Networks and Distributed Systems	Project 4: Selfish BitTorrent Client
CS3700 Spring 2015	March 30, 2015

This project is due at 11:59:59pm on April 22, 2015 and is worth 15% of your grade. You must complete it with a partner. You may only complete it alone or in a group of three if you have the instructor's explicit permission to do so for this project.

Note that there are three milestones for this project, at 11:59:59pm on April 6, 2015; April 13, 2015; and April 20, 2015. More details are in the Milestones section below.

1 Description

You will modify the trading policy of a simulated BitTorrent client in order to improve the download performance. Unlike previous projects, a significant fraction of your grade in this project will come from performance. Part of your grade will come from how your code performs when run against the solutions of your the course staff, and part will come from how your code performs against your peers.

In this project, you will have to do background reading on BitTorrent. Since we are providing you with a working implementation, you are *expected* to research the BitTorrent trading scheme, improvements to it, and possible attacks. *Simply turning in the code that we provide (or a slightly modified implementation) will result in a very low grade.* You should write selfish code, the goal of which is to improve your download speed and decrease others' download speed.

2 Requirements

We will provide you with the code you will need to make the project functional. Thus, all you must do is increase the performance of your client, while always correctly downloading the file (i.e., you should never accept part of the file that is incorrect, or print out the file in the wrong order). In particular, your code must significantly outperform the code written by the course staff, in terms of average download performance.

You should not assume that other peers will follow the protocol. Since these are the solutions of your peers, they may be malicious, they may upload bad data, they may lie to you, and the certainly will be greedy and selfish. Your solution should handle these errors gracefully, recover, download the file correctly, and *never* crash.

3 Your client program

For this project, you must use python (the BitTorrent simulator is written in python, thus C is not supported). You will write a python class that has the following basic structure:

```
class MyChosenNameForMyPeer(Peer):

  @timeout(1)
  def __init__(self, id, hashes, ids):
```

```
    Peer.__init__(self, id, hashes, ids)

    # Any initialization you would like to do

  @timeout(1)
  def next_round(self, round, messages):
    # Handle the messages
```

As shown above, your code *must* (a) extend the Peer class, (b) include the `@timeout(1)` calls before each of the two methods, and (c) make the call to the superclass constructor as shown above. You are free to add any class variables, helper functions, etc.

A few additional requirements of your code: Your code must fit into a single file (i.e., due to some limitations with how the simulator loads your code, you cannot spread your code across multiple python files). You should select a unique name for your peer (do *not* use MyChosenNameForMyPeer), and you should name your python file MyChosenNameForMyPeer.py (where MyChosenNameForMyPeer is the name of your peer; capitalization is important). Finally, your code must do all logging using the provided `self.log(message)` method; do *not* simply call `print`.

3.1 Initialization method

When the simulator begins, your class constructor (`__init__`) will be called with the following arguments:

`id` This is the id string that uniquely identifies this peer. There may be multiple instances of your peer in the simulator operating under different ids.

`hashes` A python list of hashes (strings) for the pieces in the file. This is how you find out how many pieces there are in the file. Note that you can call the provided superclass method `Peer.hash(piece)` to run the hashing function over a received piece to see if it matches the expected hash.

`ids` A few other peer ids to get you started in the network. These are not guaranteed to be your own peers, and occasionally these peers may be dead.

3.2 Next_round method

The simulator runs on a round basis, where each peer receives and sends messages each round (i.e., the system is turn-based). Each round, the simulator will call your peer's `next_round` method with the following arguments

`round` The numeric round number. Increases by 1 each round.

`messages` A list of messages that were sent to the peer in the prior round. Not all messages are guaranteed to be delivered (see below). Each message is a python dictionary.

The return value of this function can take two forms. First, the return value could be a list of messages that your peer wishes to send (again, each message is a python dictionary with the

appropriate values). Second, the return value could be a string. This should *only* be done when your peer wishes to exit. The value of the string should be the concatenation of all pieces put together (e.g., something like `"".join(pieces)` if you are keep the pieces in the list `pieces`). You should make sure that all of your pieces are correct and are in the right order; if you exit with an incorrect string, the simulator will treat you as if you exited last. Regardless, once you return a string, your peer will no longer exist.

3.3 Messages

All peers in the simulator communicate via datagram-like messages (i.e., the system works in a connectionless manner). Each message is a python dictionary, and the message is required to have keys `from`, `to`, and `type`. Each of these are strings with the obvious meanings. Additionally, there is one additional required field: `pieces`. The value of this key is a bit-vector (a python list containing 1s and 0s) indicating whether the node that is sending the message has the corresponding piece (a 1 indicates so). You *must* include all four of these required fields on every message, or your message will be rejected by the simulator. Messages *may* have additional keys, and your client should accept such messages (and silently ignore them if it does not understand them).

Not all messages are guaranteed to be delivered. The simulator enforces "bandwidth" limits on peers, and will drop messages if the peer sends or receives too much in a single round. The course staff reserve the right to modify how this works over the course of the project.

There are four basic types of messages that you will use to implement your peer's functionality:

Peer-request message This message has the type `peers-request`, and has no other required keys. This message indicates that the sending client would like to know some of the peers with whom the destination is communicating with.

Peer-response message This message has the type `peers-response`, and contains an additional key `peers` which maps to a list of peer ids. The peer is free to choose how many peers to include in its response, but the recommended behavior is to select a random 3 peers.

Piece-request message This message has the type `piece-request`, and has the additional key `piece` containing the numeric value of the piece requested (e.g., piece 73). This message indicates that the sending client would like the receiving client to send it that piece.

Piece-response message This message has the type `piece-response`, and has the additional keys `piece` (containing the numeric value of the piece requested) and `data` (containing the piece itself).

You are free to create messages however you see fit, but your code should look something like

```
{ "from": self.id, "to": message["from"], "type": "peers-response",
  "peers": random.sample(self.peers.keys(), 3) }
```

3.4 Rules of the road

In this project, you are encouraged to manipulate, lie, cheat, and steal your way to finishing first. However, there are certain behaviors which are and are not allowed:

- You *may not* attack or exploit the simulator infrastructure directly (e.g., calling functions outside of your code, changing others' memory, etc). You must limit your behavior to sending and receiving messages.
- You *may not* conduct denial-of-service attacks on the simulator infrastructure (e.g., running the simulator out of memory, hogging the CPU, etc). The simulator is build to withstand some, but not all, such attacks.
- You *may not* launch any subprocesses, threads, or any other code outside of the two methods specified above. Of course, you're encouraged to modularize your code as you see fit (within your single .py file), but don't launch any threads or processes.
- You *may not* communicate between multiple instances of your own peer via any mechanism other than sending of messages through the simulator (e.g., via global variables, side channels, etc). This would defeat the purpose of the assignment.
- You *may* create new message types (e.g., to try and identify other instances of your peer).
- You *may* falsify any messages fields that you wish (including `from` and `to`).
- You *may* lie to other peers about which pieces you have.
- You *may* upload incorrect data to other peers.
- You *may* treat other instances of your own peer differently than others.

Violations of these rules will result in a 0 on the project. If you have any specific questions about whether a certain behavior is in or out of scope, please contact the course staff.

4 Starter code

The starter code contains an example peer in the file `SimplePeer.py`. You should *not* modify this code, but you are welcome to use this code as a basis for your peer. Regardless, when implementing your code, be sure to use the `self.log(message)` method for any debugging output. As described above, simply create a file `MyChosenNameForMyPeer.py` (it must end in `Peer.py`) and the simulator will automatically find it.

You can compare different implementations of your peer by simply having multiple `...Peer.py` files in your directory; the simulator will find and launch instances of all of them. Just be sure to name them differently (and the class name and the filename have to match).

The starter code contains a version of the simulator that lets you run your code against the provided starter SimplePeer. You should be able to run this simulator on any CCIS Linux machine; you can do so by running

```
./simulate
```

All debugging information will be written to standard output. As the simulator runs, it will print out all `log()`-ed output, including the round number and the peer who issued the log message. Once the simulator is done, it will output the order that the peers (correctly) finished in. Obviously, you want to be near the top of the list.

5 Shark tank

In order for you to test your code against others', we will be providing a simulator that will periodically run your code against the code of your peers. Called the Shark Tank, it will allow you to try out new approaches to maximize your performance, to test how well your code is working, and to make sure that your code works in the test harness that we will use to grade. The Shark tank is a modified version of the simulator that is designed to run all students' code together; it shares many properties of the simulator but is different in a number of ways.

5.1 Submitting your code

After each run of the simulator, if your peers all successfully download the file, the simulator will ask you if you want to submit this version of your client to the shark tank. If you do, this code will be included in the next run.

```
Peer MyPeer successfully completed the run.  Would you like to post it to
the shark tank? [yn]
```

You are allowed to have multiple peer names in the shark tank, but the shark tank will only run at most the most recent three distinct peers.

5.2 Shark tank logs

The shark tank is configured to run every 10 minutes. The results will be posted periodically in `/course/cs3700sp15/stats/project4/results`, with each file representing one run. The file contains the order that peers exited (correctly), starting with the first. To help you in debugging, the most recent 10 log files are located in `/course/cs3700sp15/stats/project4/logs`. These can be quite large, so be careful with copying them to your local home directory.

You can view the aggregated shark tank results by calling the program

```
/course/cs3700sp15/bin/project4/printstats
```

This program takes the latest 20 runs of the shark tank and prints out each peer's average probability of finishing before the other peers (e.g., a score of 1.0 would indicate that your peer *always* finishes first). The output is a sorted list of peer names:

```
GreedyPeer                : 0.737
SimplePeer                : 0.587
FooPeer                   : 0.566
MyPeer                    : 0.522
EvilPeer                  : 0.070
```

In this case, GreedyPeer is performing the best.

5.3 Grading in the shark tank

Your code will be graded relative to both the simple peer provided by the course staff and the submissions of your classmates (i.e., on average, what order does your code finish in). In general, about 50% of your performance grade will come from how well you perform against both of

these. The course staff reserve the right to tweak this assignment over the course of the project. Additionally, the course staff may introduce additional malicious and evil peers into the shark tank over time; you are expected to handle these gracefully.

6 Submitting your project

6.1 Registering your team

You and your partner should first register as a team by running the `/course/cs3700sp15/bin/register` script. You should pick out a team name (no spaces or non-alphanumeric characters, please) and run

```
/course/cs3700sp15/bin/register project4 <teamname>
```

This will either report back success or will give you an error message. If you have trouble registering, please contact the course staff.

You must register your team by 11:59:59pm on April 3, 2015.

6.2 Milestones

In order to ensure that you are making sufficient progress, the functionality grade will be computed as the semester progresses. There are three milestone deadlines, each at 11:59:59pm on April 6, 2015; April 13, 2015; and April 20, 2015. At each of these times, a "snapshot" of the shark tank will be taken, and each will constitute 10% of your grade for the project (i.e., the course staff will run a separate shark tank with all of the submitted code).

For each of the milestones, you must also turn in the current version of your code. You can do so by running

```
/course/cs3700sp15/bin/turnin project4-milestone<X> <dir>
```

Where `<dir>` is the name of the directory with your submission and `<X>` is the milestone number (1 for April 6, 2015, 2 for April 13, 2015, and 3 for April 20, 2015). Again, the script will print out every file that you are submitting, make sure that it prints out all of the files you wish to submit! You should receive an email confirmation of your submission.

6.3 Final submission

For the final submission, you should submit you (thoroughly documented) code along with a plain-text (no Word or PDF) README file. In this file, you should describe your high-level approach, the challenges you faced, a list of properties/features of your design that you think is good, and an overview of how you tested your code.

You should submit your project by running the `/course/cs3700sp15/bin/turnin` script. Specifically, you should create a `project3` directory, and place all of your code and README files in it. Then, run

```
/course/cs3700sp15/bin/turnin project4 <dir>
```

Where <dir> is the name of the directory with your submission. Again, the script will print out every file that you are submitting, make sure that it prints out all of the files you wish to submit! You should receive an email confirmation of your submission.

You must submit your project by 11:59:59pm on April 22, 2015.

7 Grading

The grading in this project will consist of

15% Style and documentation

35% Program functionality

10% April 6, 2015 sharktank status

10% April 13, 2015 sharktank status

10% April 20, 2015 sharktank status

20% Final sharktank status

You are, however, going to be graded on how gracefully you handle errors. Remember, network-facing code should be graded defensively; you should always assume that everyone is trying to break your program. To paraphrase John F. Woods, "Always code as if the [the remote machine you're communicating with] will be a violent psychopath who knows where you live."

8 Advice

A few pointers that you may find useful while working on this project:

- Check the Piazza forum for question and clarifications. You should post project-specific questions there first, before emailing the course staff.
- Finally, get started early and come to the instructor's office hours and TA lab hours. You are welcome to come to the lab and work, and ask the TA any questions you may have.