CS 357: Final Project

Aaron Pinto and Zack Romrell

May 2022

Abstract

A general problem in most peer-to-peer (P2P) systems is free-riding – when a user takes resources and fails to contribute back to the system. The BitTorrent Protocol incentivizes cooperation by breaking up files into smaller pieces, creating a repeated game on the piece level for each user. The main BitTorrent reference client also incentivizes collaboration through its tit-for-tat strategy. However, past research has revealed that its upload strategy is significantly altruistic, and the BitTyrant client capitalizes on this expected altruism to achieve a faster download rate relative to its peers, demonstrating that BitTorrent is not robust to selfishness. The FairTorrent client seeks to maximize fairness by uploading to peers with the lowest deficit of net uploads (uploads minus downloads). Our overall goal is to implement each of these three clients and analyze how they perform with each other. This will help contribute to our understanding of client's performance in a given swarm and help us discover optimizations to existing clients, further maximizing their overall performance. Below we analyze the effect of multiple factors on client performance like the size of the file, number of peers, and the interaction of different strategies. We also consider small optimizations one can make for BitTyrant and combine these to create an optimized BitTyrant client for a given swarm. Our main findings can be summarized to (1) BitTorrent is indeed susceptible to strategic manipulation by BitTyrant (2) BitTyrant can optimize its u_p initialization, δ and γ values to decrease relative average completion times and (3) the rarest first requesting strategy can utilize internal peer and piece randomization to optimize performance. Lastly we look into a self-made client, BitExpose, that takes advantage of BitTyrant's strategy.

1 Introduction

Peer-to-Peer Systems Overview In the late 1990s, as broadband internet access was becoming more accessible and scaling around the world, file sharing became increasingly popular. Users were incentivized to use file sharing systems in order to access free content. Peer-to-peer (P2P) file sharing systems are a specific method of gaining access to files in which users (*peers*) do not solely download files from others, but additionally upload files themselves. In this system, the costs of uploading are redistributed to each peer rather than placed solely on the server, the traditional *client server model*. Early P2P file sharing systems suffered from various problems such as legality issues regarding downloads of copyrighted files[1], but the main issue that decreased these platform's traffic was free-riding. This quality occurs when a peer consumes resources (downloads files) from others, but fails to contribute resources (upload files) themselves. Gnutella, a decentralized P2P file sharing system introduced in 2000, had no incentives for a peer to upload and as a result, peers increasingly chose to free-ride. As the percentage of peers engaging in free-riding increased, Gnutella increasingly lost market share until in 2013, the system was responsible for only 1% of global P2P traffic, down from their peak estimated market share of about 40%.

BitTorrent and Relevant Terminology BitTorrent was introduced as a decentralized P2P system that allows multiple users to download a file. In this system, a user that wants a file generally visits a website that maintains a searchable directly of torrents that link to *.torrent* files. Once a user downloads a .torrent file, they open it up with a BitTorrent client. The .torrent file contains the relevant metadata, and includes the name and size of the desired file. A desired file is broken up into *pieces*, and further divided into *blocks*. The .torrent file additionally contains a digital fingerprint of data blocks to be downloaded as well as a *tracker* that is responsible for coordinating

[1] P2P file sharing system Napster was shut down in 2001 due to their system encouraging copyright infringement

peers interested in the same desired file. To introduce more BitTorrent specific language, the set of peers uploading or downloading the same file is called a *swarm*. The peers actively downloading pieces of a file are called *leechers* whilst the peers that possess the entire file and are uploading pieces of the file are called *seeders*. Each peer can initiate a connection with another peer and two connected peers are called *neighbors*. As peer can maintain connections with many other peers, the set of connected peers is called a *neighborhood*. Each peer in a neighborhood actively sends messages to its neighbors detailing which pieces they possess. A peer determines who to download from and upload to based on the specific client/strategy they follow. We will briefly explain the BitTorrent reference client, the BitTyrant client, and the FairTorrent client in following three subsections. Implementation of these clients is covered in Section 2.

BitTorrent Reference Client With BitTorrent's introduction by Brahm Cohen, he also introduced the BitTorrent reference client as the proposed method of interacting with this protocol. The reference client has specific download and upload functions that attempt to reduce free-riding by providing incentives to upload. Each peer requests to download pieces based on a *rarest first* strategy. In other words, peers will request pieces that have the lowest availability amongst their neighbors. Each peer maintains S *active slots*, which is the maximum amount of peers they can upload to. Each peer will upload to S-1 peers that they received the highest average download speeds from. Approximately, every 30 seconds, a peer will optimistically unchoke and upload to a randomly selected peer in their neighborhood. The purpose of the optimistic unchoking slot is to make a new connection in the hopes that this random peer will reciprocate and upload a desired piece in return. The upload capacity a peer contributes is equally split between all peers in their active set. The upload strategy promotes cooperation and rewards a faster download speed to its uploaders. The rarest first request strategy prevents bottle-necks, making sure all pieces to a file are downloaded and available to be shared and was initially thought to be resilient to strategic manipulation [1].

BitTyrant Client As past research showed the presence of significant altruism in the BitTorrent reference client, BitTyrant tries to minimize unnecessary uploads and maximize download speeds for the same upload contribution by maintaining estimates of a peers' download to upload speed ratios and uploading to the peers with the largest ratios. The client also does not employ equal-split upload contributions and seeks to increase reciprocation throughput. The unequal split chooses to upload the minimum expected bandwidth needed for reciprocation to the users with the greatest expected download rate. In the event a peer p has been uploaded to by another peer p' in the preceding three rounds, p will slightly decrease the amount it uploads to p' hoping to maintain its download rate at a lower upload rate. Conversely, if a peer p'' has not uploaded to p in the preceding round, p will slightly increase its upload to p'' to increase the likelihood of reciprocation. This strategic behavior for uploads has proven to perform better for BitTyrant [2].

FairTorrent Client Further research has explored strategic manipulation to target fairness for client performance rather than download speeds. The FairTorrent client dynamically maintains a list of peer deficits, which are then used to decide which peer the client uploads to. The deficit that the FairTorrent maintains for each peer p is the the amount the FairTorrent uploads to p minus the amount the FairTorrent downloads from p. The deficits are then sorted from smallest to largest and the FairTorrent client uploads to the peers in the order of increasing deficits until their bandwidth

is used up. In other words, this client uploads to the peers to which it owes the most. FairTorrent has proven to provide better fairness measures than other BitTorrent clients and provides up to five times better download speeds for contributing peers [3].

2 Methodology and Contributions

2.1 Implementation

Below we briefly touch on specific values and nuances we incorporated into our implementation and present any assumptions made. We implemented the following clients based off of information provided in the research papers references below.

Requesting BitTorrent, BitTyrant and FairTorrent all utilize the same *rarest first* requesting strategy, in which the client requests pieces in order of lowest availability in the swarm. Our initial implementation sorted the pieces by rarity then requested each peer in possession of that piece in the sorted sequence. This made it so when pieces had the same rarity the order would default to the lower pieceId, resulting in a very similar set of initial requests for competing clients. To avoid very similar request profiles for each client throughout the simulation, we made the first 5 rounds of requests randomized. We will also discuss other alterations to this function which ensured a more unique request list for peers. One concern and possible area for optimization is that multiple requests to different peers for the same (most rare) piece can take up multiple slots of the constant maximum number of requests space, decreasing the total number of possible requests that can be fulfilled in a given round. Although this is an inefficiency there is no easy way to sidestep this issue without possibly deviating from the rarest first protocol. Although these clients all have identical requesting strategies, they all differ in their uploading strategies [1] [2] [3].

BitTorrent Unchoking Strategy Our implementation of the BitTorrent client unchokes (uploads) to only four peers. This client decides three of these unchoke slots by choosing the three peers that have uploaded to it the most in past rounds. The fourth unchoke slot is given to a random peer in the swarm who is distinct from the previous three unchoked peers. This optimistic unchoke slot is given to the same random peer for three rounds whilst the other three unchoke slots are updated each round. We do not explore the effects and potential benefits present by changing these round counts; however, in any given swarm these values can be optimized. Our implementation follows the remaining details of the algorithm providing an equal split of the BitTorrent's bandwidth [1].

BitTyrant Unchoking Strategy Our implementation of the BitTyrant client sorts peers (in decreasing order) based on the ratio of expected downloads to expected uploads required for reciprocation ($\frac{d_p}{u_p}$ for a peer p). The client unchokes peers in this order until BitTyrant runs out of bandwidth. However, one can adjust the amount of bandwidth that BitTyrant uploads to peers to a lower value to minimize bandwidth sacrificing completion time. Later on, we analyze the effects of adjusting this value and discuss how it can possibly both minimize bandwidth and round completion. For simplicity, our BitTyrant client initializes the expected upload bandwidths required for reciprocation to 1 for each peer. There are better ways of estimating this value in a real BitTorrent environment, however, for our use this value is a fine starting point and we later experiment to find better initialization values for certain swarms. Similarly, our implementation initializes expected

downloads for each peer to a random integer between the minimum and maximum bandwidth of the simulation divided by four. This is due to the assumption that a BitTyrant client is a leecher in a swarm of BitTorrent peers who equally split their bandwidth amongst four unchoked peers. There are better ways of initializing this variable in the real world. However, this is satisfactory for the simulation. Immediately after BitTyrant receives a download from a peer they update the expected download variable for that peer to the corresponding bandwidth. At the end of each round, the BitTyrant client analyzes the behavior of its peers and updates these variables respectively. If BitTyrant has unchoked peer p and p has not unchoked them in return, BitTyrant increases its expected upload for reciprocation ($u_p = (1 + \delta)u_p$). If p has unchoked the BitTyrant client for the previous three rounds, they will lower their expected upload for reciprocation ($u_p = (1 - \gamma)u_p$). Deterring slightly from Piatek et al., our implementation sets $\delta = 0.1$ and $\gamma = 0.1$. However, there exists more optimal values for performance in a given swarm. Later on fixing other test parameters we experiment with these values to see how much they effect overall performance. Overall, these specifications aim to increase upload contributions while minimizing upload bandwidth to still receive reciprocation and different values will be better or worse at the given task.

FairTorrent Unchoking Strategy The FairTorrent client maintains a deficit list that is updated each round. The deficit is initialized to zero for each peer in the swarm in round zero. For each following round, deficits are updated based on the download and upload history of the FairTorrent client. In our implementation, the deficit for a peer is incremented by the number of blocks FairTorrent uploaded to the peer and decremented by the amount of bandwidth the peer uploaded to it. Depending on the extent of "fair" you want to abide by you may want to adjust this metric. The deficit list is then re-sorted in increasing order of deficit and peers are unchoked in this order. If a peer did not request the FairTorrent in the current round, FairTorrent skips over them and resumes its search for the peer with the next lowest deficit who has requested a piece. Our implementation of FairTorrent continues to unchoke peers until they run out of bandwidth, equally splitting bandwidth between the unchoked peers. However, if a given user has a bandwidth constraint or wants to operate in a given range this value can be adjusted.

2.2 Goals

Whilst following client implementations and the identification of possible optimizations to certain strategies, our main goal has been to better understand how these peers interact in actual swarms. Below we will discuss the types of simulations we ran and what parameters we have fixed and changed to better understand the factors that both enhance and diminish the performance of certain clients in a given environment. We will further try to generalize these results to properties of the participating clients and determine and test some alterations one can make to selfishly or globally improve performance.

3 Simulations and Analysis

3.1 Simulation Parameters

Before diving into the our collected data and analysis we will first get you familiar with what parameters exist for the simulation and how to run the simulation.

```
//General Parameter Format
python3 sim.py --loglevel=info --numPieces=m --blocksPerPiece=n --minBw=16 --maxBw=64
    --maxRound=1000 --iters=40 Seed,2 client1,x client2,y
```

Above is a standard format for the command prompts used below to collect data about client's and their performances. At the end of the simulation each user's average upload bandwidth and average round completion over the 40 iterations is displayed. We utilized data directly from the average output at the end of the simulation. In our analysis we mainly focus on the average round completion, however, one can also perform similar experiments focusing on the bandwidth. The number of pieces is a parameter that we vary quite often with our tests. As the number of pieces increases the total length of the simulation also increases because it takes longer for all the peers to acquire the larger set. Altering this parameter allows us to determine the long term effects of certain differences and better understand how client's strategies interact in the long-run. Typically, we leave blocks per piece unchanged at 16 because the simulation run time can get quite long as you increase this parameter along with piece size, however, for one test we do use this parameter to analyze slight deviations of randomization in the rarest first requesting strategy. Throughout all of our tests, we keep the minBw, maxBw, maxRound, iters, and the number of seeds constant to further help us zone in on the other parameter changes and strategy deviations. We decided on 40 iterations to keep our simulation run time shorter for the larger cases like numPieces = 256 and blocksPerPiece = 256. Lastly, we vary which clients we perform tests on and the number of clients we use. In all of our tests, the number of the clients, $x + y$, sum to ten. This number is arbitrarily chosen and ten clients in a swarm has no significance. First, we analyze our implemented BitTorrent client against the given FreeRider client that came along with the simulation framework. Then we analyze our implemented BitTyrant client with BitTorrent to confirm implementation correctness and analyze how their strategies interact under a variety of parameters. Within that section, we adjust BitTyrants initialization values for U_p, γ, and δ to better understand possible optimizations and confirm our general intuition about the strategies as a whole. Next we briefly touch on possible ramifications to the the rarest first request strategy and how/why they behave differently under certain parameter changes and not others. Finally, we quickly touch on a very simple self-implemented client, BitExpose, that is very inefficient against BitTorrent, however, takes advantage of BitTyrant.

3.2 BitTorrent vs. FreeRider

First, we wanted to analyze a swarm of BitTorrent versus FreeRider clients to better understand how increasing the file size affects their relative performance. First, it is important to note that the FreeRider client does not upload to other peers and it requests random pieces each round. Also, FreeRider relies on taking advantage of BitTorrent's optimistic unchoking slots, since FreeRider would not be one of a BitTorrent's top three uploading peers. Due to these characteristics FreeRider performs better in a population of more BitTorrent clients. In a converse scenario of a swarm fully consisting of FreeRiders, since they do not upload anything, the substitution of a BitTorrent client in exchange for a FreeRider client should increase overall performance globally. This is shown in simulations we ran for a swarm of ten clients. When testing how BitTorrents perform against Freeriders for an increasing number of pieces, we see that the average round completion for BitTorrent clients increase linearly at roughly 100-125% for a 100% increase in the number of

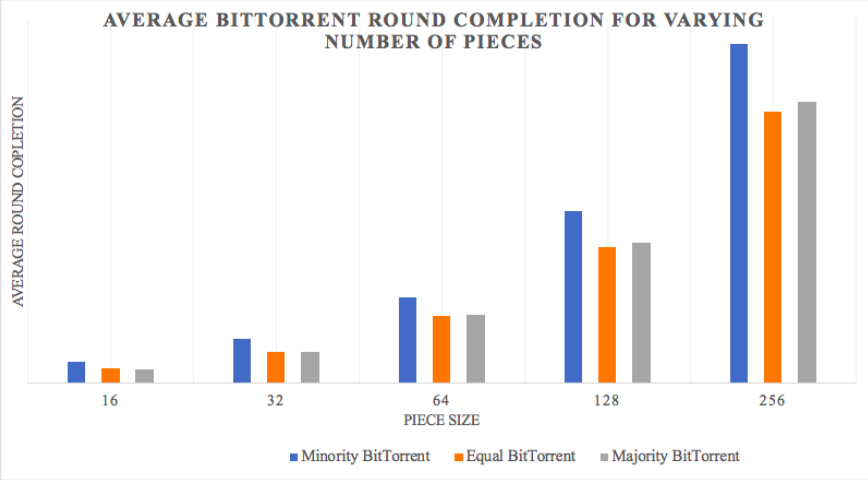

Figure 1: BitTorrent clients' average completion approximately double for a doubled number of pieces.

pieces in all cases. See *Figure 1.* These results are consistent with how Cohen et al. (2003) would expect BitTorrent to perform against FreeRiders, and it also shows that the BitTorrent client is susceptible to strategic manipulation [1].

Although we did not specifically experiment how changing the duration of BitTorrent's optimistic unchoke slot affects performance for the FreeRider client, we hypothesize that increasing the duration of the optimistic unchoke slot to $r > 3$ would increase the variation of the FreeRider's average round completion as the 'lucky' unchoked FreeRiders would benefit from faster average completion times, but the 'unlucky' FreeRiders that are not unchoked would experience slower average completion times. Conversely for $r < 3$ we expect the variation for FreeRider's performance to be lower due to a similar argument.

3.3 BitTorrent vs. BitTyrant with Optimizations

Next, we wanted to analyze swarms involving both BitTyrant and BitTorrent clients. Later on we introduce possible modifications to BitTyrant, and analyze their effects on performance. First, we compared the performances of BitTyrants and BitTorrents in varying populations. For a fixed size of 10 peers in the swarm we tested client proportions of 1:9, 5:5, and 9:1 for numPieces varying from 16 to 256. In all populations we noticed that in the 16 piece case BitTorrent outperformed BitTyrant. This is largely do to the fact that with a smaller number of pieces the simulation finishes very quickly and BitTyrant isn't able to effectively tap into the latent altruism. With a larger number of pieces we noticed that BitTyrant was greatly outperforming BitTorrent in the 1:9 case, however, in the other two cases BitTyrant was still falling behind. In this initial test BitTyrant was set to initialize the expected upload bandwidth for reciprocation to 1. In each of these swarms, we varied BitTyrant's initialized upload bandwidth for reciprocation to other values and gathered simulation data for, $u_p = 5, 10$ on top of the initial data for $u_p = 1$. We noticed that in the 5:5 and 9:1 cases when BitTyrant initialized u_p to 5 BitTyrant was beating out BitTorrent. Given any specific swarm there exists a initialization value that maximizes BitTyrant's performance. *Figure 2* shows the trend of the varying initialized u_p for the minority, 1:9, swarm. This figure shows that as

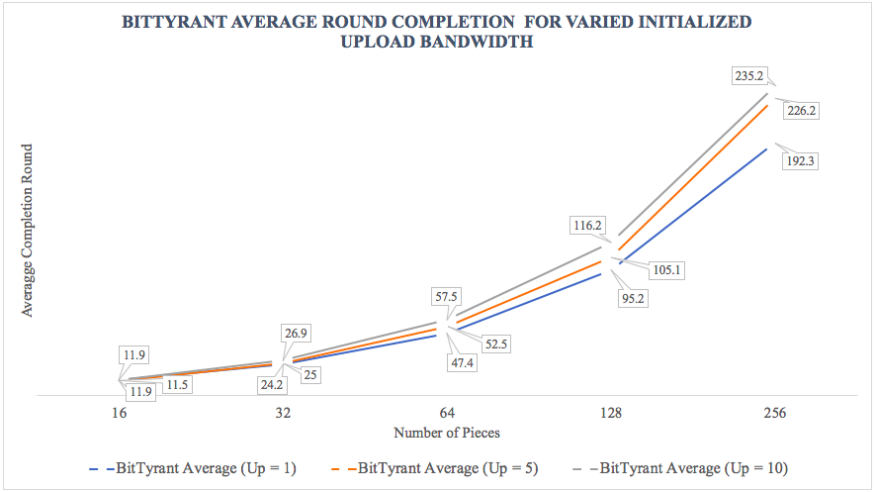

Figure 2: BitTyrant average completion rounds decrease for lower initializations of u_p for the described scenario.

u_p increases from one to ten, the BitTyrant client's performance is worse off. Thus, in this minority swarm, it is in BitTyrant's best interest to initialize u_p to 1. Since the BitTorrent clients are not strategically choosing who to upload to, it is more efficient to start uploading less and converge to the optimal uploading bandwidth from below rather than over uploading initially and working down from a greater bandwidth. Due to this finding, we use $u_p = 1$ when optimizing BitTyrant parameters in a 1:9 BitTyrant BitTorrent swarm. We also observed as you increase u_p from 5 to 10 in 5:5 and 9:1 BitTyrant:BitTorrent scenarios, BitTyrants finish earlier than BitTorrents by larger margins (see in supplementary data Excel sheet). These trends show that BitTyrant performance is dependant on how well BitTyrant client's parameters are initialized and how quickly BitTyrant adjusts its variables to maximize its download bandwidth for a minimal upload. Zoning in on these qualities led us to investigate another possible optimization for BitTyrant.

In our presentation we analyzed optimal δ and γ variable fixing the other, however, for a given swarm we wanted to experiment with and find the optimal values for both. These parameters are in the BitTyrant unchoking algorithm and help adjust u_p based on its peers actions. Finding optimal values for these parameters would have significant implications for a BitTyrant client for a specific file size and swarm. Piatek et al. (2008) initializes $\delta = 0.2$ and $\gamma = 0.1$ parameters, stating that smaller values lead to higher performance. Our analysis has confirmed this finding and indicates that even smaller values may produce faster relative completion times for BitTyrant clients for a given swarm. We experimented in a swarm of 1 BitTyrant and 9 BitTorrents for 128 pieces and ran a total of 81 simulations ranging these parameter values from $0.06 \leq \delta \leq 0.14$ and $0.06 \leq \gamma \leq 0.14$ at intervals of 0.01. *Figure 3* shows a graphical representation of our results. The figure shows that the BitTyrant client finish, on average, 24.87 rounds earlier than the BitTorrent clients for a $\delta = 0.06$ and $\gamma = 0.13$. If the BitTyrant client solely cared about minimizing the average round completion, a valiue of $\delta = 0.09$ and $\gamma = 0.09$ provide a minimal average round completion of 94.1 rounds. These values serve as a close approximation for the optimal parameter values for a fixed swarm of 1 BitTyrant and 9 BitTorrents, $u_p = 1$, 128 pieces, 16 blocks per piece, minimum bandwidth = 16, maximum bandwidth = 64 and 40 iterations. Since these simulations have a degree of variance the results only provide a close estimate. These optimal delta and gamma values should calculated and

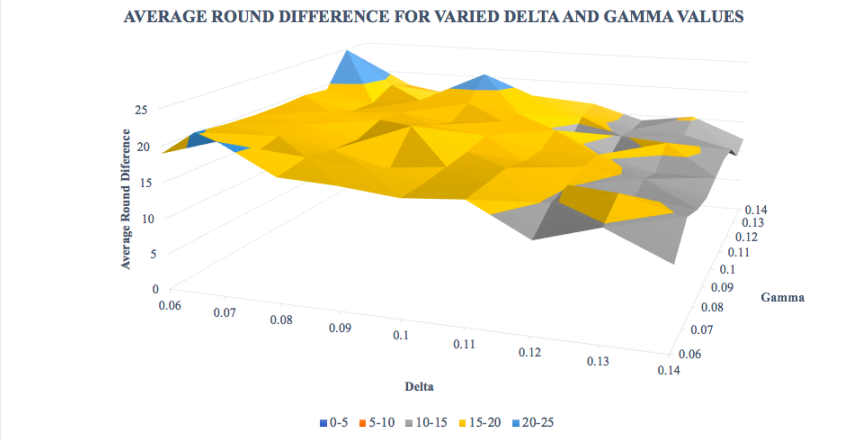

Figure 3: In 1:9 BitTyrant:BitTorrent swarms, optimizing δ and γ values show optimal relative BitTyrant round completion times.

used for a given swarm if a BitTyrant client cares greatly about its performance. This experiment is quite costly to run and due to the multitude of variety in real-world swarms, it may be impractical to calculate the optimal parameter values for a given swarm. However, optimal parameter values do exist and future research could investigate how the BitTyrant algorithm can dynamically collect data and zone in on these values as the rounds progress. Also, one can further analyze how these parameters behave in a variety of swarms and create a rule of thumb of a close estimate to start at. Shout out the computer lab for helping us complete a majority of our simulations.

3.4 Nuances with Rarest First Request

As mentioned above, our initial implementation of rarest first sorted the pieces by rarity then requested each peer in possession of that piece in the order of the sorted sequence. This made it so when pieces had the same rarity the order of requests sent defaulted to the lower pieceId, creating a low variation in request lists for competing clients. We also explored randomizing the piece and peer Id ordering prior to the sorting to ensure a more unique request list for every client.

```
//Initial sort... allPieces = {key = pieceId, value = [peers.id's who have the following
    piece]}
allSortedPieces = dict(sorted(allPieces.items(), key=lambda item: len(item[1])))

//New sort that randomizes piece and peer Id
keys = list(allPieces.keys())
// 1
random.shuffle(keys) // randomizing of piece Id's
allRandomPieces = {}
for key in keys:
    // 2
    newList = allPieces[key]
    random.shuffle(newList) // randomizing of peeer Id's within list
    allRandomPieces[key] = newList
```

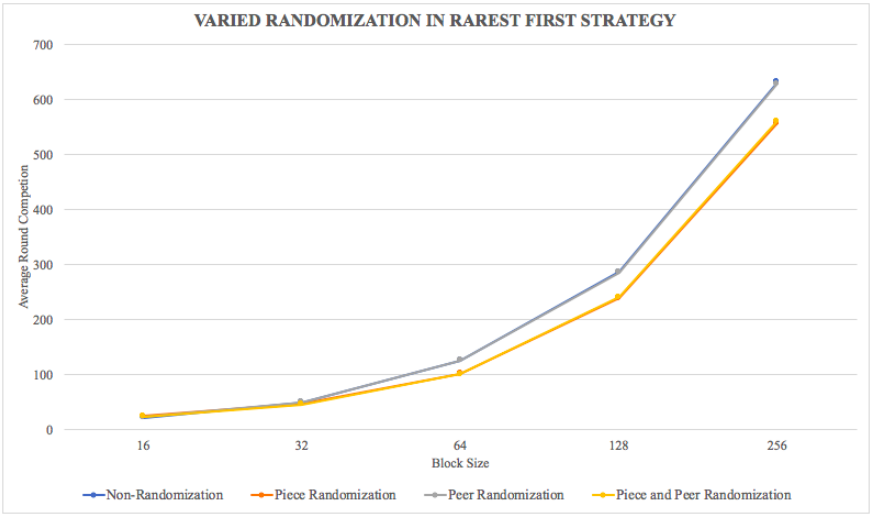

Figure 4: Varied randomization in BitTorrent client rarest first requesting strategy show peer and piece randomization finishing earlier on average for fixed piece size = 32

```
    // 1
    allRandomPieces[key] = allPieces[key]

allSortedPieces = dict(sorted(allRandomPieces.items(), key=lambda item: len(item[1])))
```

The initial sorting algorithm involves no randomization while the second one includes cases for piece randomization and peer randomization. When performing tests on the following strategies we considered the following 4 cases: no randomization, piece randomization only (1), peer randomization only (2), and both piece and peer randomization. We first compared the average round completion of these variations using 10 BitTorrent clients versus themselves while leaving blocksPerPiece fixed at 16 and changing numPieces. One would expect that piece randomization for rarity ties would create a larger more unique pool of pieces in the wild, overall contributing to a faster average round completion, however, we found that all the strategies performed relatively similar when the blocksPerPiece was 16. As we tested these variations against each other fixing numPieces to 32 and increasing blocksPerPiece up to 256 we noticed that the strategies that incorporated piece randomization's finished much earlier in the larger block cases. These tests are shown in *Figure 4*.

This drastic change in behavior seems unexpected, however, we believe it actually intuitively make sense. Since the min and max bandwidth is fixed at 16 and 64 clients are more quick to complete a piece when the number of blocks is lower. This makes it so the rarest piece ordering switched up very quickly and often making the piece randomization optimization almost useless. However, as blocks per piece increase with a fixed min and max bandwidth the randomized piece strategy becomes more effective while the no randomization strategy suffers with all peers requesting for the same pieces. In both cases the peer Id randomization did not contribute to an overall decrease in avg round completion. This specification may be helpful if one wants to interact with more unique peers in their lifetime, however, shows no clear correlation to a decrease in average round completion. Overall, depending on how pieces are divided up into blocks randomizing the

way you request for pieces will decrease you and the overall swarm's average round completion.

3.5 BitTorrent vs. FairTorrent

As FairTorrent does not attempt to minimize average round completion, we implemented this client as an exercise to further understand client behaviors. We investigated and collected data on how they performed against BitTorrent clients and also copmared FairTorrents performance against BitTyrant. As both BitTorrent and FairTorrent's implementations exhibit "fairness" qualities we hypothesized they would have similar average completion times. While running simulations for fixed swarm sizes and varying proportions of FairTorrents to BitTorrents, we saw improvements in contributing peers' download performances. In our tests, we only saw slight improvements, and in our specific small scale tests, we did not reach up to five times better download times as Sherman et al. predicts [3]. We hypothesize that we would see this result for larger scale tests.

3.6 BitExpose

BitExpose is a simple client that we implemented to demonstrate BitTyrant's susceptibility of being taken advantage of. BitExpose simply requests randomly and uploads its max bandwidth randomly to peers. When BitExpose is ran in a swarm of BitTyrants it outperforms them by inflating its $\frac{d_p}{u_p}$ ratio in each of the BitTyrant's algorithm leading to them being constantly unchoked. Do to the length of this report we will leave BitExpose at this. However, you can test and play with BitExpose yourself! There exists cases where BitExpose loses to all BitTorrents, beats out all BitTyrants, and BitTorrent loses to all BitTyrants.

4 Conclusion

Overall, this project has helped us better understand BitTorrent as a whole and learn how to efficiently analyze and use the simulation to better understand how peers interact with each other and identify and incorporate optimizations. Unfortunately, there are quite a few limitations to the analysis one can do with the simulator. The simulator does not model the facts that peers can enter/leave the swarm at will, peers can participate in multiple swarms at once, and lastly our analysis poorly captures the great diversity of BitTorrent clients in the wild. Despite these limitations if we were given more time we would love to try to implement a BitTyrant2 which collects data on past swarm performances and seeks to further identify optimal initialization values for future swarms. This analysis would require a great deal of math, however, it is definitely possible given the theory we have learned in this course over the past semester! Thank you!

5 Acknowledgement

We would like to acknowledge Professor Shikha Singh for her constant help and guidance throughout this research. Her help was greatly appreciated.

References

[1] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72. Berkeley, CA, USA, 2003.

[2] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in bittorrent. In *Proc. of NSDI*, volume 7, 2007.

[3] Alex Sherman, Jason Nieh, and Clifford Stein. Fairtorrent: bringing fairness to peer-to-peer systems. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 133–144, 2009.