

Overview

ByteTorrent is a distributed file-sharing system inspired by BitTorrent. ByteTorrent enables numerous peers to swarm and share a file in an efficient manner. Each peer connects to one or more peers, the more peers you are connected to the quicker you can download the file (unless your bandwidth is fully utilised). In the testing environment used during the development of the protocol each peer will connect to every other peer forming a full-mesh topology. However a full-mesh would not be feasible when used over the Internet where there could potentially be thousands of users sharing the file as this could actually degrade performance. ByteTorrent reduces the upstream bandwidth and server load for the original sharer of the file by enabling peers to connect to each other.

This document will outline the two protocols used by ByteTorrent the Tracker Protocol and the Peer-to-Peer Protocol, how to compile and run the ByteTorrent Tracker and Client, and explain the software design and limitations of our solution. This document covers the Tracker first and then the Peer-to-Peer Client.

Overview	1
Tracker Protocol Specification.....	1
Common Terminology	1
Registering a complete file	3
Registering a partial file.....	3
Getting a list of files.....	5
Requesting file information	5
Getting a list of peers with a requested file.....	6
Unregister from the Tracker	6
Tracker User Manual	9
Compiling the ByteTorrent Tracker	9
Running the ByteTorrent Tracker	9
Tracker Design Documentation	11
Class Diagram.....	11
Class List and Functionality.....	11
Key Design Choices.....	12
Limitations	12
Peer-to-Peer Protocol Specification	13
File Request	14
File Offer.....	14
Piece Request	15
Piece Transfer.....	15
Piece Update	17
Finished.....	17
Error Message	18
Client User Manual	19
Compiling the ByteTorrent Client	19
Running the ByteTorrent Client.....	19
Client Design Documentation.....	24
Class Diagram.....	24
Key Design Choices.....	28
Development of the Design	28
Breaking a file into pieces and transferring the pieces	29
Requesting a Piece	29
Putting the Pieces Back Together.....	30
GUI	30
Error Handling	30
Limitations	31

Tracker Protocol Specification

The Tracker Protocol a text-based client/server protocol which is used to coordinate the swarm of peers. It uses six different messages to do this 'RegisterComplete', 'RegisterPartial', 'GetList', 'GetFileInfo', 'GetPeers' and 'Unregister'. It is implemented on top of TCP/IP for reliability and scalability. The Tracker protocol is text based protocol due to it being easier to implement and test. Due to the low amount of traffic to the Tracker it doesn't need the efficiency that a binary protocol could potentially offer.

Each message consists of text-based fields using '|' to delimiter each field. The first field in each request is the command to be executed, then a '|' and a list of parameters depending on the message passed delimited by '|'.

The Tracker should disconnect each connection after the request is finished. This is because a client just sharing a file will keep the connection open indefinitely if the tracker doesn't disconnect it leading to a potential denial of service attack.

Common Terminology

Filename: the name of the file that is to be shared

File Size: the size of the file in bytes

Piece Size: the size of each piece in bytes except for the last piece which needs to be calculated. The piece size is variable and is determined by the client that first registers the complete file with the tracker ($1 - 2^{31}$ bytes).

We recommend a variable piece size for the files transferred. This to maximize the throughput by keeping the overhead low, but still keeping the number of pieces high enough that swarming will happen. We recommend the following piece sizes depending on file size:

File size	Piece size
>2gb	2mb
1gb-2gb	1mb
512mb-1gb	512kb
256mb-512mb	64kb
128mb-256mb	32kb
64mb-128mb	16kb
8kb-64mb	8kb
<8kb	File size

Port: the port a peer is listening to for incoming connections

Hash: the MD5 of the complete file used to check the integrity of the downloaded file. We intended to have an MD5 hash of the complete file however it was felt that it was beyond the requirements of the projects and it was omitted from our implementation. As our protocol uses smaller pieces than BitTorrent does, having hashes of individual pieces would have required too much overhead. This unfortunately means that if a peer is acting maliciously it won't be detected until you have the complete file. Errors in the transferring of the actual pieces from none malicious peers should be detected by TCP.

Registering a complete file

A client sharing a complete file sends a 'RegisterComplete' command to the Tracker with information about the file it is sharing and the port which other peers need to connect, to download the file. The Tracker stores this information along with the source IP as a complete file record for future reference. The source IP is determined from the connection because a client behind a NAT router may not know its live IP address.

If the client has previously registered a partial file using the 'RegisterPartial' command the partial record should be removed and replaced with the complete file record. If the piece size is greater than the file size the tracker should not store the record and should respond with an error message.

Request:

RegisterComplete | Filename | File Size | Piece Size | Port | Hash

Response:

OK if the request was executed successfully or **error|error message** if an error occurs; where **error message** is an explanation of what went wrong.

Example:

Client: RegisterComplete|test1.doc|12324|8192|22000|
AF59ED5F98F210AFE87FE34A987239AF
Tracker: OK

Registering a partial file

A client downloading a file sends a 'RegisterPartial' command to the Tracker so that other peers can connect to the client and downloaded the pieces that it has already downloaded. 'RegisterPartial' is needed in addition to 'RegisterComplete' because the Tracker needs to distinguish between complete and partial files. If the piece size is greater than the file size the tracker should not store the record and should respond with an error message.

Request:

RegisterPartial | Filename | File Size | Piece Size | Port | Hash

Response:

OK if the request was executed successfully or **error|error message** if an error occurs; where **error message** is an explanation of what went wrong.

Example:

Client: RegisterPartial|test2.doc|12345|8192|22010|
AF59ED5F98F210AFE87FE34A987239AF

Tracker: OK

Getting a list of files

A client wanting to know which files are available sends a 'GetList' command to the Tracker. The Tracker responds with a list of files that currently have at least one complete file registered.

Request:

GetList

Response:

Filename 1 | Filename 2 | ... | Filename n

Example:

Client: GetList

Tracker: test1.doc|test2.doc|test3.doc

Requesting file information

Before a client can download a file it needs to know the file size and the piece size this is done by sending a 'GetFileInfo' command to the Tracker.

Request:

GetFileInfo | Filename

Response:

Filename | File Size | Piece Size | Hash

Example:

Client: GetFileInfo|test2.doc

Tracker: test2.doc|12342|8192|AF59ED5F98F210AFE87FE34A987239AF

Getting a list of peers with a requested file

A client that is ready to download a file sends a 'GetPeers' command to the Tracker. The Tracker should return a list of IP addresses and ports of peers who are sharing the requested file.

A client will periodically connect to the tracker to get an updated list. The client is responsible for filtering the list as needed

Request:

GetPeers | filename | file size | piece size | hash

Response:

IP 1:port | IP 2:port | IP3:port |...| IP n:port n

The tracker returns a list of IP and listening ports for all peers sharing the file. The IP and port for each peer is separated by a ':'. The first peer listed should have the complete file all other peers listed can have either a complete or partial file.

Example:

Client: GetPeers|test2.doc|12342|8192|
AF59ED5F98F210AFE87FE34A987239AF

Tracker: 192.168.1.10:22000|192.168.1.53:22010|192.168.1.232:23223

Unregister from the Tracker

Before shutting down a client will issue an 'Unregister' command to the Tracker informing it that no more peer-to-peer connections are to be made to the client for the specified port and all file records are to be removed. Existing peer-to-peer connections are still able to continue unless the client disconnects them.

Request:

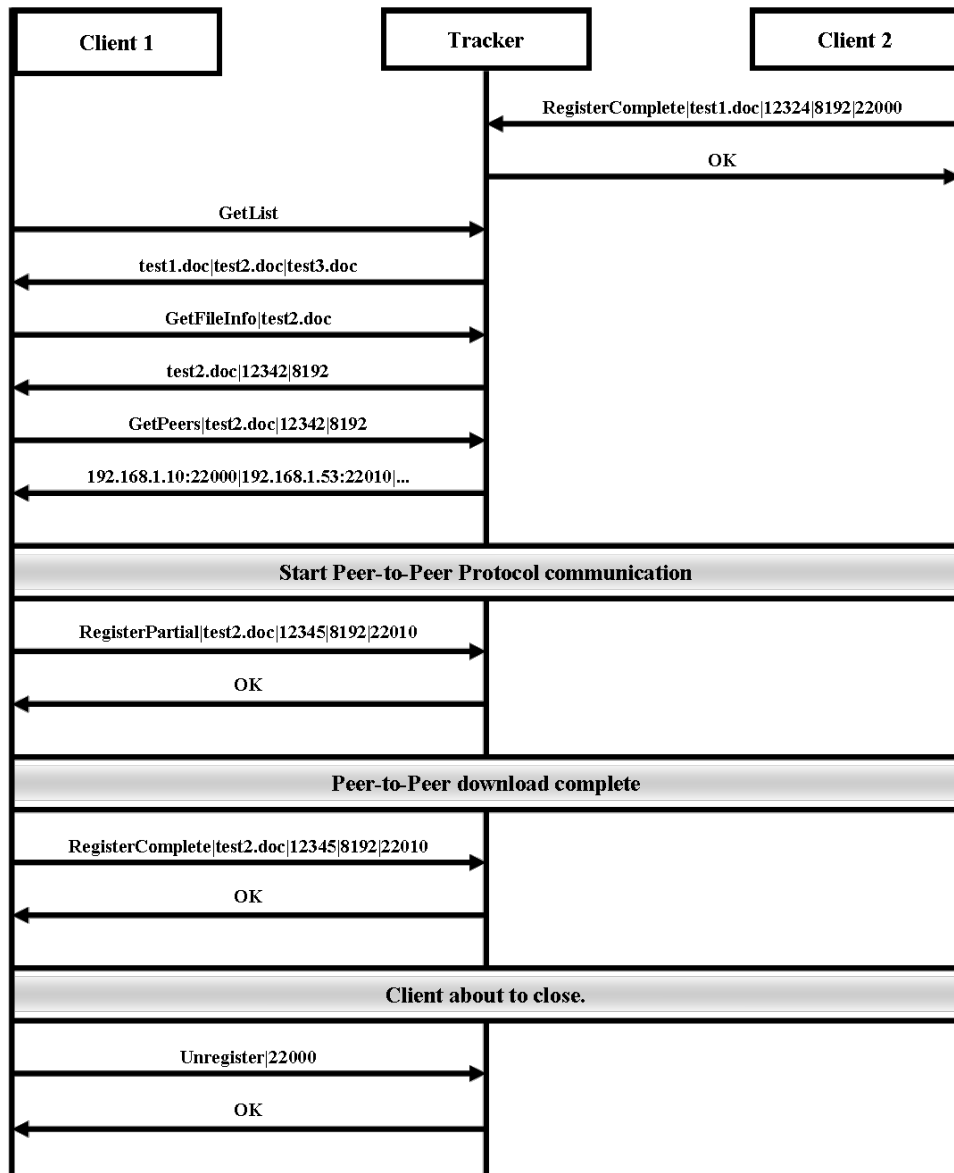
Unregister | port

Response:

OK if the request was executed successfully or **error|error message** if an error occurs; where **error message** is an explanation of what went wrong.

Example:

Client: Unregister|22000
Tracker: OK



Tracker User Manual

The Tracker is a central registry for all peers on the network. It contains information about which files are available on the network and by which peers.

Compiling the ByteTorrent Tracker

The source code can be found on the CD in \Tracker\ . This folder also contains a precompiled executable jar file (Tracker.jar) and the manifest file used in the creation of the jar file.

To compile the Client, simply run **javac *.java** on the source files supplied.

To make the jar file, either run **makejar.bat** or **jar cmf main.txt Tracker.jar *.class** after running **javac *.java**.

Running the ByteTorrent Tracker

The Tracker can be started in two different ways and has a normal and debug mode:

Starting the Jar File Version:

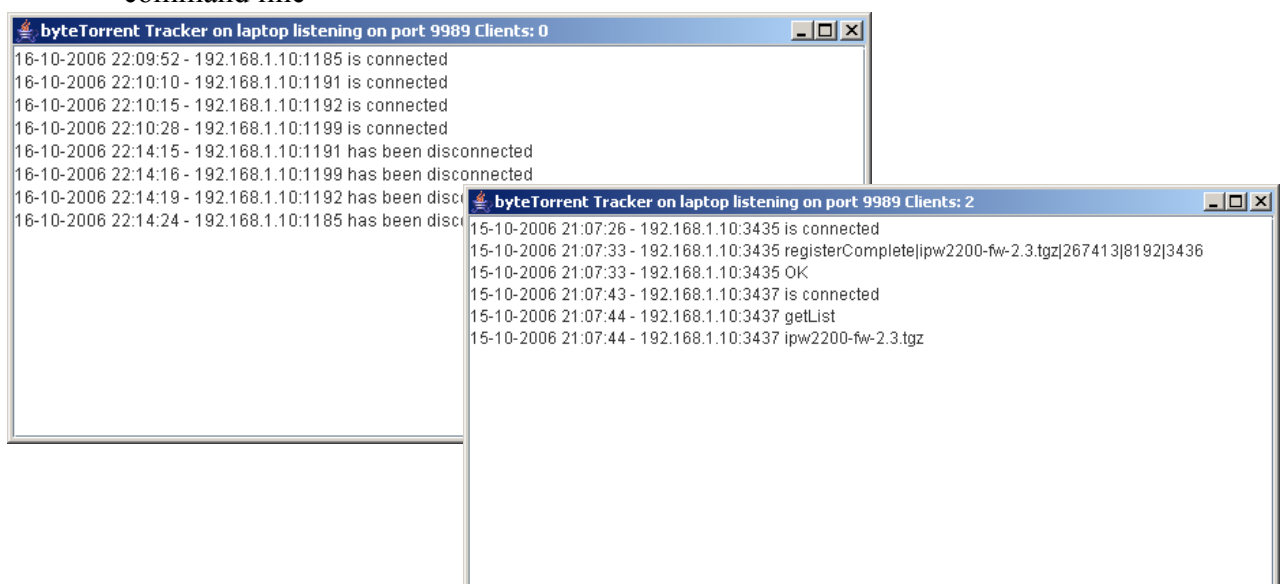
Enter **java -jar Tracker.jar** at the command-line or by **double clicking Tracker.jar** from with the GUI (if supported by OS) to start the Tracker.

To start the jar file in debug mode enter **java -jar Tracker.jar debug** at the command line.

Starting the Class File Version:

Enter **java Tracker** at the command-line to start the Client.

To start the class version in debug mode enter **java Tracker debug** at the command line

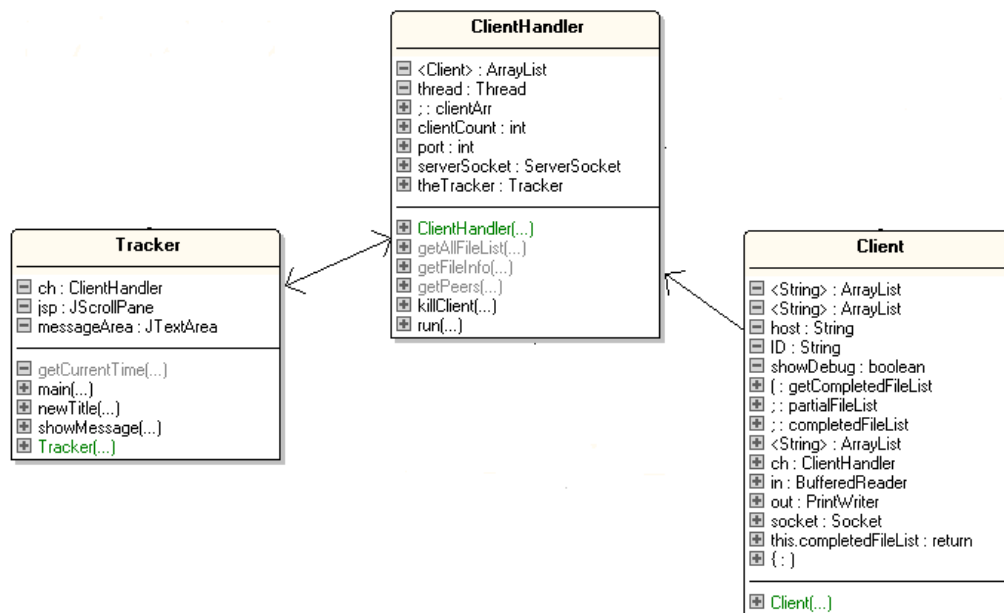


Normal Mode (right): lists all connecting and disconnecting clients and the time of these events.

Debug Mode (left): in addition to listing all connecting and disconnecting clients, the debug mode also lists the actual messages passed to and from the clients like the shared/downloaded files, and the syntax of the received and sent messages.

Tracker Design Documentation

Class Diagram



Class List and Functionality

Class Name	Functionality
Tracker	Tracker class represents the tracker of the program in form of a GUI. It acts as a main class. It shows information about connected clients and disconnected clients with timestamp on each occurrence.
ClientHandler	ClientHandler class performs the task of handling each connection made by client. It accepts incoming connections, terminates connections, and keeps track of all clients. This class also manages the requests made by clients and provides appropriate responses to them.
Client	Client class represents the connected clients. Each client object is created for each connected client. This class holds client's shared files.

Key Design Choices

- The use of threading by implementing the Runnable interface. This allows the tracker to accept and handle multiple incoming connections from clients. Each client can run and send requests to the tracker simultaneously
- The ClientHandler object holds an ArrayList of Client objects. Each Client object represents a connected client and holds its own shared files. This method simplifies the task of keeping track of connected clients and their respective shared files.
- Using unallocated port number to avoid conflict with other known port numbers which are reserved for various applications such as 23 for Telnet, 53 for Domain Name System (DNS), 80 for HTTP, and so on. In this case the tracker listens for connections on port number 9989.
- The Tracker can run in a 'normal' and a 'debug' mode. The 'normal' mode only shows when the connections and disconnections occur. The 'debug' mode also shows all communication.

Limitations

- If clients disconnect from the tracker without using the supplied 'Unregister' command, the tracker will think they are still connected and keep offering their shared files.
- If a client is used at the same computer that is running the Tracker, it is important that the user is connecting to the tracker using its local IP address (if used on the LAN) or the live IP (if used on the Internet), and not localhost, even if the Tracker is running locally.
- The user does not have the functionality to change the port number the tracker is listening on as it has been hard coded. If the port is in use, no client will be able to connect to the tracker.
- You are unable to share different files with the same name due to the design of the protocol.
- The Tracker does not fully implement all features of the Tracker Protocol for example it requires a client to constantly keep the connection open to function properly.
- Even if all the pieces are available for the file on the network you are only able to start downloading if there is at least one client with the complete file registered

Peer-to-Peer Protocol Specification

The Peer-to-Peer Protocol is used to handle the actual downloading of files in the ByteTorrent network. It is a peer-to-peer protocol but relies on the Tracker Protocol for basic co-ordination. Each peer is considered to have a server and a client component. The client component is used to request and download files, and the server component to share and transfer files.

Due to the distributed nature of the protocol it is highly fault tolerant, if one peer fails you can continue getting the file from the other peers. The ByteTorrent Peer-to-Peer Protocol is a binary protocol implemented on top of TCP/IP for reliability and scalability. As TCP does a CRC we have not implemented one in our protocol. The binary design was chosen due to efficiency and because the data to be transferred is binary in nature, a text-based protocol would not have made as much sense. Each peer is able to connect to multiple peers to share files in an efficient manner Figure 1

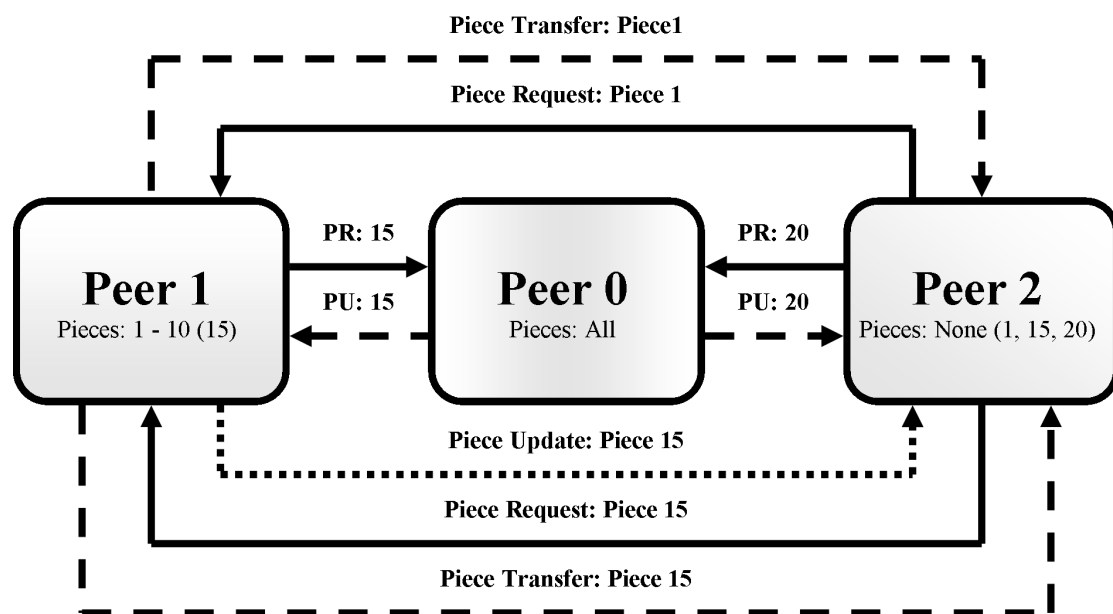


Figure 1: Multiple Peers sharing a file (initial handshaking excluded for simplicity)

File Request

The File Request is sent by a Client to request a file from the Server. The message contains a file request header, the filename, file size and piece size as retrieved from the Tracker using the Tracker protocol.

Request (Client → Server):

[Type 01] [File Name] [File Size] [Piece Size]

Type: a byte containing the type for a File Request '0x01'

File Name: a UTF formatted string containing the name of the requested file

File Size: a long containing the file size of the requested file

Piece Size: an integer containing the size of each piece to be transmitted, except for the last piece which can be smaller

Response:

Expected response is a File Offer [Type 02] if the file is available or an Error Message [Type 255] if an error occurs.

File Offer

A File Offer is used by Server in response to a File Request sent by a Client. The message contains a file offer header, the total pieces, the number of pieces the client has, and the complete list of pieces the client has, presented as '0' if the client does not have the piece, and a '1' if the client have the piece available.

Request (Client ← Server):

[Type 02] [Total Pieces] [Have Pieces] [List of Pieces]

Type: a byte containing the type for a File Offer '0x02'

Total Pieces: integer containing the total number of pieces of the offered file.

Have Pieces: integer containing the number of pieces offered

List of Pieces: zero or more bytes containing a list of pieces that the Server currently has. If 'Have Pieces' is equal to 0 or 'Num of Pieces' the 'List of Pieces' is not transmitted to keep overhead as low as possible.

If 'Total Pieces' is evenly divisible by 8 the list is 'Total Pieces' / 8 bytes long otherwise it is ('Total Pieces' / 8) + 1 bytes long. If the server has pieces 0, 8 and 10 'List of Pieces' would be 2 bytes long the first byte would contain [10000000] and the second [10100000]. The last byte is padded with 0s as needed. Please note that a 0 based index is used for pieces.

Response:

Piece Request [Type 03] or an Error Message [Type 255] if an error occurs.

Piece Request

A Piece Request is used by a Client to request a piece from the list of pieces previously obtained from Server's File Offer for download. The message contains a piece request header and the index of the piece the client requests. A client is able to request another piece before the first requested piece is returned but the Server does not guarantee they will be returned in the requested order.

Request (Client → Server):

[Type 03] [Piece Number]

Type: a byte containing the type for a Piece Request '0x03'

Piece Number: an integer containing the 0 based index for the requested piece

Response:

Piece Transfer [Type 04] or an Error Message [Type 255] if an error occurs. If a client fails to receive a piece requested in a reasonable time or receives an Error Message it should consider the peer has failed and request the piece from another peer.

Piece Transfer

A Piece Transfer is sent from a Server to a client in response to a Piece Request. The message contains a piece transfer header, the piece number requested in the piece request, the piece size, and the payload of the transferred data equal to the piece size. There is no CRC checking of the Piece Data as we rely on TCPs inbuilt error checking.

Request (Client ← Server):

[Type 04] [Piece Number] [Piece Size] [Piece Data]

Type: a byte containing the type for a Piece Transfer '0x04'

Piece Number: an integer containing the 0 based index for the piece being transmitted.

Piece Size: an integer containing the size of the Piece Data in bytes

Piece Data: one or more bytes containing part of the file being transmitted

Response:

There is no acknowledgement of a Piece Transfer to keep overhead low.

Piece Update

If a peer has downloaded a new piece its Server informs the attached Clients of the newly available piece. This does not happen if a 'List of Pieces' was not sent in the File Offer response because it had all pieces. The message sent across contains a piece update header, and the updated piece number.

Request (Client \leftarrow Server):

[Type 05] [Piece Number]

Type: a byte containing the type for a Piece Update '0x05'

Piece Number: an integer containing the 0 based index of the piece that the Server now has available

Response:

No response expected

Finished

If a peer has finished downloading it sends a finished message to the peers to which it has been downloading from so that they can shut down gracefully.

Request (Client \rightarrow Server):

[Type 06] [Piece Number]

Type: a byte containing the type for a Finished command '0x06'

Response:

No response expected

Error Message

An Error Message is sent between the peers if an error occurred to which the other peer should be informed at which point the connection should be gracefully disconnected. The message passed across is an error message header, the error code and a human readable message.

Request (Client \leftrightarrow Server):

[Type 255] [Error Code] [Message]

Type: a byte containing the type for an Error Message '0xFF'

Error Code: a byte containing the error code for the error

Message: a UTF formatted String containing a human readable error message that is appropriate for the type of error

Response:

No response expected

Common Errors:

Code	Message	Description
01	Invalid Index	Used when a requested piece index is below 0 or above the last piece index
02	File Not Found	Used when the requested file is not found
03	Invalid Request	Used if a client attempts a request that is not defined
04	Peer Busy	Used if the peer is receiving more request than it can handle in a reasonable amount of time
FF		Can be used for any non-standard message

Client User Manual

The Client is used to connect to the tracker and also to other peers.

Tracker: a Tracker is a central registry for all peers on the network. It contains information about which files are available on the network and by which peers.

Peers: a peer is another client on the network that is able to upload and download a file that is being share.

Compiling the ByteTorrent Client

The source code can be found on the CD in \Client\ . This folder also contains a precompiled executable jar file (ByteTorrent.jar) and the manifest file used in the creation of the jar file.

To compile the Client, simply run **javac *.java** on the source files supplied.

To make the jar file, either run **makejar.bat** or **jar cmf main.txt ByteTorrent.jar *.class** after running **javac *.java**.

Running the ByteTorrent Client

The Client can be started in two different ways:

Note: the executable jar file and the class files on the CD will need to be copied to a hard drive before being run as the program saves the downloaded files in the directory which is run from.

Starting the Jar File Version:


Enter **java -jar ByteTorrent.jar** at the command-line or by **double clicking ByteTorrent.jar** from with the GUI (if supported by OS) to start the Client.

Starting the Class File Version:

Enter **java ByteTorrent** at the command-line to start the Client.

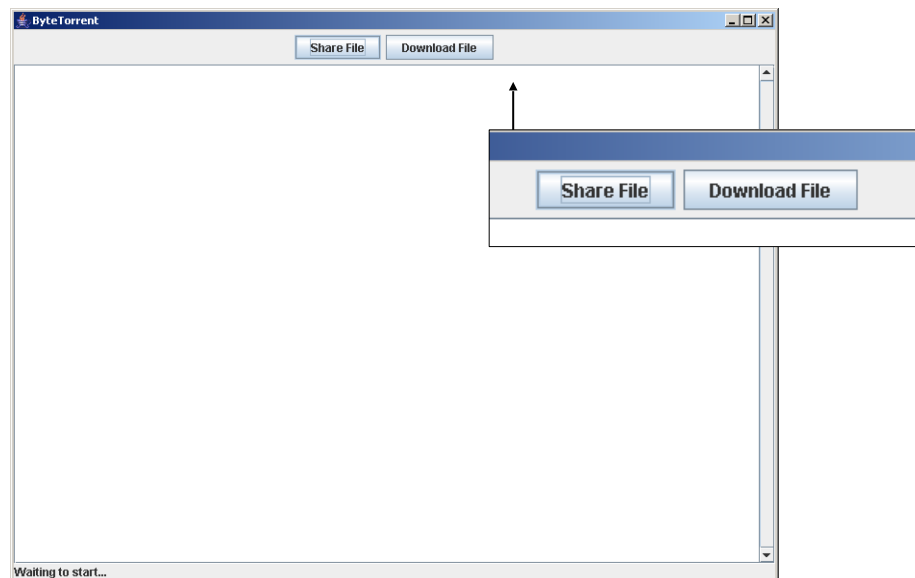
Once started the client GUI displays a screen prompting for the Tracker IP. The IP of the local computer is the default entry because it was frequently tested with the Tracker and the clients on the same computer. If the Tracker isn't running please start it as described in the previous section.

Input ✕

 **Please Enter the Tracker's IP Address:**

Note: If using the client on the same computer as the Tracker and Clients on other computers will be connecting to the Client the computers real IP address and not a loopback address.

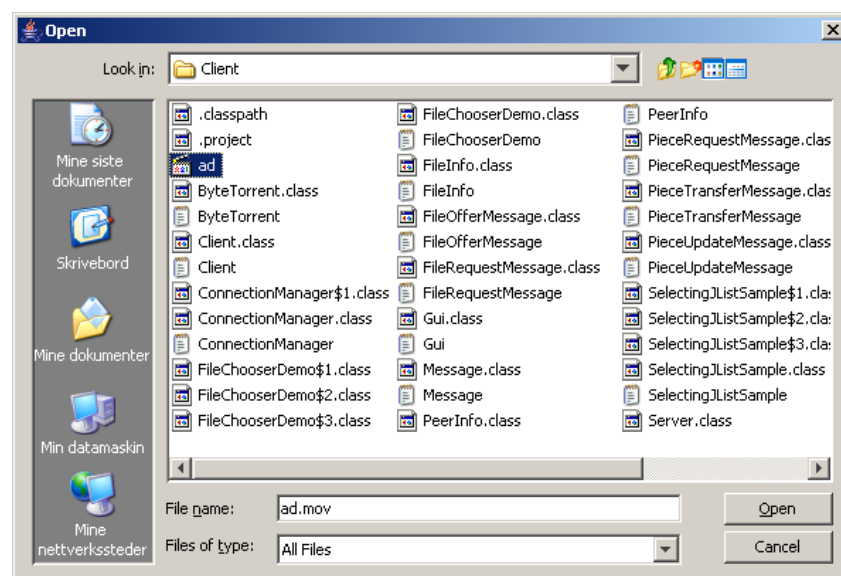
After typing the IP address of the Tracker, the main ByteTorrent screen shows:



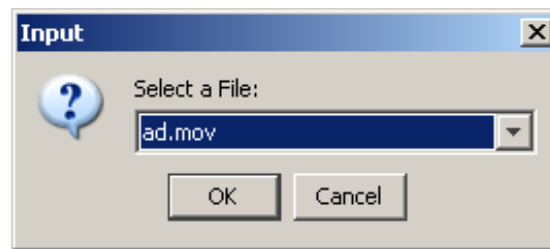
This gives the user two choices, either to share a file, or to download a file.

If you click 'Share File', a file chooser appears letting you can explore your hard drive, find the file you want to share, and then double click the filename or select the file and click open. Sharing a file means that you are going to become the Seed for the file and others will be able to connect to you.

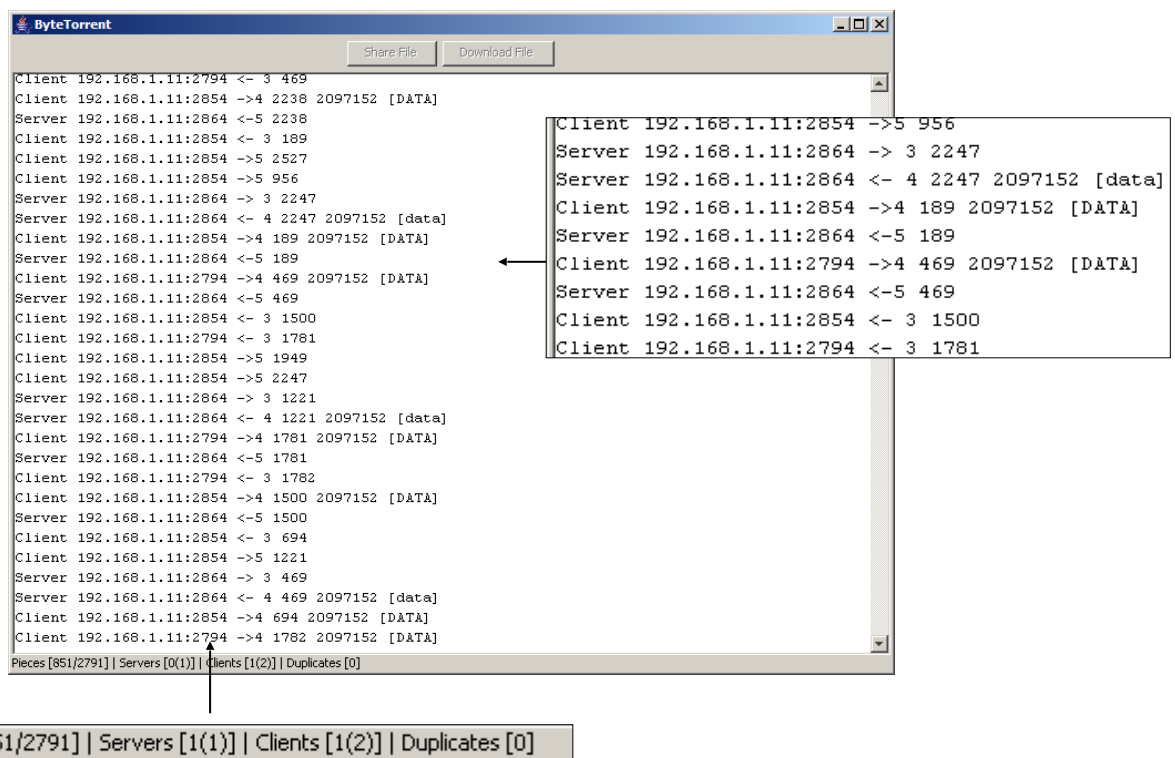
Seed: a seed is a peer that has the complete file to share



To download a file, the “download file” button is chosen from the main menu. This will give the users a dropdown list of all files shared by clients connected to the same Tracker.



Click OK on the chosen file, and the file-transfer starts.



The status bar at the bottom of the window will show you important information about the current progress:

Pieces[have/total]: Shows you how many pieces you have download and how many pieces there are in the complete file. Not shown if you are the seed.

Servers[current(maximum)]: Shows the number of Server threads currently running which is also the number of peers that have connected to you. The maximum number that has been running is also shown.

Clients[current(maximum)]: Shows the number of Client threads running which is also the number of peers you have connected to. The maximum number that has been

running is also shown. The number of Clients should be 1 higher than the number of servers.

Duplicates[current]: Shows the current number of duplicate pieces that have been downloaded. Occasionally a peer will request a particular piece from multiple peers this number shows how well our next piece algorithm is performing

Note: if the number of pieces you have stops increasing and the current number of clients is below the max number of clients it most likely indicates that the seed has fail. If a new seed joins the network downloading should resume.

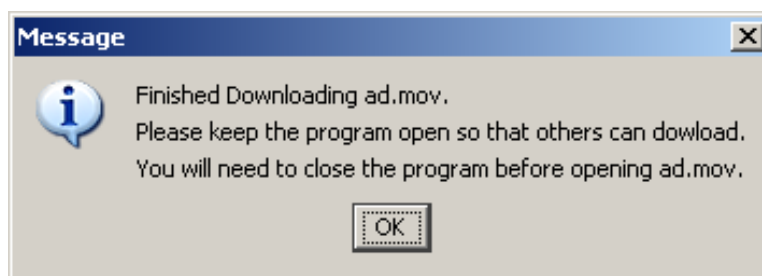
The text area lets you see which pieces are being transferred between peers in the following format:

[Server/Client] [<-/->] 'remote IP': 'remote port' 'message type' 'parameters'

<- means the data was sent by the Server/Client

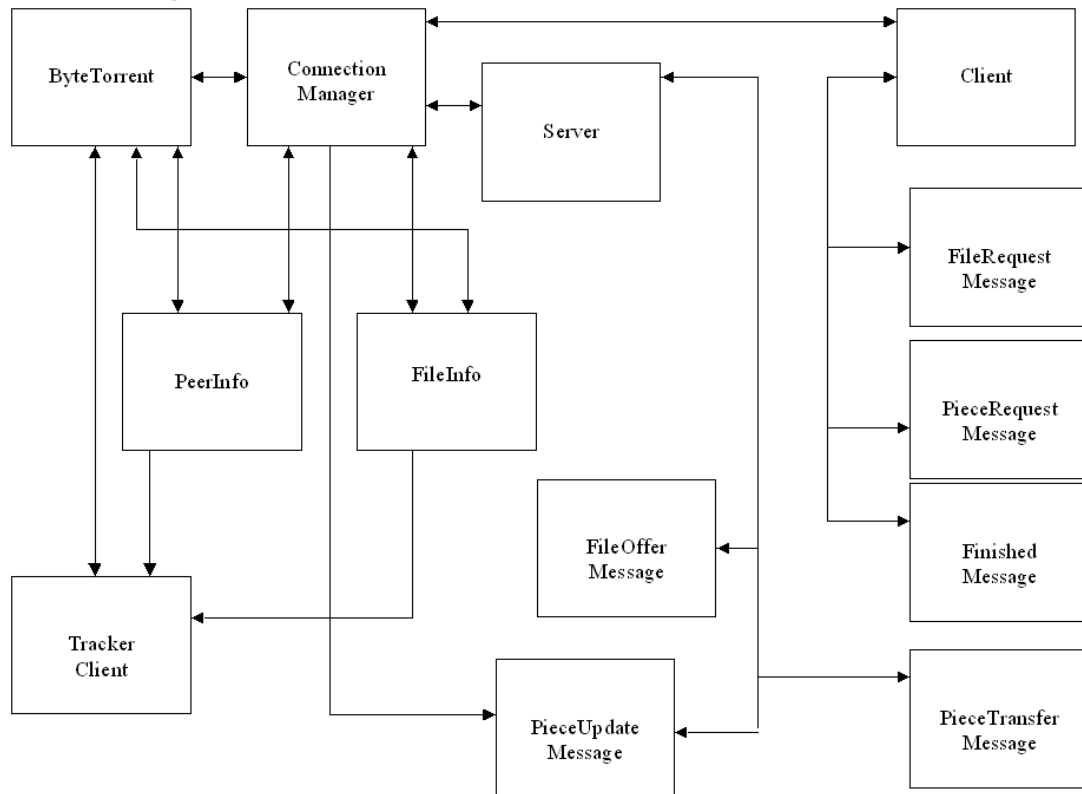
-> means the data was received by the Server/Client

When the file is finished, the program displays a message, telling the user that the file cannot be used until the program is closed, hence unlocking the file.



Client Design Documentation

Class Diagram



Class List and Functionality

Class	Functionality
ByteTorrent	The ByteTorrent class is the entry point to the ByteTorrent program. It is responsible for providing the GUI and for interacting with the ConnectionManager and TrackerClient classes. It retrieves FileInfo and PeerInfo objects from the TrackerClient and passes them to the ConnectionManager .
TrackerClient	The TrackerClient class provides a layer of abstraction between a client and the tracker. A client only needs to know about the objects returned by the TrackerClient class and not the specifics of the TrackerProtocol . It was designed to enable the client team to continue working while waiting for the Tracker team.
ConnectionManager	The heart of the ByteTorrent client. ConnectionManager is responsible for listening for incoming connections and initiating outgoing connections. ConnectionManager starts a Server thread to handle incoming connections and a Client threads to establish outgoing connections to other peers. Server and Client threads rely on ConnectionManager to handle the actual reading and writing to from the shared file.
Server	The Server class processes incoming 'File Request', 'Piece Request' and Finished commands. It sends 'File Offer', 'Piece Transfer' and 'Piece Update' commands by using the corresponding instance of the Message interface (FileOfferMessage , PieceTransferMessage and PieceUpdateMessage).
Client	The Client class sends 'File Request', 'Piece Request' and 'Finished' commands by using the corresponding instance of the Message interface (FileRequestMessage , PieceRequestMessage and FinishedMessage). It processes 'File Request', 'Piece Request' and 'Piece Update' commands sent to it by a remote Server .
FileInfo	The FileInfo class is responsible for storing the filename, path, file size and calculates and stores the piece size and the number of pieces.

PeerInfo	The PeerInfo class is used to store the IP address and port for a peer. A PeerInfo array is used to handle the list of peers returned from the Tracker.
-----------------	---

Display (Interface)	Objects that implement the Display interface can be registered with a ConnectionManager . This interface provides a 'write' method that implementations can use to log information about piece transfers the console, the GUI or a file.
ConnectionMonitor	Objects that implement the Display interface can be registered with a ConnectionManager . This interface provides methods that are used to provide updates on the number of pieces, current number of clients and servers and the number of duplicate pieces (as seen in the status bar of the GUI).
Message (Interface)	Message is an Interface that is implemented by the six message types (below) that are used to format the data that is sent via a Client or Server . It provides a 'toBytes()' methods that returns an array of bytes that will be transmitted across the network.
FileRequestMessage	FileRequestMessage is responsible for formatting a 'File Request' to be sent to a remote Server to initiate a file transfer.
FileOfferMessage	FileOfferMessage is responsible for formatting a 'File Offer' to be sent back to the requesting Client from a remote Server .
PieceRequestMessage	PieceRequestMessage is responsible for formatting a 'Piece Request' to be sent to a remote Server to request a piece.
PieceTransferMessage	PieceTransferMessage is responsible for formatting a 'Piece Transfer' to be sent back to a Client it contains the piece of the file requested.
PieceUpdateMessage	PieceUpdateMessage is used when the ClientManager needs to send a 'Piece Update' out all Servers (all the others are only used internally by Server and Client).
FinishedMessage	FinishedMessage is responsible for formatting a 'Finished' command to be sent to a Server by a Client to indicate that it has all the pieces and will be disconnecting.

Key Design Choices

Development of the Design

As all ByteTorrent groups were instructed to try and get basic transmitting of a file sent across the network in pieces before the actual assignment specification was handed out our client was started before we had the complete requirements. Our design has gone through a number of revisions before and after the specification was published. We originally planned to implement a number of features that were dropped after we realised they were not required for the assignment and instead focussed on the main task. The features dropped include support for resuming of downloads, downloading and sharing of multiple files at the same time and on the same port, using of the one socket for both downloading and uploading of files so that instead of accessing the Tracker multiple times to find out new peers you simply download from the people who connect to you.

The client started with three files classes Server, Client and FileHolder. FileHolder was responsible for the writing and reading of pieces to and from the hard drive using a random access file. It was decided that all Clients and Servers access the file through FileHolder because you can't actually write to or read from multiple parts of the file at the one time and this would later enable us to keep track of the pieces we have in a centralised location.

The Server class was then modified to be multithreaded so it could handle multiple clients. FileHolder was renamed to FileTracker and received support for files greater than 2GB. The 'read' and 'write' methods of the RandomAccessFile class take an integer offset to support files greater than 2GB you first need to use the 'seek' method which has a long offset. FileTracker also had the foundations of a registry of files that are being downloaded and uploaded so that the Server could handle multiple files on the same IP Port. The registry was removed after the assignment specification was published stating that it only needed support for a single file.

Threading support was also added to the Client class and FileTracker was modified to keep track of pieces that were already downloaded by using a Java BitSet. Code to convert the BitSet to an array and back again was added so a record of pieces can be sent across the network. By this point the protocol had been modified slightly from the original. As there was no Tracker ready TrackerClient, PeerInfo and FileInfo were created to fill in for the Tracker whilst the Tracker was being worked on. A command-line interface was also under construction.

A couple of temporary classes were made so that number of Server and Clients could be linked up and basic swarming tested. It was planned to merge the Server and Client class together to form a Connection class which could act as a client, a server or both at the same time. The Connection class was going to be partially controlled externally by the ConnectionManager class. The Connection class would be passed messages which could be queued, prioritised or sent immediately. Doing this properly was more complex than envisioned. The existing Server and Client were modified to use the

various Message types that were made but the classes were never merged. The ConnectionManager was made responsible for starting the modified Server and Client classes and it also took on the tasks previously assigned to FileTracker/Holder.

Breaking a file into pieces and transferring the pieces

There are a couple of steps involved in splitting a file into pieces. When a file is first shared, the FileInfo class determines the piece size that is appropriate for the file. This keeps number of pieces to a reasonable amount and prevents errors such as a piece size greater than the file size being registered. Pieces are read from and written to using a RandomAccessFile and a record of all pieces that the peer currently has is stored in a BitSet. The BitSet and RandomAccessFile are controlled by the ConnectionManager.

As the last piece may be of a smaller size than the rest of the pieces the size of the piece is transmitted along with the piece data as stated in the Peer-to-Peer Protocol specification. Although a peer is able to determine the piece size by the index which is transmitted, we decided to send the size anyway as it makes the implementation simpler.

A zero based index is used for the pieces as this is the indexing scheme commonly used by Java. To determine where a piece is located or it's offset within the file its piece index is multiplied by the piece size (normal piece size not the last piece size). Using the 'seek' method of the RandomAccessFile and the calculated offset the piece can be read from or written to using the appropriate 'readFully' and 'write' methods. As the piece data takes time to travel 'readFully' is used instead of 'read' as 'readFully' waits for all the data to be read. Although there are 'readFully' and 'write' methods that take an offset as a parameter the only accept an integer therefore we use 'seek' which takes a long offset before reading or writing so we can support files greater than 2GB.

Once the piece data is read a PieceTransferMessage is created to format the data to be sent as required by the Peer-to-Peer Protocol specification. Finally the data is sent via a Server thread. There is no acknowledgement of a piece transfer as outlined in the Peer-to-Peer protocol specification.

Requesting a Piece

A simple algorithm for determining the next piece a Client should download was added to FileTracker and later moved to the ConnectionManager. The Client passes a BitSet, which indicates which pieces that the remote peer it is connected to has to the ConnectionManager. The ConnectionManager compares it to the pieces it has and returns the index of the first piece that the remote peer has and that you need. The ConnectionManager keeps a record of the last index returned and starts its next search after this index plus 10% of the total number of pieces. If there are few pieces remaining or each peer only has a few pieces the occasional duplicate piece can be

requested. This wouldn't be good for production software but for the assignment the simplicity was of a higher priority.

Putting the Pieces Back Together

When the file is created its size is set to the size of the complete file by using the 'setLength' method of the RandomAccessFile. This makes it easy to put the file back together. As each piece is received from a peer the file offset is calculated and the piece is written to the file. The bit corresponding to the pieces index is set in the BitSet that is used to keep a record of the pieces received. Once all pieces are downloaded the user is informed that the file is complete. An MD5 hash could be calculated at this point to verify the file is correct. However our implementation does not currently do this as it was felt that it was beyond the requirements of the project. As our protocol uses smaller pieces than BitTorrent does, having hashes of individual pieces would have required too much overhead. This unfortunately means that if a peer is acting maliciously it won't be detected until you have the complete file.

GUI

As it was getting very hard to see what was going on when the program was running we developed a simple GUI that enabled us to better monitor what was happening in the program. Before the GUI we had a lot of debug information written to the console but it whooshed by too quickly to be useful. The GUI enabled us to better study the piece transfers that occurred after the download had completed and monitor the number of pieces downloaded, the number of servers and clients running and also the number of duplicate pieces. To make the GUI simpler to use the option to enter ports was removed and they were instead randomly selected free port by the ConnectionManager before being registered with the Tracker. The random selection enables multiple clients to be tested on the one computer without the need for user to enter a port. The GUI enabled us to detect a bug which resulted in clients disconnecting prematurely when connecting to a server that didn't have any pieces that were needed instead of waiting for pieces to become available. We may not have been able to detect this bug if we were preoccupied by extra features.

Error Handling

There are many places where errors can occur in a complex system for example users can enter invalid data, data may be corrupted during transmission or a malicious user can deliberately try to exploit the system. Some of these errors require special care and attention to handle properly and could leave a user with corrupt data if not handled properly.

As we currently control the whole system and the main purpose of our implementation is to show that the protocol works, our main focus for error handling was preventing human error and not focusing on preventing potential attacks from malicious users. Our GUI ensures that users only share files that are on their computer

as they can't enter file names and that an appropriate piece size is used as it is calculated by FileInfo. We have also designed our protocol and client to be able to handle files that are greater than 2GB which are increasingly common these days. A 32-bit integer in java can represent a maximum value of 2^{31} or one byte less than 2GB, we used longs to prevent integer overflow occurring when large files are used.

We use TCP to ensure that data is being transferred without corruption. However this doesn't protect users against peers deliberately sending invalid data. Before our ByteTorrent client should be used by the general public it will need to be hardened against potential attacks.

Limitations

- Does not enable you to pause and resume a partially complete download. However to add pause and resume functionality a pause command could be added that instructed the Server and Client threads to stop downloading uploading. Then you could get the list of pieces from the ConnectionManager and store them to a file on the hard drive for later retrieval. A less efficient method is to scan the partially downloaded file on the drive for missing pieces equally to the piece size.
- Can potentially use a lot of memory which can cause problems if testing with multiple clients on the same computer.
- You can only download or share a single file at the same time. However, we did temporarily have a version that could handle multiple files on different ports but it had some issues.
- No error checking of the downloaded file to see if it was corrupted during the download as we relied on TCP/IP for all error checking.
- Users are not able to select the directory to which a downloaded file is stored instead the files are downloaded to the working directory. Therefore it needs to be run from writeable media.
- It was only designed with the performance required of an Internet connection and not that of a LAN so some modification may be required for optimal performance on a LAN
- If the seed fails it will need to rejoin on a different port otherwise clients will fail to detect it.
- There is no way to split a piece into smaller pieces like BitTorrent does. This results in a larger number of pieces to keep track of.
- The 'Error Message' specified in the protocol was not implemented.
- Due to the complexity involved there may still be bugs present in the system...