Do incentives build robustness in BitTorrent?

Michael Piatek* Tomas Isdal* Thomas Anderson* Arvind Krishnamurthy* Arun Venkataramani†
Technical Report–TR 2006-11-02

Abstract

A fundamental problem with many peer-to-peer systems is the tendency for users to "free ride"—to consume resources without contributing to the system. The popular file distribution tool BitTorrent was explicitly designed to address this problem, using a tit-for-tat reciprocity strategy to provide positive incentives for nodes to contribute resources to the swarm. We show that although BitTorrent has been fantastically successful, its incentive mechanism is not robust to selfish clients. Through performance modeling parameterized by real-world traces, we demonstrate that high capacity peers provide low capacity peers with an unfair share of aggregate swarm resources. We use these results to drive the design and implementation of *BitTyrant*, a selfish BitTorrent client that provides a median 70% performance gain for a 1 Mbit client on real Internet swarms. We further show that this selfishness, when applied universally, can hurt average per-swarm performance compared to today's BitTorrent client implementations.

1 Introduction

A fundamental problem with many peer-to-peer systems is the tendency of users to "free ride"—consume resources without contributing to the system. In early peer-to-peer systems such as Napster, the novelty factor sufficed to draw plentiful participation from peers. Subsequent peer-to-peer systems recognized and attempted to address the free riding problem; however, their fixes proved to be unsatisfactory, e.g., "incentive priorities" in Kazaa could be spoofed; currency in MojoNation was cumbersome; and the AudioGalaxy Satellite model of "always-on" clients has not been taken up. More recently, BitTorrent, a popular file distribution tool based on a *swarming* protocol, showed that a tit-for-tat strategy was effective at incenting peers to contribute resources to the system and discouraging free riders.

The tremendous success of BitTorrent suggests that tit-for-tat is successful at inducing contribution from rational peers. Moreover, the bilateral nature of tit-for-tat allows for enforcement without a centralized trusted infrastructure. The consensus appears to be that "incentives build robustness in BitTorrent" [3, 16, 2, 11].

In this paper, we question this widely held belief. To this end, we first conduct a large measurement study of real BitTorrent swarms to understand the diversity of BitTorrent clients in use today, realistic distributions of peer upload capacities, and possible avenues of strategic peer behavior in popular clients. Based on these measurements, we develop a simple model of BitTorrent to correlate upload and download rates of peers. We parametrize this model with a measured distribution of peer upload capacities and discover the presence of significant *altruism* in BitTorrent, i.e., a minority of high capacity peers provide low capacity peers with an unfair share of aggregate swarm resources. Intrigued by this observation, we revisit the following question: *can a selfish peer game BitTorrent to significantly improve its download performance for the same level of upload contribution?*

Our primary contribution is to settle this question in the affirmative. Based on the insights gained from our model, we design and implement *BitTyrant*, a modified BitTorrent client designed to benefit selfish peers. We evaluate *BitTyrant* performance on real swarms, establishing that high and even moderate capacity peers can significantly improve download performance while reducing upload contributions. For example, a client with 1 Mb/s upload capacity receives a median 70% performance gain from using *BitTyrant*. Moreover, the strategic behavior of *BitTyrant* is executed simply through policy modifications to existing clients without any change to the BitTorrent protocol. The key idea is to carefully select peers and contribution rates so as to maximize download per unit of upload bandwidth.

How does use of *BitTyrant* by many peers in a swarm affect the performance of the swarm? We find that all peers, regardless of upload capacity, benefit from adoption, irrespective of whether or not other peers are using *BitTyrant*. Peers not using *BitTyrant* can experience degraded performance due to the absence of altruisitic contributions from selfish peers. Taken together, these results suggest that in fact "incentives do not build robustness in BitTorrent", at least as currently implemented. In the presence of selfish users that constrain upload contribution or exploit altruism, performance degrades.

In addition to our primary contribution, *BitTyrant*, our efforts to measure and model altruism in BitTorrent are independently noteworthy. First, although modeling BitTorrent has seen a large body of work (see Section 6), our model is both simpler and suffices to capture the correlation between upload and download rates for real swarms. Second, existing studies recognizing altruism in

*Dept. of Computer Science and Engineering, Univ. of Washington
†Dept. of Computer Science, Univ. of Massachusetts Amherst

BitTorrent consider small simulated settings or few torrents that poorly capture the diversity of deployed BitTorrent clients, peer capacities, churn, and network conditions. Our evaluation is more comprehensive. We use trace driven modeling to drive the design of *BitTyrant*, which we evaluate on more than 100 popular, real world swarms as well as synthetic swarms on PlanetLab. Finally, we make *BitTyrant* available publicly as well as anonymized traces gathered in our large-scale measurement study, which covers hundreds of torrents and hundreds of thousands of IP addresses.

The remainder of this paper is organized as follows. Section 2 provides an overview of the BitTorrent protocol and our measurement data, which we use to parametrize our model. Section 3 develops a simple model illustrating the sources and extent of altruism in BitTorrent. Section 4 presents *BitTyrant*, a modified BitTorrent client for selfish peers, which we evaluate in Section 5. In Section 6, we discuss related work and conclude in Section 7.

2 BitTorrent overview

This section presents an overview of the BitTorrent protocol, its implementation parameters, and the measurement data we use to seed our model.

2.1 Protocol

BitTorrent focuses on bulk data transfer, separating the problems of data lookup and retrieval. All users in a particular swarm are interested in obtaining the same file or set of files. In order to initially connect to a swarm, peers download a metadata file, called a *torrent*, from a content provider, usually via a normal HTTP request. This metadata specifies the name and size of the file to be downloaded, as well as SHA-1 fingerprints of the data blocks (typically 64–512 KB) that comprise the larger file to be downloaded. These fingerprints are used to verify data integrity. The metadata file also specifies the address of an HTTP *tracker* server for the torrent, which coordinates interaction between peers participating in the swarm. Peers contact the tracker upon startup and departure as well as periodically as the download progresses, usually with a frequency of 15 minutes. The tracker maintains a list of currently active peers and delivers a random subset of these to clients, upon request.

A user in possession of the complete file, called a *seed*, redistributes small blocks to other participants in the swarm. Peers exchange blocks and control information with a set of directly connected peers we call the *local neighborhood*. This set of peers, obtained from the tracker, is unstructured and random, requiring no special join or recovery operations when new peers arrive or existing peers depart. The control traffic required for data exchange is minimal: each peer transmits messages indicating the data blocks they currently possess and messages signaling their interest in the blocks of other peers.

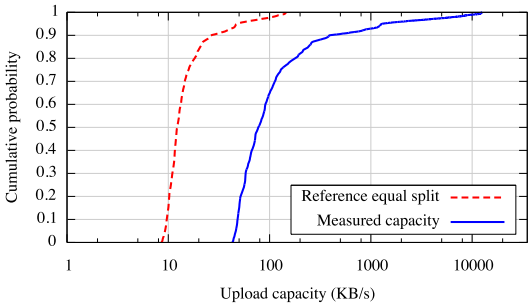


Figure 1: Cumulative distribution of raw bandwidth capacity for BitTorrent peers as well as the "equal split" capacity distribution for active set peers, assuming clients use the reference implementation of BitTorrent.

We refer to the set of peers to which a BitTorrent client is currently sending data as its *active set*. BitTorrent uses a rate-based tit-for-tat strategy to determine which peers to include in the active set. Each round, a peer sends data to *unchoked* peers from which it received data most rapidly in the past 10 second interval. This strategy is intended to provide positive incentives for contributing to the system and inhibit free-riding. However, peers will also send data to a small number of randomly chosen peers who have not "earned" such status. Such peers are said to be *optimistically unchoked*. Optimistic unchokes serve to bootstrap new peers into the tit-for-tat game as well as to facilitate discovery of new, potentially better sources of data. Peers that do not send data quickly enough to earn reciprocation are removed from the active set during a tit-for-tat round and are said to be *choked*.

Modulo TCP effects and assuming last-hop bottleneck links, each peer provides an equal share of its available upload capacity to peers to which it is actively sending data. We refer to this rate throughout the paper as a peer's *equal split rate*. This rate is determined by the upload capacity of a particular peer and the size of its active set. In the official reference implementation of BitTorrent, active set size is proportional to $\sqrt{\text{upload capacity}}$ (details in Appendix); although in other popular BitTorrent clients, this size is static.

2.2 Measurement

BitTorrent's behavior depends on a large number of parameters: topology, bandwidth, block size, churn, data availability, number of directly connected peers, active tit-for-tat transfers, and number of optimistic unchokes. Furthermore, many of these parameters are a matter of peer policy unspecified by the BitTorrent protocol itself. These policies may vary among different client implementations and defaults may be overridden by explicit user configuration. To gain an understanding of BitTor-

Implementation	Percentage share
Azureus	47%
BitComet	20%
μtorrent	15%
BitLord	6%
Unknown	4%
Reference	2%
Remaining	10%

Table 1: BitTorrent implementation usage as drawn from measurement data.

rent's behavior and diversity of implementation in the wild, we first conducted a measurement study of live BitTorrent swarms to ascertain client characteristics.

By making use of the opportunistic measurement techniques presented by Madhyastha et al. [13], we gather empirical measurements of BitTorrent swarms and hosts. Our measurement client connected to a large number of swarms and waited for an optimistic unchoke from each unique peer. We then estimated the upload capacity of that client using the multiQ tool [10]. Previous characterizations of end-host capacities of peer-to-peer participants were conducted by Saroiu, et al. [17]. We update these results using more recent capacity estimation tools. We observed 301595 unique BitTorrent IP addresses over a 48 hour period during April, 2006 from 3591 distinct ASes across 160 countries. The upload capacity distribution for typical BitTorrent peers is given in Figure 1 along with the per-peer upload rate distribution that would arise from all peers using the reference BitTorrent implementation with no limit on upload rates.

3 Modeling altruism in BitTorrent

BitTorrent's tit-for-tat reciprocity strategy is designed to correlate download performance with upload contribution. A peer with high upload contribution should see that contribution reflected in high download rates when compared with peers contributing less. Although tit-for-tat achieves this correlation, in practice the balance of upload contribution and download rate is often unfair, resulting in altruistic contributions from high capacity peers. We define altruism per peer as its upload contribution in excess of total downloaded data.

We set out to understand fundamental sources of altruism in BitTorrent. However, the diversity of implementations and the myriad of parameters makes understanding fundamental properties challenging. In this section, we take a restricted view and develop a model of altruism in BitTorrent arising from our observed upload capacity distribution and the default parameter settings of the reference implementation of BitTorrent.

We make several assumptions to simplify our analysis and provide a conservative bound on altruism. Because our assumptions are not realistic for all swarms, our modeling results are not intended to be predictive. Rather, our modeling results suggest potential sources of altruism and the reasons they emerge in BitTorrent swarms today. We exploit these sources of altruism in the design of our real world selfish client, *BitTyrant*, discussed in Section 4.

- *Representative distribution*: The CDF shown in Figure 1 is for bandwidth capacity of observed IP addresses over many swarms. The distribution of a typical swarm may not be identical. For instance, high capacity peers tend to finish more quickly than low capacity peers, but they may also join more swarms simultaneously. If they join only a single swarm and leave shortly after completion, over the lifetime of a torrent the relative proportion of low capacity peers increases.
- *Uniform sizing*: Peers, other than the modified client, use the active set sizing recommended by the reference BitTorrent implementation. In practice, other BitTorrent implementations are more popular (see Table 1); These BitTorrent clients have different active set sizes. As we will show, aggressive active set sizes tend to decrease altruism, and the reference implementation uses the most aggressive strategy among the popular implementations we inspected. As a result, our model provides a conservative estimate of altruism.
- *No steady state*: Active sets are comprised of peers with random draws from the overall upload capacity distribution. If churn is low, over time tit-for-tat may match peers with similar equal split rates, requiring higher send rates to receive reciprocation from high capacity peers. We argue in the next section that BitTorrent is slow to reach steady-state, particularly for high capacity peers.
- *High block availability*: Swarm performance is limited by upload capacity, i.e., peers will always be able to find interesting data with other peers. We find that although the reference BitTorrent implementation is designed to ensure high availability of interesting blocks, in practice, poor default settings in some clients may degrade block availability for high capacity peers.

These assumptions allows us to model altruism in BitTorrent in terms of the upload capacity distribution only. The model is built on expressions for the probability of tit-for-tat reciprocation, expected download rate, and expected upload rate. In this section, we focus on the main insights provided by our model. The precise expressions are listed in detail in the Appendix.

3.1 Tit-for-tat matching time

Since our subsequent modeling results assume that swarms do not reach steady state, we first examine the

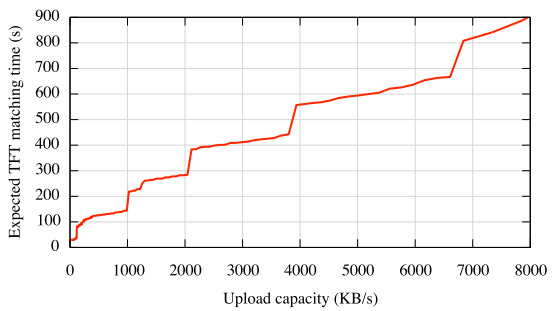


Figure 2: Assuming a peer set of infinite size, the expected time required for a new peer to discover enough peers of equal or greater equal split capacity to fill its active set.

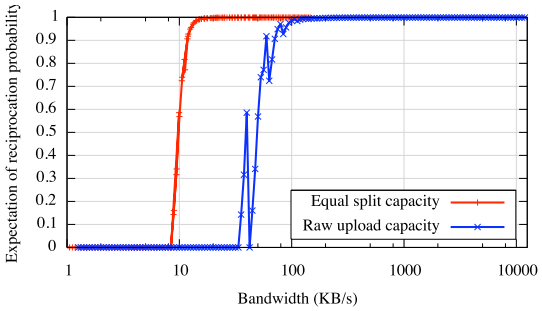


Figure 3: Reciprocation probability for a peer as a function of raw upload capacity as well as reference BitTorrent equal split bandwidth. Reciprocation probability is not strictly increasing in raw rate due to the sawtooth increase in active set size (see Table 2 in Appendix).

convergence properties of the tit-for-tat strategy BitTorrent uses to match peers of similar capacity. By default, the reference BitTorrent client optimistically unchokes two peers every 30 seconds in an attempt to explore the available set of peers in the local neighborhood for better reciprocation pairings. Since all peers are performing this exploration simultaneously, every 30 seconds a peer can expect to explore two candidate peers and be explored by two candidate peers. Since we know the equal split capacity distribution, we can express the probability of finding a peer with equal or greater equal split capacity—in a given number of 30 second rounds. Taking the expectation and multiplying it by the size of the active set gives an estimate of how long a new peer will have to wait before filling its active set with such peers.

Figure 2 shows this expected time for our observed bandwidth distribution. These results suggest that TFT as implemented does not quickly find good matches for high capacity peers, even in the absence of churn. For example, a peer with 50 Mb/s upload capacity would transfer more than 4 GB of data before reaching steady state. In practice, convergence time is likely to be even longer. We consider a peer as being "content" with a matching once its equal split is matched or exceeded by a peer in its active set. However, one of the two peers in any matching that is not exact will be searching for alternates and switching when they are discovered. The long convergence time suggests a potential source of altruism: high capacity clients are forced to peer with those of low capacity while searching via optimistic unchokes.

3.2 Probability of reciprocation

A node Q sends data only to those peers in its active transfer set, reevaluated every 10 seconds. If a peer P sends data to Q at a rate fast enough to merit inclusion in Q's active transfer set, P will receive data during the next tit-for-tat round, and we say Q reciprocates with P.

Reciprocation from Q to P is determined by two factors: the rate at which P sends data to Q and the rates at which *other* peers send data to Q. If all other peers in Q's current active set send at rates greater than P, Q will not reciprocate with P.

Figure 3 gives the probability of reciprocation in terms of both raw upload capacity and, more significantly, equal split rate. The sudden increase in reciprocation probability suggests a potential source of altruism in BitTorrent: equal-split bandwidth allocation among peers in the active set. Beyond a certain equal split rate ($\sim$ 14 KB/s in Figure 3), reciprocation is essentially assured, suggesting that further contribution may be altruistic.

3.3 Expected download rate

Each TFT round, a peer P receives data from both TFT reciprocation and optimistic unchokes. Reciprocation is possible only from those peers in P's active set and depends on P's upload rate, while optimistic unchokes may be received from any peer in P's directly connected peer set, regardless of upload rate. In the reference BitTorrent client, the number of optimistic unchoke slots defaults to 2 and is rotated in a round robin fashion among peers in the local neighborhood. As each peer unchokes two peers per round, the expected number of optimistic unchokes P will receive is also two for a fixed local neighborhood size.

Figure 4 gives the expected download throughput for peers as a function of upload rate for our observed bandwidth distribution. The sub-linear growth suggests significant unfairness in BitTorrent, particularly for high capacity peers. This unfairness improves performance for the majority of low capacity peers, suggesting that high capacity peers may be able to better allocate their upload capacity to improve their own performance.

3.4 Expected upload rate

Having considered download performance, we turn next to upload contribution. Two factors can control the upload rate of a peer: data availability and capacity limit.

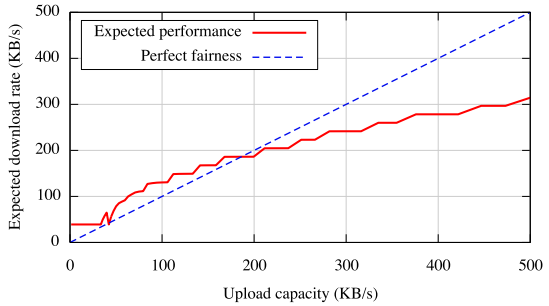


Figure 4: Expectation of download performance as a function of upload capacity. Although this represents a small portion of the spectrum of observed bandwidth capacities, $\sim 80\%$ of samples are of capacity $\leq$ 200KB/s.

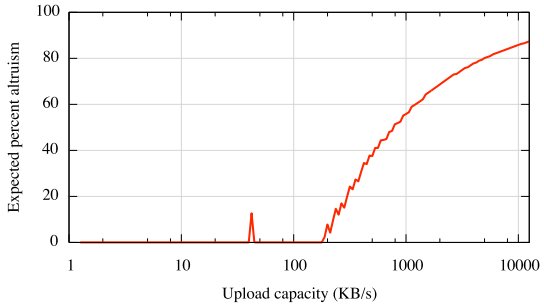


Figure 5: Expected percentage of upload capacity which is altruistic as defined by Equation 5 as a function of rate. The sawtooth increase is due to the sawtooth growth of active set sizing and equal split rates arising from integer rounding (see Table 2).

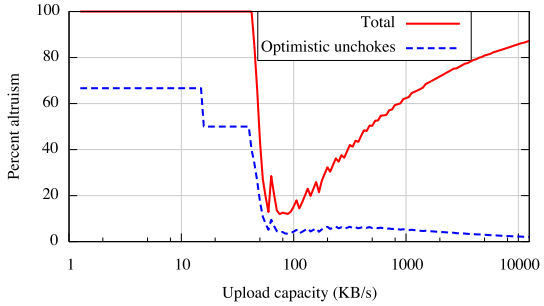


Figure 6: Expected percentage of upload capacity which is altruistic when defined as upload capacity not resulting in direct reciprocation.

When a peer is constrained by data availability, it does not have enough data of interest to its local neighborhood to saturate its capacity. In this case, this capacity is wasted and utilization suffers. Because of the dependence of upload rate on data availability, it is crucial for good utilization that BitTorrent clients download at a rate fast enough to saturate their upload capacity. We have found that indeed this is the case in the reference BitTorrent client because of the square root growth rate of its active set size.

In practice, most popular clients do not follow this dynamic strategy and instead make active set size a configurable, but static, parameter. For instance, the most popular BitTorrent client in our traces, Azureus, suggests a default active set size of four—appropriate for many cable and DSL hosts, but far lower than is required for high capacity peers. We explore the impact of active set sizing further in Section 4.1.

3.5 Modeling altruism

Given upload and download throughput, we have all the tools required to compute altruism. We consider two definitions of altruism intended to reflect two perspectives on what constitutes selfish behavior. We first consider altruism to be simply the difference between expected upload rate and download rate. Figure 5 shows altruism as a percentage of upload capacity under this definition and reflects the asymmetry of upload contribution and download rate discussed in Section 3.3. The second definition is *any* upload contribution that can be withdrawn without loss in download performance. This is shown in Figure 6.

In contrast to the original definition shown in Figure 5, this data suggests that *all* peers make altruistic contributions that could be eliminated. Sufficiently low bandwidth peers almost never earn reciprocation, while high capacity peers send much faster than the minimal rate required for reciprocation. Both of these effects can be exploited. Note that low bandwidth peers, despite not being reciprocated, still receive data in aggregate faster than they send data. This is because they receive indiscriminate optimistic unchokes from other users in spite of their low upload capacity.

3.6 Validation

Our modeling results suggest that at least part of the altruism in BitTorrent arises from the sublinear growth of download throughput as a function of upload rate. We validate this key result using our measurement data. Each time a BitTorrent client receives a complete data block from another peer, it broadcasts a 'have' message indicating that it can redistribute that block to other peers. By averaging the rate of have messages over the duration our measurement client observes a peer, we can infer the peer's download rate. Figure 7 shows this inferred download rate as a function of equal split rate, i.e. the throughput seen by the measurement client when optimistically unchoked. This data is drawn from our measurements and includes 63482 peers.

Comparing these results in Figure 4 suggests an even higher level of altruism than that predicted by our model.

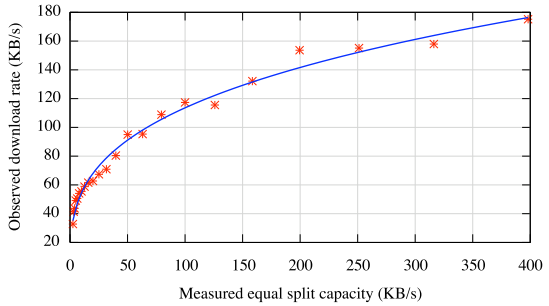


Figure 7: Measured validation of sublinear growth in download throughput as a function of rate. Each point represents an average taken over all peers with measured equal split capacity in the intervals between points. A best-fit curve for the data is shown.

We believe this is due to more conservative active set sizes in practice than those assumed in our model. Equal split rate, the parameter of Figure 7, is a conservative lower bound on total upload capacity, shown in Figure 4.

4 Building *BitTyrant*: A selfish client

The modeling results of Section 3 suggest that altruism in BitTorrent serves as a kind of progressive tax. As contribution increases, performance improves, but not in direct proportion. In this section, we describe the design and implementation of *BitTyrant*, a modified BitTorrent client optimized for selfish users. We chose to base *BitTyrant* on the Azureus client in an attempt to foster adoption, as Azureus is the most popular client in our traces.

A natural first approach to gaming BitTorrent's incentive mechanism is the well-known Sybil attack [4]. If performance for low capacity peers is disproportionately high, a selfish user can simply masquerade as many low capacity clients to improve performance. Also, by flooding the local neighborhood of high capacity peers, low capacity peers can artificially inflate their chances of tit-for-tat reciprocation by dominating the active transfer set of a high capacity peer. In practice, these types of attacks are mitigated by a common client option to refuse multiple connections from a single IP address. Resourceful peers might be able to coordinate a Sybil attack from multiple IP addresses, but such an attack is beyond the capabilities of most users. We focus instead on practical strategies that can be employed by all users of even limited technical sophistication and resources.

The unfairness of BitTorrent has been noted in previous studies [2, 5, 7], many of which include protocol redesigns intended to promote fairness. However, a clean-slate redesign of the BitTorrent protocol ignores a different but important incentives question: how to get users to adopt it? As shown in Section 3, the majority of BitTorrent users benefit from its unfairness today. Designs intended to promote fairness globally at the expense of the majority of users seem unlikely to be adopted. Rather than focus on a redesign at the protocol level, we focus on BitTorrent's robustness to strategic behavior—can a protocol-compatible client significantly reduce its own contribution but exploit the altruism of other peers?

4.1 Maximizing reciprocation

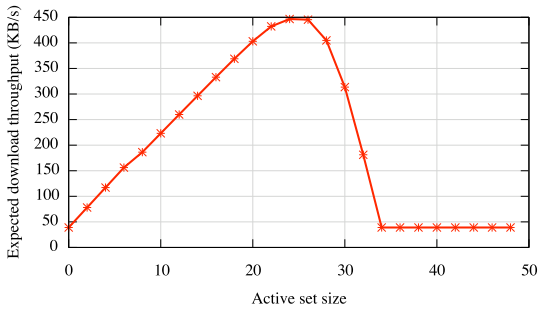
The modeling results of Section 3 and the operational behavior of BitTorrent clients suggest the following three strategies to improve performance.

- *Maximize reciprocation bandwidth per connection*: The reciprocation bandwidth of a peer is dependent on its upload capacity and its active set size. By discovering which peers have large equal split capacities, a client can optimize for a higher reciprocation bandwidth per connection.
- *Maximize number of reciprocating peers*: A client can expand its active set to maximize the number of peers that reciprocate until the marginal benefit of an additional peer is outweighed by the cost of reduced reciprocation probability from all peers.
- *Deviate from equal split*: On a per-connection basis, a client can lower its upload contribution to a particular peer as long as that peer continues to reciprocate. The bandwidth savings could then be reallocated to new connections, resulting in an increase in the overall reciprocation throughput.

The modeling results indicate that these strategies are likely to be effective. The largest source of altruism in our model is unnecessary contribution to peers in a node's active set. The reciprocation probability shown in Figure 3 indicates that strategically choosing equal split bandwidth can reduce contribution significantly for high capacity peers with only a marginal reduction in reciprocation probability. A peer with equal split capacity of 100 KB/s, for instance, could reduce its rate to 15 KB/s with a reduction in expected probability of reciprocation of only 1%. However, reducing from 15 KB/s to 10 KB/s would result in a decrease of roughly 40%.

The reciprocation behavior points to a performance trade-off. If the active set size is large, equal split capacity is reduced, reducing reciprocation probability. However, an additional active set connection is an additional opportunity for reciprocation. To maximize performance, a peer should increase its active set size until an additional connection would cause a reduction in reciprocation across all connections sufficient to reduce overall download performance.

If the equal split capacity distribution of the swarm is known, we can derive the active set size that maximizes the expected download rate. For our observed bandwidth distribution, Figure 8 shows the download rate as a func-



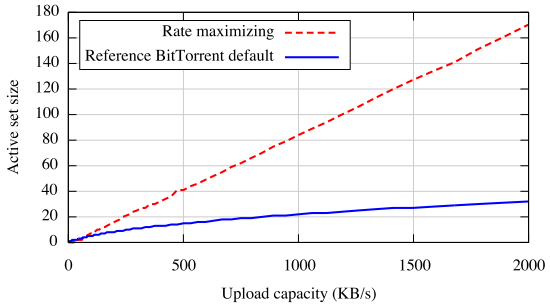


Figure 8: *Left:* The expected download performance of a client with 300 KB/s upload capacity for increasing active set size. *Right:* The download performance-maximizing active set size for peers of varying rate. The selfish maximum is linear in upload capacity, while the reference implementation of BitTorrent suggests active size $\sim \sqrt{\text{rate}}$. Although several hundred peers may be required to maximize throughput, most trackers return fewer than 100 peers per request.

tion of the active set size for a peer with 300KB/s upload capacity as well as the active set size that maximizes it. The graph also implicitly reflects the sensitivity of reciprocation probability to equal split rate.

Figure 8 is for a single selfish peer and suggests that selfish high capacity peers can benefit much more by manipulating their active set size. Our example peer with upload capacity 300 KB/s realizes a maximum download throughput of roughly 450 KB/s. However, increasing reciprocation probability via active set sizing is extremely sensitive—throughput falls off quickly after the maximum is reached. Further, it is unclear if active set sizing alone would be sufficient to maximize reciprocation in an environment with several selfish clients.

These challenges suggest that *any* a priori active set sizing function may not suffice to maximize download rate for selfish clients. Instead, they motivate the dynamic algorithm used in *BitTyrant* that adaptively modifies the size and membership of the active set and the upload bandwidth allocated to each peer (see Figure 9).

In both BitTorrent and *BitTyrant*, the set of peers that will receive data during the next TFT round is decided by the unchoke algorithm once every 10 seconds. *BitTyrant* differs from BitTorrent as it dynamically sizes its active set and varies sending rate per connection. For each peer p, *BitTyrant* maintains a measure of the estimated upload capacity required for reciprocation, u_p as well as the download throughput d_p received when p reciprocates. Peers are ranked by the ratio d_p/u_p and unchoked in order until the sum of u_p terms for unchoked peers exceeds the upload capacity of the *BitTyrant* peer.

The rationale underlying this unchoke algorithm is that the best peers are those that reciprocate most for the least number of bytes contributed to them, given accurate information regarding u_p and d_p. Implicit in the strategy are the following assumptions and characteristics:

- The strategy attempts to maximize the download rate for a given upload budget. The ranking strategy corresponds to the value-density heuristic for the knapsack problem. In practice, the download benefit (d_p) and upload cost (u_p) are not known a priori. The update operation dynamically estimates these rates and, in conjunction with the ranking strategy, optimizes download rate over time.
- A *BitTyrant* client is designed to tap into the latent altruism in most swarms by unchoking the most altruistic peers. However, it will continue to unchoke peers until it exhausts its upload capacity even if the marginal utility of upload is sublinear. This potentially opens *BitTyrant* itself to being cheated, a topic we return to later.

For each peer p, maintain estimates of expected download performance d_p and upload required for reciprocation u_p.

Initialize u_p and d_p assuming the bandwidth distribution in Figure 2.

d_p is initially the expected equal split capacity of p.

u_p is initially the rate just above the step in the reciprocation probability.

Each round, rank order peers by the ratio d_p/u_p and unchoke those of top rank until the upload capacity is reached.

$$\underbrace{\frac{d_0}{u_0}, \frac{d_1}{u_1}, \frac{d_2}{u_2}, \frac{d_3}{u_3}, \frac{d_4}{u_4}}_{\text{choose } k \mid \sum_{i=0}^{k} u_i \leq cap}, \ldots$$

At the end of each round for each unchoked peer:

If peer p does not unchoke us: $u_p \leftarrow (1+\delta)u_p$

If peer p unchokes us: $d_p \leftarrow$ observed rate.

If peer p has unchoked us for the last r rounds: $u_p \leftarrow (1-\gamma)u_p$

Figure 9: *BitTyrant* unchoke algorithm

- The strategy can be easily generalized to handle concurrent downloads from multiple torrents. A client can optimize the aggregate download rate by ordering the d_p/u_p ratios of all connections across torrents, thereby dynamically allocating upload capacity to different torrents. Torrents can be prioritized using per-torrent weights for the d_p/u_p ratios.

The algorithm is based on the ideal assumption that peer capacities and reciprocation requirements are known. We discuss how to predict them next.

Determining upload contributions: The *BitTyrant* unchoke algorithm must estimate u_p, the upload contribution to p that induces its reciprocation. We initialize u_p based on the distribution of equal split capacities seen in our measurements, and then periodically update it depending on whether p reciprocates for an offered rate. In our implementation, u_p is decreased by $\gamma = 10\%$ if the peer reciprocates for $r = 3$ rounds, and increased by $\delta = 20\%$ if the peer fails to reciprocate after being unchoked during the previous round. We use small multiplicative factors since the spread of equal split capacities is typically small in current swarms. Although a natural first choice, we do not use a binary search algorithm, which maintains upper and lower bounds for upload contributions that induce reciprocation, because peer reciprocation changes rapidly under churn and bounds on reciprocation-inducing uploads would eventually be violated.

Estimating reciprocation bandwidths: For peers that unchoke the *BitTyrant* client, d_p is simply the rate at which data was obtained from p. Note that we do not use a packet-pair based bandwidth estimation technique as suggested by Bharambe [2], but rather consider the average rate of download over a round. Based on our measurements, not presented here due to space limitations, we find that packet-pair based bandwidth estimates do not accurately predict peers' equal split capacities due to variability in active size sets and end-host traffic shaping. The observed rate over a longer period is the only accurate estimate, a sentiment shared by Cohen [3].

Of course, this estimate is not available for peers that have not uploaded any data to the *BitTyrant* client. In such cases, *BitTyrant* approximates d_p for a given peer p by measuring the frequency of block announcements from p. The rate new blocks arrive at p provides an estimate of p's download rate, which we use as an estimate of p's total upload capacity. We then divide the estimated capacity by the Azureus recommended active set size for that rate to estimate p's equal split rate. This strategy is likely to overestimate the download rates of unobserved peers, serving to encourage their selection from the ranking of d_p/u_p ratios. At present, this preference for exploration may be advantageous due to the high-end skew in altruism. Discovering high end peers is rewarding: between the 95^{th} and 98^{th} percentiles, reciprocation throughput doubles. Of course, this strategy may open *BitTyrant* itself to exploitation, e.g. if a peer rapidly announces false blocks. We discuss how to make *BitTyrant* robust in Section 4.3 and 5.

4.2 Sizing the local neighborhood

Existing BitTorrent clients maintain a pool of typically 50–100 directly connected peers. The set is sized to be large enough to provide a diverse set of data so peers can exchange blocks without data availability constraints. However, the modeling results of Section 4.1 suggest that these typical local neighborhood sizes will not be large enough to maximize performance for high capacity peers, which may need an active set size of several hundred peers to maximize download throughput. Maintaining a larger local neighborhood also increases the number of optimistic unchokes received.

To increase the local neighborhood size in *BitTyrant*, we rely on existing BitTorrent protocol mechanisms and third party extensions implemented by Azureus. We request as many peers as possible from the centralized tracker at the maximum allowed frequency. Recently, the official BitTorrent protocol has incorporated a DHT-based distributed tracker that provides peer information and is indexed by a hash of the torrent. We have increased the query rate of this as well. Finally, the Azureus implementation includes a BitTorrent protocol extension for gossip among peers. Unfortunately, the protocol extension is push-based; it allows for a client to gossip to its peers the identity of its other peers but cannot induce those peers to gossip in return. As a result, we cannot exploit the gossip mechanism to extract extra peers.

A concern when increasing the size of the local neighborhood is the corresponding increase in protocol overhead. Peers need to exchange block availability information, messages indicating interest in blocks, and peer lists. Fortunately, the overhead imposed by maintaining additional connections is modest. In comparisons of *BitTyrant* and the existing Azureus client described in Section 5, we find that average protocol overhead as a percentage of total file data received increases from 0.9% to 1.9%. This suggests that scaling the local neighborhood size does not impose a significant overhead on *BitTyrant*.

4.3 Additional cheating strategies

We now discuss more strategies to improve download performance. We do not implement these in *BitTyrant* as they can be thwarted by simple fixes to clients. We mention them here for completeness.

Exploiting optimistic unchokes: The mainline BitTorrent client optimistically unchokes peers randomly,

a strategy that is robust to manipulation. Azureus, on the other hand, makes a weighted random choice that takes into account the number of bytes exchanged with a peer. If a peer has built up a deficit in the number of traded bytes, it is less likely to be picked for optimistic unchokes. We observe that high capacity peers are likely to have trading deficits with most peers as a consequence of high altruism. A cheating client can exploit this mechanism by disconnecting and reconnecting with a different client identifier, thereby wiping out the past history and increasing its chances of receiving optimistic unchokes, particularly from high capacity peers. This exploit becomes ineffective if clients maintain the IP addresses for all peers encountered during the download and maintain peer statistics after disconnections.

Downloading from seeds: Earlier versions of BitTorrent clients used a seeding algorithm wherein seeds upload to peers that are the fastest downloaders, an algorithm that is prone to exploitation by fast peers. More recent versions use a seeding algorithm that performs unchokes in a strict round-robin order, spreading data in a uniform manner and is therefore more robust.

Falsifying block availability: A client would prefer to unchoke those peers that have blocks that it needs. Thus, peers can appear to be more attractive by falsifying block announcements and increase the chances of being unchoked. In practice, this exploit is not very effective. First, a client is likely to consider most of its peers interesting given the large number of blocks in a typical torrent. Second, false announcements could lead to only short-term benefit as a client is unlikely to continue transferring once the cheating peer does not satisfy issued block requests.

5 Evaluation

In this section, we evaluate *BitTyrant*, exploring the performance improvement possible for a single selfish peer in synthetic and current real world swarms as well as the behavior of *BitTyrant* when used by all participants in synthetic swarms.

Evaluating altruism in BitTorrent experimentally and at scale is challenging. Traditional wide-area testbeds such as PlanetLab do not exhibit the highly skewed bandwidth distribution we observe in our measurements, a crucial factor in determining the amount of altruism. Alternatively, fully configurable local network testbeds such as Emulab are limited in scale and do not incorporate the myriad of performance events typical of operation in the wide-area. Further, BitTorrent implementations are diverse, as shown in Table 1.

To address these issues, we perform two separate evaluations. First, we evaluate *BitTyrant* on real swarms with torrents drawn from popular aggregation sites to measure real world performance for a single selfish client. This provides a concrete measure of the performance gains a user can achieve today. To provide more insight into how *BitTyrant* functions, we then revisit these results on PlanetLab where we evaluate sensitivity to various upload rates and evaluate what happens if *BitTyrant* is universally deployed.

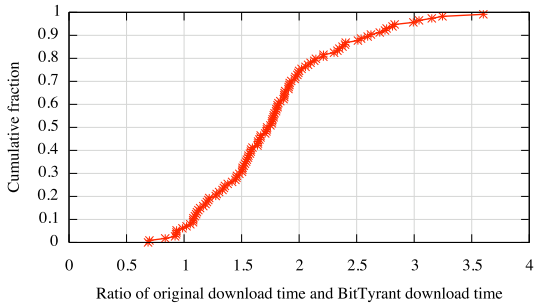


Figure 10: CDF of download performance for 114 real world torrents. Shown is the ratio between download times for an existing Azureus client and *BitTyrant*. Both clients were started simultaneously on machines at UW and were capped at 128 KB/s upload capacity.

5.1 Single selfish peer

To evaluate performance under the full diversity of realistic conditions, we crawled popular BitTorrent aggregation websites to find candidate swarms. We ranked these by popularity in terms of number of active participants, ignoring swarms distributing files larger than 1 GB. These swarms are typically for recently released files and have sizes ranging from 300–800 peers, with some swarms having as many as 2000.

We then simultaneously joined the swarm with a *BitTyrant* client and an unmodified Azureus client with recommended default settings[1]. We imposed a 128 KB/s upload capacity cap on each client and compared completion times to represent a relatively well provisioned peer for which Azureus has a recommended active set size. A CDF of the ratio of original client completion time to *BitTyrant* completion time is given in Figure 10. These results demonstrate the significant, real world performance boost that users can realize by switching to *BitTyrant*. The median performance gain factor for *BitTyrant* is a factor of 1.72 with 25% of downloads finishing at least twice as fast with *BitTyrant*. We expect relative performance gains to be even greater for clients with greater upload capacity.

These results provide insight into the performance properties of real BitTorrent swarms, some of which limit *BitTyrant*'s effectiveness. Because of the random set of peers that BitTorrent trackers return and the high skew

[1] `http://www.azureuswiki.com/index.php/Good_settings`

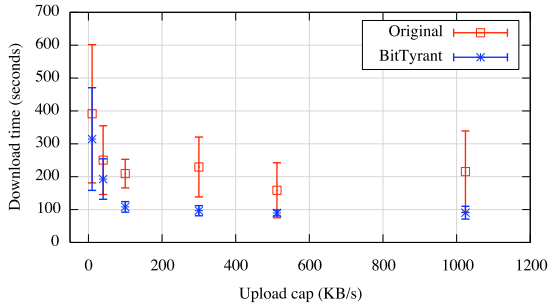


Figure 11: Download times and sample standard deviation comparing performance of a single *BitTyrant* client and an unmodified Azureus client on a synthetic PlanetLab swarm.

of real world equal split capacities, *BitTyrant* cannot always improve performance. For instance, in *BitTyrant*'s worst-performing swarm, only three peers had average equal split capacities greater than 10 KB/s. In contrast, the original client received eight such peers. Total download time was roughly 15 minutes, the typical minimum request interval for peers from the tracker. As a result, *BitTyrant* did not recover from its initial set of comparatively poor peers. To some extent, performance is based on luck with respect to the set of initial peers returned. More often than not, *BitTyrant* benefits from this, as it always requests a comparatively large set of peers from the tracker.

Another circumstance for which *BitTyrant* cannot significantly improve performance is a torrent whose aggregate performance is controlled by data availability rather than the upload capacity distribution. In the wild, swarms are often hamstrung by the number of peers seeding the file—i.e. those with a complete copy. If the capacity of these peers is low or if the torrent was only recently made available, there may simply not be enough available data for peers to saturate their upload capacities. In other words, if a seed with 128 KB/s capacity is providing data to a swarm of newly joined users, those peers will be able to download at a rate of at most 128 KB/s regardless of their capacity. Because many of the swarms we joined were recent, this effect may account for the 12 swarms for which download performance differed by less than 10%.

These scenarios can hinder the performance of *BitTyrant*, but they account for a small percentage of our observed torrents overall. For most real swarms today, users can realize significant performance benefits from the selfish behavior of *BitTyrant*.

Although the performance improvements gained from using *BitTyrant* in the real world are encouraging, they provide little insight into the operation of the system at scale. We next evaluate *BitTyrant* in synthetic scenarios on PlanetLab to shed light on the interplay between swarm properties, selfish behavior, and performance. Because PlanetLab does not exhibit the highly skewed bandwidth distribution observed in our traces, we rely on application level bandwidth caps to artificially constrain the bandwidth capacity of PlanetLab nodes in accordance with our observed distribution. However, because PlanetLab is often oversubscribed and shares bandwidth equally among competing users, not all nodes are capable of matching the highest values from our observed distribution. To cope with this, we scaled by $1/10^{\text{th}}$ both the upload capacity draws from our distribution as well as relevant experimental parameters such as file size, initial unchoke bandwidth, and block size. This was sufficient to provide overall fidelity to our intended distribution.

Figure 11 shows the download performance for a single *BitTyrant* client as a function of rate averaged over six trials with sample standard deviation. This experiment was hosted on 350 PlanetLab nodes with bandwidth capacities drawn from our scaled distribution. Three seeds with combined capacity of 128 KB/s were located at UW serving a 5 MB file. We did not change the default seeding behavior, and noticed that varying the combined seed capacity had little impact on overall swarm performance after exceeding the average upload capacity limit. To provide synthetic churn with constant capacity, each node's *BitTyrant* client disconnected immediately upon completion and reconnected immediately.

The results of Figure 11 provide several insights into the operation of *BitTyrant*:

- *BitTyrant* does not simply improve performance, it also provides more consistent performance across multiple trials. By dynamically sizing the active set and preferentially selecting peers to optimistically unchoke, *BitTyrant* avoids relying on the randomization of existing TFT implementations, which have slow convergence for high capacity peers (Section 3.1).
- There is a rate of diminishing returns for high capacity peers, and *BitTyrant* can discover it. For clients with high capacity, the number of peers and their available bandwidth distribution are significant factors in determining performance. Our modeling results from Section 4.1 suggest that the highest capacity peers may require several hundred available peers to fully maximize throughput due to reciprocation. Limited resources on PlanetLab encumber evaluation at this scale, and real world swarms are rarely this large. In these circumstances, *BitTyrant* performance is consistent, allowing peers to detect and reallocate excess capacity for other uses.
- Low capacity peers can benefit from *BitTyrant*. Although the most significant performance benefit comes from intelligently sizing the active set for high capac-

ity peers (see Figure 8), low capacity peers can still improve performance with strategic peer selection, providing them with an incentive to adopt *BitTyrant*.

- Fidelity to our specified capacity distribution is consistent across multiple trials. Comparability of results is often a concern on PlanetLab, but our results suggest a minimum download time determined by the distribution that is consistent across trials spanning several hours. Further, the consistent performance of *BitTyrant* in comparison to unmodified Azureus suggests that the variability observed is due to protocol and strategy differences and not PlanetLab variability.

5.2 Many selfish peers

We next examine the performance of *BitTyrant* when used by all peers in a swarm. Our experimental setup included 350 PlanetLab nodes with upload capacities drawn from our scaled distribution simultaneously joining a 5 MB torrent with combined seed capacity of 128 KB/s with all peers departing immediately upon download completion. Initially, we expected fairness to improve and performance to degrade since high capacity peers would finish more quickly, reducing capacity in the system. Surprisingly, performance improved overall and altruism increased. These results are summarized by the CDFs of completion times comparing *BitTyrant* and the unmodified Azureus client in Figure 12. Although *BitTyrant* attempts to allocate bandwidth to maximize reciprocation, bandwidth is not withheld by peers with excess capacity. As we observe in Figure 11, the point of diminishing returns in *BitTyrant* for upload contribution is realized at a rate far lower than that typical of unmodified Azureus swarms. In such a case, a swarm composed entirely of *BitTyrant* clients will not have less capacity, but it may make more efficient use of that capacity due to intelligent peer and rate selection, particularly during the early bootstrapping phase of a torrent with simultaneous arrivals (as in our experiment) or a flash crowd.

These results are consistent with our model. In a swarm where the upload capacity distribution has significant skew, high capacity peers require many connections to maximize throughput due to reciprocation. However, evaluation of *BitTyrant* for single torrent performance does not reflect typical behavior, as most users participate in many torrents simultaneously [7]. All popular BitTorrent implementations support simultaneous download of multiple torrents, but few make any attempt to intelligently allocate bandwidth among torrents. Those that do typically allocate some amount of a global upload capacity to each torrent, which is then split equally among peers in statically sized active sets. A global connection limit is often enforced. *BitTyrant*'s unchoking algorithm transitions naturally from single to multiple torrents. Rather than allocate bandwidth among torrents, as existing clients do, *BitTyrant* allocates bandwidth among connections, optimizing aggregate download throughput for all torrents.

If high capacity peers run *BitTyrant* and participate in multiple simultaneous swarms, the performance implications for low capacity and single-torrent peers are significant. Multiple swarms provide a setting where active set sizes can grow to the sizes necessary for high capacity peers. Rather than contributing excess resources to a single swarm, *BitTyrant* peers participating in multiple torrents can allocate excess capacity to other swarms. To model this effect, we limited the upload capacity of all high capacity peers in our scaled distribution to 100 KB/s, the point of diminishing returns observed in Figure 11. A CDF of performance under the capped distribution for an experiment with simultaneous joins is shown in Figure 12. As expected, aggregate performance decreases. More interesting, however, is that the *BitTyrant* determined rate is stable, i.e. a higher capacity peer cannot achieve significantly faster download rates.

The upload capacity distribution of the swarm defines a stable point of diminishing returns under *BitTyrant*'s strategies. Peers with capacity exceeding that rate are likely to allocate excess bandwidth elsewhere, reducing aggregate capacity and average performance, particularly for low capacity peers. This is reflected in comparing the performance of single clients under the scaled distribution (Figure 11) and single client performance under the scaled distribution when constrained (Figure 12). The average completion time for a low capacity peer moves from 314 to 733 seconds. Average completion time for a peer with 100 KB/s of upload capacity increases from 108 seconds to 190.

Our evaluation of *BitTyrant* at scale gives rise to number of concerns arising from selfish behavior:

- Average performance for individual swarms and the majority of peers degrades. If high capacity peers can participate in many torrents or otherwise limit altruism, total capacity per swarm decreases. This reduction in capacity lengthens download times for all users of a single swarm regardless of capacity. Although high capacity peers will see an increase in aggregate download rate across many swarms, low capacity peers that cannot successfully compete in multiple swarms simultaneously will see a large reduction in download rates, but each individual peer is incented to be selfish because their performance improves relative to that of standard clients, even when everyone is selfish.
- New users experience a lengthy bootstrapping period. To maximize throughput selfishly, *BitTyrant* unchokes peers that send fast. New users without data are bootstrapped by the excess capacity of the system only.

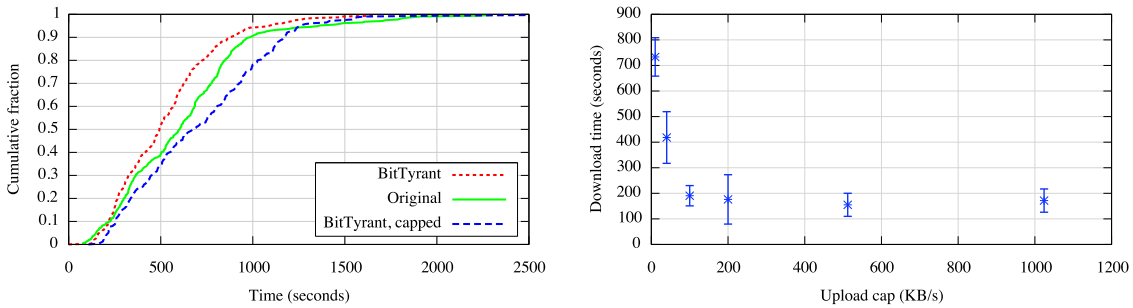


Figure 12: *Left:* CDFs of completion times for a 350 node PlanetLab experiment. *BitTyrant* and original client use the original scaled capacity distribution. Capped *BitTyrant* reduced caps on all high capacity nodes ($\geq$ 100 KB/s) to 100 KB/s. *Right:* The impact of selfish *BitTyrant* caps on performance. Error bars give sample standard deviation over six trials. Download times at all bandwidth levels increase and very high capacity peers gain little from excess contribution.

Bootstrapping time may be reduced by reintroducing optimistic unchokes in *BitTyrant*, but it is not clear that selfish peers have any incentive to do so.

- Peering relationships are not stable. *BitTyrant* was designed to exploit the significant altruism that exists in BitTorrent swarms today. As such, it continually reduces send rates for peers that reciprocate, attempting to find the minimum rate required. Rather than attempting to ramp up send rates between high capacity peers, *BitTyrant* tends to spread available capacity among many low capacity peers, potentially causing inefficiency due to TCP effects [15].

To work around this third effect, we propose extending *BitTyrant* to advertise itself at connection time using the Peer ID hash. Without protocol modification, *BitTyrant* peers can recognize one another and switch to a block-based tit-for-tat strategy that ramps up send rates conservatively until capacity is reached. This is easily implemented and requires no protocol changes. *BitTyrant* clients can simply choke other *BitTyrant* peers whose block request rates exceeds their send rates. By gradually increasing send and request rates to other *BitTyrant* clients, fairness can be preserved while maximizing reciprocation rate with fewer connections. In this way, *BitTyrant* can provide a deployment path leading to the conceptually simple strategy of block-based tit-for-tat by providing a selfish, short-term incentive for adoption by all users—even those that stand to lose from a shift to block-based reciprocation.

We do not claim that *BitTyrant* is strategy proof, even when extended with block-based tit-for-tat, and leave open the question of whether further strategizing can be effective as future work. However, a switch to block based tit-for-tat among mutually agreeing peers would place a hard limit on altruism and limit the range of possible strategies.

6 Related work

Modeling and analysis of BitTorrent's incentive mechanism and its effect on performance has seen a large body of work since Cohen's [3] seminal paper. Our effort differs from existing work in two fundamental ways. First is the *conclusion*: we refute popular wisdom that BitTorrent's incentive mechanism makes it robust to strategic peer behavior. Second is the *methodology*: most existing studies consider small simulated settings that poorly capture the diversity of deployed BitTorrent clients, selfish peer behavior, peer capacities, and network conditions. In contrast, we explore BitTorrent's strategy space with our implementation of a selfish client and evaluate it using analytical modeling, experiments under real network conditions, and testing in the wild.

The canonical tit-for-tat strategy was first evaluated by Axelrod [1], who showed using a competition that the strategy performs better than other submissions when there are many repeated games, persistent identities, and no collusion. Qiu and Srikant [16] specifically study BitTorrent's rate-based tit-for-tat strategy. They show that if peers strategically limit their upload bandwidth (but split it equally) while trying to maximize download, then, under some bandwidth distributions, the system converges to a Nash equilibrium where all peers upload at their capacity. These results might lead one to believe that BitTorrent's incentive mechanism is robust as it incentivizes users to contribute their entire upload capacities. Unfortunately, our work shows that BitTorrent fails to attain such an equilibrium for typical file sizes in swarms with realistic bandwidth distributions and churn, which our selfish client exploits through strategic rate selection.

Bharambe et al.[2] simulate BitTorrent using a synthetically generated distribution of peer upload capacities. They show the presence of significant altruism in BitTorrent and propose two alternate peer selection algorithms based on (i) matching peers with similar band-

width, and (ii) enforcing tit-for-tat at the block level, a strategy also proposed by [9]. Fan et al.propose strategies for assigning rates to connections [5], which when adopted by all members of a swarm would lead to fairness and minimal altruism. The robustness of these mechanisms to strategic peer behavior is unclear. But more importantly, these proposals appear to lack a convincing evolution path — a peer adopting these strategies today will severely hurt its download throughput as the majority of deployed conformant clients will find such a peer unattractive. In contrast, we demonstrate that *BitTyrant* can drastically reduce altruism for a single selfish client today incenting its adoption.

Shneidman et al.[18] identify two forms of selfish manipulation based on Sybil attacks [4] and a third based on uploading garbage data. Liogkas et al.[12] propose downloading only from seeds and also identify an exploit based on uploading garbage data. However, there exist reasonable fixes to minimize the impact of such "byzantine" behavior. A third exploit by Liogkas et al. involves downloading only from the fastest peers, but the strategy does not take into account the upload contribution required to induce reciprocation. In contrast, our selfish client maximizes download per unit of upload bandwidth and can drastically reduce its upload contribution by varying the active set size and not sharing its upload bandwidth uniformly with active peers.

Hales and Patarin [8] argue that BitTorrent's robustness is not so much due to its tit-for-tat mechanism, but more due to human or sociological factors that cause torrents with a high concentration of altruistic peers to be preserved over selfish ones. They further claim that releasing selfish clients into the wild may therefore not degrade performance due to the underlying natural selection. It is unclear how to validate such hypotheses without building and releasing real selfish clients—one of our contributions.

Massoulie and Vojnovic [14] model BitTorrent as a "coupon replication" system with a particular focus on efficiently locating the last few coupons. One of their conclusions is that altruism is not necessary for BitTorrent to be efficient. However, their study does not account for strategic behavior on part of peers.

Other studies [2, 7, 11] have pointed out the presence of significant altruism in BitTorrent or suggest preserving it [11]. In contrast, we show that the altruism is not a consequence of BitTorrent's incentive mechanism and can in fact be easily circumvented by a selfish client.

7 Conclusion

We have revisited the issue of incentive compatibility in BitTorrent and arrived at a surprising conclusion: although tit-for-tat discourages free riding, the bulk of BitTorrent's performance has little to do with tit-for-tat. The dominant performance effect in practice is altruistic contribution on the part of a small minority of high capacity peers. More importantly, this altruism is not a consequence of tit-for-tat; selfish peers—even those with modest resources—can significantly reduce their contribution and yet improve their download performance. Thus, BitTorrent works so well simply because most people faithfully use downloaded client software as-is without trying to cheat the system.

Based on the insights gathered from a simple model of BitTorrent's altruism, we built *BitTyrant*, a selfish client that cleverly selects peers so as to optimize the amount of download per unit of upload bandwidth. We found that *BitTyrant* improves performance for all peers that use it. Nevertheless, in practice, *BitTyrant* will hurt the performance of individual swarms as high capacity peers reach a point of diminishing returns and are incented to either withhold their upload contribution or invest it in other swarms. Low capacity peers do not enjoy such a luxury. As the majority of peers have low capacity, they will see degraded performance compared to BitTorrent today.

The *BitTyrant* source code and distribution are publicly available at

`http://www.cs.washington.edu/homes/piatek/BitTyrant/`

References

[1] R. Axelrod. *The Evolution of Cooperation*. Basic Books, 1985.

[2] A. Bharambe, C. Herley, and V. Padmanabhan. Analyzing and Improving a BitTorrent Network's Performance Mechanisms. In *IEEE INFOCOM*, 2006.

[3] B. Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, 2003.

[4] J. R. Douceur. The Sybil attack. In *International Workshop on Peer-to-Peer Systems*, 2002.

[5] B. Fan, D.-M. Chiu, and J. Liu. The Delicate Tradeoffs in BitTorrent-like File Sharing Protocol Design. In *Proc. of ICNP*, 2006.

[6] GNU Scientific Library. `http://www.gnu.org/software/gsl/`.

[7] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, and X. Zhang. Measurements, analysis, and modeling of BitTorrent-like systems. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, 2005.

[8] D. Hales and S. Patarin. How to Cheat BitTorrent and Why Nobody Does. Technical Report UBLCS 2005-12, Computer Science, University of Bologna, 2005.

[9] S. Jun and M. Ahamad. Incentives in BitTorrent induce free riding. In *Proc. of P2PECON*, 2005.

[10] S. Katti, D. Katabi, C. Blake, E. Kohler, and J. Strauss. MultiQ: Automated detection of multiple bottleneck capacities along a path. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, 2004.

[11] A. Legout, G. Urvoy-Keller, and P. Michiardi. Rarest First and Choke Algorithms are Enough. In *Proc. of IMC*, 2006.

Label	Definition	Meaning
ω	2	Number of simultaneous optimistic unchokes per peer
λ	80	Local neighborhood size (directly connected peers)
$b(r)$	Figure 1	Probability of upload capacity rate r
$B(r)$	$\int_0^r b(r)dr$	Cumulative probability of a upload capacity rate r
$\text{active}(r)$	$\lfloor\sqrt{0.6r}\rfloor - \omega$	Size (in peers) of the active transfer set for upload capacity rate r
$\text{split}(r)$	$\frac{r}{\text{active}(r)+\omega}$	Per-connection upload capacity for upload capacity rate r
$s(r)$	Figure 1	Probability of an equal-split rate r using mainline $\text{active}(r)$ sizing
$S(r)$	$\int_0^r s(r)dr$	Cumulative probability of an equal-split rate r

Table 2: Functions used in our model and their default settings in the official BitTorrent client.

[12] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting BitTorrent for fun (but not profit). In *Proc. of IPTPS*, 2006.

[13] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proc. of Operating Systems Design and Implementation*, 2006.

[14] L. Massoulié; and M. Vojnović. Coupon replication systems. *SIGMETRICS Perform. Eval. Rev.*, 33(1):2–13, 2005.

[15] R. Morris. TCP behavior with many flows. In *Proceedings of ICNP*, 1997.

[16] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proceedings of Sigcomm*, 2004.

[17] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, 2002.

[18] J. Shneidman, D. Parkes, and L. Massoulié. Faithfulness in internet algorithms. In *Proc. of PINS*, 2004.

A Modeling notes

All numerical evaluation was performed with the GSL numerics package [6]. Refer to Section 3 for assumptions and Table 2 for definitions.

Upload / download: Probability of reciprocation for a peer P with upload capacity r_P from Q with r_Q:

$$\text{p\_recip}(r_P, r_Q) = 1 - (1 - S(r_P))^{\text{active}(r_Q)} \quad (1)$$

Expected reciprocation probability for capacity r:

$$\text{recip}(r) = \int b(x)\text{p\_recip}(r, x)dx \quad (2)$$

Expected download and upload rate for capacity r:

$$\text{D}(r) = \text{active}(r)\left[\int b(x)\text{p\_recip}(r, x)\text{split}(x)dx\right] + \omega\left[\int b(x)\text{split}(x)dx\right] \quad (3)$$

$$\text{U}(r) = \min\big(r, (\text{active}(r) + \omega)\ \text{dl}(r)\big) \quad (4)$$

Altruism: Altruism when defined as the difference between upload contribution and download reward

$$\text{altruism\_gap}(r) = \max\big(0,\ \text{U}(r) - \text{D}(r)\big) \quad (5)$$

Altruism per connection when defined as upload contribution not resulting in direct reciprocation.

$$\text{altruism\_conn}(r) = \int \Big(b(x)\big((1 - \text{p\_recip}(r, x))\text{split}(r) + \text{p\_recip}(r, x)\max(0, \text{split}(r) - \text{split}(x))\big)\Big)dx \quad (6)$$

Total altruism not resulting in direct reciprocation.

$$\text{altruism}(r) = (\text{active}(r) + \omega)\text{altruism\_conn}(r) \quad (7)$$

Convergence: Probability of a peer with rate r discovering matched TFT peer in n iterations:

$$c(r, n) = 1 - S(r)^{n\ 2\omega} \quad (8)$$

Time to populate active set with matched peers given upload capacity r. Note, $s = \text{split}(r)$, and T = 30s is the period after which optimistic unchokes are switched.

$$\text{convergence\_time}(r) = T \cdot \text{active}(r)\Big(c(s, 1) + \sum_{n=2}^{\infty} n\ c(s, n) \prod_{i=1}^{n-1}(1 - c(s, i))\Big) \quad (9)$$

Unchoke probability: The distribution of number of optimistic unchokes is binomial with success probability $\frac{\omega}{\lambda}$. Because overhead is low, $\lambda \gg \text{active}(r)$ in ***BitTyrant***, we approximate $\lambda - \text{active}(r)$ by λ. The expected number of optimistic unchokes per round is ω.

$$Pr[\text{unchokes} = x] = \binom{\lambda}{x}\left(\frac{\omega}{\lambda}\right)^x\left(1 - \frac{\omega}{\lambda}\right)^{(\lambda - x)} \quad (10)$$

$$\therefore E[\text{unchokes}] = \lambda\frac{\omega}{\lambda} = \omega$$