Robert Caudill
Kenny Leftin
Andrew Nichols
William Rall

Bit Tortoise Client

Our Implementation

Our implementation of the BitTorrent protocol is called BitTortoise, and is written in Java. BitTortoise begins by generating the peer ID that the client will use. It uses Azureus style coding for the ID. Our IDs look like '-BT0001-'+12 random bytes, which are all in the displayable character range (space, digits, characters, and special characters). Once we have generated the peer ID, we take the filename of a .torrent file specified in the command line arguments, read the file, and parse through it. Once we've parsed all the data from the .torrent file we create Piece objects. Piece objects represent the pieces of the file that we want to download. They contain a piece index which is specified by the torrent file. These pieces contain an array of BlockRequest objects, each of which represent an individual segment of that piece that the client will later request. The offsets of these BlockRequests are separated by 2^14 bytes. We add all the Piece objects to a Map of outstanding requests (Mapping piece index to the piece object). When we need to request a piece from a peer, we can use this map of outstanding requests to obtain a Piece object and gather blocks to fill up the list of requests that we will send to this peer. We then continue processing more piece objects until the list has reached a full state.

After setting up this infrastructure, the next step is to send an HTTP GET request to the tracker to obtain a list of some of the other peers that are connected to the tracker. We parse the bencoded response from the tracker and extract a list of Peer objects, along with some other information. After we've gotten the initial list of peers, we create a ServerSocketChannel that listens to a port (defaults to 6881) for connections. We then enter the main loop of BitTortoise. This is the part of the program that does all the interacting with peers. BitTortoise uses a Selector to check whether or not a socket is ready to be read or be written to. Depending on the type of key we are processing, BitTortoise does different things. We have two maps, one for pending peers and one for active peers. Pending peers are peers that we obtained from the tracker with whom we have initiated a connection, but not finalized the connection (i.e. we have sent the SYN and are waiting to complete the 3-way handshake). We move these peers over from the

pending map to the active map once we finalize the connection (i.e. receive a SYN+ACK and reply with an ACK).

To fill the selector with connections to peers, at the end of each cycle, we check whether the number of peers we are connected to is above 30. If we are connected to less than 30 peers, the client attempts to connect to the peers with whom we do not currently have a connection in the list parsed from the Tracker. The client calls a non-blocking connect() (i.e. sends a SYN to the peer) and adds the Peer object to the map of pending peers. If there are not enough peers left in the list, we instead reconnect to the tracker to obtain more.

If we encounter an acceptable key from the Selector, this means that a new peer is trying to connect to us for the first time (we have received a TCP SYN). If we have less than 56 connections (which is the most that the BitTorrent Protocol suggests), then we accept the connection and add it to the Selector, if not, we ignore it. If we encounter a connectable key from the Selector, it means that we can now attempt to finish a connection with this peer (we have received a SYN+ACK in response to our SYN, we can send an ACK). If we are unable to finish the connection (i.e. this was triggered by a timeout instead of receiving a SYN+ACK), we drop the connection and remove the key from the Selector. If we are able to finish the connection, we move the peer from the pending peer map to the active peer map and send them the handshake message.

If we encounter a writeable key from the Selector, this means that we can write messages to the peer if they are part of the active peer map. If we haven't sent them our handshake, we do so. If we have exchanged handshakes, we call the sendMessage function. sendMessage figures out what message we need to send (if any) to the peer. If the last message that we attempted to send did not finish sending, it attempts to send more of that message. Otherwise, it checks several different state variables that are held within the peer object to see if it should send a certain type of message, and will send an appropriate message. All outgoing messages that should be sent to another peer are triggered by changing state in the Peer object, which causes an appropriate message to be sent the next time the connection to that peer is writable.

If we encounter a readable key from the Selector, this means that there is data waiting to be read from the peer. We call the function readAndProcess. If readAndProcess fails, we remove the peer from the list and close the connection. readAndProcess is a function that will parse any BitTorrent message type, including multiple messages received at the same time in one

byte array, and process the messages it has received in order. readAndProcess reacts to each type of message by performing the following actions (by message type received):

Receive Choking Message: We change the state to peer_choking = true.
Receive UnChoking Message: We change the state to peer_choking = false. If the peer does not have a full list of outstanding requests (i.e. size doesn't equal the maximum queuing size), we cycle through random pieces the client has that we don't have, and try to fill up the peer's outstanding requests list with BlockRequest objects.
Receive Interested Message: We change the state to peer_interested = true.
Receive Non-Interested Message: We change the state to peer_interested = false.
Receive Have Message: We add the piece index that we are given to the BitSet describing which different pieces the peer we are connected to has. If the peer now has a piece that we need (we determine this by ANDing this BitSet with the complement of the BitSet describing what pieces we have successfully received, and checking if it is non-empty), we set a state that remembers that we should send them an interested message the next time we can send a message to them.
Receive Bitfield Message: We form a BitSet from the bitfield they send us describing which different pieces the peer to which we are connected has. We store this as state (the peers completed pieces). We then AND the peer's completed pieces bitset with the complement of the bitset describing all of the pieces that we have successfully completed, and if the result is non-empty and we are not already interested, we change the state so that we will send them an interested message the next time we can send a message to them.
Receive Request Message: We check to see if we are current choking the peer. If we are then we ignore the request. If we aren't, we add the request to the peer's request list.
Receive Piece Message: We write all that we have currently received of the block to the file. Then, we check to see if all the blocks of a piece have been processed. If they have been processed then we run the SHA1 hash on the piece we received and check it against the SHA1 hash of the piece in the torrent file. If they match then we remove the piece from the outstanding pieces list. If they don't match, we throw out the data and add the block requests back to the list of outstanding requests for that piece.
Receive Cancel: Goes through the peer's request list and checks to see if there are any block requests between the boundaries. If there are, it removes them from the list.

Client Extensions

BitTortoise Dynamic Transmission Size: The general idea of this extension is that while some clients only send a static block size of 2^14, ours will send data of dynamic block size based on how fast the connection is. This extension only works if the two peers are BitTortoise clients. As described above, our client segments every piece into a list of BlockRequests. Each BlockRequest has a one-to-one correspondence with a request message. Since we can not request an entire piece with one request message, we usually have multiple block messages for each piece. These blocks contain the current offset within that piece and are usually set at 2^14 bytes apart, but they don't necessarily have to be.

Since there has been debate on the BitTorrent wiki over ideal request length, we thought we would let the client determine the most ideal length. If a client notices that another BitTorrent client is processing our requests very quickly, we would send it larger requests to reduce overhead. This is implemented by coalescing BlockRequest objects together. If a BitTorrent client is taking a long time to process our requests, we would send it smaller requests. This is implemented by splitting BlockRequests. The size of a BlockRequest always remains a power of 2.

The minimum and the maximum BlockRequest size, as well as when to update this size and under what conditions will be set after further testing.

Work Done by Each Member (self-reported)

Andrew

- Set up SVN server
- Wrote initial Receive Message Code.
- Wrote code to ping the tracker and process response including storing the peerlist.
- Abstracted code into tracker class.
- Code to create Sha-1 Hash for given data.
- Handling code for unchoke messages.
- Wrote code for beginning of Round(every ten seconds), choosing top three people and one opt unchoke.
- Wrote code to figure out which piece to randomly select from peer when we receive an unchoke message.
- Wrote draft of write up.

Rob:

- Adapted code found online to handle torrent file.
- Adapted code found online to do bencoding.
- Wrote code to compute for a given piece of data.
- Wrote initial code for generating messages to be exchanged.
- Set up a tracker and sample torrents to be used in testing and experiments.

Will:

- Wrote code to parse information returned from tracker into a list of peer objects.
- Wrote code to generate a Peer ID that is acceptable to other BitTorrent clients.
- Wrote code in BitTortoise.java so that we could use Selectors to poll sockets (java SocketChannels/ServerSocketChannels) for readability / writability / connectability / acceptability.
- Wrote code so that we could connect to remote peers at appropriate times.
- Wrote code so that we could accept connections from remote peers at appropriate times.
- Wrote code so that we could parse messages received from remote peers and modify state in Peer object and within BitTortoise (also handles when multiple messages are returned in the same byte array).
- Wrote code so that we could send messages whenever we need to, which allowed us to communicate with other users.
- Fixed misconceptions in MessageLibrary so that messages that are created use the proper/supported lengths, ids, and types for each of the arguments (there was confusion as to what were bytes/ints).
- Fixed code to select which blocks to request from remote peers. Made it so that it will select first from pieces that it has already selected blocks from, and after filling all such pieces up will pick a new random piece from which to request blocks.
- Created/Wrote/Brainstormed variables within Peer object and BitTortoise to determine what state variables needed to be kept so that messages could be sent appropriately
- Wrote code to print error messages, provide a verbose mode for testing purposes.
- Fixed a couple errors in the bencoder implementation we used.
- Fixed write-up accuracy and clarity.

Kenny:

- Set up Piece and BlockRequest logic. Implemented Piece and BlockRequest classes, as well as instantiation of them within BitTortoise.
- Handled what happened when we receive a piece message, including storing block in file and other side effects.
- Handled what happened when we receive a request message, including getting block from file and other side effects.
- Designed and began implementing Dynamic Transmission Size extension. This includes coalescing and splitting blocks.
- Formatted original handshake message that we used to communicate with other Peers.
- Miscellaneous debugging and code refactoring.
- Revised/Edited write-up.

Experiments

(Experiments still pending on working Bit Tortoise client)