UNIVERSITY OF CALIFORNIA
RIVERSIDE

Multiple Identities in BitTorrent Networks

A Project report submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Jin Sun

December 2006

Project Committee:
Dr. Michalis Faloutsos, Chairperson
Dr. Laxmi N. Bhuyan

Copyright by
Jin Sun
2006

The Project report of Jin Sun is approved:

Committee Chairperson

University of California, Riverside

Acknowledgements

First of all I want to express my deep gratitude to my advisor, Professor Michalis Faloutsos for his continuous guidance, encouragement and help during this master project.

Second, I thank my committee member, Professors Laxmi N. Bhuyan for his valuable advice and feedback.

Third, I want to thank my friends and their families both in and out of school whom I have come to know throughout the years. Their friendship has indeed made my graduate life in the U.S. very enjoyable.

Perhaps most important of all, I want to thank my parents and my husband Yu for their unrelenting love and faith in me throughout the years.

ABSTRACT OF THE PROJECT REPORT

Multiple Identities in BitTorrent Networks

by

Jin Sun

Master of Science, Graduate Program in Computer Science
University of California, Riverside, December 2006
Dr. Michalis Faloutsos, Chairperson

Peer-to-peer (P2P) file sharing systems have become ubiquitous these days and at present the BitTorrent (BT) based P2P systems are very popular and successful. It has been argued that this is mostly due to the Tit-For-Tat (TFT) strategy used in BT [1] that discourages free-ride behavior. However, Hale and Patarin [2] identify the weakness of TFT and hypothesize that it is possible to use multiple identities to cheat and the success of BT is due to other reasons, notably the lack of meta-data search. To test whether this hypothesis is true and also to better understand why BT systems are so successful, we modify the official BT source code to allow the creation of multiple processes by one BT client. These processes use different identities and cooperatively download the same file simultaneously. They download different pieces of the same file without overlapping using several

piece selection and sharing algorithms. Experiment results show that these approaches may speed up the download process in a few selected cases. However, no approach can achieve consistent speedup under any case and simply increasing the number of processes may do more harm than help. In addition, we show that the multiple-identity exploit is the most effective when all clients in the network exhibit very selfish behavior. This shows that the BT protocol is rather resilient to such exploits of using multiple identities and it encourages self-regulation among BT clients. We argue that if such exploits were easily achievable, then BT systems would have suffered breakdown long time ago.

Contents

List of Figures ix

1 Introduction 1

2 BitTorrent file sharing system 5

2.1 How BitTorrent works 5

2.2 BT code analysis 7

2.2.1 Network connection handling 8

2.2.2 Data transmissions and rate limiting 9

2.2.3 Piece selection 10

2.2.4 Storage handling 11

3 BT client implementation with multiple identities 12

3.1 Fixed-range piece assignment with no piece-sharing among processes . . . 15

3.2 Fixed-range piece assignment with piece-sharing among processes 17

3.3 Random piece assignment with no piece-sharing among processes 18
3.4 Random piece assignment with piece-sharing among processes 19

4 Detailed implementation 20

5 Experiments and Discussions 23

5.1 Experiment Setup 23
5.2 Experiment Results and Analysis 24
5.2.1 Ranged piece assignment with no piece-sharing among processes . 25
5.3 Random piece assignment with no piece-sharing among processes 31
5.3.1 Approaches that do piece-sharing 34

6 Related Work 36

7 Conclusion and Future work 38

Bibliography 40

List of Figures

3.1 Fixed piece assignment algorithm for different processes 15

5.1 Average download time with different number of seeders in the swarm (upload rate = 20 Kbps) 27

5.2 Average download time with different maximum upload speed in our modified BT client. 28

5.3 Average download time with different number of concurrent processes in our modified BT client. Processes number varies from 2 to 5 29

5.4 Average download time with different number of seeders in a swarm full of free-riders. 31

5.5 Average download time with different number of seeders. 32

5.6 Average download time with different upload speed in our modified BT client. 33

5.7 Average download time with different number of concurrent processes in our modified BT client. 33

Chapter 1

Introduction

Peer-to-peer file sharing systems enable large-scale content distribution, by allowing users to cooperate with each other and voluntarily share their resources, mostly files. There are numerous P2P clients available nowadays, such as KaZaA [3], GNUtella [4], EDonkey [5] and BitTorrent [1]. P2P applications have contributed a significant portion of Internet traffic as indicated by some measurements of the Internet backbone traffic recently [6]. Among all these file-sharing applications, BitTorrent seems to be the most popular one that survives the test of intensive use by numerous users, and it has evolved to account for a large portion of the P2P traffic on the Internet [7].

BitTorrent achieves a higher level of robustness and resource utilization than most currently known cooperative techniques [1]. It works by grouping users, who have common interests in downloading a specific file, together into swarms to cooperate with one another

to speed up the download process. BitTorrent is distinctive for the “choking/unchoking” algorithms that promote high-level reciprocation among users. The “Tit-for-Tat” (TFT) strategy encourages users to cooperate: A such player always uploads more to the peers that reciprocate and provide more data to itself.

Some people believe that the TFT strategy is the key strength for BT’s success. However, Hale and Patarin [2] highlight some weaknesses of the TFT policy and state that it is possible to fake identity in BitTorrent and get free ride because there is no mechanism to provide trusted identification authentication. This can lead to *Sybil Attack* as described in [8, 9]. Instead the authors hypothesize that its success is mostly due to the lack of metadata search and a few other factors.

Although it has been pointed out in [2] that it is possible to use fake peer IDs in BitTorrent networks, there is no detailed description and implementation of this approach in the literature according to our best knowledge. In some sense, it is reasonable for a user to try to get free-rides if possible and the user sometimes would prefer this method for his/her own benefit at the cost of other users in the swarm. Given the fact that there is no monitoring mechanism in P2P networks [10, 11], no statistics are maintained for each client thus the selfish user may easily escape from being traced. As a result, a selfish user may try to find a way to get free ride: Stop uploading file and only download from other users.

In addition to the tendency to get free-riding, a user may also wish to use multiple identities to speed up the download process. For example, the user may have two different

IPs and want to create two BT processes that can work cooperatively to download the same file without any overlapping. This is especially attractive when the file to download is very large. However, this is not currently supported by the official BT client and a few other compatible implementations. It is interesting to note that this seems to be a legitimate motive as long as these individual processes still conform to the BT protocol. However, it is unclear whether this can indeed speed up the download process. Even if so, how can these processes impact the experience of other users is still unknown.

Motivated by these observations, we have developed and implemented a modified BT client based on the official BT source code. With the modified BT client, a user can create multiple processes with different IDs and these processes can cooperatively download the same file using different piece selection and sharing algorithms. According to our best knowledge, this is the first implementation that uses the idea of multiple identities.

We have done extensive experiments with the modified BT client in the CS lab of UC Riverside. From these experiments, we find that the modified client can only achieve limited speedup under a few selected cases even though we have tried various combinations of strategies that we can think of. It is only the most effective when all the clients are very selfish and can download from seeders only who have the complete file because they do not upload anything to others. In this case, even though the speedup is significant for the modified client in comparison with other normal clients, the overall performance of the system degrades a lot so this will not happen often in reality because everyone loses by

cheating. Overall, there is no strategy that the modified client can deploy to achieve consistent speedup. Given the fact that the real-world BT system environments are much more complex and it would require users a lot of time tinkering with the various settings without guarantee of success, using multiple identities cannot help selfish users gain significant advantages over others. This shows that the BT protocol is rather resilient to such exploits. It curbs selfish behavior and encourages users to self-regulate for faster download speed. This prevents the breakdown of BT systems very effectively.

The rest of the report is organized as follows. In Chapter 2, we provide some more background information on BitTorrent systems and detailed code analysis of the official BT client implementation. At the time of writing, there is no detailed documentation on BT code organization and we believe this will be helpful for people who are interested in understanding and extending the BT client for research and experimentation. In Chapter 3, we describe how to create multiple processes with different IDs to download the same file cooperatively and propose a few different piece selection and sharing algorithms. In Chapter 4 we describe the detailed changes made to the official BT client implementation. In Chapter 5, we describe our experiments with the modified client with these algorithms in the CS lab of UC Riverside and analyze the results. In Section 6 we describe some related work. At last in Chapter 7, we conclude this report with some directions for future work.

Chapter 2

BitTorrent file sharing system

Before describing our modifications to the BT client, it is necessary to give some more details on the BitTorrent protocol.

2.1 How BitTorrent works

BitTorrent achieves efficient content distribution by swarm download. The basic idea is to split a file into equal-size *pieces* and have clients download pieces from different peers simultaneously [1]. Each piece is further split into *blocks*, which is the basic transmission unit in BitTorrent. Each node sends a request for each block it does not have. To avoid delay in piece transmissions, BT clients always have some requests pending, typically five. Each peer advertises what pieces it has by advertising *BITFIELD* or *HAVE* messages to

other connecting peers.

To begin downloading a target file, a user needs to find a *torrent* file which contains enough information about the file to download, such as file length, hash values of each piece, and the URL of a tracker. A tracker is a centralized machine that keeps track of all the peers participating in a swarm. Active peers report their status once a while (e.g. 30 minutes) including the total amount of data they have downloaded and uploaded.

Client finds other peers to download/upload by sending requests for peer lists to the tracker. Once a client gets the peer list returned by the tracker, it tries to contact them and download the pieces it needs. At the same time it also uploads the pieces it has to others. Cooperation among peers shifts the burden of content distribution to the entire swarm. Even with many simultaneous downloads, the upload burden on the central server remains quite small, since each new downloader introduces new upload capacity.

In the BitTorrent protocol, individual peer is usually identified by a tuple (IP, Port, Peer ID). Peer ID is a 20-byte string which contains two parts. The first 6–8 bytes indicate the BT client used and its version. For example, "M4-0-4–" represents Bram Cohen's original BT client version 4.0.4. The second part is a random number that varies over time. Since peer IDs can be chosen rather arbitrarily by peers themselves, they cannot be used for authentication purpose.

In addition, in the current implementation of BT tracker, it cannot differentiate legitimate IDs from faked IDs, because its function is mostly to facilitate peer-to-peer connec-

tions. So it is possible to use multiple peer IDs on behalf of a user without being detected by either the tracker or other peers.

It should also be noted that peers cannot be differentiated by IP addresses alone in BitTorrent networks for two reasons. One is that BitTorrent supports multiple connections behind *Network Address Translator* (NAT), which means that several peers behind the same NAT can have the same IP address. The second reason is that connections from behind a *proxy* are also accepted in BitTorrent.

In summary, inability to identify a unique peer in BitTorrent networks cannot prevent a user from launching multiple processes simultaneously to cooperate with each other on the behalf of the same user. The question is whether users can take advantage of this seeming weakness.

2.2 BT code analysis

The official BitTorrent client from Bram Cohen is open-source written in Python, and it provides implementation for both BT client and tracker. At the time of writing, there is no detailed document on its implementation. Because of this, we provide descriptions of some key libraries and classes implemented in the official BT client 4.0.4 to help people better understand our extensions later.

Here we focus on the command-line interface provided by BT client only (*btdownload-*

headless.py).

In *btdownloadheadless.py*, class *HeadlessDisplayer* is used to print some run-time statistics to standard output to notify user of the progress and class *DL* is used to drive the download process.

Class *DL* calls several other classes in *download.py* through its member function *run* to join the torrent and start downloading/uploading. These classes can be divided into several categories.

2.2.1 Network connection handling

A BT client needs to initiate connections to the tracker and its peers. The communication between BT clients and tracker is through HTTP and the duration of connection is relatively short. When a BT client finishes downloading and before it is closed by an end-user, it serves as a seeder and uses upload rate to decide which peers to upload more instead.

Class *RawServer* (defined in *RawServer.py*) is one of the most important classes. It multiplexes IOs (for both server and client sockets) and is responsible for closing timed-out sockets. It avoids using multiple threads or multiple processes and thus is much more efficient. Class *SingleSocket* is a simple wrapper class around the Python socket library and is also defined in *RawServer.py*. It is used to handle data in the buffer and send data through the socket. In BT client, other socket-related classes depend on the services provided by both *RawServer* and *SingleSocket* to focus on data transmissions and receptions without

being bogged down to network connection handling details.

2.2.2 Data transmissions and rate limiting

With the services provided by *RawServer*, several classes implement the BT protocol itself, handling peer-to-peer and peer-to-tracker information exchange and rate limiting. These classes include

- *Connection*

  Defined in *Connecter.py* and is used to handle BT protocol handshake between peers. Its member function *data_came_in* is called each time the socket receives data from other peers. Function *_read_message* is used to analyze each field in the received messages. A *Connection* object is created for each peer and tracker.

- *Rerequester*

  Defined in *Rerequester.py* and is used to handle BT protocol handshake between peer and tracker, e.g., request for peer list, updating its own status and handling peer list.

- *Upload*

  Defined in *Uploader.py* and is responsible for uploading pieces/blocks to peers.

- *SingleDownload*

Defined in *Downloader.py* and is responsible for downloading from peers. A *SingleDownload* object is also created for each connecting peer. It maintains a list of all the active requests it sends out, keeps track of download rate from peers and checks if remote peer has any pieces local client is interested in. Its member functions respond to different messages received from peers and then act accordingly.

- *Choker*

  Defined in *Choker.py* and is used to decide which peers to choke or unchoke. The "TFT" algorithm is implemented here.

2.2.3 Piece selection

Though this is related to data transmissions, we list it separately because it is the part upon which we make most of our changes. There are several related classes here:

- *Bitfield*

  Defined in *bitfield.py*. This is a helper class that provides easy access to bit strings that identify which pieces a peer already has and which pieces it needs to get.

- *PiecePicker*

  Defined in *PiecePicker.py*. This implements the piece selection algorithms, e.g., the rarest first algorithm and the random first algorithm. It also keeps record of which blocks in a specific piece the client has and needs to request.

In fact, we can modify the class such that a BT client only requests the pieces that we need, for example, the first half of the pieces for a file.

2.2.4 Storage handling

The *Storage* class (defined in *Storage.py*) provides lower-level functions to open, read and write files and pieces. The *StorageWrapper* class (defined in *StorageWrapper.py*) provides higher-level functions to read and write BT pieces. For example, it is possible to read or write a piece with a specific index through *StorageWrapper*.

Besides, *StorageWrapper* also ensures that pieces are assembled in order when it writes the file even though these pieces can be received out of order.

Chapter 3

BT client implementation with multiple identities

In this chapter, we describe the changes we have made to the original BT client implementation. These changes enable the creation of N BT processes (with N different IDs) for a user to download one file cooperatively.

The core idea is that the N concurrent BT processes coordinate with one another. If one process has already downloaded some pieces, then the other processes do not need to request and download the same pieces again. We will elaborate on the different approaches of assigning individual pieces to each process for downloading shortly.

When starting to download a file, a user can specify the total number of processes to be created. If N processes are created, then the download task can be shared by these

N processes. Each process has a unique Peer ID which is generated by the BT client automatically and used to contact tracker independently.

Because in the current implementation the tracker does not differentiate between several connections from one IP address by default, each process can register with the tracker successfully and request for a peer list. Given the fact that tracker randomly picks a number of peers (usually 50) from its peer database and returns a list of them to every peer, it can be expected that the chance for each process to get the same peer list is very small, especially for a large and popular swarm, although we can expect some overlaps.

Once a process receives the peer list from the tracker, it acts as a normal BT client to ensure that it will not be banned by the tracker or peers. It initiates connections to or accept connections from peers using its unique peer ID according to the BT protocol. For other peers, these processes do not exhibit any abnormal behavior other than that they have the same IP address. As long as the remote peer does not connect to more than one processes at the same time, it can treat these processes the same way as other peers.

Our approach differs in piece selection when a process sends a new request to its peers: It only chooses a piece from its own "task list", instead of from the entire piece list. By splitting the target file into several parts and assigning different parts to different processes, we cut down the actual download size by N for individual process. And we can expect that this approach may reduce the overall download time for one file though later we will see some BT dynamics prevent this from happening too easily.

There can also be some variations of the basic scheme. For example, rather than creating fixed "task lists" for individual processes, it is also possible for these processes to exchange information among themselves and decide which pieces to download next by skipping those pieces that have been downloaded or are being downloaded by other processes. Another variation is whether these processes can upload to their peers those pieces that have been downloaded or are being downloaded by other processes.

It should also be noted that we cannot create an arbitrary number of processes without limit for several reasons. First, the download capacity (bandwidth) is a limited resource and shared by all the processes concurrently if they run on the same machine. When the total number of processes increases, the download bandwidth for each process decreases, thus the download time will be affected. This can be more conspicuous when both upload and download traffic are multiplexed on the same physical link and the download traffic is also affected by the upload traffic. Second, BitTorrent protocol is built on TCP and all the processes have to share the same TCP buffer as is the case in today's most operating systems. Because TCP buffer size is usually fixed and cannot be modified at will, too many concurrent network connections would degrade TCP performance. Third, TCP flow control might also affect the performance when too many connections are established.

Following we describe the four different algorithms for piece assignment and sharing among concurrent processes in details and outline our implementation.

Algorithm: $Piece_Assignment$

1: User generates N processes to download
2: M = total number of pieces
3: X, Y = divmod(M, N)
4: **for** i in $(0\ to\ N-1)$:
5: **if** $Y \neq 0$:
6: Assign Piece$[X*i, X*(i+1)]$ to Process i
7: $Y = Y - 1$
8: **else**:
9: Assign Piece$[X*i, X*(i+1)-1]$ to Process i

Figure 3.1: Fixed piece assignment algorithm for different processes

3.1 Fixed-range piece assignment with no piece-sharing among processes

This is the most straightforward approach to share the download task among concurrent processes. The file to be downloaded (the target file) is divided evenly into N parts without any overlap. Each process is assigned a part, which is the "task list" that needs to be accomplished. Our algorithm for picking a piece within a specific range is shown in Fig. 3.1.

For example, suppose that the target file contains 101 pieces in total, and the user launches two processes to download the same file together. The file is divided in two parts: Pieces 0–50 and pieces 51–100. Process 1 will be assigned pieces 0–50 to download and process 2 pieces 51–100. Such assignment is done automatically by the program without user intervention other than specifying the number of processes and the process IDs when

invoking the BT client.

When a process sends a new request to its peers, it only chooses a piece from its own "task list", instead of from the entire piece list. By splitting the target file into several parts and assigning different parts to different processes, we cut down the actual download size by N for a single process. This approach is simple and incurs the least overhead and these processes can run on different machines.

However, in this simple approach, there is no information exchange between different processes. That is, after successfully downloading a single piece, each process will not notify other processes about this and neither will it copy this piece to others. Therefore, there is no piece sharing among processes. When the modified process is to send out a *HAVE* message, it will only announce the piece that has been downloaded by itself, and cannot report the availability of other pieces outside its "task-list". Thus it cannot upload any piece downloaded by other processes to its peers. Therefore, one possible problem with this approach is that each single process may have higher probability of being choked by others, because it can only offer a subset of the file pieces that others might want.

3.2 Fixed-range piece assignment with piece-sharing among processes

In the second approach, when generating a new request, the N concurrent processes still use the same ranged piece assignment algorithm as shown in Fig. 3.1, however it differs in that these processes also try to coordinate with one another. They exchange information among themselves about what pieces have been downloaded as a whole. That is, when each process announces what pieces they have downloaded, it will also include those pieces that have been downloaded by other processes. So each modified process is also able to upload pieces downloaded by other processes. This may help to reduce the chance of these modified processes being choked by others because they can offer to upload more pieces.

To share pieces among these different processes, there are two ways. One way is to set up dedicated connections among these processes and then they transfer the actual pieces they have downloaded to one another. However this incurs more communication overhead and implementation complexity. The other way to to have all the processes run on the same machine and each process can copy the pieces it has downloaded as individual files to a common directory. The common directory is accessible to all the processes so each process can check the pieces other processes have downloaded and upload these pieces to its peers even though they are not downloaded by itself.

One caveat with this approach is that since these processes need to run on the same

machine, they may be prevented from connecting to the same peer and they also need to share the available download and upload bandwidth and the available TCP buffer. However, because of its simplicity we choose to use this approach in our implementation.

3.3 Random piece assignment with no piece-sharing among processes

The third approach differs in its *random* piece selection algorithm: Whenever one process is about to find one piece to request, it no longer chooses from the fixed subset of pieces like the two aforementioned approaches. Instead, it will randomly pick one among all the pieces that are not requested/downloaded by any other processes. Each process needs to request the piece which the BitTorrent protocol returns (surely that piece needs to be among those that other connecting peers can provide), otherwise it may lose the opportunity to get pieces from others if they stick to the fixed piece assignment discussed earlier. When they generate less requests to other peers, they may not make the best use of the available download bandwidth.

Obviously it is necessary for each process to be aware of what pieces other processes have already had. Each process needs to check (via file) what pieces the other processes are downloading and avoids downloading the same ones. Therefore we can use the same approach by copying every piece to a common directory as individual file (e.g., *piece0*,

piece1, ...) and then any process needs to check this directory first before generating a new request.

Similar to the previous approach, this simplifies implementation although it requires all processes run on the same machine.

3.4 Random piece assignment with piece-sharing among processes

In this approach, the only difference with the previous one is that each process also checks availability of the pieces that are downloaded by other processes and announces *HAVE* messages to its peers. So each process can also upload to its peers pieces downloaded by other processes and may reduce the possibility of being choked by its peers.

Chapter 4

Detailed implementation

As discussed earlier, we use the official BT client 4.0.4 as the starting point to make our changes. We are aware that newer versions of BT client keep coming out. However, the underlying protocol remains unchanged so we can continue to use this version. To ensure compatibility with the original BT client, we have made changes such that the new BT client just behaves as the original one when invoked with the default arguments. Our changes are limited to a few files.

First in *defaultargs.py* we add a few more options. One is to specify the process ID that the BT client runs with. Second is the number of processes that will be created. In this way, each process knows which pieces it needs to download for fixed-range piece assignment approach. Third is to specify the location of log file which we can save the times when download starts and ends. The fourth option specifies where to save individual

pieces. The fifth option specifies whether processes upload pieces downloaded by other processes or not. The six option specifies whether fixed-piece selection or random-piece selection algorithm is used.

Then we derive a subclass *RangedPiecePicker* (defined in a new file *rangedpiecepicker.py*) from *PiecePicker* to override the default piece selection algorithm.

For the fixed-range piece selection case, each process can determine the piece range based on its process ID and the number of total processes created.

For the random-piece selection case, each process needs to check the common directory to see which pieces have been downloaded or are being downloaded by other pieces and will not consider them any more. Once the algorithm chooses a piece that is not requested yet, it will create the piece file exclusively to let other processes know that it will request to download this piece.

In either case, each process announces its interest in these pieces to its peers (through *INTERESTED* messages) and download/upload these pieces like normal BT clients.

When these processes finish downloading their assigned pieces, they will regard this as task completed and leave the swarm. This is different from the original BT client in that the latter will stay in the swarm for some time to act as seeder for other peers. It should be noted that these processes are perfectly legal to leave the swarm. After all, each process indeed does not have the complete file and it should not act as seeders. Such changes are made in files *download.py* and *Downloader.py*.

However, this is not the end of the whole process. When these processes leave the torrent, the pieces they have downloaded may be saved out of order or other pieces may be filled with zeros. So it is important for each process to order its pieces correctly and then write the part it takes charge of to disk. Then a separate process can assemble these parts correctly. Such changes are made in the file *StorageWrapper.py*. In the case when a common directory is used, it is also possible to skip the reordering part and just concatenate all the pieces together to get the complete file.

We will soon make our source code and commentary available through our web site to benefit the research community.

Chapter 5

Experiments and Discussions

To validate our methodology, for each of the aforementioned approaches, we conduct experiments in a swarm created in the CS lab of UC Riverside. In all experiments, we run both the modified clients and the original BT clients side by side and compare their performance. In this chapter, we describe our experiment setup and present the results with analysis.

5.1 Experiment Setup

Experiments in UCR intranet are conducted in a more controllable manner, so they can help us to assess the impact of different approaches precisely. We deploy 40 nodes to download a 30MB target file. In the swarm, there is one tracker and multiple seeders that vary from 1

to 6. Only one leecher runs our modified client (which creates actually multiple processes as described in Chapter 3), other leechers and seeders use the original BT client. We use BitTorrent's default parameter settings for all nodes except the following

- *Node Degree*

  This parameter determines the neighborhood size, that is, the number of concurrent connections a node will maintain. In our experiments, node degree is set to 10.

- *Maximum upload rate*

  We only change it for the modified client. For normal clients and seeders, this parameter defaults to 20 Kbps. We will vary the maximum upload speed of modified client in specific experiments, as noted in later chapters.

- *Number of processes created*: This parameter applies to the modified BT client only. One modified BT client can create 2–5 processes to download the same file cooperatively.

5.2 Experiment Results and Analysis

Although we have experimented with four different approaches, we first show the results for the two approaches that do not do piece-sharing and then later we discuss the results for the approaches that use piece-sharing.

5.2.1 Ranged piece assignment with no piece-sharing among processes

To evaluate this approach, we compares the performance of our modified client with the original BT client in terms of download completion time by varying the number of seeders in the swarm, maximum upload speed for the modified client and the number of concurrent processes in the modified client respectively. The results are shown in Figs. 5.1, 5.2 and 5.3 respectively.

Fig. 5.1 compares the download time between normal BT clients and the modified BT client. The latter uses two concurrent processes to download the file cooperatively. We vary the number of seeders to see how it can affect the download speed. In this set of experiments, both normal BT clients and the modified BT client have the same default upload speed of 20Kbps.

This graph clearly shows that when there is only one seeder in the swarm, the modified client can hardly download faster than the original client. When the number of seeders increases to 2, there is a significant speedup achieved by the modified client. However, when the number of seeders increases further, the speedup is more level for the modified client.

It is easy to explain the speedup for the original client because more seeders means more upload capability offered to the swarm. For the modified client, the case is a bit more complicated. When there is only one seeder in the swarm, the effect of the rarest-first

piece selection algorithm is the most conspicuous: The availability of any piece is mostly limited by the upload capability of the seeder which is only 20 Kbps. Most of the time the two processes that are created by the modified client are blocked by the availability of the requested pieces and they in fact spend much time waiting for the pieces to become available in the swarm.

When there are more than one seeders in the swarm, the modified client can achieve much higher speedup than the original client. The reason is that the availability of the pieces is much less constrained by the upload capacity of the seeders and that each process only needs to download one half of the file so the download time is much shorter.

When more seeders are available, they do not help the modified client much because the upload capacity of seeders are shared by both original clients and modified client and increasing one seeder does not increase the capacity much.

Because modified client benefits from more than one seeders as discussed above, in later experiments we focus on the case when there are three seeders in the swarm.

In Fig. 5.2, we show the effect of varying upload speed for the modified client. It can be observed that for the modified client the download time curve follows three stages. When the modified client decreases its upload speed below the norm, its download time increases even though each process only needs to download only one half of the file. The reason is that these processes can offer only half of the pieces that other peers are interested in and they may be choked more often by other peers. When they are unwilling to upload more

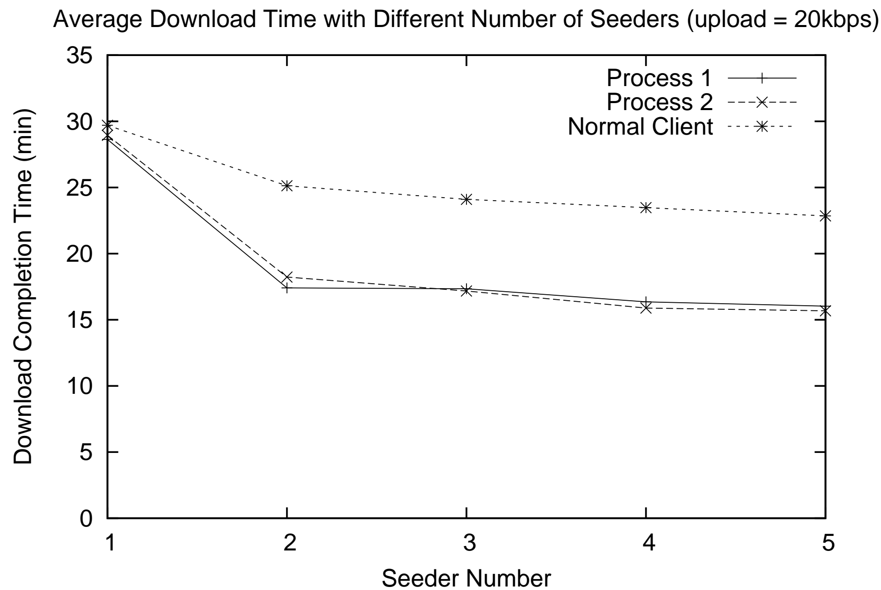

Figure 5.1: Average download time with different number of seeders in the swarm (upload rate = 20 Kbps)

data to other peers, then they are punished more severely. So it is no good for the modified client to decrease its upload speed below the norm.

When modified client increases its upload rate, the benefit of the need to download only one half of the file is more conspicuous so the download time decreases sharply. This also partially compensates for the deficiency of the limited availability of pieces from these processes. They can increase their upload rate to help reduce the possibility of being choked by others. However, further increasing the upload rate will not help much beyond a certain point because by that time these processes cannot necessarily achieve the maximum upload rate as they have limited pieces to offer and other peers will not request pieces from them so often.

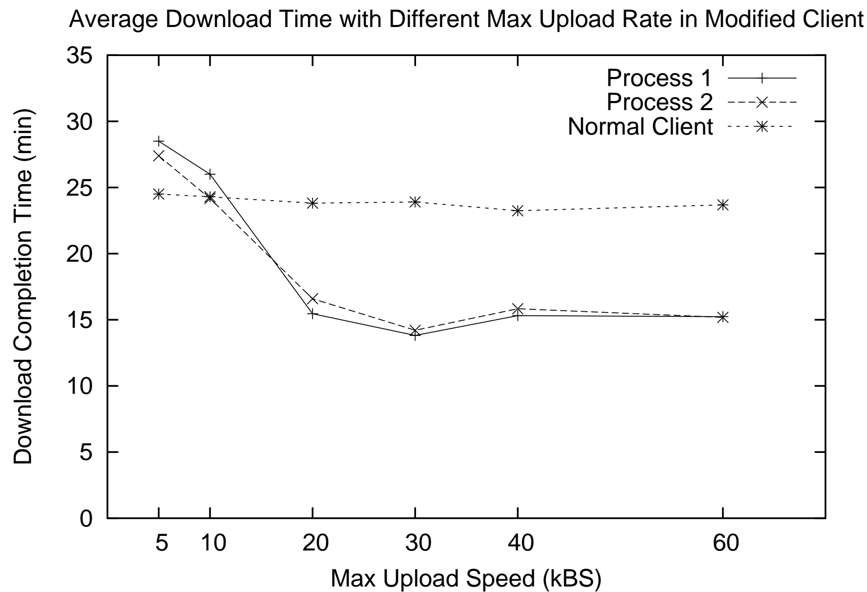

Figure 5.2: Average download time with different maximum upload speed in our modified BT client.

It is also interesting to notice that increasing upload rate may negatively affect the download speed as shown from the slight tip at 40 Kbps. This is due to the reason that these processes have their upload and download traffic multiplexed and too much upload traffic can reduce their download speed and seeders may also reduce upload rate to these processes further.

This shows that the modified client should not cheat by decreasing upload rate below the norm and it should not be too generous either. This also demonstrates BT protocol's resilience to cheating and everyone benefits by being honest and reciprocative.

It can also be noted that the download times for two processes are not always equal, but the difference between them is always less than 5%. This is as expected because their

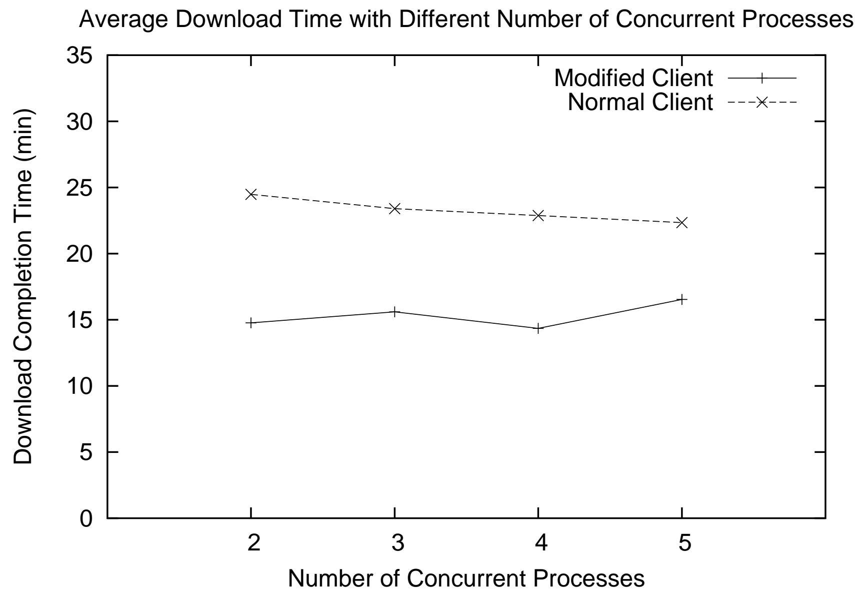

Figure 5.3: Average download time with different number of concurrent processes in our modified BT client. Processes number varies from 2 to 5

workload is almost identical and the difference is mostly due to the variation of the pieces that are available.

Fig. 5.3 compares the results when we vary the number of processes created by the modified client. As usual, here the number of seeders is fixed to three.

We can see that for normal clients, the download time decreases slightly with the increase of processes created by the modified client. This is easy to explain because with more processes participating in the swarm, they offer more upload capacity and normal clients can benefit from this.

However, for the modified client the benefit is not so clear-cut. Increasing processes reduces the workload of individual processes, however this also reduce the number of pieces

that these processes can offer to others in exchange for higher download speed. This also increases the possibility that these processes are choked by others. So it is not to the modified client's advantage to increase the number of processes arbitrarily and creating two processes has already helped a lot.

An interesting extension of this set of experiments is to simulate possible greedy behavior of users. In this scenario, all the peers do not upload data to other peers and therefore can only download from seeders. This situation can reflect the performance of BitTorrent system in a bad swarm which is full of free-riders. To achieve this, we simply set the maximum upload speed to 0 for every leecher.

In this set of experiments no one in the swarm uploads data to others except seeders. Therefore, everyone downloads from the seeders only and the choking/unchoking algorithm has no effect on the leechers. From the results shown in Fig. 5.4, we can see that the modified client achieves better performance. When the number of seeders increases, the average download time for both clients also decreases. However when running two concurrent processes, the modified client outperforms the normal client by cutting down the completion time by about 50%. This shows clearly that if many leechers in a swarm adopt selfish behavior, the modified client will have a much better chance of achieving better performance and that can hurt these selfish leechers.

It should also be noted that even though the modified clients can download faster than other clients, the overall download time for all the clients in the swarm increases a lot and

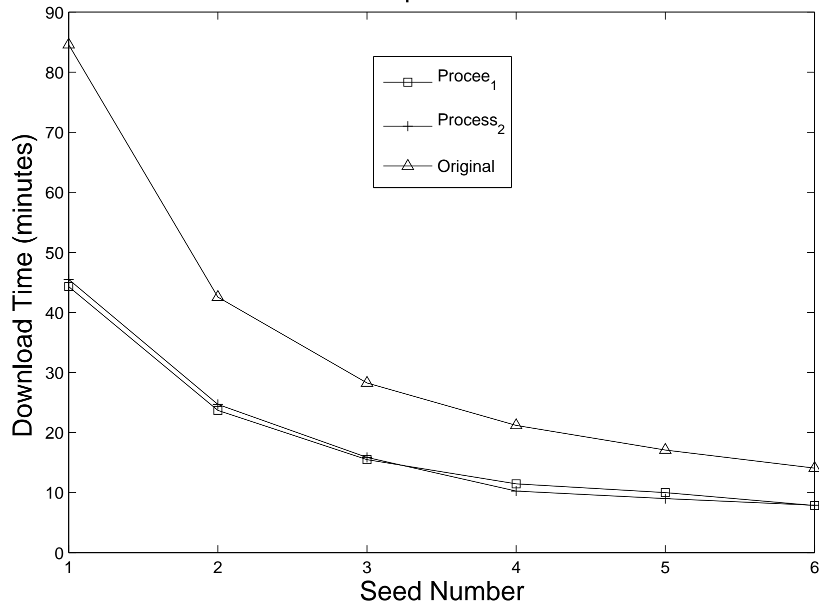

Figure 5.4: Average download time with different number of seeders in a swarm full of free-riders.

every one suffers. In reality, users have to behave to avoid the slowdown. So BT protocol effectively curbs selfish behavior and encourages self-regulation among users.

5.3 Random piece assignment with no piece-sharing among processes

In this approach, each process does not stick to a fixed-range of pieces to download, instead it randomly picks one piece that are not requested by other processes. Please note that in this approach the processes created by the modified client need to run on the same machine

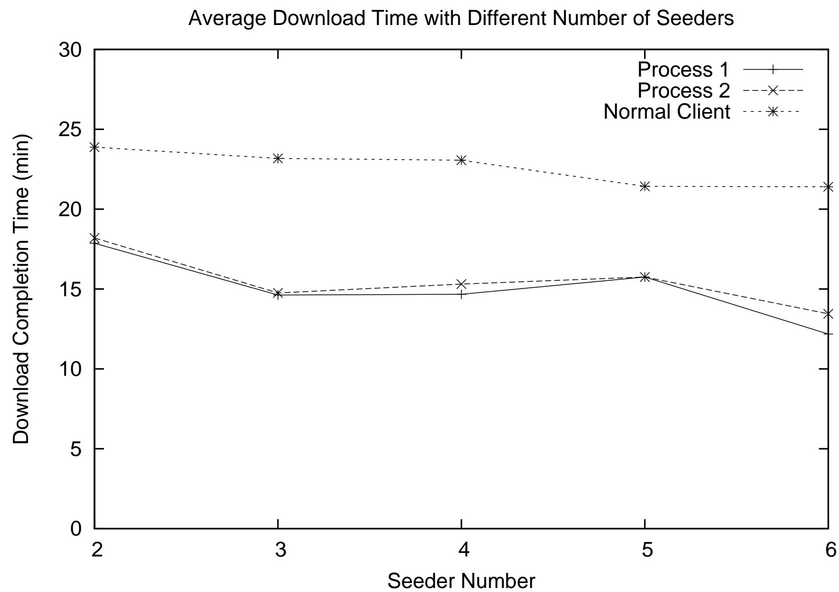

Figure 5.5: Average download time with different number of seeders.

and have to share the available download and upload bandwidth.

To evaluate this approach, we also vary the number of seeders, maximum upload speed for the modified client and the number of concurrent processes and compare the download time between the modified client and the original BT client. The results are shown in Figs. 5.5, 5.6 and 5.7.

Comparing Fig. 5.5 with Fig. 5.1, we can observe that the download time varies a lot more in this approach. The main reason is that the load for the two processes may be uneven: One process may download more pieces than the other process because once it starts downloading the rarest piece according to BT's piece selection algorithm, the other cannot request it and have to wait for the next available one. The other process may be choked more often by other peers. However, when there are more seeders in the swarm,

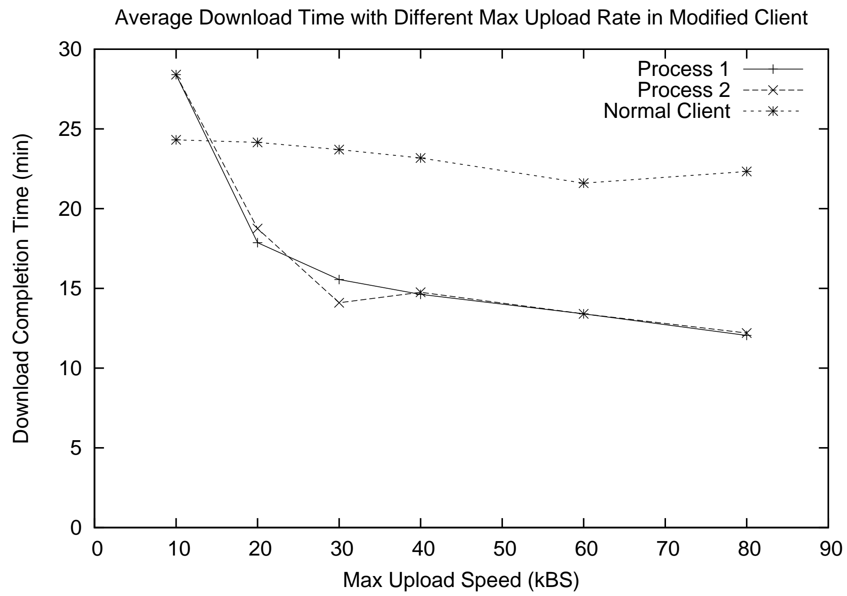

Figure 5.6: Average download time with different upload speed in our modified BT client.

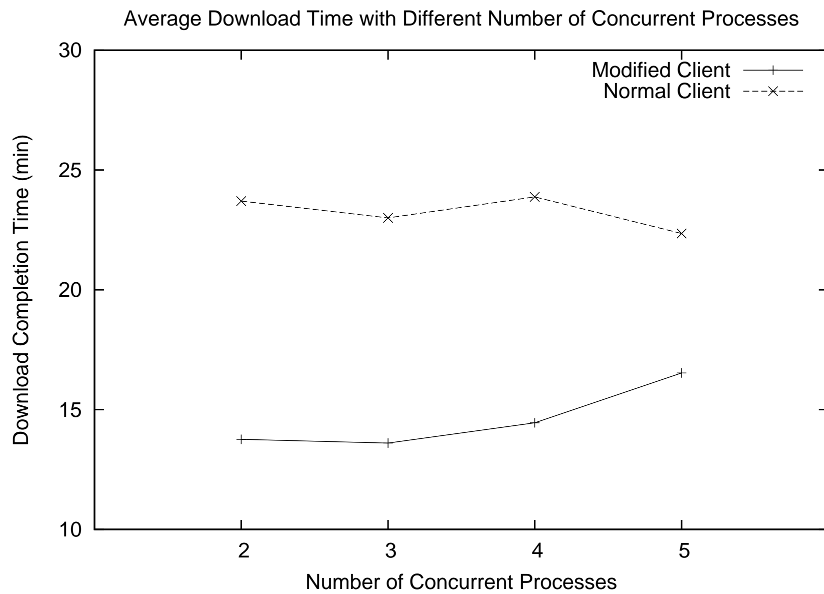

Figure 5.7: Average download time with different number of concurrent processes in our modified BT client.

then the other process may still benefit a lot from the increased upload capability offered by these seeders and the download time is shortened.

Fig. 5.6 shows that this approach benefits more by increasing the upload rate in comparison with Fig. 5.2. This can be explained as follows. Because these two processes may have uneven load, one process that has less pieces to offer may need to increase its upload rate to offer more data to its peers to reduce the possibility of being choked by others.

Fig. 5.7 shows that with the default upload rate, the download time cannot be reduced by increasing the number of processes. This is easy to explain because when there are more processes, the load of each process is even less evenly distributed and usually one process may shoulder most of the load and other processes have little to download and hence cannot contribute much to the download task.

5.3.1 Approaches that do piece-sharing

Earlier we have shown the results for the approaches that do not do piece-sharing. We have also experimented with approaches that do piece-sharing. That is, for each process that is created by the modified client, it also uploads to peers pieces that are downloaded by other processes. However, these piece-sharing approaches also yield only satisfactory results under a few cases, for example, when there are more seeders in the network or when maximum upload rate is increased. The speedup is limited due to the following reason. Although these processes may upload more pieces to their peers, this does not increase the

number of pieces that are available to themselves even though their peers are willing to reciprocate with more pieces. In this case, their upload and download rate are unbalanced and they end up with faster uploading but without faster download.

To summarize, there is no easy way to achieve consistent speedup under any case by using multiple identities with either piece selection and sharing algorithm. The TFT strategy, optimistic unchoking and rarest-piece selection algorithm are quite resilient to such potential exploits. It is the BT protocol's technical strength that makes it difficult to cheat in such systems, otherwise BT systems would have suffered breakdown long time ago.

Chapter 6

Related Work

The success of BitTorrent file sharing system has sparked great interest in studying its dynamics and possible exploits of such a system in the recent years.

The BitTorrent creator Bram Cohen [1] describes the Tit-for-Tat (TFT) strategy, the optimistic unchoking and rarest-first piece selection algorithm in detail. Since then, quite a few studies have been done to evaluate BT systems specifically.

In one of the most comprehensive studies, Pouwelse et al. [12] study the BitTorrent/Suprnova system in detail which was a defunct system that were mirrored globally and provided comprehensive lists of files to share. Their measurements and analysis are very comprehensive in that they study the availability, integrity, flashcrowds and download performance in a total system perspective, including site mirroring, torrent file distribution and maintenance, tracker, seeder and leecher uptime. They show that performance is influenced by global

components of the system and argue that decentralization is very important. The authors also show that the arrival process is not Poisson and oftentimes exhibits flashcrowd effect. This calls for more work on BT workload characterization. Since the authors have also published all the data online anomynized for further study [13], it may be possible to use the data to generate more realistic workload and evaluate various modifications to the BT protocol.

Liogkas et al. [14] proposes three exploits that selfish leechers may use to cut their download time. The first one is to download from seeders only. The second one is to download from the fastest leechers. The third one is to advertise false pieces to peers to cheat them into uploading more data to themselves. With these three exploits, the authors show that selfish leechers may get some advantages but cannot actually degrade the robustness of the BT system even though they may violate the fairness property of the system to a certain degree. The authors contribute this mostly to the TFT strategy, optimistic unchoking and rarest-first piece selection algorithm.

Our multiple process approach is orthogonal to the exploits proposed by Liogkas et al. and may combine with the first two exploits. However, the third exploit is generally infeasible: In addition to the Azureus BitTorrent client [15], the official BitTorrent client also downloads blocks belonging to the same piece from the same peer. So using the third exploits can easily get the selfish leechers black-listed.

Chapter 7

Conclusion and Future work

In this report, we have described the seeming weakness in BT systems, i.e., the possibility of using multiple identities to cheat which was first identified by Hale and Patarin [2]. However, there was no implementation that exploit such weakness. Then we describe our modification to the official BT client implementation to allow the creation of multiple processes in one BT client to download the same file cooperatively. Our extensive experiments with different piece selection and sharing algorithms show that it is possible to achieve speedup in a few selected cases including increasing the number of seeders and choosing a moderate maximum upload rate. However, no strategy can achieve consistent speedup in any case. In some cases, increasing the number of processes may even hurt the performance of the modified client. The modified BT client is only the most effective when all clients in the network exhibit very selfish behavior, however then the overall download speed of the

network degrades a lot and in reality users will not cheat because they cannot achieve any gain by doing this. This shows that the BT protocol is rather resilient to the exploit of using multiple identities. We argue that if such exploits were easily achievable, then BT systems would have suffered breakdown long time ago.

In our future work, we plan to augment the multiple identities approach with the strategies proposed by Liogkas et al. [14]. We want to explore whether it is possible to exploit the BT system further with more complex strategies and if so any counter-measures can be devised.

Bibliography

[1] B. Cohen, "Incentives Build Robustness in BiT Torrent," http://www.bittorrent.com/bittorrentecon.pdf.

[2] D. Hale and S. Patarin, "How to cheat BitTorrent and why nobody does," in *Technical Report UBLCS 05/12/05*, 2005.

[3] J. Liang, R. Kumar, Y. Xi, and K. K. Ross, "Pollution in P2P File Sharing Systems," in *IEEE INFOCOM*, 2005.

[4] "GNUtella," http://en.wikipedia.org/wiki/Gnutella.

[5] "EDonkey," http://en.wikipedia.org/wiki/EDonkey2000.

[6] T. Karagiannis, A. Broido, N. Brownlee, and M. Faloutsos, "Is P2P dying or just hiding?" in *Globecom, Dallas, TX, USA*, 2004.

[7] T. Mennecke, "BitTorrent Remains Powerhouse Network," in *Slyck News*, 2005.

[8] J. R. Douceur, "The Sybil Attack," in *The first International Workshop on Peer-to-Peer Systems(IPTPS'02)*, 2002.

[9] J. Newsome, E. Shi, D. Song, and A. Perrig, "The Sybil Attack in Sensor Networks: Analysis and Defenses," in *(IPSN'04)*, 2004.

[10] C. Buragohain, D. Agrawal, and S. Suri, "A Game-Theoretic Framework for Incentives in P2P Systems," in *International Conference on Peer-to-Peer Computing*, 2003.

[11] P. Golle, K. Leyton-Brown, I. Mironov, and M. Lillibridge, "Incentives For Sharing in Peer-to-Peer Networks," in *Proceedings of the 3rd ACM conference on Electronic Commerce*, 2001.

[12] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The Bittorrent P2P File-sharing System: Measurements and Analysis," in *International Workshop for Peer-to-Peer Systems (IPTPS) 2005*, Ithaca, NY, U.S.A., Feb. 2005.

[13] "The Delft Bittorrent Dataset for Peer-2-Peer research," http://www.peer-2-peer.org.

[14] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang, “Exploiting BitTorrent For Fun (But Not Profit),” in *International Workshop for Peer-to-Peer Systems (IPTPS) 2006*, Santa Barbara, CA, U.S.A., Feb. 2006.

[15] “The Azureus Java BitTorrent Client,” http://azureus.sourceforge.net/.