Abstract

Recent online data shows that a growing number of individuals and organizations are using BitTorrent to distribute their own or licensed material. Many report that without using BitTorrent technology and its dramatically reduced demands on networking hardware and bandwidth, they could not afford to distribute their files. In this project we have created a peer to peer le sharing/transfer program similar to BitTorrent. In a peer to peer program instead of a monolithic le being downloaded from a single site the le is broken up into chunks and each chunk is downloaded separately and from possibly different peers and then assembled into the complete le. One clear advantage is that the time required to complete the download is reduced considerably.

1 Introduction

The aim of this project was to create a variation of Bittorrent. Peers share files among themselves but unlike bittorrent there is no central tracker peer. A peer fetches all the information about each chunk of the file it wants to download. Although trackers are the only way for peers to find each other, and the only point of coordination at all, We have done all this without any tracker so all peer flood the n/w when any change has been done since he doesn't know which peer's are online currently so he broadcasts the message to all users in list ifnow any user receiving his msg will reply him if some per are not online then he will not reply still the peer which is broadcasting will try for 3 or 4 times to connect to him since UDP is unreliable so we have done 3 or 4 broadcast to ensure it now if some other peer become online then it is possible that he doesn't have all updated file then so he will broadacast to know which peer are online if any peer are online then he will send his update time of files so the other one will compare it with his update time and send him whether his files are old or now now he will download the files if he has older version so at any time all online users have sme update time is ensured now since we are comparing the update time so it will also ensure at some extent that if all user are not online at a time and an older upadate users come then he will not update his files from him so he has the newer version . A file is made up of consecutive chunks of 1MB (may be except last chunk). Chunks can be located on any peer and the le is downloaded chunk at a time from possibly different peers and then assembled into a le at the peer who requested the file. Each chunk is obtained using UDP instead of TCP. Since UDP is inherently unreliable, additional measures were undertaken to ensure reliability of each Datagram packet sent. The application also implements congestion control mechanisms.

1.1 Terminology

The application maintains the following files for each Peer.

- **main.txt** - Contains the list of all filenames shared by all the Peers. With each filename, the number of Peers sharing this file is maintained.
- **users.txt** - Contains the list of all Peers(both offline and online). This is useful for broadcasting messages.
- **shared.txt** - This file maintains the list of filenames of the files shared by this user. It also has the total number of chunks of this file and chunk numbers of the chunks of the file shared by this user.
- **unfinishedDownload.txt** - If a particular file coudn't be downloaded in a particular session(this can be because the Peers which had the chunk(s) of this file was not up), then that information is kept in this. It essentially has the information of the chunks which are yet to be downloaded.
- **masterChunk.txt** - Contains SHA hash of all the chunks ever uploaded for sharing. When a peer uploads a new file(or some of its chunks) for sharing, this file is updated by appending the SHA hashes of the chunks and broadcasted to all the peers.

2 Functional Details

Any running peer is given five options.

- Option 1 **Download**: The user is asked for a substring of the filename. Then a list of filenames is generated which match with substring provided earlier. The user selects one of these filenames. This filename is broadcasted to all the Peers. The packet contents are
  $Peer.magic + "" + Peer.fileReqStr + "" + filename$
  Peers on receiving the broadcast search for the filename in there respective shared.txt files. If a match is found, a reply is sent back containing the total number of chunks and the chunks shared ny this user. All such replies are stored in a temporary file - filename_temp.txt. An instance of the Downloader is now created by the client and the actual downlaod begins.

- Option 2 **Upload**: The user is asked the name and the path of the file to upload. Then it is checked if the file really exists and if it does then am entry is added to shared.txt. **If the file already has an entry in the main.txt then the count of number of users sharing this file is incremented. If no entry existed then a new entry is added.**
- Option 3 **Delete**: The user is asked the name of the file which it no longer wants to share. The entry of this file in shared.txt is deleted and the **count in main.txt is also reduced by 1.** If this was the only user remaining who shared the file, then the entry from the main.txt is deleted too because count in there will be 1 and decrementing it will make it 0.
- Option 4 **Download unfinshed files**: The unfinishedDownload.txt has the list of requested files which coudn't be downladed. When the user chooses this option, **another attempt is made to download the unfinished part of the file.**
- Option 5 **Unjoin Group**: This is like an unistall option. All the application files are deleted and the Peer is no longer a part of the application.

2.1 Features

- If a new user wants to join the group of existing peers, only the IP address of an existing Peer is required. it is asked to enter the IP address of one of the existing peers. The newly added user will download all the required files from that IP address. It will now broadcast to all the existing Peers the information of its additon to the group.
- For a download request the peer **is not required to enter the complete filename**; only a substring is sufficient. The filenames which have the entered **substring are printed** on screen together with a number. Only that number needs to be entered in order to download the file.
- If a file (or some of its chunks) **coudn't be downloaded during its first download request(may be because no online Peer had that chunk or the receiving Peer itself crashed), its information is stored in unfinishedDownload.txt.** Whenever the Peer exercises option 4, the download is resumed exactly from the state it was left earlier.

- The chunks are strictly downloaded on **rarest chunk first basis**. This is the safest choice in order to download all the chunks.
- A Peer can has certain number of slots pre-defined for parallel download and upload. The default value is set to **four**. This means that a Peer can download from four Peers at a time and also serve to four different peers.
- A chunk downloaded by a Peer is **automatically** shared and can be downloaded from another Peer **on the fly**(i.e. even when the downloading of file is in progress).
- If a peer **crashes from which some other peer is tranfering data, then receiver will acknowledge the user about it and take that chunk from some other peer who is available** with that chunk otherwise if there is no alternate then again peer will tell user that he is not able to download all file in this session so put it in unfinished download and will try to download next time.
- We have implemented **Congestion control** and **Congestion Avoidance** and also **Fast retransmit.** So our peers are able to determine the **maximum bandwidth** that it can use.
- Users use torrent because they want the file downloaders upload while downloading, then leave to all of its peers what pieces it has. We have implemented these things for this all chunks are named as filename_chunk_temp.txt . Now if a download request comes then the peer send the chunk from this temporary unfinished chunk file because he doesn't have the full file.

3 Implementation

3.1 Basic Transfer

A peer is identified by its IP address. It has the list of all users (in users.txt) and list of files shared on the entire application network(in main.txt). These two files are global and any change to it by a peer is communicated to all other peers. A peer also its shared files it has packet size defined also packet is defined although not set to any value but it is (at least) initialized socket is initialized datagram packet is initialized. A peer serves at port no. 2364 and listens on port 2363. When a peer is downloading a file it acts as a client and when a peer is supplying chunks to some other peer it is a server.

Also as the client and server threads run on different threads a peer can be a client and server at the same time. Each file download process begins with the peer broadcasting the filename to all the Peers. On receiving replies from the online users it creates a file - filename_temp.txt. This file is a temporary one and stores the addresses of the peers which have that file(or some of its chunks). It also stores the chunk numbers of the chunks shared by each peer alongside the address of the peer. If a file has 10 chunks in total, a line in filename_temp.txt would be like

csews2.cse.iitk.ac.in movie1.avi 10 0 1 2 3 4 5 6 7 8 9

Also the client appends the filename and the chunk nos of that file in unfinishedDownload.txt. The client thread then creates an instance of the Downloader thread which has the responsibilty of completing the downloading process. It first reads the unfinishedDownload.txt to get the chunk numbers of the chunks which are required to be donwloaded. It then parses the filename_temp.txt file to get the number of users sharing each chunk of this file. This information is used to sort the chunk numbers on the basis of rarity. So a chunk shared by least number of users is placed first in the chunkArray and so on.
The next task is to decide which chunk to be downloaded from which peer. We search the filename_temp.txt file for a host with that chunk and to the first peer obtained in the search the chunk is assigned - i.e. that chunk will be downloaded from this peer(say P). The busy bit for peer P is immediately set true At this point another thread is created which will download the chunk from peer P.
The packets transferred are Datagrams. The transfer is made reliable by cumulative acking. The sliding window protocol is used to send packets and the SWS is varied according to the congestion control mechanism. Each packet has 1028 bytes. The first four bytes are sequence numbers and the rest is data read from the file.
The end of a chunk is marked by a special packet containing the string "@#$". Once this packet is received, the Downloaded bit for this chunk is set true and the busy bit of peer is set to false. **Also, when the first chunk is downloaded completely, a broadcast message is sent to all the users announcing the sharing of the file by this peer and an entry is added to the shared.txt file of this peer.** Also as every chunk is downloaded it's number is appended in the shared.txt file and is removed from unfinishedDownload.txt. If all the chunks are downloaded then the reassembly of chunks is done and hence the task is complete. However, if some chunks couldn't be downloaded in this session then there number is

not erased from the unfinishedDownload.txt. The user can use the option 4 in the menu to again start another session for downloading the remaining chunks.

The testing of congestion control was done by introducing **probabilistic packet loss** by using java's Math.random().

3.2 Congestion Control

The implementation is similar to what TCP does for congestion control and avoidance. It works like determinig the capacity of network so we keep on determinig dynamically the capacity of network at any moment. This determination is done on the basis of what is the largest window size of sender at which the network start droping data so the bottleneck of network any router or switch whose buffer start overflowing so we increase the window size exponentially till we see the first loss. The window size is increased by one after each ack received, this is called slow start but actually it is exponential speed up because we send one packet receive the ack then send two packets receive the acks then send 4 and so on now at first loss detection we know that previously when we have send (current window size /2) packets then it is just fine and no loss has occurred but now at current window size network has went in congestion so the capacity lie between these two window size we fix our threshold window and again go in slow start.Initally the threshold window is fixed to some large value (we chose 64). This is congestion control. By keeping on doing this we are able to determine the window size at which the network not loose data. Which is when we enter in more than threshold size window by slow start mechanism. Then we enter in Collision Avoidance.

In **Collision avoidance** we increase the window size linearly. Sender is not agressive as before flooding the network with twice as much packets each time. Now Inspite of how much ack we receive we will increase the window size by after each RTT. Which implies that it is using the ack as to determine that one of its packet has safely reached the receiver and hence has leaved the network and so it is therefore safe to insert one extra packet without ading congestion, This is collision Avoidance. We have calculated the RTT by formula $\alpha(RTT_{predicted}) + (1-\alpha)(RTT)$ we get for some packet). We have taken α as 0.5.
The RTT of a packet is calculated by the time difference at which we have send the packet and the time at which we receive the ack

of that packet. Initially we see that the sender is agressive to determine the optimal window size sending too much data to determine when the loss has occured but now it is in avoidance zone now it is increasing the window slowly.

Fast Retransmit

Since the receiver is sending ack whenever it gets a packet inspite it is not the expected packet. So either he drop the packet or put it in his buffer or empty the buffer and write it in the file he will send the ack of last serial number for which he can send the ack. Now when the sender receive three duplicate acks of some packet. HE knows that the next expected packet in the ack was lost, even if the time out has not occured. The sender then retransmits the lost packet, and goes back to Slow Start mode.

Results

Here the network is has hardly any packet losses so we always remain in SLOW START and increase the window size to maximum till all packets are send. We rarely go in Collision Avoidance .So we tested our code with artificially loosing the packets. Which is by setting the sending probability as 0.7, 0.8 or 0.9. Then send the packet now not sending from sender and increasing the serial number is same as sending and loosing it so Now sender will receive duplicate acks also he see loss of packets so, he will easily enter in Avoidance and fast Retransmit zone. By decreasing the probabilty significantly we see that the sender window size remains very neer to 2 or 3 which is what expected by increasing the probabilty it boost up to 7 or 8, which is the expected behaviour.

3.3 Rarest First

Selecting pieces to download in a good order is very important for good performance. A poor piece selection algorithm can result in having all the pieces which are currently on offer or, on the flip side, not having any pieces to upload to peers you wish to. Rare pieces are generally only present on one peer, so they would be downloaded slower than pieces which are present on multiple peers for which its possible to download sub-pieces from dif-ferent places.

3.4 benifits of Rarest First

Some time the user which has shared the complete file leaves, so leaving only chunks on remaining peer now downloaders to upload. This leads to a very significant risk of a particular piece no longer being available from any current downloaders. Rarest first again handles this well, by replicating the rarest pieces as quickly as possible thus reducing the risk of them geting completely lost as current peers stop uploading. At that time, the peer has nothing to upload.When selecting which piece to start downloading next, peers generally download pieces which the fewest of their own peers have first, a technique we refer to as rarest first. This technique does a good job of making sure that peers have pieces which all of their peers want, so uploading can be done when wanted. It also makes sure that pieces which are more common are left for later, so the likelihood that a peer which currently is offering upload will later not have anything of interest is reduced.

4 Protocol

The communication between Peers is achieved by pre-defined strings which are passed through Datagram packets. These strings are broadcasted on the network as well as between any two peers. The general format of the header is:

magic + " " + reqStr + " " + filename

where reqStr is one of the following strings:

- String magic = "1234" : Every packet sent by any Peer using the application begins with this string.
- String newUserStr = "0": Whenever a new user wants to join the group, the IP address of one of the group members is asked. The new user obtains users.txt and main.txt files from that IP address. After obtaining these, the following string is broadcasted to all the Peers listed in users.txt file.

  Peer.magic + " " + newUserStr January 15, 2009+ " "
- String newUserRepStr = "1": On receiving the above packet, the receiving peer sends back an ack to the new user.
- String beaconStr = "2": Whenever an old user comes up, then it broadcasts the following packet.

magic + ” ” + beaconStr + ” ”

- String updateStr = ”4”: A Peer which just came up should have the most recent users.txt and main.txt files. This message is useful for that. The packet which is sent is

  Peer.magic +” ”+Peer.updateStr+” ”+ users.lastModified() + ” ” +main.lastModified()

  where **file.lastModified() gives the time (java) at which the file was last upadted.** A peer receiving this will process this message and will decide whose version of file is latest (i.e. updated at a later time.)

- String updateboth = ”5”:On receiving this String a newly up peer knows that his both main.txt and users.txt files need to be updated.

- String updateUsers = ”6”:On receiving this String a newly up peer gets to know that his users.txt file need to be updated.

- String updateMain = ”7”:On receiving this String a newly up peer knows that his both main.txt file need to be updated.

- String allreadyUpdated= ”8”:On receiving this String a newly up peer knows that both of his files are more recently updated than the peer who sent this message.

- String sendUserStr = ”9”: This message is sent by a newly up Peer to obtain users.txt file from a Peer who has the most updated version.

- String sendMainStr = ”10”:This is the request for main.txt

- String fileReqStr = ”11”: This message is broadcasted to all the Peers to obtain a file. The filename is also specified in the message. The message is as

  Peer.magic+” ”+Peer.fileReqStr+” ”+filename.

- String YfileReqRepStr = ”12”: On receiving the above broadcast message (11), if a Peer is sharing that file then it will send this message. The reply will also contain the chunks shared by the user.

- String NfileReqRepStr = ”13”: On receiving the above broadcast message (11), if a Peer is not sharing that file then it will reply with this message.

- String chunkDowStr = "14":This is a request message is sent to a specific Peer to download a particular chunk from that user.
- String fileUploadStr = "15":A Peer can share files for sharing at any point of time. Any share is notified to all the online users by broadcasting this message. This message is used for announcing the sharing of a file which is already shared by some other Peers.
- String newfileUploadStr = "16":A Peer can also share new files for sharing . Any new share is informed to all the online users by broadcasting this message.
- String newfileUploadRepStr= "17":This is an Ack for the message 16.
- String fileUploadRepStr = "19":This is an Ack for the message 15.
- String fileDeleteStr = "22":A user can also remove a particular file from sharing. This is notified to all the peers by broadcasting this message.
- String fileDeleteRepStr = "23":This is an Ack for message 22.

5 Possible improvements

- Our user interface is quite robust but introducing a GUI based interface would make it more user-friendly.
- Although the application maintains the masterChunk.txt file which has the SHA1 hashes of all the chunks, we have not implemented matching of the hashes of the chunks when they are downloaded. This matching can be introduced to further strengthen the integrity of the downloaded chunks.
- If a file is shared by a peer and it manually removes the file from its hard disk, then it also should automatically be delisted from the shared.txt file of this user.

6 References

1. Incentives Build Robustness in BitTorrent Bram Cohen bram@bitconjurer.org

2. CMU 2005 course on introductory computer networks - project 2. http://www.cs.cmu.edu/ srini/15-441/S05/assignments/project2/project2.pdf
3. http://www.bittorrent.com/protocol.html
4. http://java.sun.com/j2se/1.4.2/docs/api/java/net/DatagramPacket.html
5. http://java.sun.com/j2se/1.4.2/docs/api/java/net/Socket.html
6. http://java.sun.com/j2se/1.4.2/docs/api/java/io/File.html