Analyzing and Improving BitTorrent Performance

Ashwin R. Bharambe

Carnegie Mellon University

Cormac Herley
Venkata N. Padmanabhan

Microsoft Research

February 2005

Technical Report
MSR-TR-2005-03

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Analyzing and Improving BitTorrent Performance

Ashwin R. Bharambe*
Carnegie Mellon University

Cormac Herley
Microsoft Research

Venkata N. Padmanabhan
Microsoft Research

ABSTRACT

In recent years, BitTorrent has emerged as a very popular and scalable peer-to-peer file distribution mechanism. It has been successful at distributing large files quickly and efficiently without overwhelming the capacity of the origin server.

Early measurement studies verified that BitTorrent achieves excellent upload utilization, but raised several questions concerning utilization in settings other than those measured, fairness, and the choice of BitTorrent's mechanisms. In this paper, we present a simulation-based study of BitTorrent. Our goal is to deconstruct the system and evaluate the impact of its core mechanisms, both individually and in combination, on overall system performance under a variety of workloads. Our evaluation focuses on several important metrics, including peer link utilization, file download time, and fairness amongst peers in terms of volume of content served.

Our results confirm that BitTorrent performs near-optimally in terms of uplink bandwidth utilization, and download time except under certain extreme conditions. On fairness, however, our work shows that low bandwidth peers systematically download more than they upload to the network when high bandwidth peers are present. We find that the *rate-based* tit-for-tat policy is not effective in preventing unfairness. We show how simple changes to the tracker and a stricter, block-based tit-for-tat policy, greatly improves fairness.

1. INTRODUCTION

The peer-to-peer (P2P) paradigm has proved to be a promising approach to the problem of delivering a large file from an origin server to large audiences in a scalable manner. Since peers not only download content from the server but also serve it to other peers, the serving capacity of the system grows with the number of nodes, making the system potentially self-scaling. BitTorrent [3] has recently emerged as a very popular and scalable P2P content distribution tool. In BitTorrent, a file is broken down into a large number of blocks and peers can start serving other peers as soon as they have downloaded their first block. Peers preferentially download blocks that are rarest among their local peers so as to maximize their usefulness to other peers. These strategies allow BitTorrent to use bandwidth *between* peers (*i.e.*, *perpendicular bandwidth* [5]) effectively and handle flash crowds well. In addition, BitTorrent incorporates a tit-for-tat (TFT) incentive mechanism; whereby nodes preferentially upload to peers from whom they are able to download at a fast rate in return. This mechanism is especially important since studies have shown that many nodes in P2P systems tend to download content without serving anything [7].

The soundness of these architectural choices is borne out by the success of the system in actual deployment. Anecdotal evidence and accounts in the popular press indicate that BitTorrent has accounted for a large and growing share of P2P Internet traffic. Recent measurement and analytical studies [9, 11, 12] (discussed in Section 3) indicate that BitTorrent handled large distributions effectively, as well as showed desirable scalability properties. However, we believe that these studies leave a number of questions unanswered. For example:

- Biersack et al. [9] reported that clients observed high download rates. Could BitTorrent have achieved even higher bandwidth utilization in this setting? In other words, how far from optimal was BitTorrent's performance?
- BitTorrent employs a Local Rarest First (LRF) policy for choosing new blocks to download from peers. Does this policy achieve its desired objective of avoiding the last block problem?
- How effective is BitTorrent's TFT policy in ensuring that nodes cannot systematically download much more data than they upload? That is, does the system allow unfairness ?
- Previous studies have assumed that at least a fraction of nodes perform altruistic uploading even after finishing their downloads. However, if nodes depart as soon as they finish (as they might with selfish clients), is the stability or scalability of the system hurt significantly?

The answers depend on a number of parameters that BitTorrent uses. It would be difficult, if not impossible, to incorporate and control such a large space of possibilities in

*The author was an intern at Microsoft Research during this work.

an analytical or live measurement setting. Hence, in this paper, we attempt to answer these questions using a simulator which models the data-plane of BitTorrent.[1] We believe our study is complementary to previous BitTorrent studies. Details of the simulator and experimental settings are described in Sections 4 and 5. Our principal findings are:

1. BitTorrent is remarkably robust and scalable at ensuring high uplink bandwidth utilization. It scales well as the number of nodes increases, keeping the load on the origin server bounded.

2. The bandwidth of the origin server is a precious resource especially when it is limited. It is important that server deliver *unique* packets to the network at least as quickly as they can be diffused among the peers.

3. The Local Rarest First (LRF) policy performs better than alternative block-choosing policies in a wide range of environments (e.g., flash crowd, post-flash crowd situations, small network sizes, etc.) By successfully getting rid of the last block problem, it promises to be a simpler alternative to using source coding strategies (to increase the diversity of blocks in the system).

4. BitTorrent's rate based TFT mechanisms does not prevent systematic unfairness in terms of the data served by nodes, especially in node populations with heterogeneous bandwidths. We demonstrate that clustering of similar nodes using *bandwidth matching* is key to ensuring fairness without sacrificing uplink bandwidth utilization.

5. BitTorrent is good at ensuring that new peers, who initially have no packets to offer, rapidly become productive members of the network. However, it not so good, during a flash crowd, at allowing peers who have most of a file to rapidly find the few remaining blocks.

We wish to emphasize that one of the contributions of this paper is in illuminating and remedying *unfairness*, a systematic and previously unaddressed problem in BitTorrent. Note that free-riding and unfairness in P2P networks reduce their effectiveness quite significantly. We believe the changes we suggest to remedy unfairness in BitTorrent increase its usefulness.

The rest of the paper is organized as follows: in Section 2, we present a brief overview of the BitTorrent system. Section 3 discusses related analytical and measurement-based studies. Section 4 describes our simulation environment and the evaluation metrics. Section 5 presents simulation results under a variety of workloads. Finally, Section 6 concludes.

2. BITTORRENT OVERVIEW

BitTorrent [3] is a P2P application whose goal is to enable fast and efficient distribution of large files by leveraging the *upload* bandwidth of the downloading peers. The basic idea is to divide the file into equal-sized *blocks* (typically 32-256 KB) and have nodes download the blocks from multiple peers concurrently. The blocks are further subdivided into *sub-blocks* to enable *pipelining* of requests so as to mask the request-response latency [4].

Corresponding to each large file available for download (called a *torrent*), there is a central component called the *tracker* that keeps track of the nodes currently in the system. The tracker receives updates from nodes periodically (every 30 minutes) as well as when nodes join or leave the torrent.

Nodes in the system are either *seeds*, i.e., nodes that have a complete copy of the file and are willing to serve it to others, or *leechers*, i.e., nodes that are still downloading the file but are willing to serve the blocks that they already have to others. When a new node joins a torrent, it contacts the tracker to obtain a list containing a random subset of the nodes currently in the system (both seeds and leechers). The new node then attempts to establish connections to about 40 existing nodes, which then become its *neighbors*. If the number of neighbors of a node ever dips below 20, say due to the departure of peers, the node contacts the tracker again to obtain a list of additional peers it could connect to.

Each node looks for opportunities to download blocks from and upload blocks to its neighbors. In general, a node has a choice of several blocks that it could download. It employs a *local rarest first (LRF)* policy in picking which block to download: it tries to download a block that is least replicated among its neighbors. The goal is to maximize the diversity of content in the system, i.e., make the number of replicas of each block as equal as possible. This makes it unlikely that the system will get bogged down because of "rare" blocks that are difficult to find.

An exception to the local rarest first policy is made in the case of a new node that has not downloaded any blocks yet. It is important for such a node to quickly bootstrap itself, so it uses the first available opportunity (i.e., an optimistic unchoke, as discussed below) to download a random block. From that point on, it switches to the local rarest first policy.

A *tit-for-tat (TFT)* policy is employed to guard against free-riding: a node preferentially uploads to neighbors that provide it the best download rates. Thus it is in each node's interest to upload at a good rate to its neighbors. For this reason, and to avoid having lots of competing TCP connections on its uplink, each node limits the number of concurrent uploads to a small number, typically 5. Seeds have nothing to download, but they follow a similar policy: they upload to up to 5 nodes that have the highest download rate.

The mechanism used to limit the number of concurrent uploads is called *choking*, which is the temporary refusal of a node to upload to a neighbor. Only the connections to the chosen neighbors (up to 5) are *unchoked* at any point in time. A node reevaluates the download rate that it is receiving from its neighbors every 10 seconds to decide whether a currently unchoked neighbor should be choked and replaced

[1] We do not consider control-plane issues such as the performance of the centralized *tracker* used for locating peers.

with a different neighbor. Note that in general the set of neighbors that a node is uploading to (i.e., its unchoke set) may not exactly coincide with the set of neighbors it is downloading from.

BitTorrent also incorporates an *optimistic unchoke* policy, wherein a node, in addition to the normal unchokes described above, unchokes a randomly chosen neighbor regardless of the download rate achieved from that neighbor. Optimistic unchokes are typically performed every 30 seconds, and serve two purposes. First, they allow a node to discover neighbors that might offer higher download rates than the peers it is currently downloading from. Second, they give new nodes, that have nothing to offer, the opportunity to download their first block. A strict TFT policy would make it impossible for new nodes to get bootstrapped. An overview of related studies of BitTorrent [11, 9, 12] is given in Section 3.

3. RELATED WORK

There have been analytical as well as measurement-based studies of the BitTorrent system. At the analytical end, Qiu and Srikant [12] have considered a simple fluid model of BitTorrent and obtained expressions for the average number of seeds and downloaders in the system as well as the average download time as functions of the node arrival and departure rates and node bandwidth. Their main findings are that the system scales very well (i.e., the average download time is not dependent on the node arrival rate) and that file sharing is very effective (i.e, there is a high likelihood that a node holds a block that is useful to its peers).

A measurement-based study of BitTorrent is presented in [9]. The study is based on data from the "tracker" log for a popular torrent (corresponding to the Linux Redhat 9 distribution) as well data gathered using an instrumented client that participated in the torrent. The main findings are that (a) peers that have completed their download tend to remain connected (as seeds) for an additional 6.5 hours (although the authors note that this could simply be because the BitTorrent client needs explicit user action to be terminated and disconnected from the network after a download completes), (b) the average download rate is consistently high (over 500 kbps), (c) as soon as a node has obtained a few chunks, it is able to start uploading to its peers (i.e., the local rarest first policy works), and (d) the node download and upload rates are positively correlated (i.e., the tit-for-tat policy works).

Another study based on a 8-month long trace of BitTorrent activity is presented in [11]. Some of the findings in this study are different from those reported in [9], perhaps because of the broader range of activities recorded in the trace (statistics are reported for over 60,000 files). The average download bandwidth is only 240 Kbps and only 17% of the peers stay on for one hour or more after they have finished downloading. In general, there are a few highly reliable seeds for each torrent, and these are far more critical for file availability than the much larger number of short-lived seeds. The workload used for our simulations is based on this finding — we typically have one or a small number of long-lived seeds and assume that the other nodes depart as soon as they have finished downloading.

Gkantsidis and Rodriguez [8] present a simulation-based study of a BitTorrent-like system. They show results indicating that the download time of a BitTorrent-like system is not optimal, especially in settings where there is heterogeneity in node bandwidth. They go on to propose a network coding [1] based scheme called Avalanche that alleviates these problems.

Our study differs from previous research in the following important ways: first, while the analytical study reported in [12] presents the steady state scalability properties of BitTorrent, it ignores a number of important BitTorrent parameters (e.g., node degree (d), maximum concurrent uploads (u)), and environmental conditions (e.g., seed bandwidth, etc.) which affect uplink bandwidth utilization. Secondly, previous studies only briefly allude to free-riding; in this paper, we quantify systematic unfairness resulting due to optimistic unchoke and present mechanisms to alleviate it.

4. METHODOLOGY

To explore aspects of BitTorrent that are difficult to study using data traces [9, 11] or analysis [12] we adopted a simulation-based approach for understanding and deconstructing BitTorrent performance. Our choice is motivated by the observation that BitTorrent is composed of several interesting mechanisms that interact in many complex ways depending on the workload offered. Using a simulator provides the flexibility of carefully controlling the input parameters of these mechanisms or even selectively turning off certain mechanisms and replacing them with alternatives. This allows us to explore system performance in scenarios not covered by the available measurement studies [9, 11], and variations on the original BitTorrent mechanism. In this section, we present the details of our simulator and define the metrics we focus on in our evaluation.

4.1 Simulator Details

Our discrete-event simulator models peer activity (joins, leaves, block exchanges) as well as many of the associated BitTorrent mechanisms (local rarest first, tit-for-tat, etc.) in detail. The network model associates a downlink and an uplink bandwidth for each node, which allows modeling asymmetric access networks. The simulator uses these bandwidth settings to appropriately delay the blocks exchanged by nodes. The delay calculation takes into account the number of flows that are sharing the uplink or downlink at either end, which may vary with time. Doing this computation for each block transmission is expensive enough that we have to limit the maximum scale of our experiments to 8000 nodes on a P4 2.7GHz, 1GB RAM machine. Where appropriate, we point out how this limits our ability to extrapolate our findings.

Given the computational complexity of even the simple

model above, we decided to simplify our network model in the following ways. First, we do not model network propagation delay, which is relevant only for the small-sized control packets (*e.g.*, the packets used by nodes to request blocks from their neighbors). We believe that this simplification does not have a significant impact on our results because (a) the download time is dominated by the data traffic (*i.e.*, block transfers), and (b) BitTorrent's pipelining mechanism (Section 2) masks much of the control traffic latency in practice. Second, we do not model the dynamics of TCP connections. Instead, we use a fluid model of connections, which assumes that the flows traversing a link share the link bandwidth equally. Although this simplification means that TCP "anomalies" (e.g., certain connections making faster progress than others) are not modeled, the length of the connections makes at least short-term anomalies less significant. Finally, we do not model shared bottleneck links in the interior of the network. We assume that the bottleneck link is either the uplink of the sending node or the downlink of the receiving node. While Akella et al. [2] characterize bandwidth bottlenecks in the interior of the network, their study specifically ignores edge-bottlenecks by conducting measurements only from well-connected sites (*e.g.*, academic sites). The interior-bottlenecks they find are generally fast enough (≥ 5 Mbps) that the edge-bottleneck is likely to dominate in most realistic settings. Hence we believe that our focus on just edge-bottlenecks is reasonable.

Finally, we make one simplification in modeling BitTorrent itself, by ignoring the *endgame mode*[4]. This is used by BitTorrent to make the end of a download faster by allowing a node to request the sub-blocks it is looking for in parallel from multiple peers. However, neglecting the endgame mode does not qualitatively impact any of the results presented here, since our evaluation focuses primarily on the steady-state performance. Also, this simplification has little or no impact on metrics such as fairness and diversity.

For some of our experiments we also augment the core BitTorrent mechanisms with some new features including block-level TFT policies, bandwidth estimation, etc. Section 5 provides the details at the relevant places.

4.2 Metrics

We quantify the effectiveness of BitTorrent in terms of the following metrics: (a) link utilization, (b) mean download time, (c) content diversity, (d) load on the seed(s), and (e) fairness in terms of the volume of content served. The rest of the section presents a brief discussion of the above metrics.

Link utilization: We use the mean utilization of the peers' uplinks and downlinks over time as the main metric for evaluating BitTorrent's efficacy.[2] The utilization at any point in time is computed as the ratio of the aggregate traffic flow on all uplinks/downlinks to the aggregate capacity of all uplinks/downlinks in the system; *i.e.*, the ratio of the actual flow to the maximum possible.

Given the ad-hoc construction of the BitTorrent network and its decentralized operation, it is unclear at the outset how well the system can utilize the "perpendicular" bandwidth between peers. For instance, since download decisions are made independently by each node, it is possible that a set of nodes decide to download a similar set of blocks, reducing the opportunities for exchanging blocks with each other.

Notice that if all the uplinks in the system are saturated, the system as a whole is serving data at the maximum possible rate. While downlink utilization is also an important metric to consider, the asymmetry in most Internet access links makes the uplink the key determinant of performance. Furthermore, by design, duplicate file blocks (*i.e.*, blocks that a leecher already has) are never downloaded again. Hence, the *mean download time* for a leecher is inversely related to the average uplink utilization. Because of this and the fact that observed uplink utilization is easier to compare against the optimal value (100%), we do not explicitly present numbers for mean download time for most of our experiments.

Fairness: The system should be fair in terms of the number of blocks served by the individual nodes. No node should be *compelled* to upload much more than it has downloaded. Nodes that *willingly* serve the system as a seed are, of course, welcome, but involuntary asymmetries should not be systematic, and free-riding should not be possible. Fairness is important for there to be an incentive for nodes to participate, especially in settings where ISPs charge based on uplink usage or uplink bandwidth is scarce.

As described in Section 2, BitTorrent incorporates a tit-for-tat (TFT) mechanism to block free-riders, i.e., nodes that receive data without serving anything in return. However, it is important to note that this is only a *rate-based* TFT algorithm. For example, a node with a T1 uplink can still open upload connections to a group of modems, if it knows of no alternative peers. In such a case, it will end up serving many more blocks than it receives in return. Also, with the optimistic unchoke mechanism, a node willingly delivers content to a peer for 30 seconds even if it does not receive any data from the peer. These factors can potentially result in unfairness in the system. Our objective is to quantify the amount of unfairness and also to propose mechanisms designed to prevent such unfairness with minimal sacrifice in performance (in terms of link utilization or download time).

Optimality: Throughout this paper we will refer to a system as having optimal utilization if it achieves the maximum possible link utilization, and having complete fairness if every leecher downloads as many blocks as it uploads. We will refer to the system as being overall optimal if it has optimal utilization *as well as* complete fairness. Note that a heterogeneous setting can have differing degrees of fairness at the same level of bandwidth utilization. Consider an example where 50 % of the leechers have download/upload capacity of $100/50$ kbps (Type I) and 50 % have $50/25$ (Type II) and the file has B blocks. Now consider three simple

[2] In our discussion, we use the terms up*load*/down*load* utilization synonymously with up*link*/down*link* utilization.

scenarios:

- Type I leechers only serve other Type I leechers, and Type II leechers only serve Type II.
- Type I leechers only serve Type II leechers, and Type II leechers only serve Type I.
- Each Type I leecher serves half Type I and half Type II leechers. Similarly for Type II leechers.

In all three of these cases, it is possible for the utilization to be optimal. In Scenario 1 both Type I and Type II leechers upload and download B blocks. In Scenario 2 a Type II leecher uploads only $B/2$ blocks before it has downloaded B and is finished; while the Type I leechers will have downloaded $B/2$ and uploaded B by the time their peers disconnect (which seems unfair). In Scenario 3 a Type I leecher uploads $3B/2$ and downloads B while a Type II leecher uploads $B/2$ and downloads B. While all of the scenarios keep bandwidth utilization at its maximum, we consider only scenario 1 to be optimal.

Content diversity: As noted above, the system's effectiveness in utilizing perpendicular bandwidth depends on the diversity of blocks held by the leechers in the system. So we would like to measure the effectiveness of BitTorrent's local rarest first (LRF) mechanism (Section 2) in achieving diversity. We quantify diversity using the distribution of the number of replicas of each block in the system. Ideally, the distribution should be relatively flat, i.e., the number of replicas of each block should approximately be the same.

Load on the seed(s): This is defined as the number of blocks served by the seed(s) in the system. In our presentation here, we normalize this metric by dividing it by the number of blocks in the file. So, for example, a normalized load of 1.5 means that the seed serves a volume of data equivalent to 1.5 copies of the file.

In the specific scenario where nodes depart as soon as they finish their download, this metric is equivalent to the load on the origin server, which is the sole seed in the system. For the system to be scalable, the load per seed should remain constant (or increase only slightly) as the number of leechers in the system increases.

5. EXPERIMENTS

5.1 Workload Derived from a Real Torrent

In order to set the stage for the experiments to follow, we first examine how our simulator performs under a realistic workload. A workload consists of two elements that specify the torrent: (a) node arrival pattern, and (b) uplink and downlink bandwidth distribution of the nodes. To derive realistic arrival patterns, we use the tracker log for the Redhat 9 distribution torrent [9]; thus we have the arrival times of clients in an actual torrent. Unfortunately, the tracker logs have no information about the bandwidths of the arriving clients. So as an approximation, we use the actual client bandwidth distribution reported for Gnutella clients [13]. While discretizing the CDFs presented in [13], we excluded the tail of the distribution. This means that dial-up modems are eliminated, since it is unlikely that they will participate in such large downloads, and very high bandwidth nodes are eliminated, making the setting more bandwidth constrained. Table 1 summarizes the distribution of peer bandwidths. We set the seed bandwidth to 6000 kbps.

Downlink (kbps)	**Uplink** (kbps)	**Fraction**
784	128	0.2
1500	384	0.4
3000	1000	0.25
10000	5000	0.15

Table 1: Bandwidth distribution of nodes derived from the actual distribution of Gnutella nodes [13].

In order to make the simulations tractable, we made two changes. First, we used a file size of 200 MB (with a block size 256 KB), which is much smaller than the actual size of the Redhat torrent (1.7 GB). This means the download time for a node is smaller and the number of nodes in the system at any single point is also correspondingly smaller. Second, we present results only for the *second* day of the flash crowd. This day witnesses over 10000 node arrivals; however, due to the smaller file download time, the maximum number of active nodes in the system at any time during our simulations was about 300.

The results of the simulation are summarized in Table 2. As can be seen the uplink utilization at 91% is excellent, meaning that the overall upload capability of the network is almost fully utilized. However, this comes at the cost of considerable skew in load across the system. Observe that the seed serves approximately 127 copies of the file into the network. Worse, some clients uploaded 6.26 times as many blocks as they downloaded, which represents significant unfairness.

Metric	**Vanilla BitTorrent**
Uplink utilization	91%
Normalized seed load	127.05
Normalized max. #blocks served	6.26

Table 2: Performance of BitTorrent with arrival pattern from Redhat 9 tracker log, and node bandwidths from Gnutella study.

The findings reported in Table 2 raise a number of interesting connections, including:

1. How robust is the high uplink utilization to variations in system configuration and workload? The various aspects of system configuration include the number of seeds and leechers, their arrival and departure patterns (e.g., leechers leaving immediately after completing their download or a seed departing prematurely), bandwidth distribution, etc.?

2. Can the fairness of the system be improved without hurting link utilization?
3. How well does the system perform when there is heterogeneity in terms of the the extent to which leechers have completed their download (e.g., new nodes coexisting with nodes that have already completed most of their download)?
4. How sensitive is system performance to parameters such as the node degree (i.e., the number of neighbors maintained by leechers) and the maximum number of concurrent uploads?

To answer these questions, we present a detailed simulation-based study of BitTorrent in the sections that follow. The key advantage of a simulation-based study is that it can provide insight into system behavior as the configuration and workload parameters are varied in a controlled manner.

5.2 Road-map of Experiments

We use the following default settings in our experiments, although we do vary these settings in specific experiments, as noted in later sections:

- File size: 102400 KB = 100 MB (400 blocks of 265 KB each)
- Number of initial seeds: 1 (the origin server, which stays on throughout the duration of the experiment)
- Seed uplink bandwidth: 6000 Kbps
- Number of leechers that join the system (n): 1000
- Leecher downlink/uplink bandwidth: 1500/400 Kbps
- Join/leave process: a flash crowd where all nodes join within a 10-second interval. Leechers depart as soon as they finish downloading.
- Node degree (d): 7. Node degree defines the size of the neighborhood used to search for the local rarest block.
- Limit on the number of concurrent upload transfers (u): 5 (includes the connection that is optimistically unchoked)

As a very gross simplification the parameters that affect the evolution of a torrent are: (1) the seed(s) and its serving capacity, (2) the number of leechers that wish to download, (3) the policies that nodes use to swap blocks among themselves, (4) the distribution of node upload/download capacities and (5) the density of the arrivals of the nodes(*e.g.*, leecher arrival pattern such as flash crowd). To tackle the effects sequentially we start in Section 5.3 by examining only (1), (2) and (3). That is, we consider a homogeneous setting where all leechers have the same downlink/uplink bandwidth (1500/400 Kbps by default, as noted above) and only a flash crowd is considered. We explore the impact of the number of leechers, the number of initial seeds, aggregate bandwidth of seeds, bandwidth of leechers, and the number of concurrent upload transfers (u). We also evaluate BitTorrent's LRF policy for picking blocks for different settings of node degree (d), and compare it with simpler alternatives such as random block picking.

Then in Section 5.4 we examine (4) and turn to a heterogeneous flash-crowd setting where there is a wide range in leecher bandwidth. We consider 3 kinds of connectivity for leechers: high-end cable (6000/3000 Kbps), high-end DSL (1500/400 Kbps), and low-end DSL (784/128 Kbps). Our evaluation shows that BitTorrent can display systematic unfairness to the detrimant of high bandwidth peers; and we suggest a number of approaches to remedy the problem.

Finally, in Section 5.5, we turn to (5) and consider workloads other than a pure flash-crowd scenario. In particular we consider cases where leechers with very different "objectives" coexist in the system. For instance, new nodes in the post-flash crowd phase will be competing with nodes that have already downloaded most of the blocks. Likewise, an old node that reconnects during the start of a new flash crowd to finish the remaining portion of its download would be competing with new nodes that are just starting their downloads. We wish to determine how well BitTorrent's mechanisms work in such settings.

5.3 Homogeneous Environment

In this section, we study the performance of BitTorrent in a setting consisting of a homogeneous (with respect to bandwidth) collection of leechers. Unless specified otherwise, we use the default settings noted in Section 5.2 for file size (102400 KB), seed bandwidth (6000 Kbps), leecher bandwidth (1500/400 Kbps), and join/leave process (1000 leechers join during the first 10 seconds and leave as soon as they finish downloading).

5.3.1 *Number of nodes*

First we examine the performance of the system with increasing network size. We vary the number of nodes (*i.e.*, leechers) that join the system from 50 to 5000. All nodes join during a 10 second period, and remain in the system until they have completed the download. The goal is to understand how performance varies with scale. Figure 1 plots the mean utilization of the aggregate upload and download capacity of the system (*i.e.*, averaged across all nodes and all time). We find that the upload capacity utilization is close to 100% regardless of system size. (Utilization is a little short of 100% because of the start-up phase when nodes are unable to utilize their uplinks effectively.) The high uplink utilization indicates that the system is performing almost optimally in terms of mean download time. The downlink utilization, on the other hand, is considerably lower. Clearly the total download rate cannot exceed the total upload rate plus the seed's rate. Thus the download utilization will generally be limited by the upload capacity (when leechers have greater download than upload capacity). An exception is

when the number of leechers is so small that they can directly receive significant bandwidth from the seed; this can be seen in Figure 1 by the slight rise in download utilization when the network size is under fifty nodes.

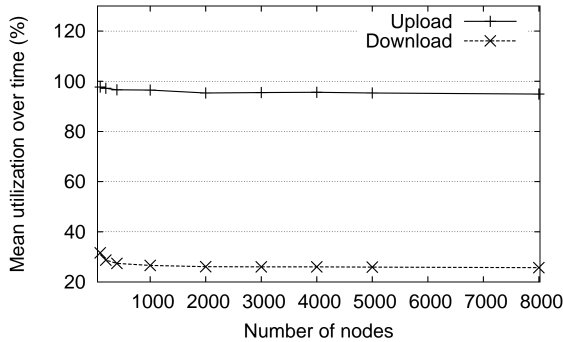

Figure 1: Mean upload and download utilization of the system as the flash-crowd size increases. Observe that the mean upload utilization is almost 100%, even as the network size increases. The download utilization is upper bounded by the ratio of the leechers upload to download bandwidths.

Another important measure of scalability is how the work done by the seed varies with the number of leechers. We measure this in terms of the normalized number of blocks served, *i.e.*, the number of blocks served divided by the number of blocks in one full copy of the file. Ideally, we would like the work done by the seed to remain constant or increase very slowly with system size. Figure 2 shows that this is actually the case. The normalized number of blocks served by the seed rises sharply initially (as seen from the extreme left of Figure 2) but then flattens out. The initial rise indicates that the seed is called upon to do much of the serving when the system size is very small, but once the system has a critical mass of 50 or so nodes, peer-to-peer serving becomes very effective and the seed has to do little additional work even as the system size grows to 8000.

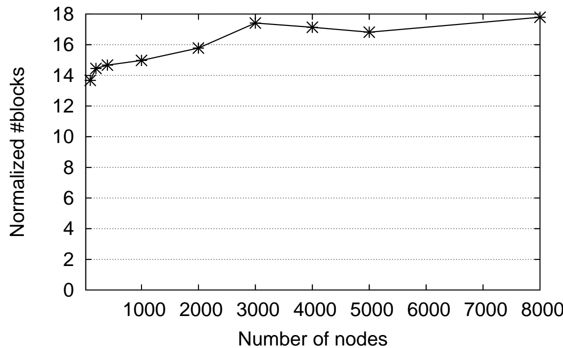

Figure 2: Contribution of the seed as the flash-crowd size increases. Observe that the amount of work done by the seed is almost independent of the network size, indicating that (at least in this scenario) the system scales very well.

In summary, BitTorrent performance scales very well with increasing system size both in terms of bandwidth utilization and the work done by the seed.

5.3.2 Number of seeds and bandwidths of seeds

Next we consider the impact of numbers of seeds and aggregate seed bandwidth on the performance of BitTorrent. We first consider the case where there is a single seed, and then move on to the case of multiple seeds. We fix the number of leechers that join the system to 1000.

Figure 3 shows the mean upload utilization (which in turn determines the mean file download time) as the bandwidth of a single seed varies from 200 Kbps to 1000 Kbps. The "nosmartseed" curve corresponds to default BitTorrent behavior. We see that upload utilization is very low (under 40%) when the seed bandwidth is only 200 Kbps. This is not surprising since the low seed bandwidth is not sufficient to keep the uplink bandwidth of the leechers (400 Kbps) fully utilized, at least during the start-up phase. However, even when the seed bandwidth is increased to 400 or 600 Kbps, the upload utilization is still considerably below optimal.

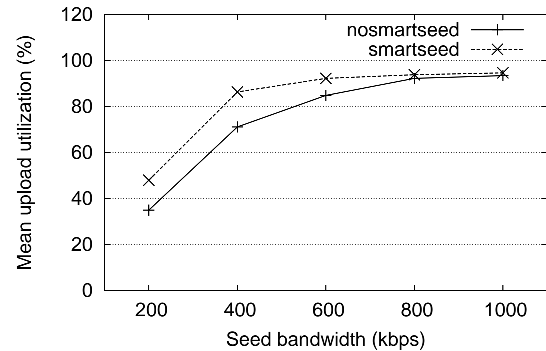

Figure 3: Upload utilization as the bandwidth of the seed is varied. By avoiding duplicate block transmissions from the seed, the "smartseed" policy improves utilization significantly.

Part of the reason for poor upload utilization is that seed bandwidth is wasted serving duplicate blocks prematurely, *i.e.*, even before one full copy of the file has been served. To see that this is so, examine the "nosmartseed" curve in Figure 4. This plots the total number of blocks served by the seed by the time one full copy of the file is served, as a function of seed bandwidth. Whenever this total number of blocks served is higher than the unique number of blocks in the file (400), it indicates that duplicate blocks were served prematurely. We believe this to be a problem since it decreases the block diversity in the network. That is, despite the Local Rarest First (LRF) policy, multiple leechers connected to the seed can operate in an uncoordinated manner and independently request the same block.

Once identified there is a simple fix for this problem. We have implemented a *smartseed* policy, which has two components: (a) The seed does not choke a leecher to which it has transferred an incomplete block. This maximizes the opportunity for leechers to download and hence serve complete blocks. (b) For connections to the seed, the LRF policy is replaced with the following: among the blocks that a leecher is looking for, the seed serves the one that it has served the least. This policy improves the diversity of blocks in the

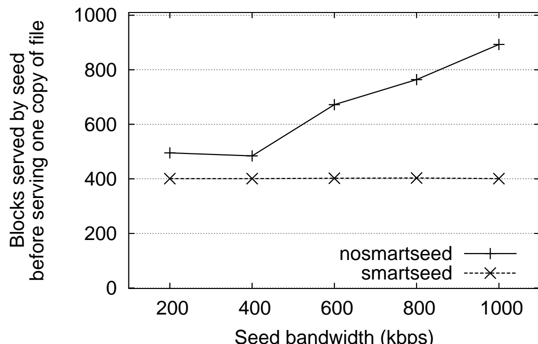

Figure 4: Variation of the total #blocks served by the seed before it has served at least one copy of each block in the file.

system, and also eliminates premature duplicate blocks, as shown in Figure 4. This results in noticeable improvement in upload utilization, especially when seed bandwidth is limited and precious (Figure 3).

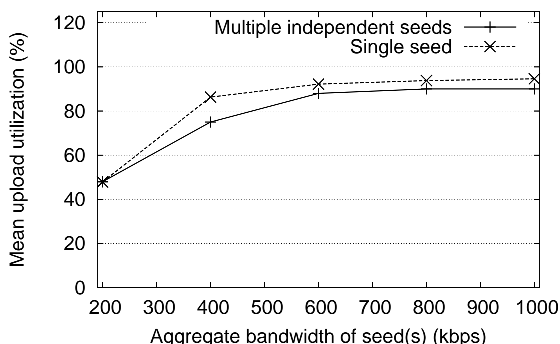

Figure 5: Upload utilization for a single seed versus multiple independent seeds. The lack of coordination among the independent seeds results in duplicate blocks being served by different seeds and a corresponding penalty in uplink utilization.

Finally, Figure 5 compares the cases of having a single seed and having multiple independent seeds, each with 200 Kbps bandwidth, such that the aggregate seed bandwidth is the same in both cases. All seeds employ the smartseed policy. The upload utilization suffers in the case of multiple seeds because the independent operation of the seeds results in duplicate blocks being served by different seeds, despite the smartseed policy employed by each seed.

In summary, we find that seed bandwidth is a precious resource and it is important not to waste it on duplicate blocks until all blocks have been served at least once. The "smartseed" policy, which modifies LRF and the choking policy for the seeds' connections, results in a noticeable improvement in system performance.

5.3.3 Block choosing policy and Node degree

Next we address the question of the block choosing policy. As mentioned earlier the LRF policy appears to be one of the key ingredients in BitTorrent. Here, we investigate how important it is, and show when it matters and when it does not. We will assume that the seed employs the *smartseed* strategy introduced in the last section and comment only qualitatively on the results otherwise.

Before describing our experiments let us quickly revisit the intuition behind the LRF policy. Since any rare block will automatically be requested by many leechers, it is unlikely to remain rare for long. For example, if a rare block is possessed by only one leecher, it will be among the first blocks requested by any nodes unchoked by that leecher. This, of course, decreases its rareness until it is as common in the network as any other block. This should reduce the coupon collector or "last block problem" that has plagued many file distribution systems [6]. These arguments are qualitative. The goal of this section is to measure how well LRF actually performs.

We investigate 3 issues. First, we compare LRF with an alternative block choosing policy in which each leecher asks for a block picked at random from the set that it does not yet possess but that is held by its neighbors. Second, we examine how the effectiveness of LRF varies as the seed bandwidth is varied. Since a high-bandwidth seed delivers more blocks to the network, the risk of blocks becoming rare is lower. Third, we examine the impact of varying the node degree, d, which defines the size of the neighborhood used for searching in the LRF and random policies.

Figure 6 summarizes the results with regard to the following issues: (a) random *vs.*LRF, (b) low seed bandwidth (400 Kbps) *vs.*high seed bandwidth (6000 Kbps), and (c) node degree, $d = 4, 7$, and 15. In all cases, the leechers had down/up bandwidths of 1500/400 Kbps. Observe that the low bandwidth seed has only as much upload capacity as one of the leechers.

The general trend is that uplink utilization improves with increases in both seed bandwidth and node degree. When node degree is low ($d = 4$), leechers have a very restricted local view. So LRF is not effective in evening out the distribution of blocks at a global level, and performs no better than the random policy. However, when node degree is larger ($d = 7$ or 15) and seed bandwidth is low, LRF outperforms the random policy by ensuring greater diversity in the set of blocks held in the system. Finally, when the seed bandwidth is high, the seed's ability to inject diverse blocks into the system improves utilization and also eliminates the performance gap between LRF and the random policy. Thus, LRF makes a difference only when node degree is large enough to make the local neighborhood representative of the global state and seed bandwidth is low.

In Figure 7 we graph the average number of *interesting* connections available to each leecher in the network for the case of $d = 7$. The connection between a node and its peer is called *interesting* if the node can send useful data to its peer. As stated in the caption, each point here represents the mean number of interesting connections (averaged over all the nodes in the system) at a particular point in time. Observe that in the high seed bandwidth case there is little difference between the LRF and the random block choos-

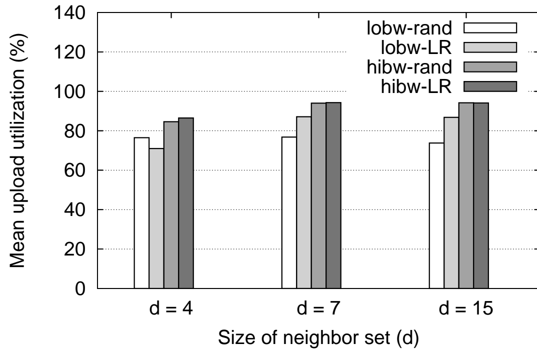

Figure 6: Upload utilization for LRF and Random policies for different values of the node degree, d. LRF performs better only when the node degree is large and the seed bandwidth is low.

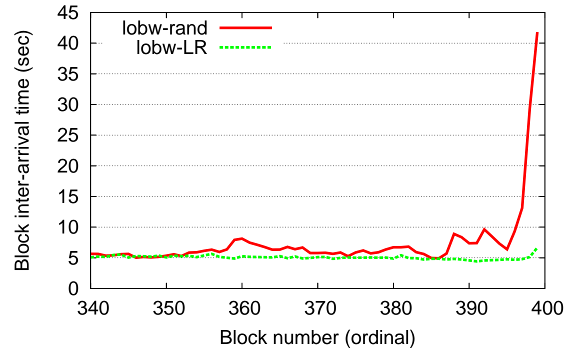

Figure 8: Inter-arrival times for blocks at the tail end of the file. Each point represents the mean time to receive the kth block, where the mean is taken over all nodes. Random clearly shows the last-block problem.

ing policies (the top 2 curves in Figure 7). In the low seed bandwidth case the difference is very pronounced. Observe that with the LRF policy, the number of *interesting* connections is significantly higher, especially towards the end of the download. This underlines the importance of the LRF policy in the case where seed bandwidth is low.

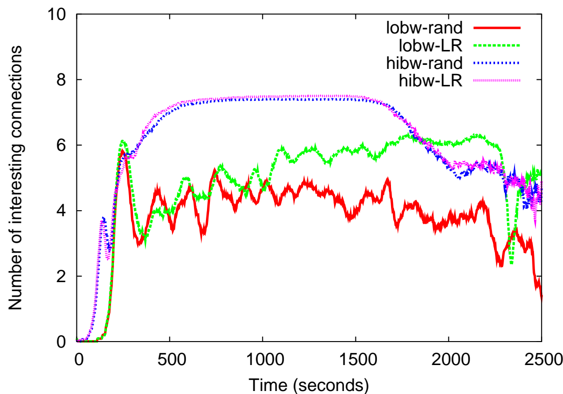

Figure 7: Variation of the number of *interesting* connections over time for $d = 7$ and various settings of seed bandwidth and block choosing policy. Each point represents the mean across all nodes present in the system at that time.

Next we plot in Figure 8 the inter-arrival time between blocks in the case of a low-bandwidth seed. This is the time between the receipt of consecutive distinct blocks, averaged across all nodes. We plot this for both the LRF and the random block choosing policies, with $d = 7$ in both cases. Recall that the file size is 400 blocks, so the figure only shows the inter-arrival time of the last few blocks. The sharp upswing in the curve corresponding to the random policy clearly indicates the last-block problem. There is no such upswing with LRF.

In summary, our results indicate that the LRF policy provides significant benefit when seed bandwidth is low and node degree is large enough for the local neighborhood of a node to be representative of the global state. Nevertheless, we find that the node degree needed for LRF to be effective is quite modest relative to the total number of nodes in the system. Specifically, in a configuration with 8000 nodes, we find that LRF is effective for $d = 7$, which corresponds to each node having direct visibility to a neighborhood that represents only 0.09% of the system. However, given the scaling limitations of our simulator, we are not in a position to extrapolate this result to larger system sizes.

5.3.4 Concurrent Uploads

In BitTorrent, each node uploads to no more than a fixed number of nodes ($u = 5$, by default) at a time. This fixed upload degree limit presents two potential problems. First, having too many concurrent uploads delays the availability of full blocks to the network. That is, if a leecher's upload capacity is divided between u nodes, there can be a considerable delay before any of them has a complete block that they can start serving to others. Second, low peer downlink bandwidth can constrain uplink utilization. That is, a leecher uploading to a peer can find its *upload* pipe underutilized if the receiving node actually becomes the bottleneck on the transfer (*i.e.*, has insufficient available *download* bandwidth to receive as rapidly as the sender can transmit).

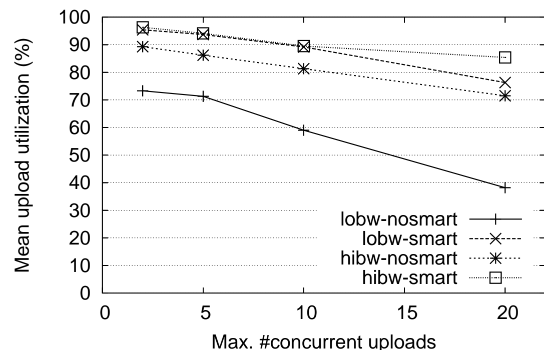

Figure 9: Utilization for different values of the maximum number of concurrent uploads (u).

Figure 9 graphs the mean upload utilization as a function of the maximum number of concurrent uploads permitted (*i.e.*, u) for low and high bandwidth seeds. We show the results both with and without the *smartseed* fix. (Since u can be no more than d, we used $d = 60$ rather than 7 in this experiment, to allow us to explore a wide range of settings

for u.) As u increases (and the *smartseed* fix is not applied), the probability that duplicate data is requested from the seed increases, causing link utilization to drop. The drop in utilization is very severe when seed bandwidth is low, since in such cases, as we have seen before, good performance critically depends on the effective utilization of the seed's uplink. We see utilization dropping gradually even when the *smartseed* fix is applied. The reason is that a large u causes the seed's uplink to get fragmented, increasing the time it takes for a node to fully download a block that it can then serve to others.

To address both the problems of underutilization and fragmentation of the seed's uplink, we propose the following fix: instead of having a fixed upload degree, a node should unchoke the minimum number of connections needed to fully utilize the available bandwidth on its upload link. In practice, however, we may want to have somewhat more than the minimum number of connections, to accommodate bandwidth fluctuations (say due to competing traffic) on any one flow. We plan to investigate this in future work.

5.4 Heterogeneous Environment

In this section, we study the behavior of BitTorrent when node bandwidth is heterogeneous. As described in Section 4.2, a key concern in such environments is fairness in terms of the volume of data served by nodes. Recall, that in the Redhat torrent given in table 2, some nodes uploaded 6.26 times as many blocks as they downloaded; and we wish to avoid such unfairness. This is especially important since uplink bandwidth is generally a scarce resource. BitTorrent only implements a *rate-based* TFT policy, which can still result in unfairness in terms of the volume of data served. This section quantifies the extent of the problem and presents mechanisms that enforce stricter fairness without hurting uplink utilization significantly.

A node in BitTorrent unchokes those peers from whom it is getting the best download rate. The goal of this policy is to match up nodes with similar bandwidth capabilities. For example, a high-bandwidth node would likely receive the best download rate from other high-bandwidth nodes, and so would likely be uploading to such high-bandwidth nodes in return. To help nodes discover better peers, BitTorrent also incorporates an optimistic unchoke mechanism. However, this mechanism significantly increases the chance that a high bandwidth node unchokes and transfers data to nodes with poorer connectivity. Not only can this lead to decrease in uplink utilization (since the download capacity of the peer can become the bottleneck), it can also result in the high bandwidth node serving a larger volume of data than it receives in return. This also implies that the download times of lower bandwidth nodes will improve at the cost of higher bandwidth nodes.

We now consider two simple mechanisms that can potentially reduce such unfairness: (a) Quick bandwidth estimation (QBE), and (b) Pairwise block-level TFT. Note that enforcing fairness implies that the download time of a node will be inversely related to its *upload* capacity (assuming that its uplink is slower than its downlink).

5.4.1 Quick Bandwidth Estimation

In BitTorrent, optimistically unchoked peers are rotated every 30 seconds. The assumption here is that 30 seconds is a long enough duration to establish a reverse transfer and ascertain the upload bandwidth of the peer in consideration. Furthermore, BitTorrent estimates bandwidth only on the transfer of blocks; since all of a node's peers may not have interesting data at a particular time, opportunity for discovering good peers is lost.

Instead, if a node were able to quickly estimate the upload bandwidth for all its d peers, optimistic unchokes would not be needed. The node could simply unchoke the u peers out of a total of d that offer the highest upload bandwidth.

In practice, a quick albeit approximate bandwidth estimate could be obtained using lightweight schemes based on the packet-pair principle [14] that incur much less overhead than a full block transfer. Also, the history of past interactions with a peer could be used to estimate its upload bandwidth.

In our experiments here, we neglect the overhead of QBE and effectively simulate an idealized bandwidth estimation scheme whose overhead is negligible relative to that of a block transfer.

5.4.2 Pairwise Block-Level Tit-for-Tat

The basic idea here is to enforce fairness directly in terms of blocks transferred rather than depending on rate-based TFT to match peers based on their upload rates. Suppose that node A has uploaded U_{ab} blocks to node B and downloaded D_{ab} blocks from B. With pairwise block-level TFT, A allows a block to be uploaded to B if and only if $U_{ab} \leq D_{ab} + \Delta$, where Δ represents the unfairness threshold on this peer-to-peer connection. This ensures that the maximum number of *extra* blocks served by a node (in excess of what it has downloaded) is bounded by $d\Delta$, where d is the size of its neighborhood. Note that with this policy in place, a connection is (un)choked depending on whether the above condition is satisfied or not. Also, there is no need for the choker to be invoked periodically.

Thus, provided that Δ is at least one (implying that new nodes can start exchanges), this policy replaces the optimistic unchoke mechanism and bounds the disparity in the volume of content served. However, it is important to note that there is a trade-off here. The block-level TFT policy may place a tighter restriction on data exchanges between nodes. It may so happen, for example, that a node refuses to upload to any of its neighbors because the block-level TFT constraint is not satisfied, reducing uplink utilization. We quantify this trade-off in the evaluation presented next.

5.4.3 Results

We now present performance results for vanilla BitTorrent as well as the new mechanisms described above with respect to three metrics: (a) mean upload utilization (Figure 10), (b)

unfairness as measured by the maximum number of blocks served by a node (Figure 11), and (c) mean download time for nodes of various categories (Figure 14). All experiments in this section use the following settings: a flash-crowd of 1000 nodes joins the torrent during the first 10 seconds. In each experiment, there are an equal number of nodes with high-end cable modem (6000 Kbps down; 3000 Kbps up), high-end DSL (1500 Kbps down; 400 Kbps up), and low-end DSL (784 Kbps down; 128 Kbps up) connectivity. We vary the bandwidth of the seed from 800 Kbps to 6000 Kbps. Seeds always utilize the *smartseed* fix.

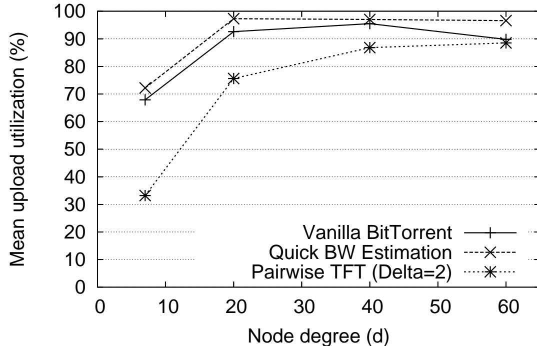

Figure 10: Mean upload utilization for (a) vanilla BitTorrent, (b) BitTorrent with QBE, and (c) with the pairwise block-level TFT policy.

Figure 10 shows the mean upload utilization of BitTorrent and other policies in a heterogeneous setting, as a function of node degree. We find that utilization is sub-optimal in many cases, and especially low with pairwise block-level TFT, when the node degree is low ($d = 7$). The reason is that when the node degree is low, high-bandwidth nodes sometimes have only low-bandwidth peers as neighbors. This restricts the choice of nodes that the high-bandwidth node can serve to such low-bandwidth nodes, despite the QBE heuristic. A bandwidth bottleneck at the *downlink* of the low-bandwidth peer would reduce the uplink utilization at the high-bandwidth node. This degradation is particularly severe with pairwise block-level TFT, since in this case the high-bandwidth node is constrained to upload at a rate no greater than the *uplink* speed of its low-bandwidth peers. In all cases, uplink utilization improves as the node degree becomes larger, since the chances of a high-bandwidth node being stuck with all low-bandwidth peers decreases.

The interaction between high-bandwidth nodes and their low-bandwidth peers also manifests itself in terms of a disparity in the volume of data served by nodes. Figure 11 plots the maximum number of blocks served by a node normalized by the number of blocks in the file. The seed node is not included while computing this metric. We would like to point out that Jain's fairness index [10], computed over the number of blocks served by each node, is consistently close to 1 for all schemes implying the schemes are fair "on the whole".

However, as Figure 11 shows, some nodes can still be very unlucky, serving more than 7 times as many blocks as they

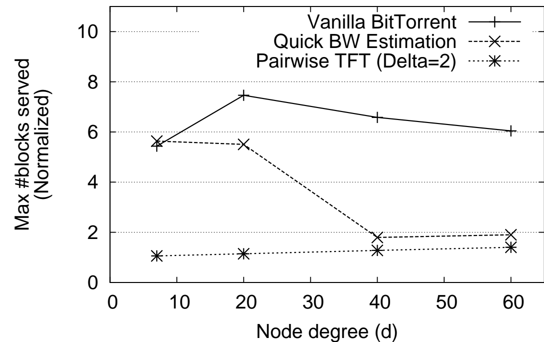

Figure 11: Maximum number of blocks (normalized by file size) served by any node during an experiment for (a) vanilla BitTorrent, (b) BitTorrent with QBE, and (c) with the pairwise block-level TFT policy.

receive in certain situations. All of these unlucky nodes are in fact high-bandwidth nodes. The pairwise block-level TFT policy eliminates this unfairness by design. Figure 11 bears this out. Also, the QBE heuristic reduces unfairness significantly when the node degree is large enough that block transfers between bandwidth-mismatched nodes can be avoided.

Bandwidth-matching tracker policy

To alleviate the problems resulting from block transfers between bandwidth-mismatched nodes, we investigate a new *bandwidth-matching tracker* policy. The idea here is for the tracker to return to a new node a set of candidate neighbors with similar bandwidth to it. This can be accomplished quite easily in practice by having nodes report their bandwidth to the tracker at the time they join. (We ignore the possibility of nodes gaming the system by lying about their bandwidth.) Having bandwidth-matched neighbors would avoid the problems arising from bandwidth-mismatched pairings.

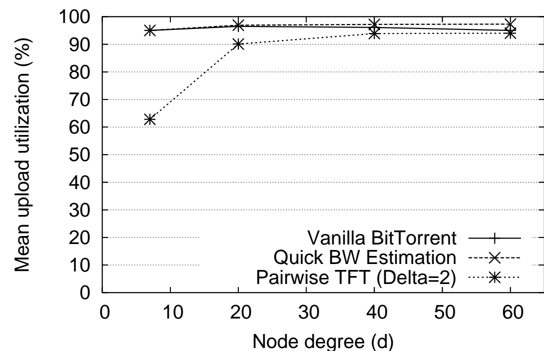

Figure 12: Mean upload utilization with the bandwidth-matching tracker policy in use for (a) vanilla BitTorrent (but for the new bandwidth-matching tracker policy), (b) BitTorrent with QBE, and (c) with the pairwise block-level TFT policy. Compare with Figure 10.

Care is needed in designing this policy. Having the tracker strictly return only a list of bandwidth-matched peers runs the risk of significantly diminishing the resilience of the peer-to-peer graph, by having only tenuous links between "clouds" of bandwidth-matched nodes. In fact, we have found several instances in our experiments where groups of clients were

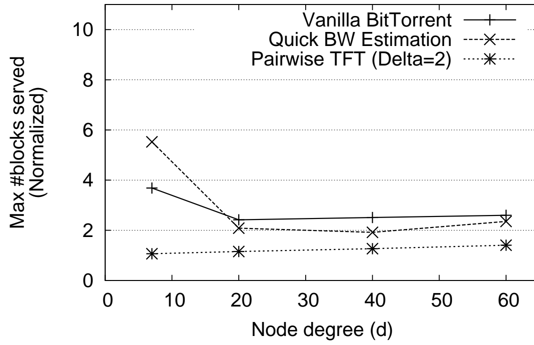

Figure 13: Maximum number of blocks (normalized by file size) served by any node with the bandwidth-matching tracker policy in use for (a) vanilla BitTorrent (but for the new bandwidth-matching tracker policy), (b) BitTorrent with QBE, and (c) with the pairwise block-level TFT policy. Compare with Figure 11.

disconnected from the rest of the network and the disconnection did not heal quickly because the tracker, when queried, would often return a list of peers that are also in the disconnected component.

To avoid this problem, we employ a hybrid policy where the tracker returns a list of peers, 50% of which are bandwidth-matched with the requester and 50% are drawn at random. The former would enable the querying node to find bandwidth-matched neighbors whereas the latter would avoid the disconnection problem.

Figures 12 and 13 show the upload utilization and fairness metrics, respectively, with the (hybrid) bandwidth-matched tracker policy in place. We find a significant improvement in both metrics across a range of values of node degree, as can be seen by comparing Figures 12 and 10 and Figures 13 and 11.

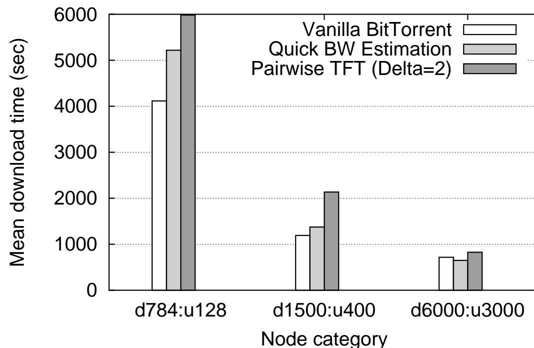

Figure 14: Download times for nodes of different categories for various schemes.

Finally, Figure 14 presents another view of the performance of these policies by plotting the mean download time for each category of nodes. We present results for the setting where seed bandwidth is 1500 kbps and $d = 20$. On the whole, we find that even for vanilla BitTorrent, download times for nodes decrease as the download and upload capacities of the nodes increase. Thus, the system appears to be fair.

However, a comparison with the QBE and block-level TFT policies reveals that, with vanilla BitTorrent, nodes with low uplink bandwidth can actually finish faster – this is because they can get connected to high-bandwidth nodes. The QBE and block-level TFT policies, on the other hand, attempt to minimize such unfairness by connecting nodes of similar bandwidths with each other. A consequence of desiring high fairness is that download times of nodes become inversely proportional to their uplink capacities. In a similar vein, we expect that high-bandwidth nodes should have lower download times since they no longer subsidize other nodes. However, this happens only for the QBE heuristic. In case of the block-level TFT policy, reduced uplink utilization nullifies this benefit and increases download times slightly.

In summary, we find that a bandwidth-unaware tracker combined with the optimistic unchoke mechanism in BitTorrent results in nodes with disparate bandwidths communicating with each other. This results in lower uplink utilization and also creates unfairness in terms of volume of data served by nodes. However, it is possible to obtain a reasonable combination of high upload utilization and good fairness with simple modifications to BitTorrent. Whereas the pairwise block-level TFT policy achieves excellent fairness and good upload utilization, the QBE heuristic achieves excellent upload utilization and good fairness. The hybrid bandwidth-matching tracker policy is critical to both.

5.5 Other Workload

In this section, we consider node arrival patterns other than a pure flash crowd. We also consider the case where the seed departs prematurely, i.e., before all nodes have completed their download.

5.5.1 Divergent Download Goals

Thus far we have focused on the performance of BitTorrent in flash-crowd scenarios. While a flash-crowd setting is important, it also has the property that each node is typically in “sync” with its peers in terms of the degree of completion of its download. For instance, all nodes join the flash crowd at approximately the same time and with none of the blocks already downloaded.

However, there are situations, such as the post-flash-crowd phase, where there may be a greater diversity in the degree of completion of the download across the peers. This in turn would result in a divergence in the download goals of the participating nodes — those that are starting out have a wide choice of blocks that they could download whereas nodes that are nearing the completion of their download are looking for specific blocks.

Here we consider two extremes of the divergent goals scenario. In the first case, a small number of new nodes join when the bulk of the existing nodes are nearing th completion of their download. This might reflect a situation where new nodes join in the post-flash-crowd phase. In the second case, a small number of nodes that have already completed the bulk of their download (at some point in the past) rejoin

the system during a subsequent flash crowd to complete their download. The majority of their peers in this case would be nodes that have not downloaded much of the file.

Performance of Nodes in the Post-Flash Crowd Phase

A post flash-crowd scenario is different from a flash-crowd in that there may be a wide range in the fraction of the download completed by each node. Nodes that have been present in the system longer are typically looking for a more specific set of blocks. Thus, it may be harder for a newcomer to establish a TFT exchange with such older nodes, which could lead to increased download times as well as greater load on the seed. Our goal here is to investigate whether this problem actually happens and how severe it is.

We start with a flash crowd of 1000 nodes joining in the first 10 seconds of the experiment. Then, a batch of 10 nodes is introduced into the system at 1800 seconds, when the flash-crowd nodes have finished downloading approximately 80% of the file-blocks. All nodes have down/up bandwidths of 1500/400 Kbps. We use two settings for seed bandwidth: 800 Kbps (low) and 6000 Kbps (high). The seed node utilizes the *smartseed* fix.

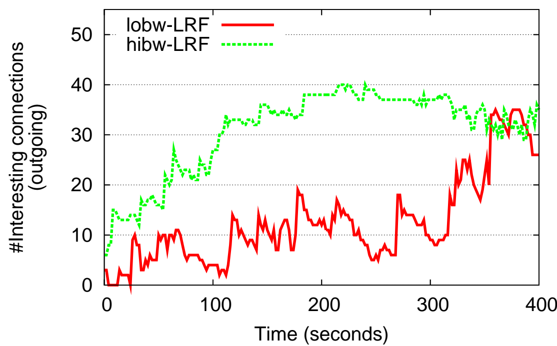

Figure 15: Number of *interesting* outgoing connections of a randomly sampled post flash-crowd node for various configurations.

Figure 15 plots the number of *interesting* outgoing connections over time for a randomly chosen newly joined node *until all the flash-crowd nodes leave*. An outgoing connection is deemed interesting if the node in question has some block that its peer needs. Note that the newcomer would be interested in content from almost all its peers during the first several seconds since it does not have any block to start with. Thus, for every interesting connection, the newcomer can establish a TFT exchange with its peer.

Figure 15 shows that a newcomer is quickly able to gather blocks that are interesting to at least a few of its peers, as seen from the non-zero count of interesting connections in the figure. The reason that a newcomer is quickly able to establish interesting connections to its peers is as follows: if p is the probability that a downloaded block is interesting to some neighbor, and if this probability is the same and independent for each neighbor, then the probability that a downloaded block is useful to at least one neighbor is $1-(1-p)^d$. This probability increases very quickly with d, even if p is relatively small. Thus, while a large degree, d, may not be necessary for a flash-crowd situation, making the degree very small can negatively impact TFT performance for new nodes in the post-flash-crowd phase.

Performance of Pre-seeded Nodes

We now consider the case where a small number of have already completed the bulk of their download (i.e., nodes that have been "pre-seeded" with the bulk of the blocks) rejoin the system during a subsequent flash crowd to complete their download. The key question is whether and to what extent such pre-seeded nodes are penalized because they are looking for specific blocks whereas the majority of nodes in the system are interested in most of the blocks (since they have few blocks).

Again, we start with a flash-crowd of 1000 nodes joining in the first 10 seconds. After that, a new node is introduced every 200 seconds into the system. Each new node is seeded with a random selection of k% blocks – this simulates a situation where the node completed k% of its download, disconnected, and then re-joined during a subsequent flash-crowd to finish its download. Ideally, a node that is pre-seeded with k% of the blocks should take approximately $(1-\frac{k}{100})T$ time to download the remaining blocks, where T is the mean time to download the entire file. ($T = 2000$ seconds, for this setting.) However, a pre-seeded node could take longer because the specific blocks that it is looking for may be hard to find, a penalty that we would like to quantify.

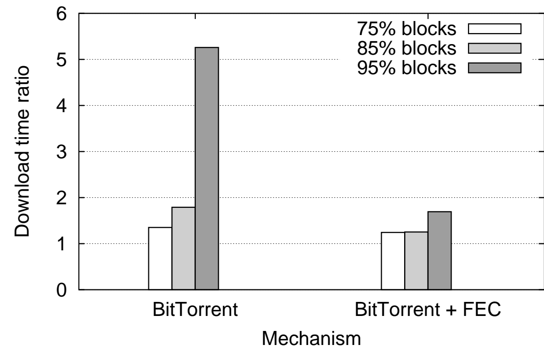

Figure 16: Download time ratios for a pre-seeded node introduced into the system at 200 seconds into the flash crowd. We show results both for vanilla BitTorrent and BitTorrent with source-based FEC.

Figure 16 plots the ratio of actual download time to the expected download time for such a "pre-seeded" node that joined 200 seconds into the flash crowd, for different values of k. A ratio close to 1.0 indicates that a pre-seeded node does not have to wait substantially longer than ideal. We use a seed bandwidth of 6000 Kbps in this experiment; thus, the seed has injected at least one copy of each block into the system at approximately 135 seconds.

From the bars labeled "BitTorrent" in Figure 16, we see that as the number of blocks required by the pre-seeded node decreases, the likelihood of the node taking longer than ideal to finish increases.[3] There are two reasons for this behavior:

[3]Note that this increase is in the *ratio* of the actual to ideal download times,

first, each block takes a non-trivial amount of time to spread from the seed to every node in the system. The maximum possible fanout of this distribution tree is bounded by $u = 5$ (refer Section 5.2). Furthermore, the degree d of the pre-seeded node determines how quickly it can "intercept" this distribution tree. The second reason is that a pre-seeded node is looking for specific blocks, and would like these blocks to be replicated quickly. However, BitTorrent's LRF policy dictates that all blocks get replicated equally so that none remains rare. This "resource-sharing" across blocks decreases the distribution rate of the specific blocks desired by the pre-seeded node, resulting in larger download times.

Notice that pre-seeded nodes are delayed basically because they are looking for *specific* blocks. If the source were to employ FEC and inject a large number of *equivalent* coded blocks into the system, pre-seeded nodes would have more choices for blocks to download and hence should be able to reduce the download time penalty. We repeated the above experiment with the source introducing 100% additional FEC coded blocks. As shown in the bars labeled "BitTorrent+FEC" in Figure 16, the download time ratio with FEC are substantially lower. The download time ratio is close to 1.0 for $k = 75\%$ and 85%, and well under 2.0 even when $k = 95\%$.

Summary

Our experiments with the divergent goals scenarios indicates that BitTorrent tends to "equalize" the performance of newly joined nodes that have fewer or more blocks than the average node. The ones that have fewer blocks are "pulled up" since the LRF mechanism is able to ensure that the new nodes quickly become effective in TFT exchanges. The ones that have a larger number of blocks get "pulled down" (even if the penalty may not be much in terms of absolute time) because the LRF policy does not preferentially replicate the specific blocks that such nodes are looking for. A simple application of source-based FEC can significantly reduce the severity of this problem.

5.5.2 Premature Seed Departure

We also experimented with flash-crowd scenarios where the origin server leaves the system after serving exactly one copy of each block. If blocks are dispersed quickly and widely by BitTorrent, this should not matter and most nodes in the flash-crowd should be able to finish. We observed this behavior consistently except in heterogeneous environments where seed bandwidth was low. In such cases, the higher bandwidth nodes which are connected to the seed get their last block from the seed and exit immediately without serving these blocks to any other node. If the seed bandwidth is not constrained, all unique blocks are injected into the system by the seed much earlier than any individual node finishes. This ensures that these very rare and crucial blocks get replicated at least a few times.

Hence, we conjecture that if leechers stay on to serve a small number (1-2) of extra blocks in the system after finishing their downloads, all nodes can finish with high probability even when the origin server departs.

not in the absolute difference between these times.

6. SUMMARY AND CONCLUSION

In this paper, we have described a series of experiments aimed at analyzing and understanding the performance of BitTorrent in a range of scenarios. We focused our attention on two main metrics: utilization of the upload capacity of nodes, and unfairness in terms of the volume of data served by nodes.

Our findings, which we believe have not been reported in the literature to date, are summarized as follows: *(a)* BitTorrent's rate-based Tit-For-Tat (TFT) policy fails to prevent unfairness across nodes in terms of volume of content served. This unfairness arises principally in heterogenous settings when high bandwidth peers connect to low bandwidth ones. *(b)* The combination of Pairwise block-level TFT (Section 5.4.2) and the bandwidth matching tracker (Section 5.4.3) almost eliminates the unfairness of BitTorrent with a very modest decrease in utilization. *(c)* Seed bandwidth is critical to conserve when it is scarce; it is important that the seed node serve unique blocks at first (which it alone can do) to ensure diversity in the network, rather than serve duplicate blocks (a function that can be performed equally well by the leechers). *(d)* The Local Rarest First (LRF) policy is critical in eliminating the "last block" problem and ensuring that arriving leechers quickly have something to offer other nodes.

Acknowledgments

We thank Phil Chou, Kamal Jain, Pablo Rodriguez, and Aditya Ramamoorthy for participating in discussions and for their insightful suggestions. We thank Ernst Biersack for providing us the Redhat tracker log, and Sharad Agarwal for his comments an earlier draft of this paper.

7. REFERENCES

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network Information Flow. *IEEE Trans on Info Theory*, 46(4):1204–1216, Jul. 2000.

[2] A. Akella, S. Seshan, and A. Shaikh. An Empirical Evaluation of Wide-Area Internet Bottlenecks. In *IMC*, 2003.

[3] BitTorrent. `http://bittorrent.com`.

[4] Bram Cohen. Incentives Build Robustness in BitTorrent. 2003. `http://bittorrent.com/bittorrentecon.pdf`.

[5] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed Content Delivery Across Adaptive Overlay Networks. *SIGCOMM*, Aug. 2002.

[6] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data . *SIGCOMM*, Sep. 1998.

[7] E. Adar and B. Huberman. Free riding on Gnutella. Technical report, Xerox PARC, 2000.

[8] C. Gkantsidis and P. Rodriguez. Network Coding for Large Scale Content Distribution. Technical Report MSR-TR-2004-80, Microsoft Research, 2004.

[9] M. Izal, G. Urvoy-Keller, E.W. Biersack, P. Felber, A. Al Hamra, and L. Garcés-Erice. Dissecting BitTorrent: Five Months in a Torrent's Lifetime. *PAM*, Apr. 2004.

[10] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, 1991.

[11] J.A. Pouwelse, P. Garbacki, D.H.J. Epema, and H.J. Sips. A Measurement Study of the BitTorrent Peer-to-Peer File-Sharing System. Technical Report PDS-2004-003, Delft University of Technology, The Netherlands, April 2004.

[12] D. Qiu and R. Srikant. Modeling and Performance Analysis of BitTorrent-like Peer-to-Peer Networks. *SIGCOMM*, Sep. 2004.

[13] Stefan Saroiu and P. Krishna Gummadi and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, Jan 2002.

[14] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A measurement study of available bandwidth estimation tools. In *IMC*, 2003.