Distributed Concurrent Downloading of Common Content in a BitTorrent Peer Group

Sukhbir Singh
R.B.I.E.B.T

Abstract- **In the BitTorrent peer-to-peer file sharing protocol, it is observed that when a group of peers known to each other (due to geographical proximity or otherwise) are aiming for the same file, it is downloaded by a single peer and then shared within the group or it is downloaded individually. This is an inefficient approach as the fidelity rests on one peer or all peers (promoting redundancy), thus resulting in wastage of bandwidth and time. In this paper, we propose a solution to counter these problems by encouraging equal participation of the peer group in the download process. We show how the bandwidth of each peer is utilized to the maximum (thus reducing download time) by segregating a file into smaller sized parts with each peer downloading a part of the file in parallel and then merging the individual parts to form the complete data. We also add to the above proposition how one peer can effectively resume the interrupted download of another peer, remotely. Using simulation, we validate and demonstrate the effectiveness of the proposed strategy by taking real world examples showing large amounts of data being transferred at massive speeds and in a very short period of time.**

***Keywords-* parallel downloading; peer-to-peer/ content distribution works; bittorrent**

I. INTRODUCTION

The BitTorrent protocol is the most common and widely implemented peer-to-peer file sharing protocol for transferring large amounts of data, accounting for more than 45 – 78% of all peer-to-peer traffic, which roughly amounts to 27 – 55% of all Internet traffic, depending on the geographical location [1]. The plethora of websites which index content that can be downloaded using the protocol serve 200 million peers and index 4 million torrent files [2]

The protocol allows downloading of content by a single peer only; neither the protocol nor any of its known implementation provides support for a commonly occurring scenario wherein all the peers in a known peer group are aiming to download a common file which is required by each peer individually. By such a peer group, we meant to include those peers which are in close geographical proximity; meaning they can exchange data physically (over sneakernet) or over a local area network [3]. The most suitable examples would be, though not limited to – students in a dormitory or a group of friends living within the same city. In such a case when the entire peer group is aiming for the same file, either a single peer downloads it and shares it within the group *or* each peer in the group downloads its own copy of the file.

Both the above mentioned approaches are inefficient – if a single peer is downloading the given file, the entire download process from initiation to completion is dependent upon it. If all the peers in the group are downloading their own copy individually, it leads to redundancy. Though ultimately all the peers are aiming for the same file, yet they are not contributing their resources proportionally to the download process; it depends on a single peer or it depends on each peer individually. There is unequal participation in the process and hence waste of bandwidth and time in both the cases. A conventional BitTorrent download is handled in this way for common data that is to be shared within a peer group as the protocol itself provides no support; peers have to rely on uneconomical means to achieve such a transfer.

To tackle the above problem, we propose the concept of *distributed concurrent downloading* within peers which can exchange data physically or over a local area network. In this paper, we describe this approach in detail and show how it can be used to transfer very large amounts of data in a very short period of time while maintaining the integrity of the download. After analyzing the data that is obtained from our experiment, we show how this approach:

1) Drastically increases download speeds by a factor which depends on the number of peers in the group and their relative bandwidths.

2) Improves the scalability of the global swarm that is downloading the said file by providing an increase in the number of seeders.

3) Supports resuming the interrupted download of a peer by another peer in the group, remotely.

We first begin by describing the concept of distributed concurrent downloading, how it works and the need for its implementation.

II. CONCURRENT DOWNLOADING

In a BitTorrent download, it is observed that if the same content is required by a group of peers known to each other, it is downloaded by a single peer and shared amongst the group or it is downloaded by each peer individually. Conventionally, this approach is followed, however as described above it is slow and inefficient. There is rampant wastage of peer bandwidth and time. This problem can be simplified if the file to be downloaded is split into smaller parts with each peer in the group downloading a part of that file in parallel and then

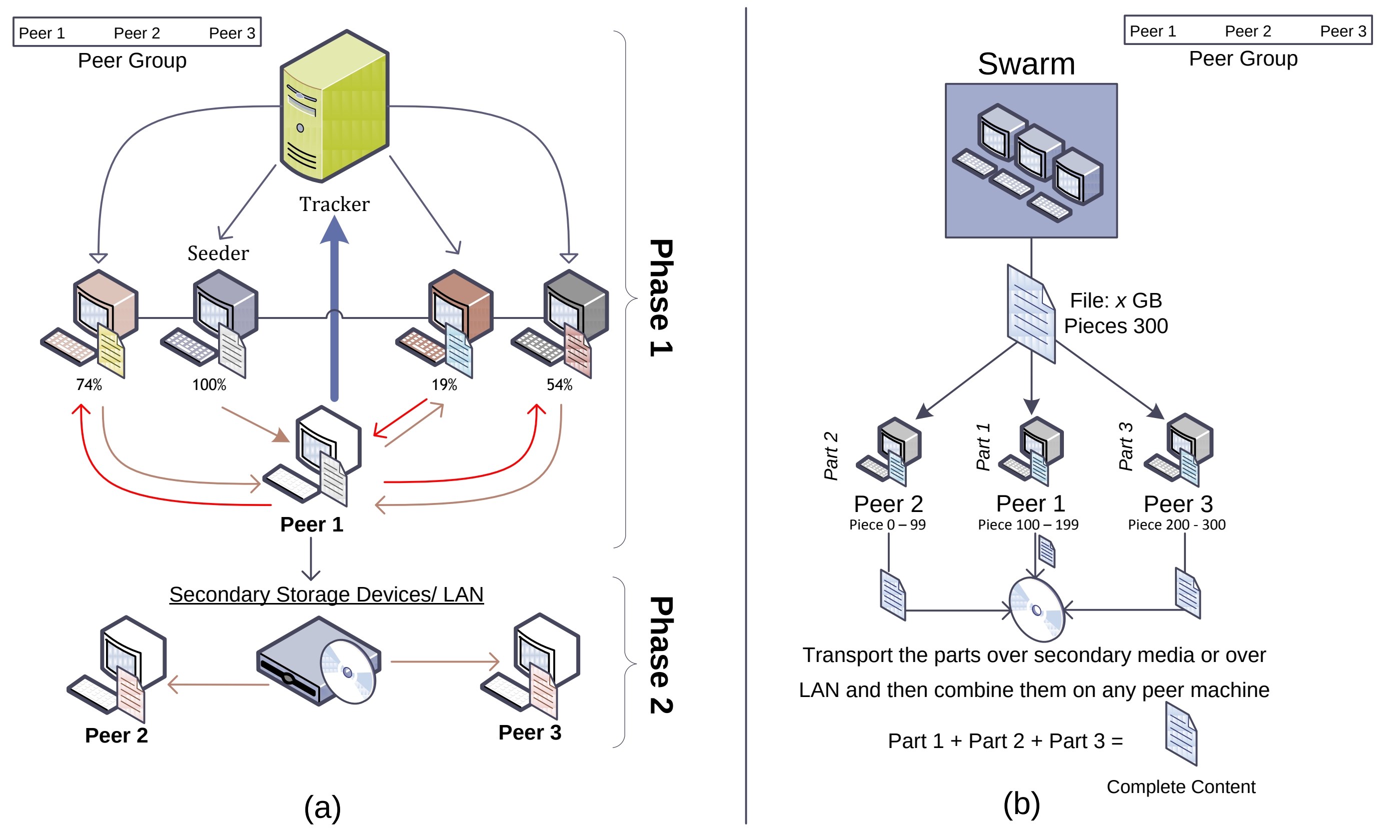

Fig. 1. The two ways in which common data is downloaded. (a) shows the conventional BitTorrent approach. (b) shows our solution with increased efficiency.

merging it later. Hence the word, *distributed concurrent sharing*.

To explain our approach, we present the two aforementioned cases diagrammatically in Fig. 1(a) and Fig 1(b). The former shows how data is usually transferred while the latter shows our solution and its operation.

A. The Usual Approach

In the first case, a single peer downloads the data, shown in *Phase 1*, as in any normal BitTorrent download. We assume that *Peer 1* has a high speed internet connection, so the peer group decides that that peer will download the content and then share it later within the group. *Peer 1* initiates the download, as shown in *Phase 1*. After the download is complete, *Peer 1* then transfers the data (shown in *Phase 2)* through a secondary storage device such as a flash drive or over a LAN (if they are in a local area network) to *Peer 2* and *Peer 3*.

This approach is inherently flawed. The download depends entirely on *Peer 1; Peer 2* and *Peer 3* are mere spectators in the process. They want the data, yet they are not contributing their bandwidths to the download. Also, both the peers are at the mercy of *Peer 1* to fetch the content for them; they have no say in the process. If *Peer 1* has to abort the download, or if it fails, the entire group suffers.

If each peer resorts to downloading its own copy of the content, it will lead to redundancy. After all, if all the peers are aiming for the same file, they should be able to participate in downloading it collectively. So this approach is also ineffective.

As we have seen, neither the BitTorrent protocol nor any of its implementation offers a conclusive solution to this problem. Peers ultimately have to rely on the uneconomical methods presented in Fig. 1(a). We intend to remove all these drawbacks in our solution, which follows next.

B. Our Solution

We propose a simple and viable solution. Refer to Fig. 1(b). The content to be downloaded is divided among the peer group in such a way that the content is (virtually) split into parts, with each peer in the group downloading a part of the content. After each peer has downloaded its part, the individual parts are collected on any peer machine and are merged to form the complete content. This approach is highly economical and utilizes the resources of each peer to the maximum. We next illustrate this with an example.

Before we begin, we completely exclude the protocol; all the changes are to be made on the client-side, thus allowing our solution to be deployed easily. We make a few assumptions, but the intention is purely to simplify understanding of the concept. As we show later, the assumptions hold true.

In a normal BitTorrent download, a file is split into different *pieces*, with the peer downloading multiple *pieces* of a file to form the complete file [4]. To allow for a file to be split on the client side to implement our approach, we fix piece count between the peers, thus allowing the file to be split virtually.

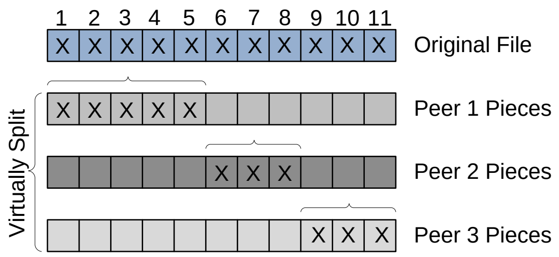

Fig. 2. The file is virtually split by changing the piece count each peer downloads.

Refer to Fig. 2. Let us suppose that the file to be downloaded is made up of *11 pieces.* Theoretically, the number of pieces a file contains depends on its size [4]. However, our assumption on a small scale of piece count holds true for larger files also, since only the piece count will increase, the method will remain exactly the same.

We present a step-by-step approach as to how the peers can successfully perform a download using our concept. We also explain side-by-side the technicality of our setup.

1) The peer group has *3* peers, which are all aiming for the same file. The peers interact among themselves using verbal communication or social networking sites, email, chat, or any other medium of communication, the *piece* count each peer will download. The peer group is an arbitrary number; theoretically it is limited only by how many peers are aiming for the same content, or how many peers can actually transfer data physically with each other. For larger file sizes, there can be more peers and for smaller, less.

2) *Peer 1* has the fastest internet connection, so the peer group decides that *Peer* 1 will download the maximum number of pieces, while *Peer 2* and *Peer 3* will download fewer pieces. The *piece* count is finalized - *Peer 1* will download 1 – 5, *Peer 2* 6 – 8 and *Peer 3* 9 – 11, as shown in Fig. 2. The piece count each peer downloads is also a purely arbitrary number, so as to allow the peer group to decide on it according to various resource constraints. This makes it possible for a peer with a higher download rate to download more pieces, while a peer with a slower connection can choose to download fewer pieces. This is another plus point of our approach; we utilize the bandwidth of each peer to the maximum in the most effective manner possible.

3) After the peer group and *piece* count is fixed, all the peers initiate the download using a *torrent* file as they normally would in a conventional BitTorrent download. When the client starts the download, it asks each peer their respective *piece* count. Since that has already been decided, the peers enter their counts and the download starts.

We would like to point out here that even though we are restricting *piece* count, there is no change in the way the file is usually downloaded. All the BitTorrent protocol specifications still hold true. The mechanisms the protocol uses to transfer files, such as *tit-for-tat*, *rarest piece first* [5], all hold true. This is because what we are effectively doing is restricting piece count. By doing that, the client doesn't request *pieces* other than those specified by the peer. This is equivalent to a client not requesting *pieces* it has already downloaded, which is what happens in a conventional download. The protocol functions the way it normally would even if it is forced to restrict downloading between a specified *piece* limit.

4) After each peer finishes downloading its part of the content, the merging stage comes up. This involves transfer of the content parts from peer machines to any given peer machine (even outside the group), so that the parts can be merged to form the complete content. We note a few important things here.

First, we take into account the decreasing cost of secondary storage media [6], and increasing data transfer speeds for them. We also refer to [7] the speed limits of a LAN, and to [8] for the maximum speed in a WLAN. From [6, 7, 8] it infers that transporting data over secondary storage devices or a LAN, is *faster* and *cheaper;* in respect to our approach, it implies that the transporting the individual parts is much more economical than the approach shown in Fig. 1(a). So moving the individual data parts is not a hassle, though it might seem so. Even with files of larger sizes, with fast storage media available, data can be physically shared easily.

The peers use the means above to transfer the data amongst them. Any given peer machine is selected, preferably the peer which is closer to the group collectively, so that each peer can bring over its part of the file for merging. Note that it is not necessary for the peer machine to be selected for the merger to be that of a peer in the group, any machine can be selected; all that is required is the *torrent* file that was used to initiate the download and the individual file parts. An important issue comes up in this regard, which is maintaining file integrity for each part of the file.

BitTorrent already calculates the SHA1 checksum for all pieces in a file. [4] This data is stored in the *torrent* file for the content that is downloaded. The client then uses the checksums in that file to check for any bad data after all the pieces of the file have been downloaded.

Since the file has checksums for individual pieces, verifying integrity is easy. Refer to Fig. 2, suppose *Peer 1* is selected for merging the file. *Peer 2* and *3* bring their parts of the file over to *Peer 1*'s machine. The client software on *Peer* 1 initiates checksum verification (shown later) for each part received by comparing it against the checksum in the *torrent* file. Since *Peer 1* had downloaded pieces 1 – 5, the client matches the checksum only for those pieces. If a bad piece is found, the client reports it; only that *piece* is to be downloaded again, not the entire file (or part).

C. Positive gains from our solution

If our solution is applied to the downloading process, we notice some positive side effects which strengthen our solution's effectiveness even more. We present them below and support them with experimental data later.

1) Since the file is split virtually, we notice a very interesting side-effect. Consider the same peer group in Fig. 2. Let us suppose that *Peer 1* has to abort its download due to some reason. The other peers are unaffected due to our approach, however the download still suffers since *Peer 1* was contributing the maximum bandwidth. We now put forward a method that allows the download to be resumed remotely by another peer.

When a peer is downloading a part of the file, the client keeps note of all the pieces that have been downloaded, or are to be downloaded. When a peer abandons the download, the client has all the information necessary for another peer to resume it. Consider our previous example. The piece count of *Peer 1* was restrained to 1 – 5. Now during the download suppose that it downloaded pieces {1, 2, 5}. Piece 3 and 4 was still left but the peer had to pause/ abort the process.

Next consider another peer is made to join the group to resume the aborted download of *Peer 1*. *Peer 1* publishes its piece list, i.e. the detailed piece statistics of its download, which is already maintained by the client and sends them over to *Peer 4,* who is the new peer in the group. *Peer 4* then feeds the piece list to its *torrent* client, and finds out which pieces are to be downloaded and which are to be ignored. The pieces which *Peer 1* downloaded are ignored, while the pieces which are still left, piece 3 and 4, are put to download. When the downloading completes, we have 4 parts of the file. All other sub-processes are exactly the same; instead of 3 parts there are 4 now, but the overall process including that of verification is the same.

Hence, we see that using this approach it is possible to sustain the download process even if one or more than one peer exists. This ensures maximum efficiency and the download uptime is enhanced since there are no bottlenecks.

2) We also notice that the scalability of the file download is increased manifold. If our approach is followed, there are theoretically more *seeders* in the swarm than there are in a conventional download. In a normal download usually, there are more *peers* than *seeders.* In our case, there are more *seeders* than the *peers,* which is how a swarm should be.

The explanation is simple. During the download, theoretically a single peer is downloading the file. Because the file is split on the basis of piece count, virtually there is only peer in the swarm that is utilizing the swarm bandwidth. Though in reality there are many peers downloading the file, but in fact all of them are downloading a single copy of that same file, and hence the swarm treats that as a single peer.

Also, when the download has finished and the peers have exchanged the file, all peers have now become *seeders* - since they have the complete file now, they will be contributing it fully to the swarm. So even though a single peer started the download, it evolved into multiple *seeders* later thus increasing the scalability of the file many times over.

III. Experimental Data

To validate the effectiveness of our proposed strategy, we present below experimental data to confirm the same.

Consider a case where there are a total of n peers in the group x_1, x_2 … x_n downloading the content in our specified manner. The download speeds be d_1, d_2 ... d_n If there was only one peer downloading the time required would have been:

$$\frac{Total\ Size}{Total\ Download\ Speed\ of\ one\ Peer}$$

Now if the peer group takes the responsibility, the total size for each peer reduces to a part of the file. Let the parts be p_1, p_2 … p_n. Download time for i^{th} peer become:

$$\frac{Piece\ Size}{Download\ Speed\ of\ one\ peer} \text{ or } \frac{P_i}{D_i}$$

$$\text{Total download time} = max\ (\frac{P_i}{D_i});\ \ 1 \leq i \leq n$$

If we consider the virtual concept, the download speed of one peer has been increased to sum of all the peers. Thus, download speed for considering the whole system as one peer can be reduced to:

$$\frac{p_1 + p_2 + p_3 + \cdots + p_n}{d_1 + d_2 +\ d_3 + \cdots + d_n}$$

$$\text{or, } \frac{\sum_1^n P_i}{\sum_1^n D_i}$$

Deciding on the division of pieces between the peers:

It is very obvious that the peer with the highest speed would get the maximum bunch of download to implement the algorithm properly.

$$piece\ count \propto \text{(peer download speed)}^m\ ;$$
$$\forall\ m \text{ as positive integer} \qquad (1)$$

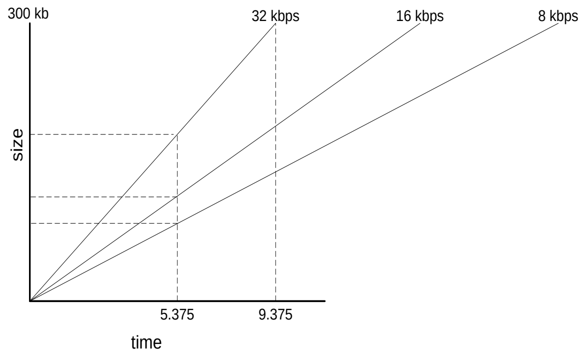

Fig. 3. Reduction in download time.

Consider a case with a data of *300* kb to be downloaded by three peers with download speeds *8* kbps, *16* kbps and *32* kbps on an average.

Individually,

Peer 1 would take (300 / 8) = 37.5 seconds

Peer 2 would take (300 / 16) = 18.75 seconds

Peer 3 would take (300 / 32) = 9.375 seconds

Now, dividing the 300 kb data into three pieces of length p_1, p_2 and p_3, we observe from Fig. 3, that for certain configurations the download time is drastically reduced.

We require that the total download time should be less than 9.375 seconds as can be achieved if only *Peer* 3 downloads. Considering that *Peer* 1 downloads *x* kb of data, *Peer* 3 downloads *y* kb and the remaining (300—*x*—*y*) is done by *Peer 2*.

Time for *Peer 1* = *x* / 32

Time for *Peer 2* = (300—*x*—*y*) / 16

Time for *Peer 3* = *y / 8*

The algorithm will be feasible only when time for downloading the piece allotted to the slower peers is *less* than what is required for the fastest one. Thus we require,

$$x / 32 > (300 - x - y) / 16, \text{ and}$$

$$x / 32 > y / 8$$

Solving for boundary conditions:

$$x = 171 \text{ kb},$$

$$y = 43 \text{ kb}$$

TABLE I
OBSERVED DOWNLOAD SPEEDS OF PEERS IN A BITTORRENT DOWNLOAD IN A TIME FRAME OF 8 SECONDS

Peer 1	Peer 2	Peer 3
7.2	28.2	44.2
8.1	31	44.4
7.6	33.4	45
7.8	27.5	43.8
8.2	30.7	43.6
6.3	32.1	44
6.9	29.9	44.3
7.5	28.4	44.7

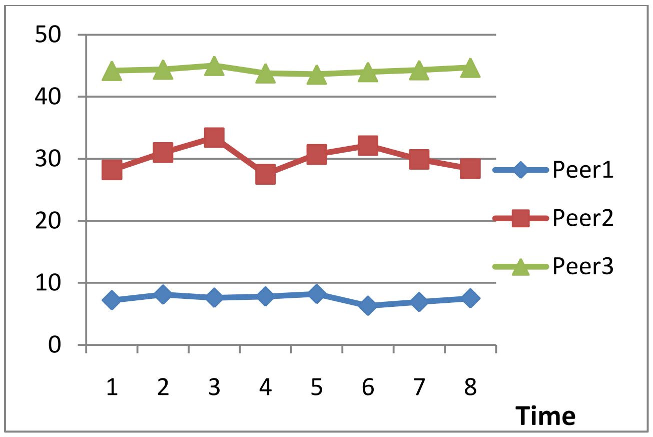

Fig. 4. The varying download speed of our selected peers in a BitTorrent download.

This would amount to a time of 5.4 seconds approximately, which is *way less* than what was required by *Peer* 1.

Also, in synchronization with the prescribed formula, the pieces are in the ratio of 1:2:4, which can be found out by putting *m = 1* in the (1).

Our actual analysis was limited to *3* peers (see Table 1, Table II and Fig. 5) only because taking into account the nature of our experiment, a smaller sample size would be better in showing results in a coherent manner rather than a large sample size where it would be difficult to make out what is being said. As we have proved above, our result is equally valid no matter how many peers are there.

To carry out the actual analysis, the root mean square (rms) values are used to approximate the minimum possible error count. (See Table 2)

The *rms* download speeds are in the ratio

$$44.25: 30.21: 7.47$$

Using the algorithm devised in (1), the *piece* count should be divided as:

$$44.25^m: 30.21^m: 7.47^m$$

TABLE II
ACTUAL TIME TAKEN FOR DOWNLOADING 38.72 MB IN PIECES

Download Size	Peer 1	Peer 2	Peer 3
0	0	0	0
3	401.6	99.3	67.8
6	803.21	198.6	135.6
9	1204.81	297.9	203.4
12	1606.42	397.2	271.2
15	2008.03	496.52	338.98
18	2409.63	595.8	406.8
21	2811.2	695.13	474.57
24	3212.85	794.43	542.4
27	3614.45	893.74	610.16
30	4016.06	993.05	677.96
33	4417.67	1092.35	745.76
36	4819.27	1191.65	813.6
39	5220.88	1290.96	881.3

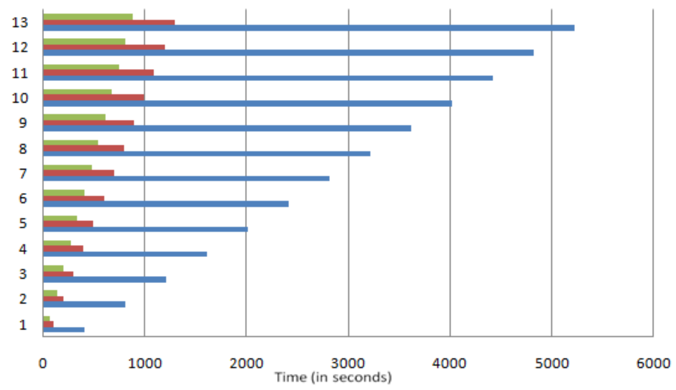

Fig. 5. Pieces v/s download time measurements

The possible solution to the above equation with positive *m* comes out as:

7:5:1

Time required for download in the above case:

$max\{(Peer3) \rightarrow 7 \text{ pieces}, (Peer2) \rightarrow 5 \text{ pieces}, (Peer\ 1) \rightarrow 1 \text{ piece}\}$

This comes out to be *474.57* seconds. If only *Peer3* downloaded the whole content, time required was *881.3* seconds. Thus, the time is drastically reduced with the participation from each peer as per the algorithm.

III. Application

Our approach has wide area of application, thus making its implementation prolific. We present some of the most prominent ones below.

1) Students in a dormitory can make use of distributed downloading and collaborate on downloading data. Consider a common case and the requirement which led to development of this idea originally. A Linux distro is released as DVD ISO image for download via BitTorrent and is required by a group of students in a dormitory (also called flash crowd effect [11]). Now instead of each student in the dormitory downloading its own copy of the distro or a single student downloading and then giving it to the group, the students use our approach and download the ISO image in parts. After the downloading is complete, the students then merge the individual parts over their high speed LAN connection. Everyone gets the complete Linux distro theoretically within a few minutes, taking into account the high speed internet access in universities and the fast data transfer rates over a LAN.

This is also applicable to friends living within the same city that are in a position to exchange data physically.

2) In developing countries, fast internet access is still not widespread enough [9] [10]. Even if the availability is there, a connection with high speed is expensive. So downloading of large amounts of data is not viable. Our approach is perfectly suited for use in such a scenario. If peers make use of our approach, downloading of large files is also possible.

So basically we see two specific domain applications of what we proposed. Firstly it makes it possible to download files of very large sizes even, since the speed depends on the number of peers in the group. If larger file sizes are to be downloaded, more peers in the group can be added and so on. Secondly, it is a *social* BitTorrent concept, where peers socialize with each other and download content. If a peer group wants to download content and it uses our approach, every peer in the group benefits. Thus, the potential this idea has is immense.

IV. Related Work

Realizing the need of such an approach where peer groups download data in parallel, there has been some work done in this field already. Though not related to our work in general, the research carried out laid stress on the need for a social based file sharing network, which is exactly what we propose. Azureus (now Vuze) [12], one of the most popular BitTorrent clients implemented a feature called *Friend Boost.* Under this, it was possible for a peer to upload *pieces* at a higher speed (preferentially) to another peer which was designated as a *Friend.* However, this feature was later dropped from the client [13]. It failed because it still did not solve the problem of *distributed sharing* – what we proposed and proved in our paper. Also, an important mention is F2F or *friend-to-friend* networks, which is another attempt to create a social file sharing based network [14]. However, as far as we are aware, our idea has not been implemented in any BitTorrent implementation or otherwise.

References

[1] BitTorrent Still King of P2P Traffic. http://torrentfreak.com/bittorrent-still-king-of-p2p-traffic-090218/
[2] Isohunt Tracker Statistics. http://isohunt.com/stats.php?mode=btSites
[3] Sneakernet Redux: Walk Your Data. http://www.wired.com/culture/lifestyle/news/2002/08/54739
[4] BitTorrent Protocol Specification v1.0. Retrieved from http://wiki.theory.org/BitTorrentSpecification#Metainfo_File_Structure
[5] B. Cohen, “Incentives Build Robustness in BitTorrent”. *1st Workshop on Economics of Peer-to-Peer Systems*, 2003.
[6] J. Palm, “The Digital Black Hole”. Retrieved from www.tape-online.net/docs/Palm_Black_Hole.pd
[7] Ethernet Speeds. Retrieved from http://en.wikipedia.org/wiki/Ethernet#Varieties_of_Ethernet
[8] Wireless LAN Association High-Speed Wireless LAN Options. Retrieved from http://www.wlana.org/pdf/highspeed.pdf
[9] The cost of internet access in developing countries. Retrieved from http://www.itu.int/asean2001/documents/pdf/Document-16.pdf
[10] Internet Usage in India. Retrieved from http://www.internetworldstats.com/asia/in.htm
[11] J. A. Pouwelse, “The BitTorrent P2P File-Sharing System: Measurement and Analysis”. In *IPTPS*, 2005.
[12] Vuze BitTorrent client. www.vuze.com
[13] Discontinuation of Friend Boost. http://blog.vuze.com/2009/11/12/vuze-update-less-is-more/
[14] W. Galuba, “Friend-to-Friend Computing: Building the Social Web at the Internet Edges”. http://lsirpeople.epfl.ch/galuba/papers/f2f.pdf