Understanding and Deconstructing BitTorrent Performance

Ashwin R. Bharambe

Carnegie Mellon University

Cormac Herley
Venkata N. Padmanabhan

Microsoft Research

January 2005

Technical Report
MSR-TR-2005-03

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Understanding and Deconstructing 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 very successful at distributing large files quickly and efficiently without overwhelming the capacity of the origin server, even under extreme flash crowd conditions.

In this paper, we present a simulation-based study of BitTorrent. The goal is to deconstruct the BitTorrent 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 show that BitTorrent performs near-optimally in terms of uplink utilization, download time, and fairness, except under certain extreme conditions. We present and evaluate simple techniques designed to alleviate the suboptimal behavior encountered under such workloads. On the whole, our findings point to the remarkable robustness of BitTorrent's performance despite (or perhaps due to) the simplicity of its mechanisms.

1. INTRODUCTION

We consider the problem of delivering a large file from an origin server to large audiences in a scalable manner. In such a scenario, the demand for the file is often unpredictable and can shoot up by several orders of magnitude within a short span of time (i.e., a "flash crowd"). This makes it difficult and expensive to engineer a central server-based distribution mechanism. The peer-to-peer (P2P) paradigm is a promising alternative. In this approach, peers not only download content from the origin server but also serve it to other interested peers in the system. Hence, in principle, the serving capacity of the system grows with size, making the system self-scaling.

BitTorrent [3] has emerged as a popular and scalable P2P tool for distributing large files. 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 the 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. This is in contrast to many previous P2P file sharing systems where a peer needed to download the complete file before it could serve any part of it. In addition, BitTorrent incorporates a tit-for-tat (TFT) incentive mechanism for swapping data. Nodes preferentially upload to peers from whom they are able to download in return. This mechanism is especially important since studies have shown that nodes in P2P systems tend to download content without serving anything [8]. Earlier systems either lacked incentive mechanisms or employed mechanisms that could be easily thwarted by modified clients [17].

The goal of this paper is to explore the impact of BitTorrent's mechanisms and parameters on its overall performance.[1] First, we would like to know if BitTorrent is able to keep all the uplinks in the system active and fully utilized. Full utilization translates into optimal mean download time as well as optimal load on the origin server. There are a number of reasons why uplink utilization may be less than full. One is that nodes make independent decisions for downloading file-blocks. As a results, it is possible that neighbors gather a similar set of blocks, thereby decreasing their utility to each other. How blocks get replicated is, in turn, governed by the block-selection policy employed by peers. The policy used by BitTorrent, Local Rarest First (LRF), may not necessarily be optimal. We would like to understand if other policies can work equally well and under what workloads the choice of the policy becomes crucial.

Second, it is unclear if BitTorrent's incentive mechanism successfully enforces fairness in environments where nodes have heterogeneous bandwidth constraints. Even if it does succeed at enforcing fairness, TFT constrains the connection between peers: it can result in a situation where a node decides not to serve its peer, despite having useful blocks to serve. Setting up effective "barter" exchanges can become especially hard under situations where nodes are interested in disjoint sets of blocks (for example, under post flash-crowd conditions).

Finally, the key mechanisms of BitTorrent depend on parameters such as the number of peers each node interacts with, the maximum number of concurrent uploads permitted, etc. These mechanisms interact in complex ways, and performance may be influenced by the particular choice of parameter settings. Our goal is to analyze these interactions and expose suboptimal behavior, if any, within the BitTorrent system. We do this with respect to multiple metrics, including download performance, fairness in terms of the

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

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

volume of content served by each node, and load imposed on the origin server. To this end, we have built a discrete-event simulator, which models protocol interactions of a BitTorrent network. Our key findings are: (a) BitTorrent scales very well with the number of nodes (i.e., downloaders) in the system, (b) origin server bandwidth is a precious resource and must be used carefully, especially when it is low, (c) LRF policy is important for small network sizes as well as under post flash-crowd workloads, (d) BitTorrent's TFT mechanisms can result in systematic unfairness in terms of volume of data served by nodes. These findings motivate us to consider variants of and alternatives to the core BitTorrent mechanisms to alleviate suboptimal behavior.

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

2. BACKGROUND AND RELATED WORK

In this section, we present a brief overview of the BitTorrent system and discuss related analytical and measurement-studies of BitTorrent.

2.1 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 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.

2.2 BitTorrent-specific Studies

There have been analytical as well as measurement-based studies of the BitTorrent system. At the analytical end, Qiu and Srikant [18] 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 [10]. 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 [16]. Some of the findings in this study are different from those reported in [10], 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 [9] present a simulation-based study of a BitTorrent-like system. They show results indicating that the download time of a BitTorrent-like system is far from 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. We also find that suboptimal behavior arises in BitTorrent. However, such behavior is only limited to extreme configurations which, in many cases, may not be practically interesting (e.g., low seed bandwidth). Furthermore, we show that the problems can be remedied with relatively simple fixes, including the use of source-based FEC. Such source coding avoids the computational complexity and security challenges of network coding, since the source can sign the coded packets as authentic.

2.3 Other Systems for Large File Distribution

A number of other approaches have been proposed for the efficient distribution of large files in the Internet. Some of these have been based on the multicast paradigm, where the content is transmitted down a distribution tree rooted at the source. In the Digital Fountain approach [6], both the source blocks and FEC blocks are multicast continuously, and clients join and stay tuned in until they have received a sufficient number of blocks to be able to reconstruct the file.

Although the original Digital Fountain proposal was based on IP multicast, it can in principle also work with overlay multicast [7, 12]. One problem, however, is that disruption of the distribution tree due to the failure or departure of ancestors, or due to network congestion would result in progressively lower throughput as we go deeper down the tree. This problem can be alleviated by employing multiple, diverse distribution trees and striping data across the trees, as in CoopNet [15].

There have also been proposals for mesh-based (i.e., BitTorrent-like) rather than tree-based distribution topologies. Byers et al. [5] present an approach for informed content delivery. The digital fountain encoding scheme is used to generate redundant blocks, which are then exchanged by peers that are connected in an overlay mesh topology. The main contribution of [5] is a set of algorithms for efficient estimation, summarization, and reconciliation of the blocks held by each peer. This problem is much simpler in BitTorrent since only (unencoded) source blocks are distributed.

Finally, Bullet [14] is another system for mesh-based data dissemination. It is quite similar to BitTorrent, but uses the distributed RanSub mechanism [13] rather than a centralized tracker as the peer discovery mechanism.

3. METHODOLOGY

We have 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 would be difficult or even impossible to achieve using live Internet measurement techniques (e.g., using tracker logs [10, 16] or by participating in a live-torrent). In this section, we present the details of our simulator and define the metrics we focus on in our evaluation.

3.1 Simulator Details

For our study, we built a discrete-event simulator of BitTorrent. The 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 5000 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 2.1) 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 long length of the connections at least makes 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 ($\geq$ 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.

We also make one simplification in modeling BitTorrent itself. We ignore the *endgame mode*[4], which 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 4 provides the details at the relevant places.

3.2 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.

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.

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 2.1) 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.

Fairness: The system should be fair in terms of the number of blocks served by the individual nodes. No node should be asked to upload much more than it has downloaded (unless it has volunteered to serve the system as a seed). 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 2.1, 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 or rank-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 without sacrificing performance (in terms of link utilization or download time).

4. EXPERIMENTS

4.1 Roadmap 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): 60 (i.e., leechers try to maintain between 40 and 80 neighbors). 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)

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

We start in Section 4.2 with a set of experiments that consider a homogeneous setting where all leechers have the same downlink/uplink bandwidth (1500/400 Kbps by default, as noted above). We vary several parameters individually: the number of leechers, the number of initial seeds, aggregate bandwidth of seeds, bandwidth of leechers, the number of blocks in the file, 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 4.3 we turn to a heterogeneous 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 focuses on the issue of fairness, which is important since, for instance, a high-end cable node is in a position to upload to a low-end DSL node much faster than the DSL node can upload in return. We consider smarter strategies for optimistic unchoke, which reduce the likelihood of a node optimistically unchoking a neighbor from whom it is unlikely to be able to download at a fast rate in return. We also evaluate variants of the TFT policy that explicitly try to enforce fairness.

Finally, in Section 4.4, we consider workloads other than the extreme flash crowd scenario considered thus far. We consider a more gradual arrival pattern and also a bursty arrival pattern with multiple bursts spaced apart in time. We also consider a scenario where nodes that already have a partial copy of the file join the network. Our objective in considering these alternative workloads is to determine how well BitTorrent's mechanisms (in particular, LRF) work in a setting where there is potentially a wide variation in the objectives pursued by the leechers in the system (*e.g.*, an old node might be looking for a few specific blocks to complete its download whereas a newcomer would be happy downloading any block). Finally, we consider the arrival pattern observed for an actual torrent (the one corresponding to the Linux Redhat 9 distribution [10]), which includes both a flash crowd and a post-flash-crowd phase.

4.2 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 4.1 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).

4.2.1 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.

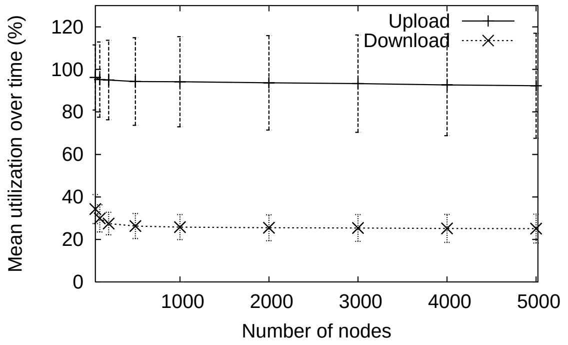

Figure 1: Mean upload and download utilization of the system as the flash-crowd size increases. Observe that the mean upload utilization is almost 100 %, even as the network size increases. The download utilization is upper bounded by the ratio of the leechers upload to download bandwidths.

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 5000.

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

4.2.2 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

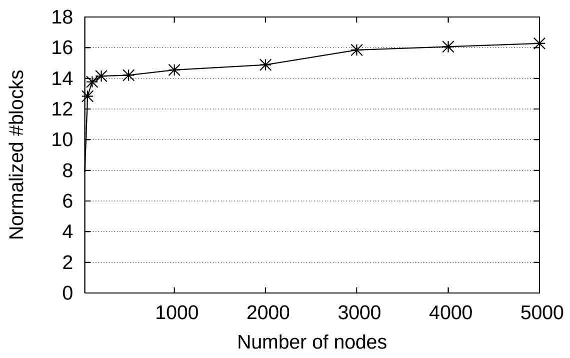

Figure 2: Contribution of the seed as the flash-crowd size increases. Observe that the amount of work done by the seed is almost independent of the network size, indicating that (at least in this scenario) the system scales very well.

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.

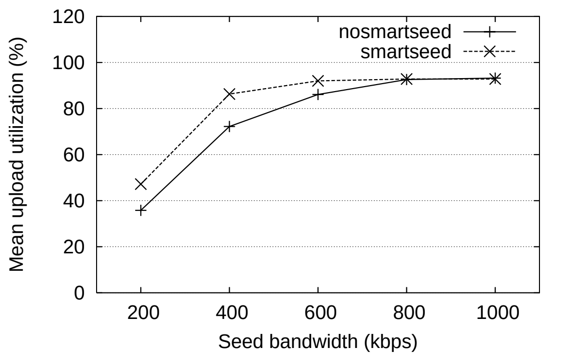

Figure 3: Upload utilization as the bandwidth of the seed is varied. When the seed has low bandwidth it is incapable of providing enough capacity to the system to keep the leechers upload pipes fully utilized. Observe also the difference between the default BitTorrent strategy "nosmartseed" and an improved "smartseed" strategy where the seed avoids serving duplicate blocks prematurely. The "smartseed" strategy improves the diversity of blocks in the system, without imposing additional burden on the seed.

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 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).

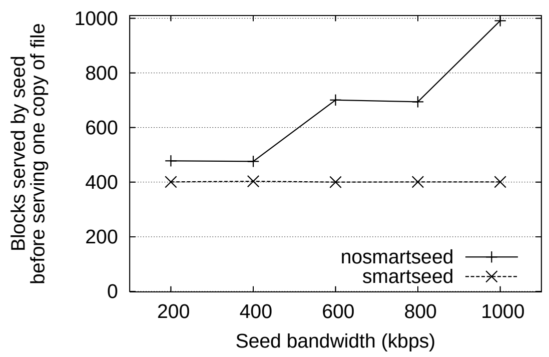

Figure 4: The total number of blocks served by the seed before it has served at least one copy of each block in the file. If we use the default BitTorrent strategy (indicated by "nosmartseed") duplicate blocks are often served before the whole file is served. For example at seed bandwidth 600 kbps approximately 700 blocks were served before each of the 400 blocks had been served at least once. The "smartseed" strategy ensures that no block is served twice before every block has been served once.

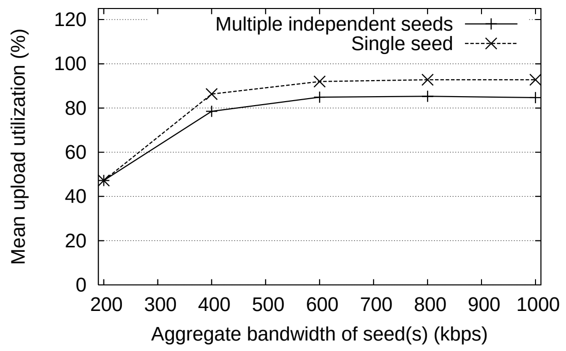

Figure 5: Upload utilization for a single seed versus multiple independent seeds. The lack of coordination among the independent seeds results in duplicate blocks being served by different seeds and a corresponding penalty in uplink utilization.

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.

4.2.3 *Impact of node degree and block choosing policy*

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. In summary, 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 constrained.

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

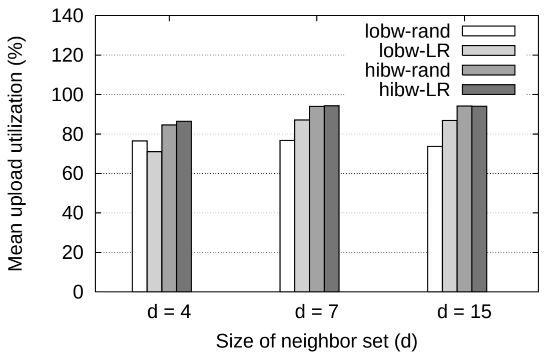

Figure 6: Upload utilization for LRF and Random policies for different values of the node degree, d. LRF performs better only when the node degree is large and the seed bandwidth is low.

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 choosing policies (the top 2 curves in Figure 7). In the low seed bandwidth case the difference is very pronounced. Observe that the LRF policy manages to find far more *interesting* connections, especially towards the end of the download. This underlines the importance of the LRF policy in the case where seed bandwidth is very limited.

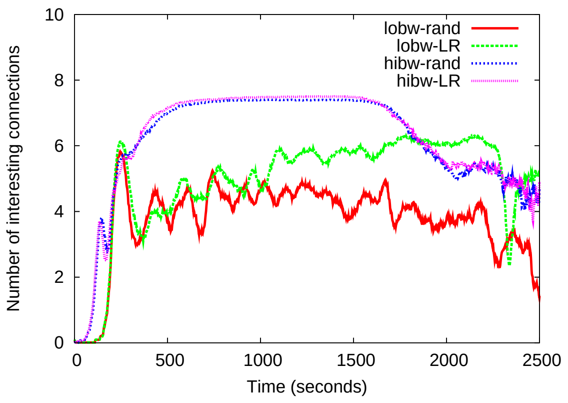

Figure 7: Variation of the number of *interesting* connections over time for $d = 7$ and various settings of seed bandwidth and block choosing policy. Each point represents the mean across all nodes present in the system at that time.

Next we plot in Figure 8 the inter-arrival times for 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, and $d = 7$ in both cases. Recall that the file size is 400 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.

Finally, we saw in Figure 6 that for very small d, utilization suffered. When leechers followed a LRF block choosing policy, but had only a set of $d = 4$ neighbors diversity was adversely affected. However, when d was increased to 7, the problem disappeared. A natural question is how large the set of neighbors needs to be, and whether this thresh-

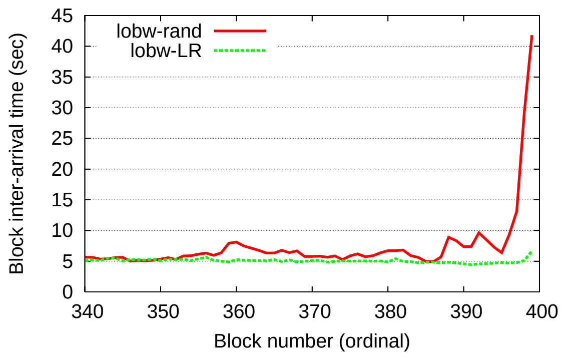

Figure 8: Inter-arrival times for blocks at the tail end of the file. Each point represents the mean time to receive the k^{th} block, where the mean is taken over all nodes. Random clearly shows the last-block problem.

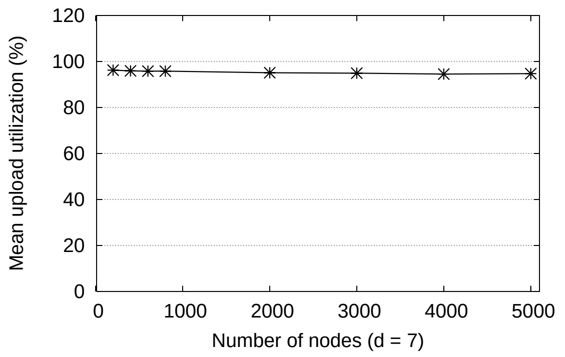

Figure 9: Utilization when $d = 7$ and number of nodes is increased. Observe that utilization drops very slowly with increasing n, indicating that even with a small neighbor set the LRF

old increases with the size of the network. In Figure 9 we graph the mean upload utilization for $d = 7$ as the number of nodes in the network n increases. We see a very slight drop in the utilization; however, this drop is only because the initial ramp up time to get all uplinks going increases as the number of nodes increases. We conclude that for the LRF policy to not degenerate, it is only necessary that d be above a certain small constant. In other words, d *does not have to scale* with n for ensuring efficient behavior (at least up to the limit of $n = 5000$ considered in our experiments).

4.2.4 Number of blocks and impact on parallelism

In this section we explore the effect of block size on utilization. For a fixed file size, Figure 10 shows the impact of number of file-blocks (N) on the overall upload utilization. If N is too small, each block becomes proportionately larger in size, thereby increasing the block-transfer quantum. This in turn implies that it takes longer for a peer that is downloading a block to turn around and start serving it, thus reducing the utilization of perpendicular bandwidth. On the other hand, if N is too large, it can result in significant overhead for communicating metadata. Figure 10 shows that for $N \geq 400$, BitTorrent is able to utilize perpendicular bandwidth effectively.

4.2.5 Concurrent Uploads

In BitTorrent, each node uploads to no more than a fixed

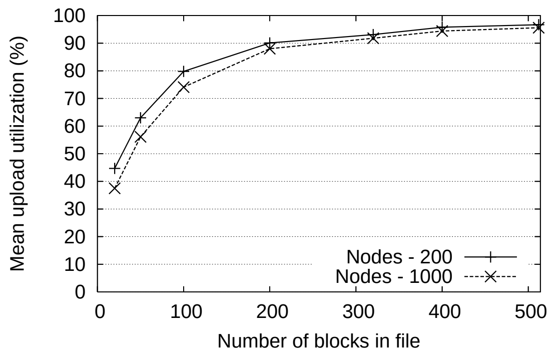

Figure 10: Variation of link utilization as the number of blocks in the file, N, varies. Observe that for small N utilization is very low.

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).

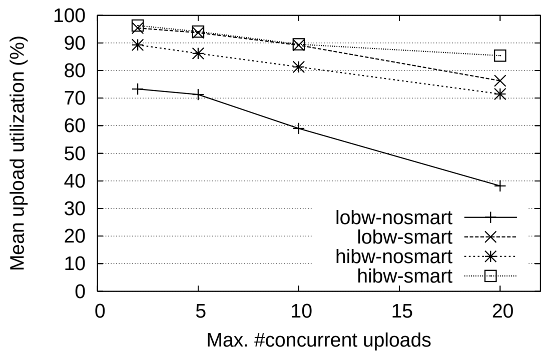

Figure 11: Utilization for different values of the maximum number of concurrent uploads (u).

Figure 11 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. 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 fill its upload link. We plan to investigate this in future work.

4.3 Heterogeneous Environment

In this section, we study the behavior of BitTorrent when node bandwidth is heterogeneous. As described in Section 3.2, a key concern in such environments is fairness in terms of the volume of data served by nodes. 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. 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 describe two simple mechanisms that can potentially reduce such unfairness: (a) Instantaneous bandwidth estimation (IBE), 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).

4.3.1 Instantaneous 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 we assume (somewhat unrealistically) that bandwidths for all d peers can be estimated instantaneously without much overhead, optimistic unchokes are not needed. A node can simply unchoke the top u peers sorted by decreasing estimated upload rates. In this case, nodes do not have to participate in data exchanges to discover the upload rate of peers.

While *instantaneous* bandwidth estimation (IBE) may not be feasible in practice, there exist many lightweight bandwidth estimation schemes [20] that incur much less overhead than a full block transfer. Also, information derived from the history of past interactions with a peer could be used to estimate its upload bandwidth. So we believe that the IBE results presented here can be approximated in practice.

4.3.2 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.

4.3.3 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 12), (b) unfairness as measured by the maximum number of blocks served by a node (Figure 13), 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.

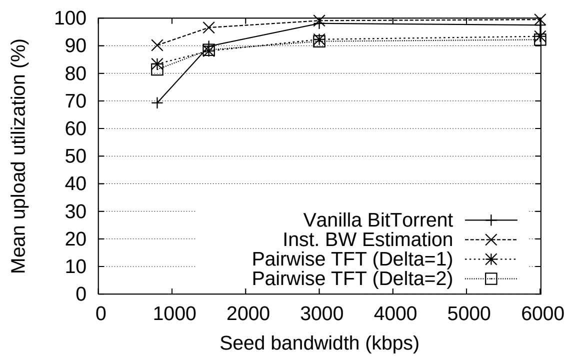

Figure 12: Mean upload utilization for *a)* vanilla BitTorrent, *b)* BitTorrent with Instantaneous Bandwidth Estimation, and *c)* with a pairwise block-level TFT policy.

Figure 12 shows the mean upload utilization of BitTorrent and other policies in a heterogeneous setting. We find that utilization is low for BitTorrent when seed bandwidth is low. This is because of two factors: first, high-bandwidth nodes (cable modems) have uplink capacities greater than the seed's uplink capacity and hence cannot fully utilize their uplink. This is similar to the effect seen in Figure 3. Second, and more importantly, high-bandwidth nodes can end up unchoking and transferring data to nodes whose download capacity is saturated. The fixed limit on the number of upload connections (u) means that uplink utilization suffers further. We see that if nodes are "matched" properly, as is done by the ideal IBE heuristic, utilization improves significantly. Also, from Figure 12, it appears that these differences diminish rapidly as the seed bandwidth increases. This is simply because a faster seed implies high-bandwidth nodes finish much faster and *leave* the system, thus reducing the impact of low utilization of *their* uplinks on the overall uplink utilization.

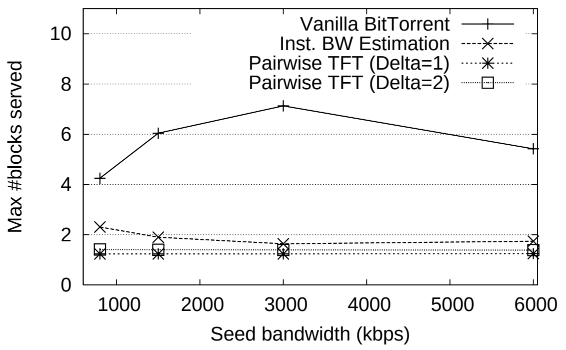

Figure 13: Maximum number of blocks (normalized by file size) served by any node during an experiment for (a) vanilla BitTorrent, (b) BitTorrent with Instantaneous Bandwidth Estimation, and (c) with a pairwise block-level TFT policy.

As described earlier in this section, the fact that high-bandwidth nodes can end up interacting with low-bandwidth nodes also manifests itself in the disparity of volume of data served by nodes. Figure 13 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 [11], 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 13 shows, some nodes can still be very unlucky, serving more than 7 times as many blocks as they receive in certain situations. It turns out (not shown in the graphs) that these unlucky nodes are all high-bandwidth cable modems. As the seed bandwidth increases, the uplink utilization of the high-bandwidth nodes also increases (due to more unique blocks being available in the system for serving). So the high-bandwidth nodes end up serving more data to lower bandwidth nodes, resulting in greater unfairness. At very high seed bandwidths, however, the download times of the high bandwidth nodes fall significantly. Hence such nodes depart the system quickly, reducing the potential for unfairness.

We find that the new mechanisms for reducing fairness are very effective. The IBE heuristic does well because it reduces the number of rate-mismatched transfers between high-bandwidth and low-bandwidth nodes. While the pairwise block-level TFT policy is expected to reduce fairness by design, an important question is how the tighter constraint it imposes impacts uplink utilization. Figure 12 shows that uplink utilization does suffer by about 6-7% for pairwise block-level TFT as compared to vanilla BitTorrent or IBE. The reason is that when a high-bandwidth node is paired up with a low-bandwidth node (which is more likely to happen towards the end of the former's download, when it is looking for a few specific blocks), the insistence of pairwise block-level TFT limits the upload rate of the high-bandwidth node to that of the low-bandwidth node, thus hurting the former's uplink utilization. Although this penalty is non-negligible, we believe it might be a reasonable price to pay for guaranteeing bounded unfairness; a contributing peer now can be assured that it will not be taken advantage of to an arbitrary degree.

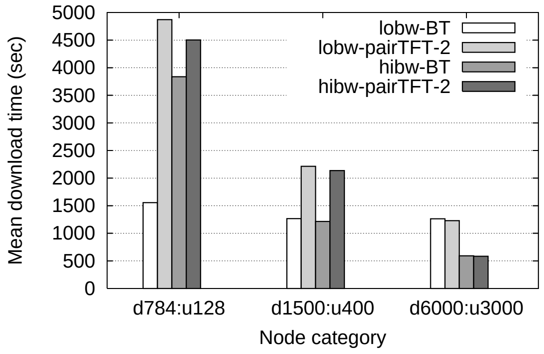

Figure 14: Download times for nodes of different categories for various schemes.

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 BitTorrent and block-level TFT for two settings: low seed bandwidth (800 kbps) and high seed bandwidth (6000 kbps). We see that, with vanilla BitTorrent and low seed bandwidth, even nodes with low uplink bandwidth can finish fast since they can get connected to high-bandwidth nodes. The block-level TFT policy achieves its goal of restoring this to a situation where the download times are roughly proportional to the node uplink capacities. Block-level TFT helps decrease the download time of high-bandwidth nodes by a small amount; however, this reduces the number of blocks served by these nodes by a significant amount, as shown in Figure 13. When the seed bandwidth is high, differences between BitTorrent and block-level TFT are less evident. This is because in both cases the high-bandwidth nodes finish downloading and depart the system quickly, denying the lower bandwidth nodes the opportunity to download much from the high-bandwidth nodes.

In summary, we find that 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, simple alternative policies — IBE and pairwise block-level TFT — can remedy the situation.

4.4 Other Arrival Patterns

Thus far we have focused on the performance of BitTorrent in flash-crowd scenarios. While performance in a flash-crowd setting is an important concern, most realistic workloads also involve a non-trivial post flash-crowd phase. It is important for participants to get good download performance in the latter settings as well. The goal of this section is to evaluate BitTorrent's performance from the perspective of post flash-crowd nodes.

4.4.1 Variation in Download Goals

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. We present results from experiments designed to investigate this issue.

Performance of Nodes in the Post-Flash Crowd Phase

In the first experiment, 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.

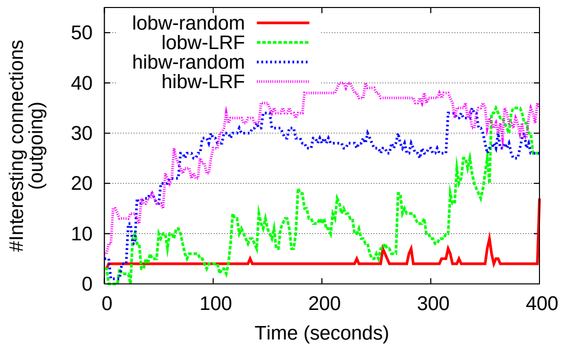

Figure 15: Number of *interesting* outgoing connections of a randomly sampled post flash-crowd node for various configurations.

Figure 15 plots the number of *interesting* outgoing connections over time for a randomly chosen post flash-crowd 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 irrespective of the block-selection policy (LRF or random) or the seed bandwidth, 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. This is not surprising: 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, we conclude that while a large degree, d, may not be necessary for a flash-crowd situation (Figure 9), making the degree very small can significantly impact TFT performance for nodes in the post flash-crowd phase.

Although both the LRF and random policies perform well, LRF performs better in general and results in a larger count of interesting connections. One exception, however, is when the seed bandwidth is low. As shown in Figure 15, the number of interesting connections starts off higher with the random policy ("lobw-random") compared to the LRF policy ("lobw-LRF"). The reason for this inversion is that the low-bandwidth seed is still injecting new blocks into the system. Since such new blocks are by definition rare and there are not many other rare blocks in the system (since the existing nodes have already downloaded 80% of the blocks), the LRF policy will impel all nodes — existing nodes as well as new nodes — to preferentially download such a new block at the same time. However, this common and concurrent interest in the same rare block diminishes the ability of a new node to find peers that are interested in subsequently downloading the same block from it. The new node would actually be better off had it picked a block to download at random, as seen from the larger number of interesting connections for "lobw-random" at the very beginning.

Thus it is important for a new node to switch to LRF only after downloading the first few blocks. It is interesting to note that BitTorrent currently employs this heuristic, but for a different reason [4], viz., to enable a new node to *download* its first block quickly.

Performance of Pre-seeded Nodes

The second experiment we perform is a slight variation of the first. 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. Furthermore, 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. We would like to see a node with k% blocks taking approximately $(1-\frac{k}{100})T$ time to download the remaining blocks, where T is the mean download time for leechers. ($T = 2000$ seconds, for this setting.) The key question, however, is whether the pre-seeded node takes longer than it should to complete its download, because of a mismatch between its goal (which is to download the (few) specific blocks that it does not have) and that of the other nodes in the system (which typically have fewer blocks and hence have more choices for blocks to download).

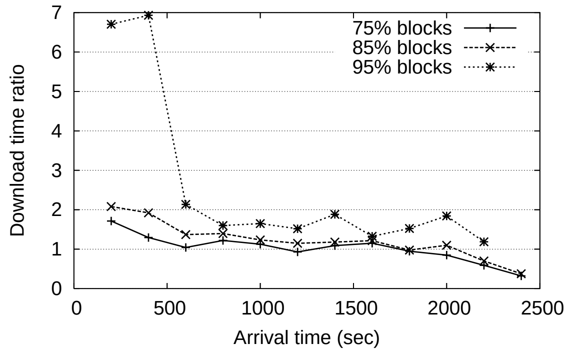

Figure 16: Download time ratios for pre-seeded nodes introduced into the system at various times in the experiment. This experiment used vanilla BitTorrent.

Figure 16 plots the ratio of actual download time to the expected download time for such "pre-seeded" nodes, 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 Figure 16, we find 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: 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 4.1). 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, and confirmed this prediction. The download time ratios for all pre-seeded nodes and for all values of k decreased to below 2.0.

4.4.2 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.

4.4.3 Workload Derived from a Real Torrent

Our goal in this section is to verify that the results presented in previous sections hold under realistic workloads. We consider two important workload parameters: (a) node arrival pattern, and (b) uplink and downlink bandwidth distribution. To derive realistic arrival patterns, we use the tracker log for the Redhat 9 distribution torrent [10]. Table 1 describes the distribution of peer bandwidth, which was derived from the Gnutella study reported in [19]. While discretizing the CDFs presented in [19], we excluded the tail of the distribution. This means that (a) dial-up modems are eliminated, since it is unlikely that they will participate in such large downloads, and (b) very high bandwidth nodes are eliminated, making the setting more bandwidth constrained. We set the seed bandwidth to 6000 kbps.

[3]Note that this increase is in the *ratio* of the actual to ideal download times, not in the absolute difference between these times.

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

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.

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.

Table 2 shows the performance of BitTorrent with the workload described above. We conclude that, even under a post-flash-crowd arrival pattern that is spread out over time, the results observed in earlier sections hold true, viz., uplink utilization is very high and there is considerable unfairness in the amount of data delivered by nodes.

5. 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.

We find that BitTorrent performance scales very well with the system size: even a seed with a modest capacity can handle large flash crowds while keeping uplink bandwidth utilization high. We show that while block-selection policies are an important factor in achieving high uplink utilization, the choice is not critical in many common settings (e.g., a well-provisioned seed). At the same time, the Local Rarest First policy employed by BitTorrent plays a key role in ensuring diversity, especially when the seed bandwidth is low and under post flash-crowd conditions. In general, our conclusion is that seed bandwidth must be utilized carefully when it is scarce.

Finally, we show that BitTorrent's rate-based tit-for-tat policy, with the accompanying optimistic unchoke mechanism, can result in systematic unfairness, with some nodes serving much more than they download. We explore possible solutions and show that peer-to-peer block-level tit-for-tat can bound such unfairness without hurting utilization significantly.

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.

6. 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] Y. Chu, S. Rao, and H. Zhang. A Case for End System Multicast. *SIGMETRICS*, Jun. 2000.

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

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

[10] 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.

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

[12] J. Jannotti, D. Gifford, K. Johnson, M.F. Kaashoek, and J. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. *OSDI*, Oct. 2000.

[13] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using Random Subsets to Build Scalable Network Services. *USITS*, Mar. 2003.

[14] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. *Proc. ACM Symposium on Operating Systems Principles*, Oct. 2003.

[15] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. *NOSSDAV*, May 2002.

[16] 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.

[17] Kazaa Lite. `http://www.k-lite.tk`.

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

[19] 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.

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