Programming Assignment Three: BitTorrent Handshake and Control

CMSC 417 Spring 2014

1 Deadline

Intermediate: March 7. Accept connections and reply to handshakes.
Full: March 14. Full handshake with all tracker-provided peers.

There is an intermediate deadline for accepting a connection on a server socket of the assignment, and replying to the handshake message with a properly-formatted handshake message. (After the handshake response, any behavior is permissible, including dropping the connection or leaving it open, for the intermediate deadline.) I expect you to implement the milestone as if you're writing the entire project, but to focus on the connection-accepting behavior. This is not too different from the connection server, just with binary messages, so you should be able to reuse substantial code.

The full assignment requires planning and elegance in implementation to remain tractable. Hack at the solution at your own peril and be prepared to refactor. In contrast to BitTorrent part one (the tracker), this phase is less sensitive to arbitrary decisions at the tracker side and more focused on correctly invoking socket operations.

Post general questions to the forum. Seriously. I consider all non-anonymous posts to the forum (even stupid questions, unless they're answerable from this spec) to be class participation. Good student answers make me happy.

2 Objective

The goals of this mini-project are:

- to integrate your server side socket code (bind, listen, accept) into the bittorrent design.
- to integrate your handling of multiple concurrent connections (select or poll).
- to use non-blocking connect.

3 Executive Summary

The code should:

1. Create a socket for accepting inbound bittorrent connections. (Intermediate deadline.)
2. Complete the tracker connection steps of part one, this time with the inbound connection port parameter set. (Intermediate deadline.)
3. Determine the "have" bitset (bitfield) of every peer listed by the tracker and every peer that connects. (At least those peers that connect within thirty seconds of startup.) (Full deadline)
4. Run for approximately one minute to accept incoming connections. (Intermediate deadline.)
5. At the end of execution, print the "have" bitset in hex format (e.g., `"> %u.%u.%u.%u:%u ffffffff8"` if the peer has all 33 pieces. Please use lower-case hex (`%02x`). (Full deadline.)
6. Disconnect from peers, and tell the tracker that you've stopped (as in part one) (Intermediate deadline.)
7. Exit

4 Read

Please read `http://wiki.theory.org/BitTorrentSpecification#Peer_wire_protocol_.28TCP.29`. Keep reading until you get to "Algorithms." Read it again.

Read the man page for connect(). It describes how nonblocking connect works, restating some of this document.

5 Specification Details

1. Groundwork. These steps are steps for you to complete, not steps in the processing of the code you will write.
   - (Recommended) Refactor and organize your tracker connection code. One gargantuan main() function will make you unhappy. Torrent file loading can be separated from parsing; consulting the tracker should be one method. Consider a per-peer callback function to be invoked for each IP address received from the tracker. This will save you from having to package up a temporary array of peers.

- (Recommended) Make effectively global state global. I will not ask you to download multiple torrents from within the same binary, so the info hash and peer id can easily be globals. Other state from this assignment may also be global. (The goal is to avoid sneaking extra parameters into all function signatures.)
- (Recommended) Run tracker connection code under valgrind to check that there aren't any stray invalid memory accesses. (I'm more worried about accessing uninitialized or freed memory, which will cause non-deterministic behavior in your code, than I am about memory leaks, which will just eventually crash.) Some functions can trigger valgrind complaints through no fault of your own: getaddrinfo is particularly bad in this respect, but properly configured valgrind/memcheck will suppress warnings about it.
- Enumerate the possible connection states, at least through the handshake. If you get it wrong, come back and add (or remove) states. (I.e., don't ask us if you've got it right before proceeding.) You need states only to describe what you expect next from a socket, not what you've just done. C's enum or preprocessor symbols are reasonable approaches to defining states.
- Define a "peer connection" data structure that will hold ip, port, socket number, have bitset (to be allocated dynamically, since you do not know at compile time how many pieces will be in the file), connection state, incomplete message data, and any other fields as they become necessary.
- Define a collection of "peer connections". Such may be an array (preferred for simplicity) or any structure you feel comfortable with. You may use geekos list.h. I have modified it slightly to include stdbool.h (for the bool data type) and to define KASSERT to assert. A global variable is okay. As usual, you *must not* rely on data structure libraries that you did not personally write. 100 entries should be sufficient.
- Implement a helper method to find the "peer connection" corresponding to a given socket (file descriptor). This will be useful when discovering that a socket is ready for reading or writing.

2. Create the server socket (bind, listen) and determine its operating-system-assigned port (getsockname). You do not need to determine your own IP address(es). If you decide to choose a port, your code must find another port if your chosen port is in use. (Intermediate deadline.)
3. Determine the number of pieces in this file. The calculation is a simple function of the length of the pieces element in the info value. This will be the length of the bitset.
4. Note the time (gettimeofday) as the execution start time; one minute from now, you will dump state and exit. (Intermediate deadline.)

5. Query the tracker for peers. Ensure that port is now the port to which your listening socket is bound. (Intermediate deadline.)

6. Populate a "peer connection" structure for each peer.

7. Connect to all peers discovered from the tracker.

   (a) Create a socket for the peer

   (b) Set the socket into non-blocking mode. (fcntl)

   (c) Invoke the connect(). Expect it to return -1 with errno set to EINPROGRESS. (corrected from EAGAIN). It may succeed immediately if connecting locally, which is likely to happen on the submit server.

   (d) If it succeeds (returns zero, not -1 with EINPROGRESS), send the handshake (below).

8. Determine the amount of time left for this execution. (gettimeofday compared with the start time.) One minute of wall clock time is the limit for this part of the assignment (plus any time required for the stopped request). If the time remaining is less than zero, you can skip steps.

9. Construct the fd_set bitsets or pollfd structures by inserting all the (new) sockets. Then call select() or poll() at the core of your code: the new sockets that have not been fully connected in the list to be notified for writing; the connected sockets and the server socket for reading.

10. Handle errors. If select or poll break down because there's a bad file descriptor you're passing in, nothing good will happen until that tainted file descriptor is expunged.

11. Iterate through the ready sockets:

    - If ready for writing on an unconnected socket, finish the connection by calling connect() again. If the connect is successful, send the handshake and return the socket to blocking mode. If the connect() fails, close the socket and mark the peer invalid (or remove it from the collection). Mark the updated state of the peer.
    - If ready for reading on the server socket, call accept(), then getpeername() and populate a connection structure. Mark the updated state of the peer. Ensure this new socket will go into the select/poll structure to read the handshake message when it arrives. (Intermediate deadline.)
    - If ready for reading on another socket, put new data into a connection-specific buffer of unprocessed data. Then, parse the buffer as possible until you're either out of data or you're upon an incomplete message (the length of the message is longer than the bytes remaining in the buffer). Once read(), data cannot be un-read back into the kernel. It is yours to keep. You may implement a helper function to read data, resize buffers, deal with disconnection, etc.

- The messages that could have arrived on that socket might be:
  - "`exit\n`" as in pa2; This is outside the bittorrent protocol. Leave whatever while loop you have, and be sure to complete step 14 below. This is intended to help with swift testing on submit.cs.
  - an incoming handshake response (if you sent the handshake),
  - an incoming handshake request (if you accepted the connection last time around), or
  - a have message, unchoke, choke, etc. If a have or bitfield (bitset) message, update your copy of the bitset. If another type of message, consider printing debugging information (to stderr).

  Which message depends on the current connection state. A closed connection will be ready for reading; read() will return zero bytes. For that socket, close() it and ensure that it doesn't go back into the select/poll parameter. (Failure to do so will make a mess of select and poll)

12. Repeat (from the "Determine the amount of time" step) until the one minute time expires.
13. (Recommended) Close all connections. It's polite to do so, but not necessary, since those connections will be closed when the program exits.
14. Print all peer/bitset combinations to stdout. `"> %u.%u.%u.%u:%u %s"` where the string is a representation in capitalized hex of the bitset. You may use `Print_Bit_Set()` after printing the ip and port.
15. For peers that did not reply (for which no bitset information is available), printing a line is optional. I will look for bitsets only for hosts that have provided one.
16. Complete the event=stopped connection to the tracker as in the previous assignment.
17. `exit(EXIT_SUCCESS)`

6 Example Tests

The torrent files included in this assignment are:

halfway-test-live.torrent Connect to three azureus peers connected to my tracker.

halfway-test-fake.torrent Connect to the three azureus peers, without using the tracker (using the same server as in Fake.torrent). This ensures that three good peers are present. (And at the moment, one broken one from the query.)

Jay.jpg.torrent Similar to halfway-test-live, but shorter and where a single seed has all the bits.

ubuntu.torrent As before, possibly obsolete.

You are encouraged to find other live torrents, presumably for linux distributions that are not of dubious legal status. I have some in the "secret" test bin.

To test incoming connections (intermediate deadline), where of course 58703 is replaced by the port you're listening on:

```
./pa0 localhost 58703 < halfway-test-handshake
```

Remember, pa0 provided a means to send data, in this case I have included an example handshake message for the "halfway-test" torrents. If you lack a functional pa0, you may use simple "telnet localhost 58703" to exercise the connection-accepting code and the "netcat" tool to send the "halfway-test-handshake" content.

7 Overspecification

You may use geekos list.h; there is no list.c. I would ordinarily use an array and mark entries invalid, but my version of the solution uses the list and it works well enough.

You may use geekos bitset.c and bitset.h. I have included them and modified them slightly to build outside geekos. They are not feature rich implementations of a bitset, but they do share a data format with the bitset message. That is, you can directly write them out in a bitset message, and you can copy a received bitset message (just the data portion) and treat it as a bitset. The only thing bitset.c does for you is to set, test, and clear bits. I have added a `Print_Bit_Set` function, not that you needed one.

Otherwise, you may not use any glorious library that makes everything easy that does not come with the language. (For example, glib and the apache portable runtime are forbidden.) If you wrote it yourself, you're welcome to turn it in, though.

You must use the bencode and sha1 provided in the template. They're not terrible.

Torrent files may be arbitrarily large.

You may limit the number of connections to 100 (i.e., arrays of connections may be limited to 100). As in previous assignments, you might be best off using a larger number. The operating system will typically limit the number of connections to FD_SETSIZE (1024 or 4096) as described in the select man page.

O(n) search (e.g., from socket to peer connection) is fine.

8 Hints

As before, do not use str functions on character arrays that may have null bytes in the middle. Using (unsigned char *) instead of (char *) may be helpful to you (it is doubtful that you'd want the sign).

Bittorrent message lengths are integers sent in network byte order.

μTorrent, iirc, sends empty bitfield messages, then sends only have messages to inform neighbors of stored pieces. This means that you must be able to receive several messages after the handshake.

Expect errors! Most connect() attempts will fail because the listed peer is not present. Collectively, we'll likely make this even worse by adding our short-lived peers to the list. Many read() attempts may find that the connection has terminated. Check return values and at least print a message describing each failure so that failures are not silent.

Be careful printing binary data that comes from the tracker. (The peers list is binary.) If you manage to wedge your terminal, type "`echo ^V^O`" and hit enter. (Where `^` represents control, not the literal carat.)

Although I will not forbid it in test code, I recommend that you do not use threads and do not use signals (alarm()). Using both at the same time is incredible disaster. Signals can cause any system call to fail because it is interrupted. Threads can lead to synchronization problems and deadlock, and generally make code hard to run with the debugger.

TCP provides bytestreams. BitTorrent converts those bytestreams into messages by using the length/message id/content framing format described online. Reading from the TCP connection in a way that is robust to partially received messages without blocking is somewhat tricky, and will be very important once you're moving data blocks that are much larger than packets.

There is a macro in sys/time.h called timersub.

```
struct timeval start, now, end, remaining;
gettimeofday(&start, NULL);
end.tv_sec = start.tv_sec + 60;
end.tv_usec = start.tv_usec;
gettimeofday(&now, NULL);
timersub(&end, &now, &remaining);
```

I consider it good practice to annotate variables that store units of time as value_seconds, value_ms, value_us, always with the unit somewhere, especially if they're going to live for a while. A timeval or a timespec both have defined time units, but an integer does not, and so might need the reminder.

Do not attempt to support IPv6; IP addresses from the tracker are necessarily IPv4 addresses when using the compact form. This means you can store struct sockaddr_in's without having to worry about other types of addresses.

Consider mirroring GeekOS global variable naming conventions (`g_` as a prefix).

9 To Turnin

Your code MUST RUN on the submit server. You may have to test the compilation on linuxlab (which has a similar setup). Errors can occur. You may need more #include lines than you expect or they must be in the order on the man page. It is not sufficient that your code work on some machine you happen to know about. IF your code runs on linuxlab but not on submit, we'll grant you a pass on this requirement... at least until we can figure out the mismatch.

10 Finding Code On-line

If you find precisely this online, don't copy it. Cite sources in the top of your file.