CS 453 (Fall 2014): Computer Networks
Programming Assignment 1
Instructor : V. Arun

1. Overview

This assignment is designed to introduce you to socket programming. The assignment asks you to implement a program to download files using a traditional client/server approach (like HTTP) and compare it to a peer-to-peer approach (like BitTorrent).

Goal: Your goal is to write a network program, referred to as the *client*, that downloads an image file from a *server* that we maintain. The client must implement two options:

1. Client/server option: Request the server for the entire file similar to HTTP.
2. Peer-to-peer option: Request the server for addresses of other peers that possess parts of the file, called *blocks*, and download these blocks from different peers.

Your client needs to download the image as fast as possible. Note that the first option will not yield the fastest download as the server (as well as each of the individual peers) services requests at a constrained rate, so relying on only one server (or peer) to get the entire file is likely to yield a poor download rate.

The rest of this document specifies the protocol for the client/server option (Section 2); the protocol for the peer-to-peer option (Section 3); design constraints, hints and suggestions (Section 4); submission and auto-grading instructions (Section 5).

---

2. Client/server option

In the client/server option, the client requests the server for the entire file using TCP similar to HTTP. The server is running on the following <hostname, port number> combinations: **<plum.cs.umass.edu, 18765>** and **<pear.cs.umass.edu, 18765>**. You may use either one of the two servers. We support two servers just for fault-tolerance and to defend against unintentional DoS attacks when everyone decides to stress-test their client the weekend before the assignment is due (so be smart and finish early).

File request format: The client must send a request in the following format to the server to download filename, where the "\n" at the end is the newline character.

```
GET <filename>\n
```

The filename to use for this assignment is Redsox.jpg. Thus, the client must send the following string to request the file.

```
GET Redsox.jpg\n
```

File response format: The response to the file request will have a header followed by the body. The header is terminated by two newlines, i.e., the string "\n\n", which marks the beginning of the body. A typical response might look as follows.

```
200 OK
BODY_BYTE_OFFSET_IN_FILE: 0
BODY_BYTE_LENGTH: 58241

<bytes of body follow here>
```

In the client/server option, the entire file is sent in the body. The client-server option will be slow, which is why we also have a peer-to-peer option that your client must also support.

---

3. Peer-to-peer option

In the peer-to-peer option, the client must first obtain the "torrent metadata" from the tracker and use this metadata to download data blocks. The torrent metadata contains information about the number and size of blocks constituting the file and peers (<IP,port> combinations) from which the blocks may be downloaded.

3.1. Torrent metadata

Torrent metadata request format. The client must download the torrent metadata for filename by sending a UDP message in the following format:

```
GET <filename>.torrent\n
```

Thus, to request the torrent metadata for Redsox.jpg, the client must send a UDP message containing the string "`GET Redsox.jpg.torrent`" (with no newline).

This UDP-based torrent metadata server is running at **<plum.cs.umass.edu, 19876>** and **<pear.cs.umass.edu, 19876>**. As above, you may use either one of the two.

Torrent metadata response format: The response to the request for torrent metadata is in the following format:

```
NUM_BLOCKS: 6
FILE_SIZE: 58241
IP1: 128.119.245.20
PORT1: 3456
IP2: 128.119.245.20
PORT2: 4321
```

The names of the fields above are self-explanatory. `NUM_BLOCKS` is the number of blocks in the requested file. `FILE_SIZE` is the size of the entire file in bytes. All blocks except the last block are of the same size, namely, `floor[FILE_SIZE/NUM_BLOCKS]`. `IP1` and `PORT1` identify the IP address and port number of the first peer. Similarly, `IP2` and `PORT2` identify the second peer.

Each response will contain two randomly chosen valid peer identifiers. You can query the tracker multiple times to get more peer identifiers. However, the tracker is designed to rate-limit the queries, so you may not get responses promptly if you send requests too fast or may not get responses at all as UDP messages can get lost.

3.2. Data blocks

Having obtained metadata information using UDP as above, data blocks must be requested using TCP as follows.

Request format: The following request fetches a specific block

```
GET filename:<block_number>\n
```

where `block_number` is an integer identifying the block in filename. For example, you may request block 24 in Redsox.jpg by sending the string "`GET Redsox.jpg:24\n`" to any one of the peers received in the torrent metadata above. Note that the servers listed in the client/server option also act as peers and support the above request format to request specific blocks.

Specifying "*" instead of a block number returns a randomly chosen block

```
GET filename:*\n
```

Response format: The response to a block request has the following format as that of the whole body. The only difference is that the starting byte offset in the file in general will be non-zero and the size of the block will be much smaller than the size of the file, for example:

```
200 OK
BODY_BYTE_OFFSET_IN_FILE: 20000
BODY_BYTE_LENGTH: 10000

<bytes of body follow here>
```

All blocks except the last block will be of the same size.

---

4. Design constraints

Your objective is to do download the file as fast as possible. You can do whatever you like with your client program. But you **MUST** abide by the following behavior; not doing so will be considered scholastic dishonesty.

- Your code must be completely your own (but you can discuss strategies for the P2P option with each other).
- You can not launch denial-of-service (DoS) attacks on the servers.

4.1. Hints, suggestions, things to keep in mind

1. We encourage you to use multiple threads to simultaneously download blocks from different peers. Remember to carefully synchronize access to shared data structures when using threads. If you have not taken an Operating Systems course or do not otherwise have experience using threads, this part will incur a steep learning curve.
2. Plan to complete the assignment well before the due date. Successful execution of the entire program to download the file may take many minutes. So, debugging and testing may take longer than you might expect.
3. Comment your code as much as possible. Remember that documentation is as much for your own benefit as for the benefit of others who read your code.
4. An easy way to check if you got all the bytes in the file correctly is to save it to a file and view the downloaded image. This manual check will not work with the grading server as the bytes of the file are scrambled.
5. All data transfer uses TCP, but the torrent metadata tracker uses UDP.
6. Use correct <IP address, port> combinations, otherwise you will get no response.
7. You are strongly encouraged to explore and use other commands supported by the Java/Python/your-favorite-language's socket API. You are unlikely to be able to write the client program using the commands you have seen in example code shown in the class alone.
8. Blocks are numbered starting from 0. Block offsets (the position of the first byte of the block in the file) start from 0.
9. Make sure your client is resilient to long lived connections and delays in data reception. You can not expect the entire data to be received right away in a bytestream channel. The servers are designed to rate-limit data at different rates.
10. Your connections will get closed if you send bad commands. At random times, your connection may also get closed for no good reason as the server periodically changes the peer-to-peer ports.
11. You can use `telnet` as we saw in class to check that the servers actually work. For example, you can do telnet <IP> <port> and send commands such as `GET Redsox.jpg\n` or `GET Redsox.jpg:3\n` and the server will return the corresponding data (in binary) on the command line.
12. You have another utility command called `GETHDR` available to just get the header. This command works just like `GET` but only sends the header, not the body. Your client does not need to use this command. It is provided only to quickly check using `telnet` that the servers are up and running correctly.
13. The grading machine (**date**) will be slower and/or more challenging than the "development" machines (**plum** and **pear**), so use the latter for quick development and testing and the former to actually be graded and get credit. A grading server on the grading port also runs on each development machine and you can use this to debug your (near-final) client, but we will only record grades through challenge-responses performed on the grading server.

5. Submission instructions on moodle

1. *You must submit your client program*. Name the main file TorrentClient.<appropriate-extension>. Feel free to code up additional helper classes to this main file.
2. *You can use any programming language of your choice*. The nice part about network programming using the socket API is that the server and client may be written in different programming languages.
3. We suggest the following command line format for running the client in the client/server or P2P mode. For example, if you are using java, use
   java TorrentClient CS <IP> <port>
   java TorrentClient P2P <IP> <port>
   where "CS" means that the client will use the client/server option and "P2P" means that the client will use the P2P option. The IP and port are the IP and port of the (TCP-based) file server in the former case and the UDP-based torrent metadata server in the latter.

   Submit all your files as a single tarzip with your TorrentClient program in the top-level directory.

4. For PA1(a), your client only needs to implement the client-server (CS) option as described in Section 2 above. *You must save and include the downloaded Redsox.jpg file in your submission in the same top-level directory as your TorrentClient program.* Your downloaded file must be viewable as an image.We will also check that running your client that it does indeed produce the file you included.

5. For PA1(b), your client must additionally implement the peer-to-peer (P2P) option as described in Section 3 above and the three steps described in 5.1 right below.

5.1. Online Auto-grading (necessary only for PA1(b))

Your client program will be auto-graded by a grading server. The grading server is a TCP server running at **<date.cs.umass.edu, 20001>**. Here are the steps your client needs to implement to be auto-graded.

Step 1: Your client must send the command `IAM <studentID>\n` where `studentID` is your UMass ID to the grading TCP server. The server will respond with the following challenge message (without any newlines):

```
studentID_redsox.jpg : What is the 4-byte
XOR checksum, i.e., the sequential XOR of
all 4-byte words, of the file with this
name? Your testing time starts NOW!
```

Keep this grading TCP connection open because you will need it in Step 3.

Step 2. Next, your client must download the file as quickly as possible from the same machine as the grading server via the ports listed in the client-server or peer-to-peer sections above. For example, for the client-server option, your client should send the request `GET studentID_redsox.jpg\n` to **<date.cs.umass.edu, 18765>**; likewise for the peer-to-peer ports. Note that all grading is confined to a single machine, i.e., you can not download the challenge file from a different machine (e.g., from **plum** or **pear** when the grading server is running on **date**) as this will simply be processed as a bad request. Note also that the challenge file to be downloaded is customized specifically for you and is created (with random bytes) only when you send the `IAM` command in Step 1.

Step 3: After downloading the challenge file, your client must respond by sending exactly four bytes followed by a newline on the grading TCP connection opened in Step 1 above. If your answer is correct, you will receive the following response message and you are all done! You can upload your code to moodle.

```
Success! You have answered the challenge
correctly in <num_seconds> seconds.
```

If your answer is incorrect, you will get the following response message.

```
Failure! You have not answered the challenge
correctly. But you may try submitting more
answers over this connection.
```

You will be able to see your current performance stats (as well as that of your classmates') at this page. Your goal is to answer the challenge ASAP. Have fun!

6. FAQ

1. *How exactly do we compute the 4-byte XOR checksum?*

   Here is a concrete example, suppose your file consists of 13 bytes b0, b1, b2, ..., b12. Organize these bytes as 4-byte words as follows.

   w1 = b0 b1 b2 b3

   w2 = b4 b5 b6 b7

   w3 = b8 b9 b10 b11

   w4 = b12

   We need to compute w1 XOR w2 XOR w3 XOR w4. Your answer should be four bytes long. Thus, the first byte of your answer will be b0 XOR b4 XOR b8 XOR b12. The fourth and the last byte of your answer will be b3 XOR B7 XOR b11.

   For example, you should be able to verify that the 4 bytes in the XOR checksum of the string *"Always remember that you are unique. Just like everyone else."* (without the quotes) when printed as integer values (in the range [-127, 127]) are the numbers [119,15,106,62]. If you try to print these four bytes as a string, you should see "wj>". In general, the 4-byte checksum may not have nice, readable ASCII characters like in this simple example.

2. *Why is the file download so much slower on the grading server compared to the development servers?*

   That is intentional. The development machines are provided to you for quick testing and debugging. Go to the grading server only when you think you have the near-final client ready.

3. *Can I download the file quickly from the development servers in order to answer the challenge from the grading server?*

   No, you can try it and you will see that you fail. The reason is that the grading server creates a unique random file just for administering the test for you. Remember that all grading related stuff is confined to a single machine, namely, the grading server specified above.

4. *What download times are reasonable?*

   For the client-server option, there is no strategy involved. You should see a download time of around a couple mins on the grading server, and around a few seconds on the development servers. When you implement the peer-to-peer option and do it well, you will find that you can reduce the download time on the grading server significantly as well, possibly down to a few seconds. The faster, the better.

5. *I am paranoid. I just need to implement the client, right? No server, right?*

   Yes, my paranoid dear, yes. If still in doubt, read the "Goal" at the very beginning of this assignment.

6. *Is verifying using the grading server the only way of proving that our client can download the file correctly?*

   Yes, we have no other way of verifying that you downloaded the file correctly. With the grading server, you also get feedback on where you are going wrong, and you also get to see others' download times on the statistics page. Plus it is only two simple additional steps (Steps 1 and 3 in Section 5.1 above) you need to implement in your client.

7. *I am not getting any response from the server. What's up with that?*

   You are quite likely missing a newline. Read the protocol described above carefully. If you send badly formatted requests, the server will just close your connection angrily. But if you send a prefix of a correctly formatted request (e.g., without a newline), the server is going to wait for you to complete your request. What else can it reasonably do over a bytestream channel after all?

   When in doubt, try using telnet. If telnet works, the servers are working fine.

8. *I tried using readLine like we saw in class, but I run into problems.*

   There are no "lines" in a binary image file, so readLine() is not meaningful. Explore other commands to read bytes in the socket API of the language you are using. You know how many bytes to read by inspecting the header.