Wednesday, December 03, 2014 3:04 PM

# Assignment Instructions

ROB RICH RAHUL

**Sections**

- Theory of Operation
- Assignment Specifics
- Files Available
- Programming Style
- Bencoding
- Communication With the Tracker
- Communication With the Peer
- The Write-Up
- Submitting the Project
- Grading
- Resources
- Frequently Asked Questions

**Updates**

Before jumping into the project I would like to mention one thing. Most of the time your first attempt at a software project is either too slow, too buggy, or is difficult to maintain/expand. Be prepared to throw away the first versions of most of your code. You can even plan to have them as throw-aways. You don't waste time doing this because you learn where you can make improvements in your final version.

**Theory of Operation**

The third project will expand upon what you did for the first two. Your client must be able to interface with multiple peers, upload and download, accept incoming connections, publish appropriate information to the tracker (port, uploaded, downloaded, left, events, etc.) to allow remote peers to contact your client.

The client must also use the rarest-piece-first algorithm for piece selection; and perform choking and optimistic unchoking. Throughput measurement is necessary to accomplish correct (un) choking behavior. The client should be able to maintain connections to at least ten (10) peers, though not all peers need to be uploading/downloading simultaneously.

The user should be able to suspend (quit) the program at any time, and the client should NOT terminate until the user provides the appropriate input.

~~Extra credit will be available to groups that develop a Graphical User Interface (GUI) for their client. The GUI should have complete functionality: displaying all relevant information about the client, allowing user input, and be aesthetically pleasing. The extra credit GUI has a maximum value of 15% of the project grade (115% with full credit). To achieve the full 15%, the GUI will need to be very well-designed and -implemented.~~

**Assignment Specifics**

Your assignment should basically do the following:

Take as a command-line argument the name of the .torrent file to be loaded and the name of the file to save the data to. For example:

```
java my.package.RUBTClient somefile.torrent somepicture.jpg
```

See Project Part 2 for the basics of how your client should behave.

~~Your client must correctly publish its connection information and status to the tracker. This includes the port, uploaded, downloaded, left, and event arguments. Your client must accept incoming connections from other peers. It should not maintain more than one TCP connection to another peer.~~

Piece requests should be ordered based on piece rarity for all connected (not only unchoked) clients. The rarest piece available from a peer should be requested first, and if there are more to another peer.

Piece requests should be ordered based on piece rarity for all connected (not only unchoked) clients. The rarest piece available from a peer should be requested first, and if there are more than one pieces with the same rarity available, a randomized selection scheme should be used.

Every 30 seconds, the client should analyze the performance of the connected peers, choke the "worst" peer, and randomly unchoke a choked peer. Priority should be given to peers with high upload speeds while downloading (you finish sooner), or peers with high download speeds while seeding (they finish sooner), in order to maximize swarm throughput.

You may limit the number of unchoked connections to 3. No more than 6 connections should be active (unchoked) at any point.

Tracker scrapes should be performed no more frequently than the value of "min_interval" (or 1/2 "interval" if "min_interval" is not present), and no less frequently than twice the value of "interval" returned by the tracker.

How you choose to implement saving the state of your program is up to you, but an intuitive approach might be to allocate the total space to disk before downloading, and then writing the appropriate pieces into the file as they complete. It is recommended that you do not use serialization of Java objects to save client state.

**Files Available**

There are several files available in the Resources section of the site in the folder "Project files." The most important is project2.torrent, which will continue to be used for part 3. The other files available are helpful, but not necessary. Be sure to check the resources section for updates or changes.

**Programming Style**

Writing maintainable code is just as important as writing correct code. Consequently, you will be graded on your style. Below is a list of suggestions to make your code easier to understand and maintain:

Javadoc - All classes, fields, and methods that are not private must be documented with Javadoc. Supplying Javadoc comments for private classes, methods, and fields is encouraged but optional.

Comments - Comments for variables and for sections of code that does not have an obvious purpose. Please do not comment obvious statements or sections of code. For example,

```
int i = 0;    //i is an integer and the initial value is 0
```

is not useful and unnecessarily clutters the code.

Naming - Names of classes, methods, and variables should be descriptive. For example, if a class has a field containing its hash value, naming the field hash_value is much better than naming it hv .

Exceptions - Catch exceptions and do something useful with them. For example, print a statement describing what went wrong to System.err.

Easy to understand loops/recursion. When possible, it's best to avoid break statements whenever possible. When using recursion, tail-recursion is often preferable because "smart" compilers can often translate it into a loop. Return statements within an if/else clause should be avoided whenever possible.

Efficiency - Avoid being terribly inefficient. For example, if you have to sort 10M objects, it's better to sort them with a O(n*lg(n)) algorithm than a O(n2) algorithm.

Encapsulation/Objects - Encapsulation is a useful feature of object-oriented languages, and because you are writing your program in an OO language, your program must separate the functionality into multiple classes. This way you can change the implementation of a class without affecting the rest of your program. An example of separation would be to have separate FileHandler, PeerInterface, and TrackerInterface classes to manipulate files, interface with peers and interface with the tracker, respectively. If you do not separate your program into functional units, you will lose points on the Style portion of the grading.

**Bencoding (Pronounced "Bee Encoding")**

Bencoding a method of encoding binary data. Tracker responses, and interpeer communication will be bencoded. Below is how data types are bencoded according to the BT protocol. [The following list is taken fromhttp://www.bittorrent.org/beps/bep_0003.html ]

Strings are length-prefixed base ten followed by a colon and the string. They are encoded in UTF-8. For example 4:spam corresponds to 'spam'.

Integers are represented by an 'i' followed by the number in ASCII base 10 followed by an 'e'. For example i3e corresponds to 3 and i-3e corresponds to -3. Integers have no size limitation. i-0e is invalid. All encodings with a leading zero, such as i03e, are invalid, other than i0e, which of course corresponds to 0.

Lists are encoded as an 'l' followed by their elements (also bencoded) followed by an 'e'. For example, l4:spam4:egse corresponds to ['spam', 'eggs'].

Dictionaries are encoded as a 'd' followed by a list of alternating keys and their corresponding values followed by an 'e'. For example, d3:cow3:moo4:spam4:eggse corresponds to {'cow':'moo', 'spam':'eggs'} and d4:spaml1:a1:bee corresponds to {'spam': ['a', 'b']}. Keys must be strings and appear in sorted order (sorted as raw strings, not alphanumerics).

**Communication With the Tracker**

Your client must take the information supplied by the TorrentFile object and use it to communicate with the tracker. The tracker's IP address and port number will be given to you by the TorrentFile object, and your program must then contact the tracker. Your program will send an HTTP GET request to the tracker with the following key/value pairs. Note that these are NOT bencoded, but must be properly escaped [this list is taken from http://www.bittorrent.org/beps/bep_0003.html ]:

info_hash - The 20 byte(160-bit) SHA1 hash of the bencoded form of the info value from the metainfo file. This value will almost certainly have to be escaped.

peer_id - A string of length 20 which this downloader uses as its id. Each downloader generates its own id at random at the start of a new download. This value will almost certainly have to be escaped.

port - The port number this peer is listening on. Common behavior is for a downloader to try to listen on port 6881 and if that port is taken try 6882, then 6883, etc. and give up after 6889.

uploaded - The total amount uploaded so far, encoded in base ten ascii.

downloaded - The total amount downloaded so far, encoded in base ten ascii.

left - The number of bytes this peer still has to downloaded, encoded in base ten ascii. This key is important - If you do not specify how much you have left to download, the tracker assumes you are a seed and will not return any seeds in the peer list.

event - This is an optional key which maps to started , completed , or stopped (or empty , which is the same as not being present. If not present, this is one of the announcements done at regular intervals. An announcement using started is sent when a download first begins, and one using completed is sent when the download is complete. No completed is sent if the file was complete when started. Peers send an announcement using stopped when they exit.

The response from the tracker is a bencoded dictionary and contains two keys:

interval - Maps to the number of seconds the downloader should wait between regular requests. Scrapes should occur within (interval * .5, interval *2) seconds of the previous scrape.

peers - Maps to a list of dictionaries corresponding to peers, each of which contains the keys peer id , ip , and port , which map to the peer's self-selected ID, IP address or dns name as a string, and port number, respectively.

In addition to what your program did for the last project, your client should also periodically update its status to the tracker. The update period should be no less than the min_interval returned by the tracker (or half of interval, if min_interval is not present). This may change during execution, so it should be updated each time the tracker is contacted. In the event that the interval value is excessively large, you may cap it at 180 seconds.

**Communicating With the Peer**

Handshaking between peers begins with character nineteen (decimal) followed by the string 'BitTorrent protocol'. After the fixed headers are 8 reserved bytes which are set to 0. Next is the 20-byte SHA-1 hash of the bencoded form of the info value from the metainfo (.torrent) file. The next 20-bytes are the peer id generated by the client. The info_hash should be the same as sent to the tracker, and the peer_id is the same as sent to the tracker. If the info_hash is different between two peers, then the connection is dropped. You must verify the peer ID of the remote peer if it is available (e.g. from a tracker scrape), if the connection is incoming and you have no information about the connecting peer, you may omit checking the peer ID.

All integers are encoded as 4-bytes big-endian. e.g. 1,234 should be encoded as (hex) 00 00 04 d2

The peer_id should simply be random bytes. It is suggested that you make the peer ID ASCII-printable for compatibility with other peers and debugging purposes.

After the handshake, messages between peers take the form of <length prefix><message ID><payload> , where length prefix is a 4-byte big-endian value and message ID is a single decimal character. The payload depends on the message. Please consult either of the BT-related resources for detailed information describing these messages. Below is a list of messages that need to be implemented in the project.

keep-alive: <length prefix> is 0. There is no message ID and no payload. These should be sent around once every 2 minutes to prevent peers from closing connections. These only need to be sent if no other packets are sent within a 2-minute interval.

choke: <length prefix> is 1 and message ID is 0. There is no payload.

unchoke: <length prefix> is 1 and the message ID is 1. There is no payload.

interested: <length prefix> is 1 and message ID is 2. There is no payload.

uninterested: <length prefix> is 1 and message ID is 3. There is no payload.

have: <length prefix> is 5 and message ID is 4. The payload is a zero-based index of the piece that has just been downloaded and verified.

bitfield: <length prefix> is 1+X and message ID is 5. The payload is a sequence of X bytes where each bit indicates whether the peer has the piece at the bit's index.

request: <length prefix> is 13 and message ID is 6. The payload is as follows:

<index><begin><length>

Where <index> is an integer specifying the zero-based piece index, <begin> is an integer specifying the zero-based byte offset within the piece, and <length> is the integer specifying the requested length.<length> is typically 2^14 (16384) bytes. A smaller piece should only be used if the piece length is not divisible by 16384. A peer may close the connection if a block larger than 2^14 bytes is requested.

piece: <length prefix> is 9+X and message ID is 7. The payload is as follows:

<index><begin><block>

Where <index> is an integer specifying the zero-based piece index, <begin> is an integer specifying the zero-based byte offset within the piece, and <block> which is a block of data, and is a subset of the piece specified by <index> .

Below is an example of what would take place between two peers setting-up a connection and starting sharing.

The local host opens a TCP Socket to the remote peer and sends the handshake packet. The local host then listens for the remote peer to respond.

Upon receiving the handshake packet and verifying the info_hash, the remote peer responds with a similar handshake packet (except with its peer_id). The remote peer then listens for the local host to send a bitfield or other packet. The remote host can send a bitfield packet to the local host at this time.

Upon receiving the handshake and verifying the info_hash, the local host then (optionally) sends a bitfield packet which tells the remote peer which pieces it has downloaded and verified so far.

If the local host is interested in what the remote peer has downloaded, then it sends an interested packet, otherwise it sends an uninterested packet. If the remote peer is interested in what the local host has downloaded, then it sends an interested packet, otherwise it sends an uninterested packet.

When the local host, or the remote peer, is ready to download/upload to the other, it will send

an unchoke packet.
Please note that clients will, and your client should, ignore any request messages received while a remote peer is choked. A client should only upload to another client if the connection is unchoked AND the remote peer is interested. This means that a remote peer will not reply to the local hosts's request messages unless you have expressed interest AND the remote peer has sent an unchoke packet. If the remote peer sends a choke packet during data transfer, any outstanding requests will be discarded and unanswered - they should be re-requested after the next unchoke packet.

**The Write-Up**

Include your name as it appears on the roster and your student ID.

A high-level written description of how your program works. Specifically, describe the high-level interactions between the classes. This is a good way to make sure that your program doesn't have any classes depending on the implementations of other classes. You may include diagrams or illustrations if you feel it would aid your explanation or if you have a hard time explaining how your program works. However, you MUST include some written description of your program. If you know how to use LaTeX, this should be fairly easy to diagram. Also, OpenOffice.org's Draw program has an Export to PDF feature that is very simple to use. This section should be around 100 words, or 2 paragraphs.

A brief (about 1 paragraph) description of each class in the program. Describe what it does in general, the main public methods it provides, and any public fields it exposes. Note that classes should generally NOT have public fields unless they are Static (e.g. constants).

Optionally, you may include a section on feedback about the program. Please only provide constructive/useful comments or insights. What parts of the project were the most challenging and which were the easiest? Did you have trouble finding resources for the project? Were any parts confusing to you? Including feedback will help shape the direction of the next project as well as help in pointing-out trouble spots to look at during grading.

**Submitting the Project**

The project should be submitted through Sakai.

Your project MUST be managed by either Git or Mercurial (Hg) version control systems. To submit your project, you may either submit the entire repository (not only your working directory), or you can give the instructor read-only access to your repository hosted on BitBucket.org or GitHub.com.

Make sure your names are in comments at the top of every file submitted.

The "main" starting method should be in a file called RUBTClient.java.

The write-up in HTML or PDF format saved as a writeup.<html/pdf>. Please do not submit a write-up in any other formats

All files should be submitted as a compressed archive. Acceptable formats are .zip, .tgz/.tar.gz, and .tar.bz2. This file should be named <NetID>.<EXT>, where <NetID> is your eden NetID and <EXT> is the file type extension.

Late submissions will not be accepted. If you have not completed the project, submit what you have and you will be graded for partial credit.

**Grading**

Programming Style: 15%

Correctly interfacing with the tracker (retrieving peer list, periodic updates, after completing download): 10%

Correctly interfacing with the peers (handshaking, maintaining state, keep-alive): 10%

Correctly downloading from at least 2 peers simultaneously: 10%

Correctly uploading to at least 2 peers simultaneously: 10%

Correctly downloading and verifying the file: 15%

Rarest-piece-first selection algorithm: 5%

Ability to maintain at least 10 peer connections: 5%

Optimistic choking/unchoking (includes rate measurement and throttling): 10%
Write-up 10%
To ensure everyone in the group contributes to the final submission, your individual grade will be weighted by how much you contributed, up to 50% of the project grade. Contribution will be evaluated based on the repository commit logs (see above), so be sure to commit as often as practical.

**Resources**
It is strongly recommended that you bookmark or download the Sun Java 1.6 API , as well as read the following pages:
The Main BT Protocol Explanation: http://www.bittorrent.org/beps/bep_0003.html
Another BT Protocol Explanation: http://wiki.theory.org/BitTorrentSpecification
Java Cryptography: Java Cryptography Architecture (JCA) Reference Guide
A Page on Maintainable Code: http://advogato.org/article/258.html
A Page on HTTP escaping: http://www.blooberry.com/indexdot/html/topics/urlencoding.htm
Wireshark - A network traffic sniffing tool useful for watching network traffic between your client and the tracker/peer.

Any questions about the project in general should be posted to the Discussion Board tool. If you have a question that is specific to you (e.g., JVM install, Eclipse setup), send an email to your TA.

**Frequently Asked Questions**
I've been getting a lot of similar questions lately, so I'll post general forms of them as well as advice on how to solve the underlying problems. As a bit of general advice, if you aren't sure what a packet is supposed to look like, you can always fire up a working client (the mainline BitTorrent client, for example) and then use a network packet sniffing tool like Ethereal to capture the traffic and analyze it.
I can't connect to the tracker. I've tried creating a Socket and sending the bytes through myself by grabbing the InputStream and OutputStream from the Socket, but I can't get a response from the tracker. What's going wrong?
Contacting the tracker is a simple HTTP GET request. This type of operation is done by every web browser today and even some smaller programs. As a result, there exists a class java.net.URL that can easily handle this type of request. Whatever you're trying to do in Java has probably already been done a thousand times, so it's usually faster to find an existing implementation than to write one yourself.

## Submission

From <https://sakai.rutgers.edu/portal/tool/c9c9d8a4-9f53-4391-af42-c0e0b19da732/student-submit/57086>