Introduction

BitTorrent [1] is one of the most popular peer-to-peer file-sharing protocols known for sharing large files over the Internet like tv shows, movies, games, software, etc. BitTorrent gained a lot of popularity amongst several service providers as an efficient and scalable way of delivering data while reducing the load on central servers along with the download time for the peers. According to "Application Usage & Threat Report" [2] in 2013, Bittorent accounted for 3.5% of the whole world wide bandwidth which was more than half of the bandwidth allocated for file sharing. This meteoric rise in the popularity of BitTorrent was due to several innovative mechanisms introduced like tit-for-tat (TFT) and rarest first (RF). These mechanisms helped in solving the problems faced by the traditional protocols. BitTorrent has attracted many pieces of research over the years ranging from a survey of its performance [3] to modifications in the core algorithm to further increase its performance [4].

Architecture

BitTorrent works on an overlay network called "Torrent" where connections are established between the peers for the corresponding file being distributed. Along with the peers, tracker and web server are also required for file distribution in BitTorrent.

The tracker is a special node used by peers to find each other downloading the same file by keeping a log of peers that are currently downloading a file. The tracker is not directly involved in the transfer of data as it does not have a copy of the file. The information between tracker and peers is exchanged using a simple protocol on top of HTTP. The peer gives information to the tracker about which file it's downloading, ports it's listening on, etc. The tracker in return sends a response consisting of a list of other peers who are downloading the same file.

A peer is classified into two types: leecher and seed. Leecher is the one downloading the file. A seed is the one having a full file staying online to serve others. To publish a file using BitTorrent for download, the first step is to create a meta info file called a "torrent". The torrent file contains the information regarding the file like filename, size, hashing information of the pieces, and URL of the tracker. This torrent file is required by the peer who wants to download the file corresponding to that torrent file.

To download a file corresponding to a .torrent file, the peer contacts the tracker to request a list of peers downloading the same file. Tracker on receiving the request selects a number of potential neighbor peers randomly (usually 50) from the list of all active peers in the torrent. On receiving the list, the peer contacts them to create a set of neighbors. If the number of neighbors falls below 20, the peer contacts the tracker again for the list.

When creating the torrent file from the original file, the original file is cut into smaller pieces, usually 512 KB or 256Kb in size. Earlier SHA-1 hash codes of the pieces were included in the torrent file. Newer protocol shifted to SHA-256 as the earlier one was declared not secure. The downloaded data is verified by computing the hash for the downloaded data with the one present in the torrent file. This ensures that the data is downloaded free of errors and guarantees that the

real files were downloaded. As the piece is downloaded and verified, the peer downloading the file reports this to other peers in the swarm about the new piece.
The process followed in file-sharing by Bittorrent can be seen as 5 steps as shown in fig1:

1. For Peer A to download the file, Peer A first downloads the corresponding .torrent file from a web server.
2. Peer A then contacts the tracker for a list of active peers participating in the torrent.
3. The tracker returns a list of peers involved in the torrent.
4. Peer A adds all the connected peers from the list as its neighbors and sends the file piece request to each other.
5. Once the request is accepted, Peer A can exchange file pieces with the neighbors.

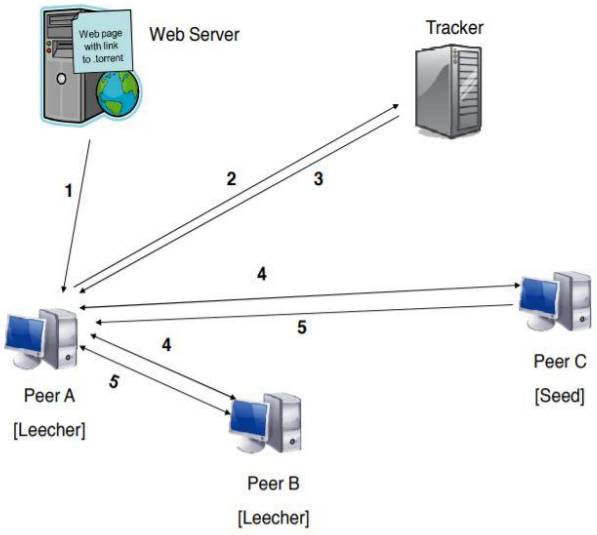

Fig1. BitTorrent file sharing process [3]

The peer downloads the pieces not only from the seeds but also from other peers, reducing the load on the seeds significantly. A peer serves only a certain number of peers at a time (typically 4). This makes the peer choose the best peers to serve while choking others. Choking is a method to temporarily refuse to upload while keeping the connection alive. The goal is to have several bidirectional connections at any time. This results in peer uploading to peers with the best download rate. So more peers will allow to download if the peer has a higher upload rate, which eventually leads to a higher download rate.
The key architecture points of BitTorrent are peer selection strategy and piece selection strategy.
The peer selection strategy focuses on maximizing the service capacity amongst the peers and the piece selection strategy tries to keep the pieces available which the peers and its neighbors might be interested in.
To understand the importance of peer selection strategy we can look at an example. Let a peer A be downloading a file. A will be interested in another peer, say B, only if B possesses pieces which A doesn't possess. The more the peers A is interested in, the better will be the

downloading ability for A. So it becomes crucial for A to have neighbors with maximum peers like B. But due to the tit-for-tat mechanism (TFT) adopted by BitTorrent, B would like to serve A only if B is also interested in A. This highlights the importance of a good peer selection algorithm. Hence the peer selection is outlined by four major mechanisms: : *tit-for-tat (TFT), optimistic unchoking (OU), anti-snubbing, and upload only*. The combination of these mechanisms try to reward the peers which contribute by uploading and punish the freeloaders which just download without uploading the pieces back. This enhances the download experience for the contributing peers. The mechanisms are as follows:

- Tit-for-tat: As the name suggests this strategy rewards the peers who can provide the best value back. As the peers are connected to only a fixed number of neighbor peers (typically 4), it makes sense to choose the ones with best downloading rates and choke others . After every 10 seconds, the downloading peer reevaluates by reassessing the downloading rates of the neighboring peers. If any choked peer promises better downloading rate than any of the unchoked ones, then the roles are swapped. This strategy encourages more contribution and punishes free-riders.
- Optimistic Unchoking: With just the TFT strategy there is no opportunity for discovering other peers which might be better than the ones being used. Hence it becomes important to discover better neighboring peers. This is done every 30 seconds by unchoking a randomly selected neighboring peer regardless of its uploading rate. If the new peer connection is better than one of the existing unchoked ones then this replaces one of them. While this helps in discovering peers with higher upload rate, it is also useful for newly joined peers to get started.
- Anti-snubbing: A peer might be getting poor download rates as it may be choked by the peers it was formerly downloading from. So if a peer doesn't get any piece from a neighboring peer in some time, the leecher assumes it is 'snubbed' by that peer. As optimistic unchoking is done every 30 seconds, the peer now does not upload to this neighboring peer any further through regular unchoke to recover download speed faster.
- Upload Only: As a leecher finished downloading the entire file, it becomes a seed. As for the TFT strategy the seed has nothing to gain by selecting peers based on downloading rates, it prefers to upload to peers with better uploading rates. This helps in replicating the pieces faster and then uploading them to other peers faster.

After the peer selection, choosing pieces to download unintelligently can also lead to situations where all peers have pieces currently available and none of the missing ones. This highlights the importance of a good piece selection algorithm. The goal is to replicate different pieces on different peers faster. This ensures that all the pieces are at least present somewhere in the network. The pieces are further broken down into smaller sub-pieces which can be downloaded from different peers. There are four major mechanisms as follows:

- Strict Priority: Here peers try to download a whole piece before requesting another piece. This means that if sub-pieces for a piece are requested then the remaining sub-pieces of

the same piece will be requested before the sub-pieces of any other piece. This is done to download the complete piece as soon as possible as only the complete pieces can be traded with others.

- Rarest First: This means that the peers prefer to download pieces which are the rarest. Each peer keeps a list of the pieces that each of its neighbors possess. This list is updated as a new piece becomes available from its neighbors. The peer then downloads the pieces rarest among its neighbors. This helps in balancing the load by spreading the pieces which are rare. As the current peer gets the rare piece which others might want, it will make other peers interested in trading with this piece further increasing the download speed. Along with removing the fear of missing pieces in the network as the seed leaves creating a bottleneck, it also helps in increasing the download speed across peers as the rare pieces are replicated over time providing multiple copies.
- Random First Piece: For a peer joining a torrent, it has nothing to upload so it becomes important to get the first piece as fast as possible so as to reciprocate for TFT strategy. Here the rarest first strategy will fail as in the rarest first strategy the rare pieces are present with a lesser number of peers, it slows down the download rate. So the peer here will download the first piece randomly to get a whole piece as fast as possible.
- Endgame Mode: This mode is activated as the peer is nearing the complete download of the file. If the final request piece is requested from a peer with a slow transfer rate, the finishing of download will be delayed. To solve this the peer requests all of its neighbors for remaining sub-pieces by broadcasting the request. This helps in completing the download of that last piece faster. As the piece is downloaded a cancels message is sent to indicate that the download is completed.

Protocol Features

BitTorrent while being complex, configures the connections to the peers without the users being involved making it self-configuring. Dynamically decisions are made inside the client of every peer, based on the list of possible peers and the algorithms presented making it easy to use for the users.

The major issue before BitTorrent was that most users tend to have different speeds for download and upload. Many traditional peer-to-peer file sharing protocols suffered by not handling this. This used to cause a user to have a very lower download speed for a file due to uploaders' downlink and uplink. This implied that one-to-one file-sharing was not an optimal solution. BitTorrent tackled this by splitting files into smaller chunks and downloading chunks from different users at the same time which resulted in better utilization of downlink bandwidth. This significantly reduced the downloading time making the downloads faster.

In traditional models, the increase in the number of downloaders increases the load on servers hosting the file. The architecture of BitTorrent turned this con around by utilizing the upload capacity of the peers that are downloading a file and making the protocol work better as the

number of downloaders increased. This enables a much faster download speed than before. Along with the better download speeds, it relieves the pressure off the servers having large files.
Further the “tit-for-tat” strategy tries to prohibit “free riders” from destroying the dynamics of the peer-to-peer network. If peers contribute more it keeps their download rates higher while if a peer blocks other peers from uploading, it will lead to itself being choked by them and thus affecting its download rate. BitTorrent encourages uploading in return for a better chance for faster download.
The cryptographic hash stored in the torrent file provides a way to promote security and authenticity. By identifying any modification or changes done to the file with the hash provided, any malicious attempts against the peers can be stopped.

Challenges

While BitTorrent proposed a unique way to improve performance with more downloaders, it created trouble for the old or unpopular files. As the unpopular files will have a lower number of users downloading them, it will be difficult to find them and fewer users to download from. This results in making the protocol effective when dealing with highly demanded files and less popular files tend not to be available.
The random neighbor selection strategy also could be modified to better use the network resources and further improve performance.

References

[1] BitTorrent: https://www.bittorrent.com/
[2] Palo Alto Networks Application Usage & Threat Report (accessed on 20th March’21): https://blog.paloaltonetworks.com/app-usage-risk-report-visualization/
[3] A Survey of BitTorrent Performance: https://ieeexplore.ieee.org/document/5451765
[4] Fair File Swarming with FOX (2006): http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.8125

Hub -> centralized servers

Centralized servers - multiple - interconnected - (lets say 4-5)
→ predefined IP address in code (hardcoded) -> initiating connection
→ every time at least 2 central server connection (fault tolerance)
→ central server use hash table for the file name and peer information DHT

Peer connect with central servers
→ peer informs file to be uploaded -- upload -- this will be saved in central server hash table
→ connection alive every 10 mins check

Search
→ contact central server with filename (search module)
→ file name exact matching
→ multiple files with the same hash stored in a different table (replication)

Download
→ get file meta and corresponding peers
→ divide the file into chunks and download parts (from which peer, which chunk) (load balancing)
→ each chunk verification by hash matching (chunk security and validation)
→ complete file hash matching (file authenticity)

< key, value > -> key = file name, value = peer ip address, timestamp, file hash, size
resolving ip address? -> node unique id?
hub having range of unique ids for peers

-> new hub joining protocol
-> new peer joining protocol
    -> unique id or ip address
-> file upload protocol
    -> file indexing
-> file download protocol
    -> hash verification
-> peer or hub runtime -> connection alive checking protocol

1) Hub
2) Dnode
3) RPC client

Hub:
Uid :: 32 bytes
Port1 :: listens for peer hubs
Port2 :: data servers communication

Cmd :: ./hub -p13000 -p2 4000

Dnode
Uid :: it will be given by any one of the hubs
Port1 :: data port for other data nodes

Port2 :: rpc client port
- Upload files
-
- Download files
-

Cmd :: ./dnode -h ip:port -p1 5000 -p2 6000

RPC client

./rpc_client -dnode ip:port -u file_name
./rpc_client -dnode ip:port -d file_name

- HUb communication

1) Join

2) -d filename
   Result :: Index_data of the file, destination servers ip address
3) -u filename
   a) Result:: destination ip addresses

File indexing

1) Split the file into chunks of 2MB
2) Hash each chunk (32 bytes)
3) Hash the concatenation of the chunk hashes, this will serve as file hash.

DHT::

1) File_name , file_hash
2) File_hash, {index_data, destination_server_uids}

In-memory data structure

Dnode_uid, dnode_ip, dnode_port, dnode_flags (32 bytes + 4 bytes + 2 bytes + 2 bytes) (40 bytes)

HUb -hub communication

1_ search for file_name
2) file_hash

2 users:

- hub
    - map for file name, file hash
    - map for file hash, file information < list of peers to download from >
    - alive peers list
- client
    - maintains map for uploaded files
    - map for file name, file hash
    - map for file hash, file information:
        - file location
        - size
        - file name

hub:

- initialization:
    - initialize DHT
    - unique id for peers initialization
    - ports open (two server ports)
- while running:
    - Check for peer state when
- upload request:
    - check file hash similar file present
        - check filename and size (replication)
        - add as another peer for file
    - else
        - create new index with this file hash
- on download request:
    - check file by file name (complete matching)
    - check if peers are alive or not for peers last active before 30 mins:
        - if not then set not active
        - if active then check for file upload by peer
    - send list to requesting peer

peer:

- initialization:
    - predefined ip address list of hubs
    - try connection with 2
    - get unique id and client details
- system start:
    - recheck hub connections

    - check files uploaded are available or not
- upload:
    - divide file into chunks and keep chunks hash
    - create file hash
    - send file details to hubs (filename, size, uid, file hash, chunks hash)
- download:
    - request list from hubs
    - from list check connection with peers
    - keep a queue for chunks
    - download chunks from different peers
    - verify each chunk
    - at the end verify whole file hash
    - seed (replicate upload)
- checkup by hub:
    - check if file is present for upload or not (last checkup before 30 mins)
    - if not then return FALSE
    - else send TRUE

- if node rejects download request:
    - send FAIL to hub
    - hub rechecks connection with corresponding peer for file
    - if refused, remove peer from list of ip address for file
        - if only 1 peer was there so delete file only

- if hub fails:
    - peers find new hub and send upload details for files

- if peer fails:
    - while downloading restart chunk download from another peer