to help discover other peers with similar upload rates. (N is typically between 5 and 10). The set of peers to which a client uploads is periodically changed based on measurements of their download rates. BitTorrent refers to the selection and deselection of a peer for uploading as *unchoking* and *choking*, respectively.

Much work has been done in studying the behavior of BitTorrent. Qiu and Srikant [23] show that under some

bandwidth distributions, the system eventually converges to a Nash equilibrium. Legout et al. show that BitTorrent peers tend to exchange data primarily with other peers with similar upload rates over a large file download [13]. 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 [15, 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] claims a median 70% performance improvement by sending at a minimum possible rate to its neighbors and observing that BitTorrent peers are willing to altruistically upload at rates higher than what they receive.

These 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. Further waste occurs because connections with discovered peers may be unstable, as the other peers are also always searching for better connections. 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 [1] has been proposed for improving fairness in BitTorrent. Instead of uploading to a small number of peers, block-based TFT enables a client to upload to all the peers in a swarm that are interested in its data, but limits the difference between what it uploads to a peer and what it downloads from that peer to a constant number of blocks. Peers are still selected in round-robin fashion. The hard limit of a constant number of blocks results in under-utilization of the peers' upload capacities. To compensate, they propose using a bandwidth-matching tracker which would match peers with similar bandwidth. However, this solution assumes that peers would not game the system by lying about their upload bandwidth. While the evaluation of block-based TFT was limited to simulation studies [1], another study of a block-based TFT policy showed poor performance and bandwidth under-utilization compared to BitTorrent [28]. SWIFT [2] proposes a model where peers use a block-based reciprocation together with a willingness to donate a small fraction of their bandwidth. It is not clear how to tune their highly-parametrized algorithm [2] beyond the simulation results presented for a realistic deployment scenario.

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 [10]. However, this assumption is not realistic as many leechers remain in the system as seeds in many BitTorrent systems [6]. Furthermore, other work suggests that fairness does not need to come at the expense of performance [29].

Various approaches have explored the use of BitTorrent for streaming applications [31, 22] These approaches focus on identifying peers with which to exchange data that are close to one another in time in processing a stream. This work is complementary to our focus on providing fine-grain fair bandwidth exchange.

Many other approaches outside of the context of BitTorrent have explored different aspects of improving fairness in P2P systems, although mostly focused on dealing with free riders. Reputation-based systems [25, 14, 5, 8] attempt to separate good contributors and free riders by associating reputations with peers. These systems provide a mechanism for selecting peers with good reputations to make it less likely that good peers will exchange data with bad free riders. Such systems suffer from problems with bootstrapping, and collusion [24], 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.

Credit-based systems [30, 19, 18, 16, 17] use virtual credit or micropayments to incentivize fair exchange of services among peers in P2P systems. Virtual credit is typically maintained over many file downloads over many days. When a peer requests a new file, its performance will likely depend on the overall distribution of credits that have been accumulated by all the participants up to that point. These systems are not compatible with commodity P2P systems. They typically require significant overhead as well as trusted third party agents to maintain credit values and verify the services provided. In contrast to BitTorrent, credit-based systems do not optimize finer granularity fairness during the current download of a file. They also cannot support quality-of-service requirements for streaming applications.

The problem of fair bandwidth allocation has perhaps been most studied in the context of scheduling packets through a router. 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. Many scheduling algorithms have been developed to address this problem [9, 20, 12, 4, 26]. 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. The notion of weights in packet scheduling corresponds roughly to download

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 download rates are not known, can change dynamically, and can be difficult to estimate.

3 FairTorrent Algorithm

FairTorrent implements a distributed algorithm that provides fair bandwidth exchange even in the presence of diverse individual peer bandwidth capacities while preserving good download performance. For compatibility with BitTorrent, FairTorrent uses the same BitTorrent protocol,torrent files, and tracker service. FairTorrent is executed individually by each peer and does not rely on any global allocation or management service beyond what is already provided by BitTorrent. To describe the FairTorrent algorithm, we use the definitions of seeds and leechers from BitTorrent and the terminology in Table 1. Section 3.1 describes the deficit-counter-based main routines of FairTorrent which exchange data between leechers. Sections 3.2–3.4 describe other important considerations including an even-split seed behavior, a new method for dealing with unchoking, and dynamic considerations.

L_i	Leecher i
μ_i	Upload rate of L_i
$Sent_{ij}$	Total bytes sent by L_i to L_j
$Recv_{ij}$	Total bytes received by L_i from L_j
DF_{ij}	Deficit of L_i with respect to L_j: $DF_{ij} = Sent_{ij} - Recv_{ij}$
$Sent_i$	Total bytes sent by L_i to leechers
$Recv_i^L$	Total bytes received by L_i from leechers
$Recv_i^S$	Total bytes received by L_i from seeds
$E(i)$	Instantaneous service error of L_i: $E(i) = Sent_i - Recv_i^L$
$E(i,t)$	$E(i)$ at time t
$E^+_{\max}$ or $E^+_{\max}(i)$	$\max_t E(i,t)$. Max positive service error of L_i
$E^-_{\max}$ or $E^-_{\max}(i)$	$\max_t(-E(i,t))$. Max negative service error of L_i
$E_{\max}$ or $E_{\max}(i)$	$\max(E^+_{\max}(i), E^-_{\max}(i))$ Max service error of L_i
EM	$\max_i(E_{\max}(i))$ Maximum service error.
$packet\_size$	Maximum message size

Table 1: FairTorrent terminology.

3.1 Leecher Behavior

We first describe the basic algorithm run by the leechers. Each leecher, L_i, maintains several variables associated with each other leecher L_j. At any time, let $Sent_{ij}$ be the total number of bytes that a peer i has sent to peer j, and $Recv_{ij}$ be the total number of bytes that a peer i has received from peer j. Each peer i that implements FairTorrent maintains a *deficit variable* DF_{ij} for each peer j, where $DF_{ij} = Sent_{ij} - Recv_{ij}$. Thus, a positive (negative) deficit implies that peer i uploaded more (fewer) bytes to peer j than it downloaded from j. The values DF_{ij} are maintained in sorted order by peer i in a list called $SortedPeerList$. Each time the peer i is ready to send the next packet FairTorrent chooses to send that packet to the peer with the smallest DF_{ij}.

Procedures 1 and 2 show the FairTorrent operations performed by L_i when it receives or sends a packet to another peer. Procedure 1 (RECVPACKET) is executed by L_i whenever a packet from some peer j is received by L_i. RECVPACKET checks that peer j is a leecher. If peer j is a leecher, 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}. For simplicity, ties between deficit values are broken using unique peer IDs.

Procedure 1 RECVPACKET($peer\ j, data\_packet\ p$)

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

Procedure 2 (SENDPACKET) is executed by L_i when it is ready to send a packet. Each peer has an upload rate μ_i, which is expressed in KB per second. Thus, every $1/(\mu_i/packet\_size)$ seconds, L_i calls procedure SENDPACKET, which tries to pick a leecher with the lowest possible value of DF_{ij}. It examines the $SortedPeerList$ starting at the lowest index (which contains the peer with the lowest DF_{ij}) and picks the first peer j' from whom there is a pending request and the connection is writable (i.e. there is room in the TCP socket buffer). FairTorrent tries to send a packet of up to $packet\_size$ bytes, but then increments $Sent_{ij'}$ and $DF_{ij'}$ with the bytes that were actually sent to j' and re-inserts j' into the $SortedPeerList$. FairTorrent uses a $packet\_size$ of 16 KB for compatibility with older BitTorrent implementations, and for simplicity given the default 16 KB sub-piece request size in BitTorrent. Other BitTorrent clients typically also use a 16 KB packet size.

It is possible that SENDPACKET may not have any data of interest to send to the peer with the lowest deficit. In this case, FairTorrent just sends data to the next best peer, allowing for maximum utilization of the leecher's upload capacity. Since the deficit DF_{ij} with the lowest-deficit peer is always maintained, data will be sent to this peer when it becomes available, and the fairness is preserved.

Procedure SENDPACKET assumes the existence of several other procedures. HPRF(j), or HAVEPENDIN-

Procedure 2 SENDPACKET

$n \leftarrow 0$; $sz \leftarrow Size(SortedPeerList)$
$j \leftarrow SortedPeerList[n]$
while $(n < sz)$ and !(HPRF(j) and CWT(j)) **do**
  $n \leftarrow n + 1$; $j \leftarrow SortedPeerList[n]$
end while
if $(n < sz)$ **then**
  $bytes \leftarrow$SEND$(j, packet_size)$
  $Sent_{ij} \leftarrow Sent_{ij} + bytes$; $DF_{ij} \leftarrow DF_{ij} + bytes$
  $SortedPeerList.ReInsert(j)$
end if

GREQUESTFROM(j), returns true if there is a pending request from peer j. CWT(j), or CANWRITETO(j), returns true if there is room in j's buffer to send a packet. SEND is the procedure that actually sends the packet from i to j.

As SENDPACKET is the most important procedure in FairTorrent, we illustrate its behavior with an example of 3 leechers in Figure 3.1. For simplicity, the example assumes leechers always have data to send to one another, expresses values in packets rather than bytes, and uses equal-size packets. Leechers L_1, L_2 and L_3 have upload capacities of $\mu_1 = 3$, $\mu_2 = 2$ and $\mu_3 = 2$ packets/s, respectively. Thus, L_1 sends a packet every $1/3$ of a second. L_2 and L_3 send packets every $1/2$ of a second. All peers send their first packet at time 0.000. The left column of Figure 3.1 shows all the clock times at which at least one of the peers sends a packet. Each peer keeps track of its deficit variables DF_{ij} shown underneath each peer at each clock time. Arrows are used to show the source and the destination of each packet. At time 0.000, all the deficit variables are 0, and each peer sends its first packet to a peer with the lowest peer ID. Thus, L_1 sends to L_2. L_2 and L_3 each send a packet to L_1. When L_1 sends a packet to L_2, it sets $DF_{12} = 1$. Before time 0.333, it will have received a packet from L_2 and will have decremented DF_{12} back to 0. Thus at time 0.333, $DF_{12} = 0$. Also, since it receives a packet from L_3, before time 0.333 it will set $DF_{13} = -1$. At time 0.333, L_1 will send its next packet. Using procedure SENDPACKET, it will pick L_3 as the peer to send the packet to because it has lower deficit: $DF_{13} < DF_{12}$. Thus L_1 sends a packet to L_3 and increments DF_{13} to 0. Figure 3.1 shows the process until the time 2.000.

At time 2.000, the system reverts to the same state as at time 0.000 as all the deficits variables are 0, and all the peers will once again send a packet. In these two seconds, each peer sent the maximum number of packets limited by its capacity and received the same number of packets from its neighbors. Thus, without knowing upload capacities of its peers, the nodes discovered very quickly the appropriate rates using FairTorrent. If we

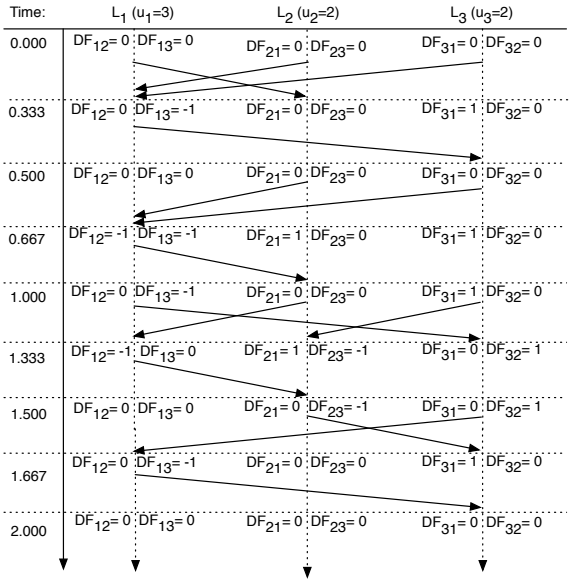

Figure 1: FairTorrent algorithm for leechers L_1, L_2 and L_3 with upload capacities of 3, 2 and 2

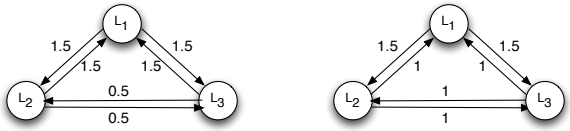

Figure 2: Peers L_1, L_2 and L_3 with upload capacities 3, 2, and 2. Bandwidth allocated under FairTorrent (left) vs Equal-Split (right).

count the arrows between each pair of peers that indicate the number of packets sent, we will see that in the 2 second cycle L_3 and L_2 exchanged 3 packets each with L_1 (or 1.5 packets/s) and they exchanged 1 packet with each other (or 0.5 packets/s). These rates are shown in Figure 2 (*Left*). 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. As we run for a long time period, the same behavior will repeat every 2 seconds and all deficit values will remain between -1 and 1.

To see the advantages of FairTorrent, we consider the same example, but 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*). Here the rates diverge. L_1 pushes 1.5 packets/s to each neighbor, but receives only 1 packet/s in return from each, resulting in unfair service as the deficits with each peer will grow by 1 every 2 seconds.

3.2 Seed Behavior

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

We currently focus on ensuring fair exchange of leecher bandwidth in a single swarm. In the case of multiple swarms, a peer may act as a seed in one swarm and a leecher in another. In this case, we may want to extend the deficit values to track deficits over multiple swarms. However, a detailed discussion of fair bandwidth exchange in multi-swarm scenarios is beyond the scope of this paper.

3.3 Unchoking Behavior

FairTorrent also needs to identify the peers who have relevant data. In BitTorrent, a peer begins by contacting the tracker to obtain IP addresses and ports of up to 50 other peers in the swarm, and then establishes TCP connections to these peers. By default, BitTorrent allows a peer to initiate up to 40 connections, and to accept up to 40 more for a total of 80 simultaneous connections with other peers who many be seeds or leechers. FairTorrent does not change this default behavior for two reason. First, in typical scenarios FairTorrent is able to provide its fairness and performance with much fewer than 80 connections. Second, we wanted to ensure a fair comparison with BitTorrent in Section 5 and thus exclude performance differences simply due to the number of simultaneous connections.

FairTorrent differs from BitTorrent because leechers are able to exchange data with many *neighbors*, where neighbors of a peer are those with which a TCP connection has already been established. Using the BitTorrent protocol, a leecher L_i will only receive requests from a leecher L_j if L_j is interested in some data that L_i has and L_i has unchoked L_j. But BitTorrent only unchokes a few peers at a time to measure their upload rates and identify high uploading peers. In contrast, in FairTorrent, leecher L_i simply unchokes any neighbor L_j that is interested in L_i's data.

FairTorrent unchokes all of its interested neighbors for two reasons. First, a high-capaciy FairTorrent leecher surrounded by low-capacity leechers may need to exchange data with many of them in order to reach covergence between its upload and the download rates. Since BitTorrent unchokes only a few peers at a time for a duration of 20 seconds by default a high-capacity leecher may take a long time to reach convergence [21]. Second, unchoking more neighbors reduces the likelihood of a data availability problem where a leecher has no data of interest to send to its peers. By talking with many peers, it is more likely that some peers will be interested in some of its data and FairTorrent can thus increase its upload capacity utilization as we show in Section 5.

FairTorrent is able to unchoke all of its interested neighbors and still provide fair bandwidth exchange because it uses the deficit counters to dynamically adapt to its neighbors' upload rates. If BitTorrent neighbors were all to unchoke one another then low uploaders would receive more bandwidth than they deserve, and high uploaders would receive less bandwidth than they deserve under the equal-split policy. Furthermore, FairTorrent can unchoke all its interested neighbors because it does not need to use the unchoking mechanism for discovering only the high uploading neighbors, as it can adapt to the upload rates of its neighbors in a more granular manner.

3.4 Dynamic Considerations

In P2P systems, peers may join and leave the system over time. A possible concern with FairTorrent is that when a leecher L_i accepts a connection from leecher L_j it sets $DF_{ij} = 0$. This setting could cause problems if, at that time, L_i has deficit variables far above 0. Then L_j will receive preferential service from L_i as DF_{ij} will be smaller compared to other deficits of L_i.

However, our analysis and experiments show that under FairTorrent service error tends to be quite small, and the average deficit DF_{ij} tends very close to 0. We expect that in a dynamic environment if a node joins an FairTorrent system it will discover a state where most DF_{ij} values in the system will be close to 0, and will not get any significant preferential treatment.

In future work, we plan to investigate the scenario of whitewashing [11] where a node free-rides, then re-joins the system with a new id to obtain some more free service. In such a system it is likely that the average DF_{ij} value will grow, and we may need to initialize DF_{ij} to a non-zero value, such as $\text{avg}_k(DF_{ik})$ value. Thus we believe, that while in a regular dynamic scenario setting DF_{ij} to a non-zero value is unnecessary it may be helpful in a scenario where whitewashing is prevalent, and with a smart use of deficits FairTorrent can actually adopt well to curb whitewashing.

4 Analysis

We will analyze FairTorrent with 3 goals – low overhead, fairness, and performance. In this section, we will give theoretical analysis of the first two objectives. Section 5 will deal with all three objectives experimentally.

4.1 Running Time

The running time of FairTorrent is quite small. Procedure RECVPACKET is dominated by the time to reinsert a peer back into the $SortedPeerList$ which takes $O(\log k)$ time for a system with k leechers. The time

for SENDPACKET consists of the while loop, plus the last few steps which are also dominated by the $O(\log k)$ time to reinsert into $SortedPeerList$. As written, the while loop may have to iterate over all peers which would incur linear overhead. As discussed in Section 3.3, in practice we do not need to check many peers to find one which has requested data. However, we can actually show that even in the worst case, the overhead is only $O(\log k)$. To implement the HAVEPENDINGREQUESTTO procedure, we maintain an array which indicates which peers have pending requests. As soon as the last request to a peer is satisfied, we remove that peer from the $SortedPeerList$. When a peer makes a new request, we insert it into $SortedPeerList$. Each insertion and deletion can be charged against data actually being sent and thus the additional overhead is only a small constant factor. CANWRITETO is implemented similarly. The first time L_i discovers that it can't write to L_j, it removes it from the $SortedPeerList$. An asynchronous call is received from the OS when the network socket for peer L_j again becomes writable. The insertion of L_j into $SortedPeerList$ upon such a call can also be charged to a previous *send* to L_j.

We now give a numerical estimate of the total overhead. If the packet_size used by all peers is P KB, and the total upload and download bandwidth is B KB/s, then the re-insertion step on receiving or sending a packet is performed at most B/P times per second. Thus, the running time of FairTorrent per second is $O((\log k) * B/P)$. Assuming $P = 16$, and a high-bandwidth peer with say 1MB/s and 3MB/s of upload and download respectively, and $k = 300$ peers, then the total running time per second is roughly $256 * \log(300) < 2500$ operations. This overhead is quite small for a modern processor.

4.2 Fairness

Informally, our notion of fairness is that each peer is able to download from other leechers at the same rate that it uploads. More formally, at any time t, each peer L_i has uploaded an amount $Sent_i = \sum_j Sent_{ij}$ and downloaded $Recv_i^L = \sum_j Recv_{ij}$, where sums are taken over all other leechers. The instantaneous service error for L_i is $E(i) = Sent_i - Recv_i^L = \sum_j DF_{ij}$. We then compute $E^+_{\max}(i)$ and $E^-_{\max}(i)$ as the maximum positive and negative error for L_i, where the maximum is taken over time. $E_{\max}(i) = \max\{E^+_{\max}(i), E^-_{\max}(i)\}$ and finally $EM = \max_i\{E_{\max}(i)\}$. EM is our final measure of error. If EM is small, then we can conclude that at all times, on all peers, the difference between upload and download rate is small. We use EM and $E_{\max}$ as a measure of fairness in an analogous way to how they are used in the Fair Queuing literature [4, 26, 9].

In Section 5, we will see that EM is small for many different networks and loads. In this section, we will show rigorously that for any 3 node network, EM is small. We conjecture that in any network where all peers implement FairTorrent under a wide variety of upload capacity distributions, the upload rate of each peer closely approximates its download rate, but do not pursue that proof in this paper.

4.3 Three-Node Case

We now prove strong fairness bounds for the special case of a 3-node network. We are given three peers L_1, L_2 and L_3, with upload rates μ_1, μ_2 and μ_3. For ease of exposition, we assume that $\mu_1 \geq \mu_2 \geq \mu_3$ and that $packet\_size = 1$. These assumptions imply that L_i uploads exactly once every $1/\mu_i$ time units. We further make the following two gentle assumptions:

•[A1)] The upload capacity (μ_i) of any leecher i is smaller than the upload capacities of the remaining nodes. (i.e. $\mu_i \leq \sum_{j \neq i} \mu_j$ for any leecher i). If this assumption does not hold, then there is no possibility of the peer getting a download rate from the other leechers equal to its upload rate.

•[A2)] A leecher always has useful data to share with its neighbors. As discussed in Section 3, this assumption generally holds.

Theorem 1 *Assume we have a 3-leecher network that satisfies assumptions A1 and A2. If each leecher implements FairTorrent, then the maximum service error* $EM \leq 4$.

Due to the lack of space we present the lemmas that make up the proof, but either omit or only sketch their proofs. We observe that the bound of 4 in the theorem is independent of the number of packets broadcast. The proof essentially shows that the 4 arises because in the 3-node case, there can be a burst of 2 packets. In general in the k-node case, there can be bursts of k packets (e.g. all peers immediately send to peer 1), but we believe that the imbalance cannot get much worse than k.

In order to prove the theorem, we need the following lemma that describes these bursts. We can list the sequence of nodes that broadcast in time order, with ties broken in favor of the higher rate peer. For the example in Figure 3.1, that sequence would be $L_1, L_2, L_3, L_1, L_2, L_3, L_1, L_1, L_2, L_3, \ldots$. We call this sequence the *broadcast list*, and we call a sequence of m consecutive appearances of the same leecher a m*-burst*.

Lemma 2 *Assume we have a 3-leecher network that satisfies assumptions A1 and A2. Then (1) Neither* L_2 *nor* L_3 *ever appear in a* 2*-burst on the broadcast list. (2)* L_1 *can never appear in a* 3*-burst on the broadcast list. (3) Between the start of two consecutive* 2*-bursts of* L_1*, the total number of broadcasts of* L_2 *and* L_3 *combined is at least as large as that of* L_1.

Lemma 3 *No peer ever accumulates a deficit $DF_{ij} > 2$.*

Proof: Due to lack of space we only provide a sketch of this lemma's proof. Let us consider the proof for L_1, the other nodes follow by similar reasoning. We abbreviate the state of the node L_1 as S_{cd} where $c = DF_{12}$ and $d = DF_{13}$. Furthermore, let's collectively call S_{start} the state where L_1 is in S_{01} or S_{10}. The proof proceeds by a careful case analysis on a very limited state diagram. We show based on Lemma 2 that L_1 can only get as far as state S_{22} before both L_2 and L_3 send packets exclusively to L_1. Since $\mu_1 \leq \mu_2 + \mu_3$ it will force L_1 back to S_{start}, thus never reaching $DF_{ij} > 2$.

Theorem 1 now follows. Since $DF_{ij} \leq 2$ at all times, then $E^+_{\max}(i) \leq \sum_{j \neq i} DF_{ij} \leq 4$. We conclude this section by conjecturing that a similar proof holds for the general k-node case. The deficits can grow as large as k, but we believe that they do not grow much larger and are independent of the amount of data transmitted.

5 Experimental Results

We implemented FairTorrent on top of the original BitTorrent Python client in only 150 lines of Python code, thereby demonstrating that FairTorrent is simple to implement. To measure the effectiveness of FairTorrent in a realistic wide-area network environment, we ran an extensive set of experiments on PlanetLab to compare the fairness and performance of FairTorrent against three other BitTorrent implementations: (1) original BitTorrent 3.9.1, a Python client which implements the documented version of the BitTorrent protocol and was the code base used for our FairTorrent modifications, (2) Azureus 3.0.4.2, one of the most popular BitTorrent clients and the latest version of that Java implementation available at the start of our experimentation, and (3) BitTyrant 1.1.1, the latest version of a strategic client that attempts to garner the highest download rate from other leechers. We instrumented each of the clients to measure fairness by logging the bytes uploaded to and downloaded from other peers every 15 seconds, and to measure performance by logging the completion times at the end of each file download. We also instrumented the clients to be consistently configured with an upload bandwidth limit to allow us to experiment with different distributions of upload bandwidth capacity.

To quantify fairness, we measure the maximum service error $E_{\max}(i)$ for each leecher i, the metric shown in Table 1 and discussed in our analysis in Section 4. Maximum service error is the largest difference between bytes uploaded and downloaded during a file download. For example, a leecher will have a maximum positive service error $E^+_{\max}$ of 10 MB if, at some point during the download, it uploaded 10 MB more data than it downloaded from other leechers. For a 32 MB file download, $E^+_{\max}$ of 10 MB implies that a system allows a leecher to contribute roughly 30% of the entire file more than the service that it receives. Similarly, a leecher will have a maximum negative service error $E^-_{\max}$ of 10 MB if, at some point during the download, it downloaded 10 MB more data than it uploaded to other leechers. Note that received seed bandwidth is not counted toward service error. $E_{\max}$ is the maximum of the $E^+_{\max}$ and $E^-_{\max}$ for a given leecher. Both high $E^+_{\max}$ and $E^-_{\max}$ represent unfairness in the system, although users may be more concerned with a high $E^+_{\max}$ as that implies they are contributing more than what they are receiving.

To quantify performance, we use a variety of statistics about the download times of the individual leechers including the average, maximum and various percentiles. We also look at the standard deviation, and, when appropriate separate the leechers into categories.

We present results for a set of experiments with a network of 50 leechers and 10 seeds file-sharing a 32 MB target file. All nodes were configured with download bandwidth capacities of 100 KB/s and upload bandwidth capacities of no more than 50 KB/s, reflecting typical scenario where most ISPs allow users a download rate at least twice their allowed upload rate. The leechers were configured in three different distributions of upload bandwidth capacities between 0 to 50 KB/s: uniform, skewed, and bimodal. The uniform distribution represents a wide range of peers with diverse upload capacities participating in a swarm. Each leecher received an upload capacity randomly selected between 1 and 50 KB/s. The skewed distribution represents a high contributing uploader participating in a swarm with low contributors. 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, with a total upload bandwidth across all low contributors of 150 KB/s. The bimodal distribution represents a population of high contributing uploaders who do not cap their bandwidth and a population of free riders participating in a swarm. The 25 high uploaders received upload capacities randomly selected between 40 and 50 KB/s, and the free riders received upload capacities randomly selected between 0 and 3 KB/s. The seeds were configured with upload bandwidth capacities of 25 KB/s each, chosen to match the average upload bandwidth capacity of the leechers in the uniform distribution, for a total of 250 KB/s of seed bandwidth.

We ran the same set of experiments for a FairTorrent network (FT), a BitTorrent network (BT), an Azureus network (AZ), and a BitTyrant network (TY). For each BitTorrent network and each distribution, we ran five experiments with five different sets of upload capacities generated randomly from the respective distribution. Thus, we collected 250 leecher measurements for each combination of network and distribution. In each exper-

iment, the leechers begin the download simultaneously and remain in the system as seeds when they complete their download. Note that once leechers complete their download, their upload bandwidth counts as seed bandwidth and does not affect the fairness measurements of data exchange among leechers. We also ran experiments with different file sizes, different numbers of seeds and total seed bandwidth, and leechers configured to leave the system when they complete their download. Since the results of these experiments were similar to the ones we present, they are omitted due to space constraints.

The captions in the figures begin with a prefix "U:", "S:", or "B:" to signify the uniform, the skewed or the bimodal distributions respectively.

5.1 Uniform Distribution

Figures 3 to 17 show the measurements for leechers with upload capacities randomly selected from a uniform distribution. Figure 3 shows a cumulative distribution function (CDF) of the maximum service error $E_{\max}$ across all five experiments. It shows the fraction of leechers whose maximum service error was always less than the given value. FairTorrent provides an order of magnitude better fairness than Azureus, the next closest network. Azureus and BitTorrent had similar fairness performance. BitTyrant had by far the worst fairness performance.

Figure 4 separates 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 $E^-_{\max}$ of FairTorrent. The figure shows a range of percentiles 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 The median FairTorrent leecher's $E^+_{\max}$ was just 79 KB, meaning that a median leecher at no time gives more than five 16 KB packets of service more than it receives. The maximum value of $E^+_{\max}$ for FairTorrent was only 436 KB, meaning that during the entire download of the file, no FairTorrent leecher gives more than 436 KB of service than what it receives from other leechers. This value is 18 to 73 times smaller than the maximum $E^+_{\max}$ of other networks. The maximum $E^+_{\max}$ for an Azureus or BitTorrent leecher was over 8 MB, more than 25% of the entire 32 MB file. $E^+_{\max}$ for BitTyrant reached 31 MB, or almost 100% of the file. These results show that FairTorrent provides much better fairness than all three other networks, and that BitTorrent, Azureus, and BitTyrant all exhibit poor fairness performance.

Figures 5 to 8 show the average upload rate versus the average leecher download rate experienced by each leecher during its download of the 32 MB file. In the ideal case, the download rate should equal the upload rate, which is represented by a reference line in each figure drawn in the background. Figure 5 shows that FairTorrent provides fairness that closely matches the ideal reference line and visually demonstrates that it attains rate convergence, where the download rate that a leecher obtains from other leechers converges to its upload rate. In contrast, Figures 6 to 8 show that BitTorrent, Azureus, and BitTyrant all have poor rate convergence. Figures 6 and 8 shows that higher contributing peers in BitTorrent and even more so in BitTyrant are likely to receive a download rate far below their contribution, while lower contributing peers will receive a higher level of service than their contribution.

Figure 9 shows the average and maximum time for leechers to completely download the target file for each network. The average download times were 939, 945, 979, and 1127 seconds for FairTorrent, BitTorrent, Azureus, and BitTyrant, respectively. While some P2P file-sharing models [10] posit a tradeoff between fairness and download performance, these results show that FairTorrent is able to achieve both the best fairness and average download performance. Furthermore, the performance of peers under FairTorrent is more directly correlated with their contribution rates. For example, the average download times for peers with higher upload rates of 40 to 50 KB/s were 690, 728, 737 and 952 seconds for FairTorrent, BitTorrent, Azureus, and BitTyrant, respectively, representing a bigger relative difference between FairTorrent and other systems than for the average times. This behavior occurs because FairTorrent rewards peers more fairly based on their contribution and naturally causes high uploaders to download faster. Figure 9 shows that FairTorrent provides an even larger improvement when measuring the maximum time across all leechers to completely download the file. The maximum download times are 1347, 1892, 1849 and 2266 seconds for FairTorrent, BitTorrent, Azureus and BitTyrant respectively. Thus, in FairTorrent all of the peers complete their download 37 to 68% faster than in other systems.

One interesting statistic here is how close can a system come to the optimal bound of the last leecher's download time. Assuming the bandwidth can be used optimally, in a system with n leechers, B total upload capacity and FS file size, the last uploader will not finish before $OPT = n * FS/B$ as each leecher needs to download FS bytes. In our experiments, $FS = 32$ MB, $n = 50$ and $B = 1500$ KB/s assuming average leecher bandwidth of 25 KB/s and 250 KB/s total bandwidth from the seeds. Thus, $OPT = 1092$ seconds. FairTorrent comes within 255 seconds of this bound while the next closest system, Azureus, is three times worse at 757 seconds more than the bound.

Figures 10 to 13 show the completion time of the leechers in each system based on their upload capacity. Figure 10 shows that FairTorrent provides a very

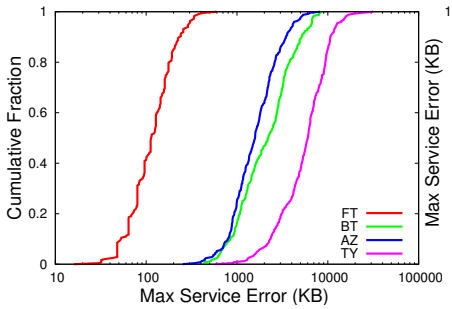

Figure 3: U: Max service error $E_{\max}$

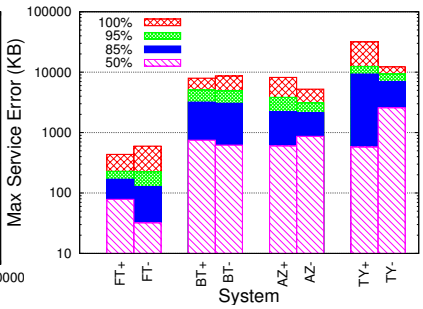

Figure 4: U: $E^{+}_{\max}$ and $E^{-}_{\max}$

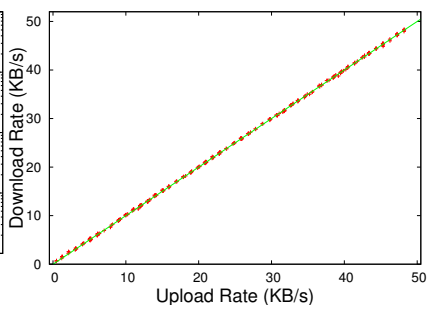

Figure 5: U: FairTorrent fairness

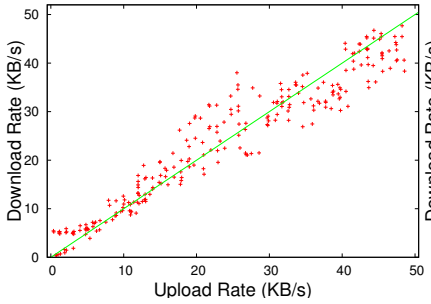

Figure 6: U: BitTorrent fairness

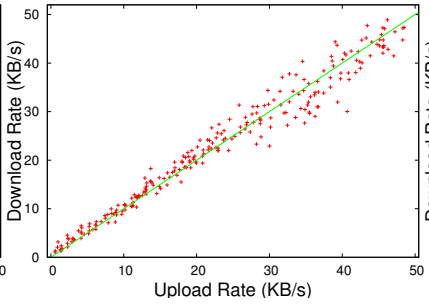

Figure 7: U: Azureus fairness

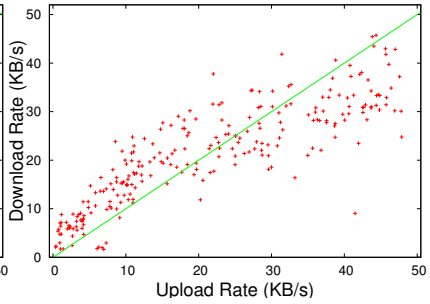

Figure 8: U: BitTyrant fairness

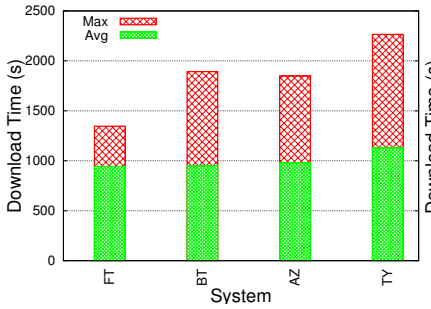

Figure 9: U: Download time

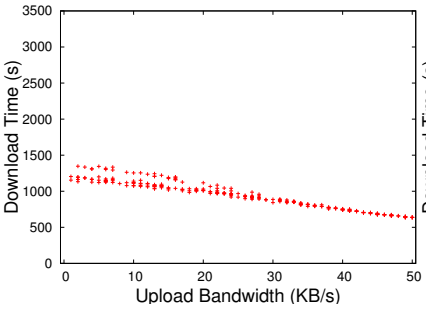

Figure 10: U: FairTorrent download time

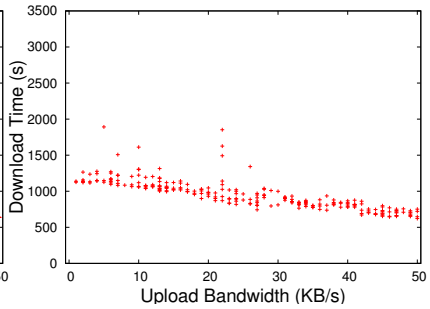

Figure 11: U: BitTorrent download time

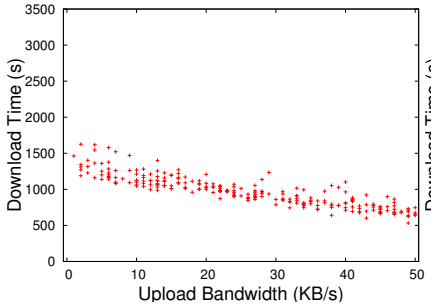

Figure 12: U: Azureus download time

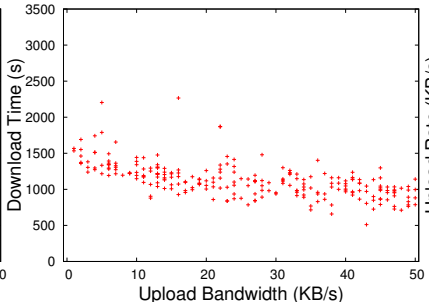

Figure 13: U: BitTyrant download time

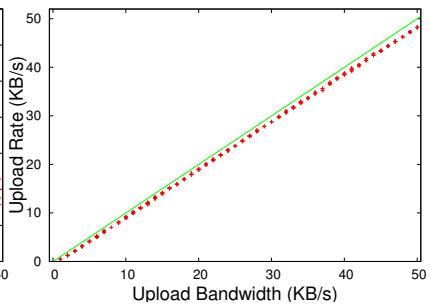

Figure 14: U: FairTorrent utilization

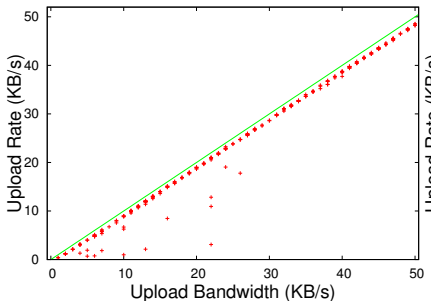

Figure 15: U: BitTorrent utilization

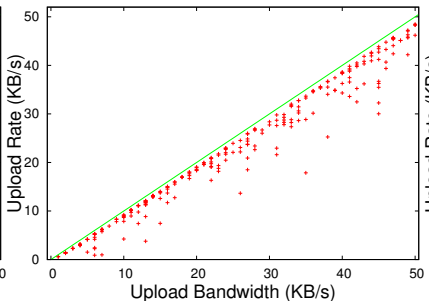

Figure 16: U: Azureus utilization

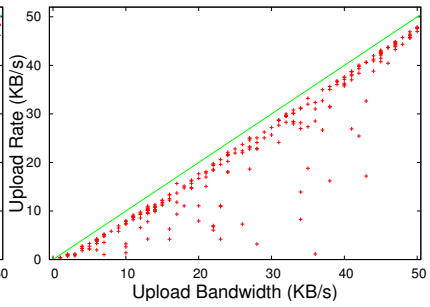

Figure 17: U: BitTyrant utilization

strong correlation between upload capacity and download times. There is very little variance in download times among leechers of the same capacity especially for the upload bandwidth above 25 KB/s. There is a slight variability for lower uploaders, because as high uploaders finish at different times for different tests there is a different amount of seed data available for the remaining leechers. Figures 11 to 13 show that the other systems have a much higher variability in download times for a given upload bandwidth. Figure 13 shows many lower contributing peers obtaining faster download times than higher contributing peers when using BitTyrant.

Figures 5 to 13 tell an interesting story of why FairTorrent provides faster maximum download times. One would expect that the maximum download time is due to a low contributing peer, which generally uploads less and therefore downloads less for most systems. However, Figures 5 to 8 show that FairTorrent peers with low upload rates have the lowest leecher download rates of all the systems since it more accurately matches download and upload rates. Based on this, one would expect that FairTorrent low contributors should take the longest to download, thus increasing the maximum download time. However, Figures 10 to 13 show that FairTorrent low contributors do not have the slowest download times. One reason for this is that FairTorrent's better fairness enables high uploading leechers to finish sooner and become seeds. This results in more available seed bandwidth earlier in the download which can be used by low contributors to increase their aggregate download rates across leechers and seeds. By enabling high contributors to finish downloading sooner and become seeds, FairTorrent can improve download performance across all peers even though it may reduce leecher download bandwidth for low contributors.

The second reason for FairTorrent's better download times can be seen in Figures 14 to 17, which show the achieved upload data rate versus the configured upload bandwidth capacity for each leecher. In the ideal case, leechers should maximize utilization of available upload bandwidth so that the upload rate should equal the upload bandwidth, as is represented by a reference line in each figure. For good performance, it is critical that a system not compromise bandwidth utilization for reducing service error or improving rate convergence. For example, a peer with 50 KB/s of available capacity that uploads and downloads at 5 KB/s will exhibit no service error and perfect rate convergence, but will have poor bandwidth utilization and performance.

Figure 14 shows that FairTorrent achieves close to 100% bandwidth utilization. The difference from ideal performance is due to BitTorrent protocol overhead, which is correctly not counted as part of the achieved upload rate, but consumes part of the available bandwidth. In contrast, Figures 15 to 17 show that all the other networks have a greater number of leechers that lie far below the ideal reference line and have poor utilization. Those leechers could not achieve the desired upload rates. This difference in utilization is part of the reason that FairTorrent achieves better download performance than the other systems. Overall, the aggregate bandwidth utilization across all leechers for each network was 95.3%, 93.7%, 89.8% and 82.8% for FairTorrent, BitTorrent, Azureus, and BitTyrant, respectively. These numbers include only the data upload rate, not the protocol overhead which was roughly 3% for each system.

There are at least two reasons that FairTorrent achieves better utilization than the other systems. First, FairTorrent is not limited by the unchoking behavior of BitTorrent and Azureus. It will send a packet to any peer with the lowest deficit from whom a request is pending. Thus, as long as there is bandwidth available and a request from any peer in the network, FairTorrent will send a packet. FairTorrent's utilization stands in sharp contrast to block-based TFT, which ends up under-utilizing bandwidth compared to BitTorrent [1]. Since FairTorrent does not set any arbitrary threshold on the deficit with its neighbors, it is able to achieve both better utilization and at the same time get high reciprocation due to small $E_{\max}$. Second, FairTorrent unlike BitTyrant, FairTorrent does not try to minimize the amount of bandwidth it sends on each connection, as this behavior leads to very low bandwidth utilization in an all-BitTyrant network.

For all networks, it is interesting to note that the gap between ideal and achieved utilization is slightly larger for leechers with higher upload bandwidth. This gap occurs because the BitTorrent protocol overhead consists primarily of HAVE messages that a leecher sends to its neighbors indicating it has a piece of the file. Higher uploaders will download pieces faster and thus send HAVE messages more frequently.

To illustrate the variability in download rate of different systems, Figure 18 shows the standard deviation in download rate observed by the leechers, as measured over consecutive 15 second intervals. FairTorrent had the lowest average standard deviation in leecher download rate at just 1.8 KB/s on average, more than three times better than the next closest system. BitTorrent, Azureus, and BitTyrant had average standard deviations of 6.0 KB/s, 8.0 KB/s and 12.3 KB/s respectively. The low standard deviation makes FairTorrent more amenable for use in BitTorrent live streaming applications such as CoolStreaming [31]. FairTorrent is beneficial in two ways. First, because of the constant and fair download rate, a peer with a higher upload capacity can receive a higher quality stream with a bit-rate closer to its upload capacity. Second, a content provider can provision less server (or seed) bandwidth because it will be less likely

for a peer's download rate to drop due to high variance.

5.2 Skewed Distribution

Figures 19 to 22 show the measurements for leechers with upload capacities randomly selected from a skewed distribution. In addition to running experiments for FairTorrent, BitTorrent, Azureus, and BitTyrant networks, we also ran the same experiments for the 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. The results of using a FairTorrent high uploader in a network of BitTorrent, Azureus, and BitTyrant peers were denoted FT/BT, FT/AZ and FT/TY, respectively.

Figure 19 shows the CDF of $E_{\max}$ across all five experiments for the respective networks. The CDF for FairTorrent stands out far to the left from all the other networks, representing between one to two orders of magnitude difference for all the cumulative fractions. It shows that FairTorrent provides fair service to both the high uploader and the 49 low contributors. FairTorrent prevents the high uploader from accumulating a large $E^+_{\max}$ and low uploaders from accumulate large $E^-_{\max}$.

Figure 20 shows the $E^+_{\max}$ and $E^-_{\max}$ of the high uploader in each system. Clearly $E^+_{\max}$ always dominates $E^-_{\max}$ as the high uploader typically serves more data than it receives in the skewed case. However, $E^+_{\max}$ for the FairTorrent case is 60 to 200 times smaller than the $E^+_{\max}$ of BitTorrent, Azureus, and BitTyrant. $E^+_{\max}$ of the high uploader was measured to be 555 KB for FairTorrent, as compared to 51 MB, 31 MB, and 113 MB for BitTorrent, Azureus and BitTyrant, respectively. When FairTorrent replaced the high uploader in other systems, it is still able to reduce $E^+_{\max}$. FairTorrent reduces $E^+_{\max}$ by a factor of 15 in comparing FT/BT to BitTorrent, by 10% in comparing FT/AZ to Azureus, and by a factor of 50 in comparing FT/TY to BitTyrant.

The smaller improvement in fairness for the high uploader in comparing FT/AZ to Azureus is due to a subtle difference in the unchoking behavior of Azureus. Azureus biases its optimistic unchoking behavior towards peers from whom it saw better data-exchange ratios in the past. This results in the highest of the low uploaders, those uploading 4 and 5 KB/s, exchanging data more consistently with one another. Unfortunately, this policy disadvantages the one high uploader as the 4 and 5 KB/s leechers only send a fraction of their bandwidth to that peer. As a result, the high uploader does not receive enough bandwidth from the other lower uploaders, resulting in a smaller fairness improvement. This unusual behavior is very unlikely in practice and only applies to this highly exaggerated skewed distribution in which the single high uploader holds 25% of the total leecher bandwidth.

Figure 21 shows the maximum and the average download rate of the single high uploader for each of the systems. 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 improved its average download rate. FairTorrent improved the download rate from 13.8 to 44.0 KB/s when comparing FT/BT to BitTorrent, from 7.2 to 45.6 KB/s when comparing FT/TY to BitTyrant, and despite only a modest improvement in fairness of FT/AZ it improved the download rate from 9.8 to 23.0 KB/s. In this very skewed case FairTorrent is able to achieve a substantial improvement in the download rate bringing it significantly closer to its upload capacity.

Figure 22 shows the maximum and the average download times for the entire set of peers and just the high uploader (labeled with letter "H") for each of the systems. The high uploader in FairTorrent completed its average download in 644 seconds, 3-5 times faster than the high uploader in BitTorrent, Azureus, or BitTyrant. When FairTorrent replaced the high uploader in other systems, it still improves download times. FairTorrent reduces download times from 1,804 to 703 seconds in comparing FT/BT to BitTorrent, from 1,859 to 1,138 in comparing FT/AZ to Azureus, and from 3,305 to 615 seconds in comparing FT/TY to BitTyrant. FairTorrent substantially reduces the download time for Azureus even though its improvement in fairness is more modest. As with other distributions, replacing the high uploader in each system with FairTorrent improved the maximum download time of the low uploaders as well. FairTorrent adopts to the upload rates of the surrounding low uploaders and is therefore able to get a high download rate, even though the low uploaders do not run FairTorrent.

5.3 Bimodal Distribution

Figures 23 to 26 show the measurements for leechers with upload capacities randomly selected from a bimodal distribution. In addition to running experiments for FairTorrent, BitTorrent, Azureus, and BitTyrant networks, we also ran the same experiments for the non-FairTorrent networks in which the high uploaders were replaced by FairTorrent to show how FairTorrent performs in the presence of free-riders running non-FairTorrent clients. The results of using FairTorrent high uploaders in a network of BitTorrent, Azureus, and BitTyrant peers were denoted FT/BT, FT/AZ and FT/TY, respectively.

Figure 23 plots the CDF of the maximum service error of the high uploaders for each set of experiments. The four curves on the left where FairTorrent runs on the high uploaders are separated from the rest of the setups by an order of magnitude. Recall that in the skewed case, a single high uploader was only able to get limited im-

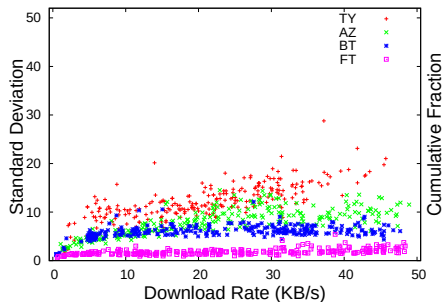

Figure 18: U: Standard deviation of the download rate

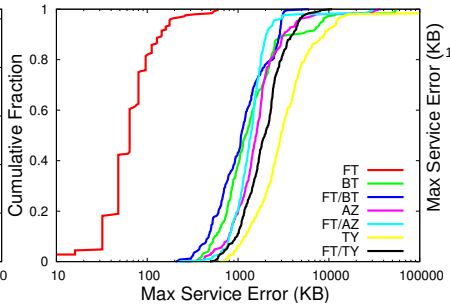

Figure 19: S: Max Service Error

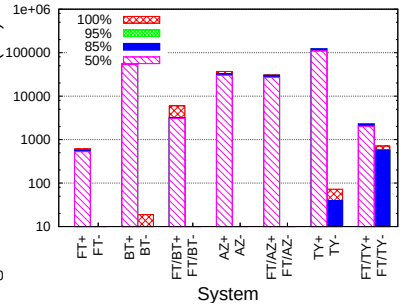

Figure 20: S: high uploader $E^{+}_{\max}$, $E^{-}_{\max}$

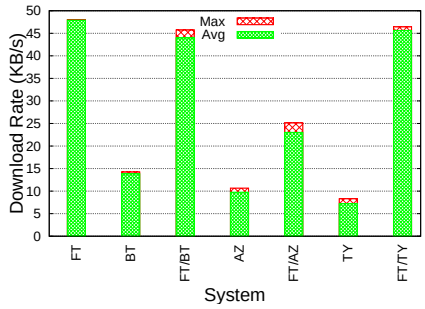

Figure 21: S: high uploader download rate

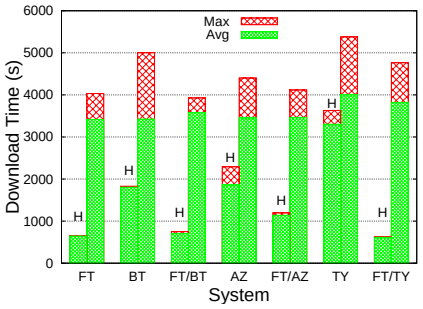

Figure 22: S: Download time

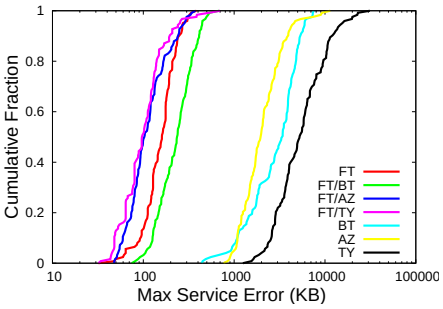

Figure 23: B: Max service error $E_{\max}$

provement in the case of FT/AZ (as compared to AZ). In contrast, here, all four tests where FairTorrent replaces high uploaders, including FT/AZ, show more than an order of magnitude of improvement in $E_{\max}$. The reason for this difference is that where more than one high uploader is present the high uploaders can exchange data with one another at a high rate. This behavior should be a very strong motivation for users who do not cap their bandwidth to run FairTorrent and be immune from the effects of a large population of free-riders.

Figure 24 shows separately the maximum $E^{+}_{\max}$ and $E^{-}_{\max}$ of the high uploaders for each system. The maximum $E_{\max}$ was 384 KB in FT. In BT, AZ, and TY, where high uploaders were replaced by FairTorrent, the $E_{\max}$ for the high uploaders was reduced by 1 to 2 orders of magnitude to be under 700 KB.

Figure 25 shows the $E^{+}_{\max}$ and $E^{-}_{\max}$ of the free-riders. The free-riders obtain the least amount of instantaneous free service ($E^{-}_{\max}$) under FairTorrent, as the maximum $E^{-}_{\max}$ is 304 KB, and the median $E^{-}_{\max}$ is 0. (meaning that a median free-rider gets no free service!) When FairTorrent replaces the high uploaders in other systems it is able to reduce the maximum $E^{-}_{\max}$ of a free-rider by a factor of 5 for FT/BT as compared to BitTorrent, a factor of 10 for FT/AZ as compared to Azureus, and a factor of 4 for FT/TY as compared to BitTyrant. Most notably, in FT/AZ, a median free-riders receives only 68 KB of free service (also a factor of 10 reduction). Thus, if the FairTorrent policy is adopted by users who do not cap their bandwidth, free-riders who run the most popular client, Azureus, will have little incentive to free-ride.

Figure 26 shows the average and maximum performance of the high uploaders in each system over a set of five tests. In each network that runs FairTorrent for high uploaders, the heavy-tail of these high uploading performers is eliminated. FairTorrent reduced the worst-case download time for a high uploader from 913 to 739 seconds for BitTorrent, from 1164 to 708 seconds for Azureus and from 1771 to 707 seconds for BitTyrant. Thus, even in the presence of 50% of free-riders (of any type client) the high uploading users who do not cap their bandwidth are immune from unlucky neighbor assignments if they run FairTorrent.

6 Conclusions

Users participate in a peer-to-peer system to be able to download files quickly. The system must distribute the limited bandwidth among the different users. Any session may contain a diverse collection of users with different bandwidth capacities, and also users who are malicious and are willing to try to subvert any protocol in order to obtain more bandwidth for their own downloads.

Fundamental to all peer-to-peer systems is some notion of fairness. Each of the common peer-to-peer systems enforce some notion of fairness, and achieve different behaviors in terms of speed of download and exactly which peers benefit more from the particular system.

In this paper, we introduced a new protocol, called

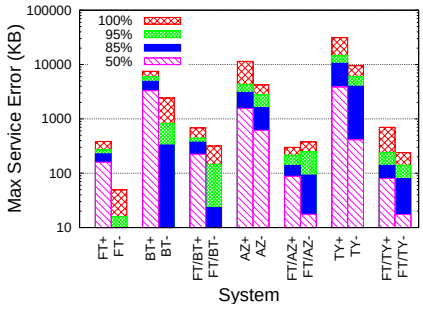

Figure 24: B: high uploaders $E^{+}_{\max}$, $E^{-}_{\max}$

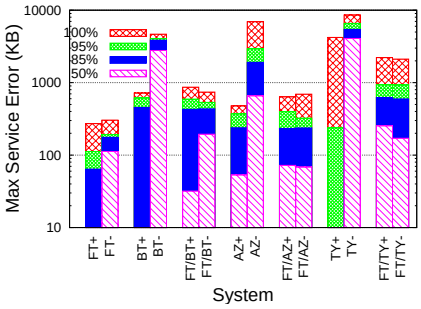

Figure 25: B: free-riders $E^{+}_{\max}$, $E^{-}_{\max}$

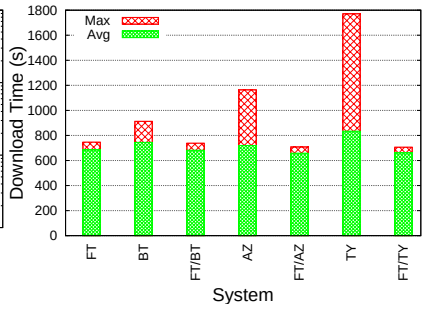

Figure 26: B: high uploaders download time

FairTorrent, which is simple to implement and has many desirable properties. We then showed that this simple protocol not only maintains fairness, but has surprising good download performance in a variety of settings. By essentially matching download rates with upload rates, under various upload capacities' distributions we achieve faster overall download times. There are several high-level explanations for the success. First, while other systems typically give free-riders more than their share of service we only give them their small share. Furthermore, when the high-uploaders adopt FairTorrent free-riders receive little free service. Second, by having these high-uploading clients complete their downloads earlier, they are able to serve as seeds and then help the legitimate lower bandwidth clients, without any adverse effects on their own performance. Finally, FairTorrent peers achieve a better utilization rate, thereby also improving the system by increasing the upload capacity. We demonstrated these results in a variety of settings and derived bounds on the worst case performance.

References

[1] A.Bharambe, C. Herley, and V.Padmanabhan. Analyzing and improving a bittorrent network's performance mechanisms. In *INFOCOM*, 2006.

[2] K. amd V. Pai and A.Mohr. Swift: A system with incentives for trading.

[3] Azureus. `http://www.azureus.com/`.

[4] J. C. R. Bennett and H. Zhang. Wf2q: Worst-case fair weighted fair queuing. In *INFOCOM*, 1996.

[5] K. berer and Z. Despotovic. Managing trust in a peer-to-peer information system. In *ACM CIKM*, 2001.

[6] A. Chow, L. Golubchik, and V. Misra. Improving bittorrent: a simple approach. In *IPTPS*, 2008.

[7] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of P2P Systems*, 2003.

[8] F. Cornelli, E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Choosing reputable servents in a p2p network. In *WWW*, 2002.

[9] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.

[10] B. Fan, D. Chiu, and J.Lui. The delicate tradeoffs in bittorrent-like file sharing protocol design. In *IEEE International Conference on Network Protocols*, 2006.

[11] M. Feldman, C. Papadimitriou, J. Chuang, and I. Stoica. Free-riding and whitewashing in peer-to-peer systems. In *Workshop on Economics and Information Security*, 2004.

[12] S. J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOM*, 1994.

[13] A. Legout, N. Liogkas, E. Kohler, and L. Zhnag. Clustering and sharing incentives in bittorrent systems. In *SIGMETRICS*, 2007.

[14] Q. Lian, Y. Peng, M. Yang, Z. Zhang, Y. Dai, and X. Li. Robust incentives via multi-level tit-for-tat. In *IPTPS*, 2006.

[15] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in bittorrent is cheap. In *5th Workshop on Hot Topics in Networks*, 2006.

[16] G. A. M. Ham. Ara: A robust audit to prevent free-riding in p2p networks. In *Inernational Conference on P2P Computing*, 2005.

[17] S. J. M. Sirivianos, X. Yang. Dandelion: Cooperative content distribution with robust incentives. In *USENIX*, 2007.

[18] T. Ngan, A. Nandi, and A. Singh. Fair Bandwidth and Storage-sharing in Peer-to-Peer Networks. In *First IRIS Student Workshop*, 2003.

[19] T. Ngan, D. Wallach, and P. Druschel. Enforcing Fair Sharing of Peer-to-Peer Resources. In *IPTPS*, 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. In *IEEE/ACM Transactions on Networking*, volume 1, June 1993.

[21] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Building BitTyrant, a (more) strategic BitTorrent client. *USENIX ;login:*, 32(4):8–13, August 2007.

[22] Pplive. `http://www.pplive.com/`.

[23] D. Qiu and R.Srikant. Modeling and performance analysis of bittorrent-like peet-to-peer networks. In *SIGCOMM*, 2004.

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

[25] H. G.-M. S. Kamvar, M.S chlosser. The eigentrust algorithm for reputation management in p2p networks. In *WWW*, 2003.

[26] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round-robin. In *SIGCOMM*, 1995.

[27] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free riding in bittorrent networks with the large view exploit. In *IPTPS*, 2007.

[28] S.Jun and M.Ahamad. Incentives in bittorrent induce free riding. In *Workshop on Economics of peer-to-peer systems*, 2005.

[29] S. Tewari and L. Kleinrock. On fairness, optimal download performance and proportional replication in peer-to-peer networks. In *IFIP Networking*, 2005.

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

[31] X. Zhang, J. Liu, B. Li, and T. Yum. Coolstreaming/donet: A data-driven overlay network for efficient live media streaming. In *INFOCOM*, 2005.