Amy Chen @amy
Moulindra Muchumari @mm1729
Dylan Herman [dah242] @oneTimePad

Phase 3
IMPLEMENTATION DETAILS

Dependency Diagram:

```
 Client --GET list of peers-> Tracker---Contact Tracker--> Tracker server
          |
          --peers list---> PeerDownloader---get bittorrent packets--> Message
                            |
                            --send/get pieces--> PieceManager --write data-> FileWriter --write to
disk-> I/O disk
```

////////////////////////////////
// Download & Upload Bitfields //
////////////////////////////////

Per peer, PeerDownload contacts all peers.

> Handshakes Peer: client tells peer its client ID & torrent hash
> Handshake from Peer: peer sends its ID and torrent hash
> SUCCESS: if handshake is successful, EXCHANGE BITFIELDS

Exchange Bitfields:
A bitfield represents the pieces that a peer has. The peer
and client exchange bitfields. On the client, we have 2 byte
slices. One slice represents the bits we have received (aka
currently own). The other slice represents pieces we are missing
(aka pieces we need to request from the peer).

```
          our pieces       [0 0 0 0 0 0]
          missing pieces  [1 1 1 1 1 1]
```

When we receive a piece, we flip the bit for the ones we receive.
To calculate the missing: !(our pieces) & (peer pieces) = (missing pieces)

Because our bitfields are represented in byte slices, the following
calculation is needed:

PIECE INDEX = (byte index * 8) + bit index

//////////////////////////////
// DOWNLOAD & UPLOAD PIECES //
//////////////////////////////

If PieceManager determines we need a piece, it returns TRUE from compareBitField. Otherwise we move on to the next peer.

We send the peer an INTERESTED message to notify we would like to download a piece(s).

We wait for an UNCHOKE message from the peer, which indicates that the peer is ready to send pieces. After receiving the unchoke message, our connection has state UNCHOKED and INTERESTED. At this point, we are able to send request messages and receive piece messages as response.

Next, we ask PieceManager for the piece index to be requested.
Then, we send the REQUEST message to the peer. After sending the request, we wait for a PIECE message for the REQUEST sent. We currently only send one REQUEST at a time and wait until the corresponding PIECE message is received to the send the next REQUEST. If the peer receives a CHOKE message at any time in this loop, we wait until we receive an UNCHOKE message to resume sending REQUESTS.

We added a time calculation of the total time it takes to download all the pieces.

///////////////////
// PIECEMANAGER //
///////////////////

PieceManager keeps track of what pieces need to be downloaded. Currently, it works with one peer at a time. It stores the peer bitfield and the client bitfield to check what pieces it is missing. PieceManager stores the requests (10 by default) in a queue to quickly send the requests

With the addition of multiple peers downloading and uploading, it was require to introduce synchronization to the piecemanaeger. The piecemanager now contains two bitfields, one called "bitfield", which is the pieces that we, the client, current are in possesion of. We send this field to whoever we speak too. The second bitfield is called the "transitField", this is marked by a peer connection [go routine] that claims responsibilty for retrieving that piece from the peer it manages. This allows from multiple connections to concurrently access the piecemanager claiming pieces and putting them into their request queues to be downloaded from a peer. When the piece is actually retrieved, it is marked in the "bitfield" that we now have it. Mutexes are used for synchronization.

All connections are responsible for registering with the piecemanager. For each connection registered, the piece manager generates a data structure that keeps track of the pieces that connection is responsible for downloading, what pieces the peer who is in that connection has, and a "haveBroadcastQueue". Have messages are sent to all peers we are connected to whenever we complete the downloading of a piece. Thus connections signal the piece manager that a piece has been downloaded and the piece manager puts a have message in all the "haveBroadcastQueue"'s of each of the connections registered. This connections then send Have messages to their peers.

When a connection fails for any reason the connection manager for that peer connection notifies the piecemananger who loops through that peer's piece request queue to flip the bits of all bits in the transitField that correspond to those pieces ,including the one it was about to request but caused the error. Thus other peers can now claim respnisbiltiy for those pieces.

The piece manager also saves the state of what pieces we have to file if the program is ever stopped. This allows the bitfield to be loaded up again upon restart.

////////////////////
//ConnectionManager//
////////////////////

Connection Managers are spawned for every peer connection created [outgoing or incoming]. It is responsible for maintaing a queue of messages to be sent to the peer it manages, performing the correct action upon receiving a specific message, and sending messages. When a connection manager is created for a tcp connection is registers with the piece manager [as described above], and unregisters upon failure or completetion.

The use of the connection and piece manager in this fashion allows us to use the same interface for both incoming and outgoing peer connections in addition to uploading and downloading.

The connection manager also contacts the piecemanager if a peer every claims to have received a new piece it did not have before (i.e. it send a Have msg to us), use UpdatePeerField.

////////////////////////
//PeerConnectionManager//
////////////////////////

Each of these components, piece manage,r connection manager and Peer Connection Manager abstract a certain level of the connection process. Piece Manager abstracts keeping track of what pieces we have and what we need. Connection Manager abstracts the actual receiving and sending of messages to a peer and how to act accordingly. Peer Connection Manager abstracts the actual TCP connections made for incoming and outgoing connections. It spawns a listener for incoming connections and attempts to make outgoing connections to all peers in the tracker's list. A connection manager is created for each of the incoming and outgoing connections.

///////////////////////
//Packet///////////////
///////////////////////
Packet abstracts the actual lower level sending and receiving of binary messages, parsing them to be read by ConnectionManager.

Multiple connections are made using go routines in parallel