YamTorrent - Python BitTorrent Client
CPSC 433: Computer Networks Final Project
by:

David Hergenroeder (david.hergenroeder@yale.edu)
Micah Rosales (micah.rosales@yale.edu)
Lincoln Swaine-Moore (lincoln.swaine-moore@yale.edu)

N.B. All the code/READMEs/etc. is stored in a GitHub repo at https://github.com/dherg/yamtorrent

Background: Background on the BitTorrent Protocol

BitTorrent is a peer-to-peer Internet file sharing protocol that was designed by Bram Cohen in 2001. It has enjoyed widespread success and adoption and is currently one of the most common protocols for peer-to-peer transfer of large files.

A BitTorrent downloads a file by joining a "swarm" of hosts that are downloading and uploading the file simultaneously. This is advantageous over other traditional client/server file transfer protocols, especially for large files that are downloaded many times, because it more efficiently distributes the work of uploading.

A torrent is created by creating a .torrent file, which contains metadata that describe the file or files that make up the torrent, such as the url of the tracker, the creation date, the size of the file, the piece length, the number of pieces that make up the file, and a 20-byte SHA1 hash for each piece of the file.

The tracker is a host that maintains and makes available a list of peers for the torrent. When a BitTorrent client wants to download a file, they obtain the .torrent file, and issue a GET request to the tracker. The tracker responds with list of peers. The client can then begin to communicate with the peers directly to begin to download the file.

Peer-to-peer communication is done in BitTorrent via the following protocol. First, a client sends an initial handshake. The handshake is required to be the first message transmitted by the client. The handshake consists of a string that identifies the protocol, a 20-byte SHA-1 hash of the .torrent metainfo file, and a 20-byte peer ID that serves as a unique identifier for the client. Upon receipt of a handshake, a peer responds with the same handshake. When the client gets a handshake back, the connection is established.

Clients keep track of the state of the connection with peers in a few ways. First, a client know what pieces the peers it is connected to has because the peers send have messages and bitfield messages. A have message is a simple message that tells a client that a peer has a

piece. Bitfield messages have a payload of a series of bytes, with each bit in the string of bits representing one piece. The bitfield contains a 1 if the peer has the corresponding piece and a 0 otherwise. For example, if a file is made up of 8 pieces, a bitfield value of 10111111 would mean that the peer has all but the second piece. Bitfields allow quick transfer of availability information between peers.

In addition to availability, clients must keep track of whether or not they are open to requests from a particular peer and whether a particular peer is open to requests from the client. Whether requests are allowed or not is referred to in BitTorrent as being “choked” or “unchoked.” This state can be changed by the sending or receipt of a choke or unchoke message. In addition, peers must signal their interest in downloading before beginning to request pieces. Signalling interest is done via the interested and not interested messages. In total, for each connection with a peer, a client must keep track of the peer’s piece availability, signal its availability, keep track of whether the peer is choking or not and interested or not, and signal to the peer whether the client is choking it or not and whether the client is interested or not.

Once a client knows that a connected peer has a piece that it needs, the peer has unchoked the client, and the client has signalled that it is interested in the peer, the client may begin requesting pieces from the peer. The pieces are usually too big to download all in one request, so the pieces are downloaded in chunks (unofficially called “blocks”). The usual block size (and the one that YamTorrent defaults to) is 16KB. A request message is sent to a peer with a piece number, an offset within the piece, and a length (the block size, except in the case of the last block of the last piece, which can be an irregular size). The peer responds with a piece message, the payload of which contains the *length* number of bytes for the corresponding block of the corresponding piece. It is the client’s job to ensure that these blocks are concatenated in the right order to reconstruct the file. Once a piece is fully downloaded, the piece’s validity is verified by taking the 20-byte SHA-1 hash of the piece and comparing it to the hash that is listed in the .torrent metadata file. After every piece is downloaded, it is the client’s job to ensure that these blocks are concatenated in the right order to reconstruct the file.

Pieces can be requested in any order: from first to last, in a random order, in order of rarest to most common (i.e. pieces which the fewest peers have first), etc. In addition, other downloading strategies may be employed, such as selectively unchoking a random peer every once in a while (optimistic unchoking), or capping the number of simultaneously unchoked peers for better TCP performance.

YamTorrent Overview

An Overview of our BitTorrent Client

Our client can be run with the command "python3 YamTorrent.py <*.torrent> [--progress|--verbose]". It is important to use Python 3 when running the client because some dependencies require Python 3, as does some of the syntax used in the code written here.

YamTorrent first scans the .torrent file in the TorrentMetadata for important information, such as the tracker URL (currently we only support HTTP), the number of pieces, and the "info" section about the file to be download, from which a SHA-1 hash is computed. YamTorrent then contacts the tracker in the TrackerConnection class whose URL was found in the torrent file with a GET request using that info hash as one of several parameters. Also sent is a peer ID that the client wishes to be known as, the amount the client has left to download of the file (initially, all of it), and a flag that tells the tracker it wants the returned information to be "compact" (this seems to be standard practice). YamTorrent peer IDs are set based on the Azureus style. They take the form -YT0001- together with 12 random bytes (-YT0001- represents YamTorrent version 1).

When the tracker responds to this GET requests, it is with information about peers to which the client can potentially connect. At this point, there is a callback to the TorrentManager's class, and the TorrentManager can begin to initiate connections with as many of those peers as it chooses.

A connection to a peer is managed by the PeerConnection class, which first initiates a handshake with the peer, after which (hopefully) that peer sends along a bitfield, which allows the client to see which pieces are available for request. The PeerConnection then listens for an "unchoke" message from the peer, at which point it changes its state to reflect that it can now send messages. The TorrentManager maintains a list of the peers it has connected to, and scans it regularly for peer connections that are both idle (not currently requesting anything from a peer) and unchoked. It then searches the bitfield of that PeerConnection for a piece to request (doing so in order of priority of pieces). If it finds one it would like, it instructs the PeerConnection to start downloading that piece. There is a callback for when the PeerConnection completes downloading that piece that stores in the TorrentManager's bitfield that it now has the piece, and seeks to the appropriate offset of a file to write that piece.

The actual downloading of the piece is managed by the PeerConnection, which sends an interested, and then sends a request for the first block. When a block is received, it stores it and initiates the request for the block. It does so until all the blocks have been receives, at which point it has a callback to the TorrentManager. It can timeout if it takes too long (see design choices).

This process continues until the whole of the file has been downloaded.

Design Choices

Twisted

This was one of the most important design choices we made. Via the recommendation of a blog post, we found this library, which enables relatively easy asynchronous programming in Python.

We used the twisted to enable multiple peer connections to be operating independently and at their own pace, without opening multiple threads. Twisted permits functions to be called when messages and received, and helps facilitate callbacks that enable the TorrentManager to be notified when important events occur at the PeerConnection level, such as a bitfield is received or a piece finishes downloading.

Buffer

Messages from peers can arrive in any number of packets, which is why the length header of a message is so important. No message should be read until the entirety of its length has arrived. To managed this aspect, we employed a buffer that was simply a bytearray. Whenever a new message arrives from a peer, it is appended to the buffer, and the front of the buffer is checked to see the length of the first message that was in there. If the whole is present, the PeerConnection processes that message as is appropriate to its type, and checks to see if there are any more complete messages in the buffer. The buffer thus offers the PeerConnection a simulation of the messages arriving as complete and distinct entities, when in reality, they are split piecemeal across many packets.

Ticks

TorrentManager uses LoopingCall to call the function timer_tick every TICK_DELAY (global variable, currently set to 5) seconds. This enables the *TormentManager* to do actions regularly. These include scanning the list of idle peers to assign them pieces to request, and scanning the list of busy peers to check for timeouts. The *TorrentManager* also manages a variable called num_ticks that it increments every tick.

Download pieces in order

The TorrentManager maintains a list of pieces it desires in order. For every idle and unchoked peer found in a regular sweep of the peers, the TorrentManager looks through its desired pieces in order, and sees whether the peer has that piece. If it does, it removes the piece from the list, and tells the piece to start downloading. Currently the list is in order from first piece to last piece. While not every peer will have every piece, and so the downloading may occur slightly out of order, having the desired order be linear with the actual order is both simple and useful. In the event of say, streaming, it is useful to have earlier pieces arrive earlier, so the file can be played faster.

Timeout

Some peers are remarkably slow. To deal with this issue, we have implemented a means of timing a piece request. When a peer is told to start requesting a piece, it stores the TorrentManager's current value of num_ticks. Thereafter, when TorrentManager loops through

all the busy peers, it compares their value of the starting tick time to the current value of num_ticks. If they are more than TIMEOUT (global variable currently set to 120) seconds apart (calculated by multiplying the difference between the tick counts by TICK_DELAY and comparing that to TIMEOUT), the TorrentManager calls the PeerConnection's method cancel_current_download, which handles sending a cancel, and resetting its state to be ready to start sending a different piece. At the TorrentManager level, it puts the number of the failed piece back in the desire queue.

Input and Output

Our client can be run with the command "python3 YamTorrent.py <*.torrent>", where <*.torrent> is a standard torrent file (though currently only HTTP trackers work).

While the client is running, it maintains a file called “<suggested name>.part” (where suggested name comes from the .torrent file), which is where the pieces are stored as they are downloaded. If this already exists when YamTorrent is run, it is overwritten.

When the file has finished downloading it is saved as “<suggested name>” and “<suggested name>.part” is deleted, unless something already exists in the directory with that name, in which case it keeps “<suggested name>.part”.

Make sure you use a VPN if you are testing it!

For an example torrent file, try The Latest Ubuntu Release.

Dependencies

We used the following modules:

sys
requests
hashlib
bencodepy
struct
socket
os
bitstring
twisted
enum
urllib
glob
pybtracker

ProgressBar (pip install ProgressBar2)

Most of these should be available through any standard distribution, and any that are not (e.g. twisted, bencodepy, etc.) are available via pip. All of the modules can be installed at once using `pip install -r requirements.txt`

Achievements and Impacts

Our client successfully decodes the torrent file, interacts with the tracker, decodes the list of peers from the tracker, contacts, connects with, and maintains state with multiple peers simultaneously, requesting and saving pieces based on a list of pieces needed.

Here, in a slightly outdated screenshot (the look of the output has changed somewhat), it is possible to see that pieces requests/receipts are working well!

```
INFO:PeerConnection:send_interested to 163.172.27.114:51413
INFO:PeerConnection:starting download piece 1596
INFO:PeerConnection:send_interested to 216.15.74.42:51413
INFO:PeerConnection:starting download piece 1597
INFO:PeerConnection:send_interested to 99.34.10.137:51413
INFO:TorrentManager:PEER IS CANCELLING PIECE 1497
INFO:PeerConnection:send_cancel piece 1497 offset=131072 length=16384 to 31.17.108.93:51413
INFO:PeerConnection:piece number 1584 complete
INFO:PeerConnection:validating piece 1584: hash matched!
INFO:TorrentManager:received_piece 1584 from 97.127.20.90:65520.
INFO:PeerConnection:piece number 1571 complete
INFO:PeerConnection:validating piece 1571: hash matched!
INFO:TorrentManager:received_piece 1571 from 109.195.147.150:50839.
INFO:PeerConnection:piece number 1574 complete
INFO:PeerConnection:validating piece 1574: hash matched!
INFO:TorrentManager:received_piece 1574 from 217.240.68.149:16881.
INFO:PeerConnection:starting download piece 1497
INFO:PeerConnection:send_interested to 31.17.108.93:51413
INFO:PeerConnection:starting download piece 1598
```

A few notes:

- Downloads are begun and finished in an interspersed way. That is, the downloading of the data doesn't happen in a synchronous way.
- "PEER IS CANCELLING PIECE 1497" demonstrates a timeout in action. The peer that was supposed to be dealing with 1497 was taking too long, and so the TorrentManager cancelled it. It is possible to see, however, that the piece was added back to the desired list, because some lines below there is the line "starting download piece 1497"
- "Validating piece 1584: hash matched!" means that the piece has finished receiving piece 1584, and that the hash of the data received matched what was in the original .torrent file. That piece can now be save, and the peer is available for the TorrentManager to request another block (unless it has been choked).

Here is what the output looks like now with the "--verbose" flag:

```
DEBUG:PeerConnection:rcv_piece: id=16 off=409600 len=16392
DEBUG:PeerConnection:send_request piece 16  offset=425984  length=16384 to 90.127.223.42:51413
DEBUG:PeerConnection:rcv_piece: id=24 off=147456 len=16392
DEBUG:PeerConnection:send_request piece 24  offset=163840  length=16384 to 88.198.224.202:51413
DEBUG:PeerConnection:rcv_keepalive
DEBUG:PeerConnection:rcv_piece: id=11 off=327680 len=16392
DEBUG:PeerConnection:send_request piece 11  offset=344064  length=16384 to 188.235.255.30:51413
DEBUG:PeerConnection:rcv_piece: id=18 off=245760 len=16392
DEBUG:PeerConnection:send_request piece 18  offset=262144  length=16384 to 82.228.241.32:51413
DEBUG:PeerConnection:rcv_piece: id=27 off=16384 len=16392
DEBUG:PeerConnection:send_request piece 27  offset=32768  length=16384 to 173.74.186.248:51413
DEBUG:PeerConnection:rcv_piece: id=25 off=0 len=16392
DEBUG:PeerConnection:send_request piece 25  offset=16384  length=16384 to 128.199.55.34:51413
DEBUG:PeerConnection:rcv_piece: id=20 off=245760 len=16392
DEBUG:PeerConnection:send_request piece 20  offset=262144  length=16384 to 195.138.83.92:60843
DEBUG:PeerConnection:rcv_piece: id=23 off=163840 len=16392
DEBUG:PeerConnection:send_request piece 23  offset=180224  length=16384 to 77.175.195.46:16881
DEBUG:PeerConnection:rcv_piece: id=22 off=81920 len=16392
DEBUG:PeerConnection:send_request piece 22  offset=98304  length=16384 to 85.93.129.132:51413
DEBUG:PeerConnection:rcv_piece: id=26 off=16384 len=16392
DEBUG:PeerConnection:send_request piece 26  offset=32768  length=16384 to 188.40.81.200:57594
DEBUG:PeerConnection:rcv_piece: id=8 off=507904 len=16392
INFO:PeerConnection:piece number 8 complete
INFO:PeerConnection:validating piece 8: hash matched!
INFO:TorrentManager:received_piece 8 from 212.51.144.37:51413.
DEBUG:PeerConnection:rcv_piece: id=16 off=425984 len=16392
DEBUG:PeerConnection:send_request piece 16  offset=442368  length=16384 to 90.127.223.42:51413
DEBUG:PeerConnection:rcv_piece: id=24 off=163840 len=16392
DEBUG:PeerConnection:send_request piece 24  offset=180224  length=16384 to 88.198.224.202:51413
```

In this example, it is possible to see the individual block requests being made by the different PeerConnections (the offset indicates which block is being requested).

Here is what the output looks like now with the “--progress” flag:

```
~/dev/yamtorrent master 1m 8s
(venv) ❯ python YamTorrent.py ubuntu2.torrent --progress
  2% (  32 of 1112) |#                                                  | Elapsed Time: 0:00:51 ETA:  0:17:22
```

Possible Improvements

Areas for Future Improvement / Possible Future Features

Order for requesting pieces
As explained earlier, right now we are requesting pieces in order. Some clients do not do this, and instead opt for strategies such as rarest-first, where they scheme which pieces they should try hardest to get.

Endgame
When most pieces have been downloaded, sometimes clients engage in an endgame where they send out requests to multiple peers so as to guarantee that one will give it to them. They then send out cancels when they receive the piece.

Torrents with multiple files
Currently, if there are multiple files in a tracker, they will all be downloaded as one file. Theoretically, the client should be able to split them as appropriate, based on information in the .torrent file.

Supporting UDP trackers
Many .torrent files use UDP trackers, so this would make our client more versatile. See Problems Encountered for more details.

Pausing and later restarting torrents
This is useful if you can only download part of something at a time--perhaps due to moving in and out of wireless service, for instance.

Creating torrents
Many torrent clients allow you create a .torrent file.

DHT and peer to peer discovery
Currently, the client only finds peers via the tracker. But other clients sometimes are able to communicate with eachother to discover more peers that the tracker might not give. Distributed Hash Tables (DHT) are also sometimes used for this purpose. Either one would speed up the download by permitting more peer connections.

Contacting multiple trackers
Many .torrent files contain not just one announce tracker, but a list of announce trackers. Contacting multiple trackers would also allow us to expand our set of peers, and hence speed up the download.

Problems Encountered

UDP
Currently we only support HTTP trackers, and not UDP trackers. We tried to support UDP trackers, but encountered several issues. Firstly, many of the UDP trackers simply did not respond to our messages. After finding torrents that used UDP trackers that did provide a response, we found that the responses they gave were extremely unreliable. The responses we received exhibited strange characteristics. First, the tracker returned only one ip address, and then returning the same ip address multiple times with multiple ports. These difficulties made us choose not to pursue a more complete UDP tracker implementation.

Yale's Firewall
Yale appears to attempt to block traffic associated with torrenting, so often it will be difficult to connect to a tracker or to peers. This issue can be mostly circumvented by using a VPN.

Peers not behaving in useful ways
Sometimes it will be possible to connect to peers, but they will not send a bitfield, or do not unchoke. This makes things slower. Timeouts help avert this problem, but not totally.

Lost Connections
Sometimes connections to peers are inexplicably lost.