BitTorrent project 1

DUE: Jul 15, 2014 10:00 PM

Assignment Details

Open Date	Jun 25, 2014 8:00 AM
Graded?	Yes
Points Possible	100.0
Resubmissions Allowed?	Yes
Remaining Submissions	Unlimited

Assignment Instructions

Sections

This is a large project -- be prepared to rewrite your code when you find that your approach isn't working. Start early! Don't expect to finish the entire project at the last minute.

Theory of Operation

In this project, you will build a simple BitTorrent(BT) client which will serve as the foundation for a fully-featured BT client you will build in project 2. For project 1, your client will load a .torrent file, interface with a tracker and a single peer and will download a single file (a JPEG) from that peer. When the file is finished downloading, you will save it to the hard disk. All communication will be done over TCP.

The file your program will read in is a .torrent file containing metadata for the file to be downloaded.

Assignment Specifics

Your assignment should basically do the following:

1. 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 -cp . RUBTClient somefile.torrent picture.jpg
   ```

2. Open the .torrent file and parse the data inside. You may use the `Bencoder2.java` class to decode the data.
3. Send an HTTP GET request to the tracker at the IP address and port specified by the TorrentFile object. The java.net.URL class is very useful for this.
4. Capture the response from the tracker and decode it in order to get the list of peers. From this list of peers, *use only the peers at IP address 128.6.171.130 with peer_id prefix RU1103*. You must extract this IP from the list, hard-coding it is not acceptable.
5. Open a TCP socket on the local machine and contact the peer using the BT peer protocol and request a piece of the file.
6. Download the piece of the file and verify its SHA-1 hash against the hash stored in the metadata file. The first time you begin the download, you need to contact the tracker and let it know you are starting to download.
7. After a piece is downloaded and verified, the peer is notified that you have completed the piece.
8. Repeat steps 5-7 (using the same TCP connection) for the rest of the file.
9. When the file is finished, you must contact the tracker and send it the *completed* event and properly close all TCP connections
10. Save the file to the hard disk according to the second command-line argument.

Files Available

There are several files available in the Resources section of the site in the folder "Project files." The most important is the "Project 1 torrent file" (project1.torrent), which is the "testbed" torrent you can use to develop your client. The other files available are helpful, but not necessary. Be sure to check the resources section for updates or changes as the semester progresses.

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:

- Comments - Comments for variables and for sections of code that do 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.
- 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 $O(n^2)$ algorithm.
- Encapsulation - Encapsulation is a useful feature of object-oriented languages, and so you should try to write classes that expose as little as possible. This way you can change the implementation of a class without affecting the rest of your program.

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-10 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:eggse 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 metadata file and use it to communicate with the tracker. The tracker's IP address and port number must be extracted from the metainfo dictionary, 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 fromhttp://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. You peer ID must NOT start with RUBT.
- *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-10 ascii.
- *downloaded* - The total amount downloaded so far, encoded in base-10 ascii.
- *left* - The number of bytes this peer still has to downloaded, encoded in base-10 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. Downloaders send an announcement using *stopped* when they cease downloading.

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 rerequests.
- *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.

Communicating With the Peer

Handshaking between peers begins with byte nineteen 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.

- 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 be randomly-generated for each process your program generates. (Meaning that it should probably change in some way each time the client is run.) Stick to alphanumerics for easier debugging.

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 byte. 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 messages 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.
- 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.

1. The local host opens a TCP Socket to the remote peer and sends the handshake message. The local host then listens for the remote peer to respond.
2. Upon accepting the incoming connection, the remote peer responds with a similar handshake message (except with its peer_id). Be sure to verify the relevant portions of the handshake message.
3. Upon receiving the handshake and verifying the info_hash, the local host then (optionally) sends a bitfield message which tells the remote peer which pieces it has downloaded and verified so far.
4. If the local host is interested in what the remote peer has downloaded, then it sends an interested message, otherwise it should simply await messages from the remote peer. If the remote peer is interested in what the local host has downloaded, then it sends an interested message.
5. When the local host, or the remote peer, is ready to upload to the other, it will send an unchoke message.
6. Once a peer is unchoked, it can send a request message for one of the pieces that the other peer announced it has completed.

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 message. If the remote peer sends a choke message during data transfer, any outstanding requests will be discarded and unanswered - they should be re-requested after the next unchoke message.

The Write-Up

- Include your name as it appears on the roster and your student ID.
- A high-level 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. Drawing a dependency diagram is a simple way to visualize these dependencies. 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.
- A brief (about 1 paragraph) description of each class in the program.
- 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?

Submitting the Project

The project should be submitted through Sakai.

- Make sure thjat author names are in the comments at the top of every file submitted.
- The "main" method should be in a file called RUBTClient.java.
- The write-up in HTML or PDF format saved as a writeup.<html/pdf>.
- 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.

Grading

- Programming Style: 15%
- Correctly setting up TCP sockets/connections: 15%
- Correctly interfacing with the tracker: 20%
- Correctly interfacing with the peer: 20%
- Correctly downloading and verifying the file: 20%
- Write-up 10%

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. If you are not sure how a particular message should work, try running a known working BitTorrent client and using a tool like Wireshark to view the network traffic and analyze the packets.

Frequently Asked Questions

1. **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

To submit your assignment, attach one or more files and then click Submit.

An * designates a required field.