Project Report

Peer-to-Peer Tube

Group 20

16116005 Aniket Singh

16116020 Harshal Mittal

16116021 Harshit Bansal

16116057 Samriddh Bhalla

16116076 Vishal Sharma

Contents

S. NO.	Particular
1.	Introduction
2.	P2P vs Client Server Model
3.	Framework
4.	File Sharing
5.	Video Streaming
6.	Our GUI
7.	Future Prospects
8.	References

Introduction

Youtube is the biggest online video sharing platform. It was created in February 2005 and since then it has only gained popularity with around 100 million hours of videos being watched every day. It is based on a client-server model. Thus to expand we need an expanding system - larger and more expensive data storage on server side, higher bandwidth, faster connections etc. The expansion must be uniform since the delay in one of these features will result in an overall delay.

Supposing one video becomes viral. It usually takes down multiple servers with it. It consumes a lot of bandwidth and a lot of unwanted buffering occurs.

Now imagine a peer to peer model. Supposing you are a user and are streaming some data. Another user requests for the same video and we can have sharing of data between the two users.

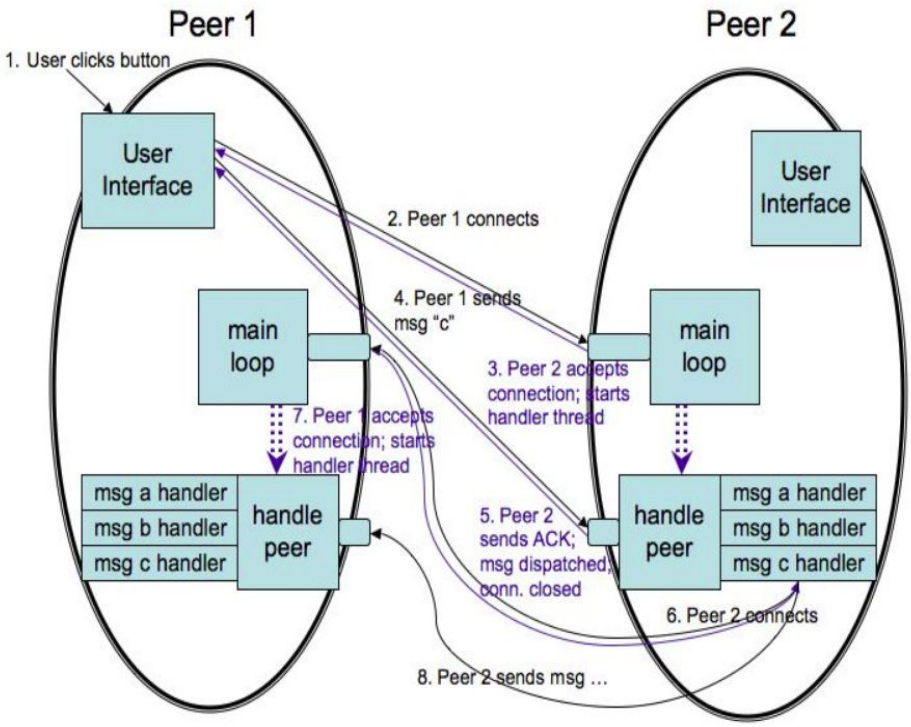
P2P vs Client Server Model

Client server architecture is a simple architecture with 2 pieces in it - a server and a client. The server has all the files in it and the client can only request for those files. P2P is a more complex architecture where each user becomes a peer of the whole group. It can become a *'client'* when it needs to access some data from the internet and also a *'server'* when some other user requests some data that it has. The p2p model is usually tougher to implement and has its own problems but its strengths are in the areas where the disadvantages of the client server model lie. Our project is to achieve the aforementioned goal in a video sharing system.

Framework

Firstly, peers are initialized. This forms an initial network/swarm of peers. Suppose a new peer wants to join this network.This new peer must know the ip address and port of one of the existing users of the network. Then the peer list of both the users is interchanged. The new peer then becomes a part of the initial peer network and any peer out of the initial peers can now access data from the new peer.

Here is a layout of the framework explained in a block diagram:



Initializing a Peer:

A new peer is initialized by the Peer class. It defines its name, to port on which it listens for connections, and maximum size of known list of peers. The constructor also initializes several fields whose use is described in the following subsections - **shutdown**, **handlers**, and **router**. The following definition illustrates the Python code for accomplishing these tasks:

```
def __init__( self, maxpeers, serverport, myid=None, serverhost =
None ):

        self.debug = 1

        self.maxpeers = int(maxpeers)
        self.serverport = int(serverport)
        if serverhost: self.serverhost = serverhost
        else: self.__initserverhost()

        if myid: self.myid = myid

        self.peerlock = threading.Lock()
        self.peers = {}
        self.shutdown = False

        self.handlers = {}
        self.router = None
```

Main Loop:

We set up a socket that listens for incoming connections from other peers. It involves creating a socket object, setting its options, binding it to a port to listen for connections.

```
def mainloop( self ):
       s = self.makeserversocket( self.serverport )
       s.settimeout(2)
       self.__debug( 'Server started: %s (%s:%d)'
                    % ( self.myid, self.serverhost, self.serverport ) )

       while not self.shutdown:
           try:
              self.__debug( 'Listening for connections...' )
              clientsock, clientaddr = s.accept()
              clientsock.settimeout(None)

              t = threading.Thread( target = self.__handlepeer)
              t.start()
           except KeyboardInterrupt:
              self.shutdown = True
              continue
           except:
              if self.debug:
                  traceback.print_exc()
                  continue

       self.__debug( 'Main loop exiting' )
       s.close()
```

Peer Connection:

The **handlepeer** method manages the requests of a newly formed connection. It reads the requests and dispatches it to the appropriate method

```
def __handlepeer( self, clientsock ):
        self.__debug( 'New child ' +   str(threading.currentThread().getName()) )
        self.__debug( 'Connected ' + str(clientsock.getpeername()) )

        host, port = clientsock.getpeername()
        peerconn = PeerConnection( None, host, port, clientsock, debug=False )

        try:
               msgtype, msgdata = peerconn.recvdata()
               if msgtype: msgtype = msgtype.upper()
               if msgtype not in self.handlers:
                      self.__debug( 'Not handled: %s: %s' % (msgtype, msgdata) )
               else:
                      self.__debug( 'Handling peer : %s: %s' % (msgtype, msgdata) )
                      self.handlers[ msgtype ]( peerconn, msgdata )
        except KeyboardInterrupt:
               raise
        except:
               if self.debug:
                      traceback.print_exc()

        self.__debug( 'Disconnecting ' + str(clientsock.getpeername()) )
        peerconn.close()
```

Routing Messages:

The peer class has a function that is used for routing messages to the destination id. It takes the destination id as parametric input and decides to which peer should the message be routed to.

When the search button is being clicked by the peer after writing the name of the file to search for, the **`sendtopeeer()`** function is being invoked with four parameters : the filename to be searched, the peerid of the receiver, the peerid of the sender and **`msgtype=QUERY`**. The function is being invoked for each peer in the list of known peers.
The *sendtopeer* function calls the router function with the peerid of the sender as argument and gets the host and port of the sender in return.

```
def sendtopeer( self, peerid, msgtype, msgdata, waitreply=True ):
            if self.router:
                  nextpid, host, port = self.router( peerid )
            if not self.router or not nextpid:
                  self.__debug( 'Unable to route %s to %s' % (msgtype, peerid) )
                  return None
            #host,port = self.peers[nextpid]
            return self.connectandsend( host, port, msgtype,
                              msgdata,pid=nextpid,waitreply=waitreply )
```

Then **`connectandsend()`** function is being invoked from inside the sendtopeer, which establishes the connection with the sender by creating a peerconn object and if the file is present, a msg containing the filename and the peerid of the sender is returned to the receiver.

```
def connectandsend( self, host, port, msgtype, msgdata, pid=None, waitreply=True ):
        file_recv = False
        if msgtype=="FGET":
                file_recv=True
        msgreply = []
        try:
                peerconn = PeerConnection( pid, host, port, debug=self.debug )
                peerconn.senddata( msgtype, msgdata )
                self.__debug( 'Sent %s: %s' % (pid, msgtype) )

                if waitreply:
                        onereply = peerconn.recvdata(file_recv, msgdata)
                        while (onereply != (None,None)):
                                msgreply.append( onereply )
                                self.__debug( 'Got reply %s: %s'
                                        % ( pid, str(msgreply) ) )
                                onereply = peerconn.recvdata(file_recv, msgdata)
                peerconn.close()
        except KeyboardInterrupt:
                raise
        except:
                if self.debug:
                        traceback.print_exc()

        return msgreply
```

File Sharing

Sharing of files is done on the P2P network. A certain protocol is followed that is given below:

Protocol Design

There are 7 types of messages that can be communicated between the peers in our implementation and each message has its seperate message handler function.

LISTPEERS : self._handle_listpeers

INSERTPEER : self.__handle_insertpeer,

PEERNAME: self.__handle_peername,

QUERY: self.__handle_query,

QRESPONSE: self.__handle_qresponse,

FILEGET: self.__handle_fileget,

PEERQUIT: self.__handle_quit

Message Description

These are 4 bit long character messages. The first 7 messages are used by nodes to interchange information and the **`REPL`** and **`ERROR`** are used to indicate the success or failure of the first 7 messages respectively. Each message has the following format:
The first 4 bytes represent the message type (which is one from above mentioned), the next 4 bytes represent the message length, and the following bytes in the **`msgdata`** comprise of the actual message data (for eg. file data in case of **`FILEGET`** message type). The following function illustrates the message:

```
def __makemsg( self, msgtype, msgdata ):
        msglen = len(msgdata)
        msg = struct.pack( "!4sL%ds" % msglen, msgtype, msglen, msgdata )
        return msg
```

Video Streaming

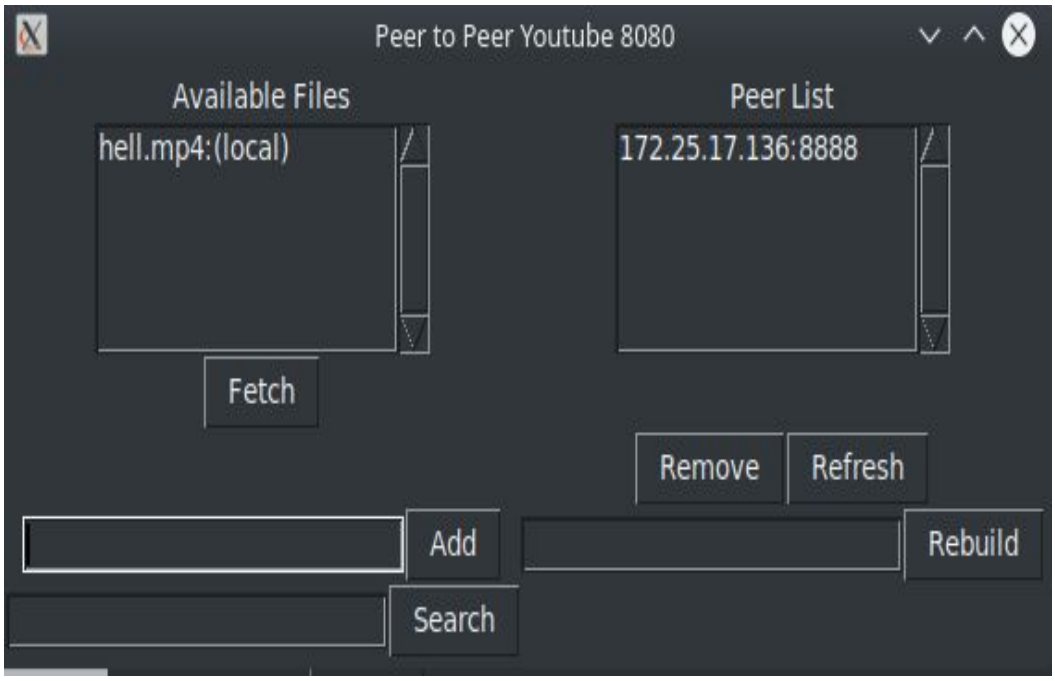
This was the main feature of our project but in order to achieve this we had to build up on the features mentioned above, most importantly file sharing. It works on a modification over the file sharing feature. Now, instead of sending a file after reading it **completely** on the client side, what we are doing is reading the video file in chunks of 2048 bytes, and sending it to the requesting peer instantly.
This buffers the content at the receiver side. We use the python **file descriptor in append mode** to write to video file as soon as chunks are received. In order to stream the video at the receiver side, we have used a python library called **python-vlc**.
The following is a snippet of the code at the ***sender side*** (this method is for handling the message type **`FILEGET`**):

```
def __handle_fileget(self, peerconn, data):
        fname = data
        if fname not in self.files:
            self.__debug('File not found %s' % fname)
            peerconn.senddata(ERROR, 'File not found')
            return
        try:
            fd = file(fname, 'r')
            filedata = ''
            while True:
                data = fd.read(2048)
                if not len(data):
                    break
                peerconn.senddata(REPLY, data)
                filedata += data
            fd.close()
        except:
            self.__debug('Error reading file %s' % fname)
            peerconn.senddata(ERROR, 'Error reading file')
            return
```

The code snippet at the ***receiver side*** is:

```
def recvdata( self, file_recv=False , fname=None):
        if not fname and file_recv:
            fname = os.path.join(os.getcwd(), fname)
        try:
            msgtype = self.sd.read( 4 )
            if not msgtype: return (None, None)

            lenstr = self.sd.read( 4 )
            msglen = int(struct.unpack( "!L", lenstr )[0])
            msg = ""
            while len(msg) != msglen:
                data = self.sd.read( min(2048, msglen - len(msg)) )
                if file_recv:
                    if not os.path.isfile(fname):
                        fd = open(fname, 'w')
                        fd.write(data)
                        fd.close()
                        print 'reading data'
                        vlc_instance = vlc.Instance('--fullscreen')
                        player = vlc_instance.media_player_new()
                        media = vlc_instance.media_new(fname)
                        player.set_fullscreen(True)
                        player.set_media(media)
                        player.play()
                    else:
                        fp = open(fname, 'a')
                        fp.write(data)
                        fp.close()

                        if not len(data):
                            print 'Data is none'
                            break
                        msg += data

                    if len(msg) != msglen:
                        return (None, None)

                except KeyboardInterrupt:
                    raise
                except:
                    if self.debug:
                        traceback.print_exc()
                    return (None, None)
                return ( msgtype, msg )
```

Our GUI



- **Add**: add local video file for sharing
- **Search** : search for a video from peers
- **Fetch** : fetch the required video file from peers if found
- **Remove** : remove a peer
- **Refresh** : check if peers are alive
- **Rebuild** : rebuild list of known peers from start

Future Prospects

- Presently the project does not have any tracker which most peer to peer networks usually have. In order to locate a file, we need to go through the network of all the peers to find whether a file is present or not. We have already implemented the backend of central tracker which will keep a list of **ip addresses vs files** stored by all the users present in the network. Now, instead of going to each peer to check whether he has a file or not, we'll be using the tracker to get the users who have the requested file, and download the files simultaneously from them. The integration of the API with the frontend is still to be done.
- Currently, we have implemented live streaming in our project. The video is being downloaded in chunks and at the same time it is being played using vlc media player. A better option could be to have the video being played directly without being downloaded, i.e. as soon as the chunk received is played, instead of getting accumulated, it gets deleted.
- Currently, the implementation is not very efficient in terms of speed of data transfer as tested in the case of large video files. The streaming can be made faster with more optimized techniques we read about.
- The implementation is in python (an interpreter). Converting the code into a compiler oriented language like C++ may improve the performance.

References

1. **Computer Networks: A Top-down Approach** by Behrouz A. Forouzan
2. **Python VLC** for video streaming
3. **Peerflix** - Streaming torrent client for Node.js
4. https://perso.telecom-paristech.fr/drossi/paper/rossi13p2p-b.pdf
5. https://github.com/ppeddabbu/P2P-File-Sharing