CS 136 Assignment 2: Peer-to-Peer BitTorrent Simulation

David C. Parkes
School of Engineering and Applied Sciences, Harvard University

Out Friday Sep 15, 2017
Due **5pm** sharp: **Friday Sep 22, 2017**
Submissions to Canvas class page

This is a group assignment to be completed by groups of **up to 2 students each**. While you are permitted to discuss your clients with other students as much as you like, each group must write their own code and explanations. In addition, there will be a small bonus for clients that perform well in the class tournament. If you want a partner and don't have one, post to Piazza. Your submissions should be made to Canvas: the code must be in .py files and the writeup of the analysis must be in PDF format. Both partners must contribute to all aspects of the assignment. It should go without saying that you must write all of the code yourself, per Harvard's Academic Integrity Policy. If you use any code whatsoever from any source other than the official Python documentation, you **must** acknowledge the source in your writeup.

Goal

In this assignment you will program a number of clients for the BitTorrent protocol and test them in a simulation environment. You will get points for implementing the strategies and your writeup (conciseness, reasoning, analysis).

Setup

Tools: Make sure Python 2.7.x (for some $x \in \{2, .., 10\}$) works on your computer system (`http://python.org/download/`). The simulator code will not run in Python version 3!

Obtaining materials: Download the BitTorrent .zip-archive from the class website (archive posted on Piazza) and make sure you have all the relevant files: `dummy.py`, `history.py`, `messages.py`, `peer.py`, `seed.py`, `sim.py`, `start.py`, `stats.py`, `util.py`.

Generating .py-files: Pick a cool group name, perhaps based on your initials, so that it will be unique in the class. Run `python start.py NAME`, substituting your group name for `NAME`. This will create appropriately named template files for your clients. For example, if your group name was "abxy":

```
> python start.py abxy
Copying dummy.py to abxystd.py...
Copying dummy.py to abxytyrant.py...
Copying dummy.py to abxypropshare.py...
Copying dummy.py to abxytourney.py...
All done.  Code away!
```

In each of the files, you will need to change `Dummy` in line 16 (where it says `class Dummy(Peer)`) to your group name and the client specification (e.g. `AbxyStd`, `AbxyTyrant`, `AbxyPropShare`, `AbxyTourney`).

The simulator: You are given a BitTorrent simulator. It has the following general structure:

- Time proceeds in integer rounds, starting at 0 (0, 1, 2, ...).
- There is a single file being shared, composed of `num-pieces` pieces, each of which is composed of `blocks-per-piece` blocks.
- There is a set of peers, which should include at least 1 seed. Seeds start with all the pieces of the file and have the maximum upload bandwidth. Other peers start with none of the pieces.
- Each peer has an upload bandwidth, measured in blocks per round. This is chosen uniformly at random from a specified range (except for seeds, who have the maximum bandwidth). Download bandwidth is unlimited.
- Each round proceeds as follows:
  1. Each peer must provide a list of `request` objects, each asking for blocks from a particular piece from a particular other peer. This is the `requests`-function.
  2. Each peer is passed the requests it has received in this round, and asked to return a list of `upload` obejcts which specify how much of its upload bandwidth to allocate to any given peer. This is the `uploads`-function.
  3. These lists are checked for consistency, i.e. requests must ask for pieces the peer has, uploads must add up to no more than the peer's bandwidth cap, etc.
  4. Peers who are being uploaded to get their blocks. Multiple requests are satisfied in order until the shared bandwidth is exhausted. For example, if peer 1 requests the last 7 blocks of piece 12, and also piece 11 starting at block 0, and peer 2 includes `Upload(from_id=2, to_id=1, bw=10)` in its list of uploads, peer 1 will get the last 7 blocks of piece 12, and the first 3 of piece 11.
  5. The events of this round are saved in a `history` object, which is available in future rounds.
- The simulation ends once all peers have all the pieces or the maximum number of rounds `max-round` is exceeded.

Building clients: Familiarize yourself with the provided code. You should not need to change any of the provided files, only modify the ones created by `start.py`. The only files you should need to read in any detail are `messages.py` and `dummy.py`, along with a skim of `history.py`.

Testing clients: Here is an initial test command to run. It sets a bunch of simulation parameters, and creates 1 seed and 2 dummy clients.

```
> python sim.py --loglevel=debug --num-pieces=2 --blocks-per-piece=2
  --min-bw=1 --max-bw=3 --max-round=5 Seed,1 Dummy,2
```

Make sure you understand the output.
`> python sim.py -h` will show all the command line options.
When analyzing the performance of your clients, you can set the log level to `info`. Use `--iters=...` to extract statistics for multiple runs of the same setup. A good starting point for your testing may be

```
python sim.py --loglevel=info --num-pieces=128 --blocks-per-piece=32
  --min-bw=16 --max-bw=64 --max-round=1000 --iters=32 Seed,2 AbxySomeclient,10
```

but we encourage you to try varying parameters.

Problem Set

1. Designing your clients:

   (a) Implement the BitTorrent *reference client* as described in Chapter 5, including rarest-first, recipocation and optimistic unchoking. This should be class `TeamnameStd` in `teamnamestd.py`. Not all the details are in Chapter 5, so you will have to make some assumptions. Explain all of the assumptions you make in your writeup.

   (b) Implement the *BitTyrant client*. This should be class `TeamnameTyrant` in `teamnametyrant.py`. You may have to introduce additional accounting and book-keeping procedures beyond what is provided by `peers` and `history`.

   (c) Implement the *PropShare client*. This should be class `TeamnamePropShare` in `teamnamepropshare.py`. The *PropShare client* allocates upload bandwidth based on the downloads received from peers in the previous round: It calculates what share each peer contributed to the total download and allocates its own bandwidth proportionally. In addition it reserves a small share of its bandwidth for optimistic unchoking (e.g., 10%). For example

   - In round k the client received $4, 6, 1, 9$ blocks from peers A, B, C, D, respectively
   - In round $k + 1$ peers A, B, E, F request pieces from the client
   - The client allocates $\frac{4}{4+6} \cdot 90\% = 36\%$ and $\frac{6}{4+6} \cdot 90\% = 54\%$ of it's upload bandwidth to A and B, respectively
   - E is randomly selected and allocated 10%

   (d) Write a client for the class competition in class `TeamnameTourney` in `teamnametourney.py`.

      i. Write a client that does not freeze the simulation.

      ii. Win the tournament: We will run the tournament in a neighborhood containing one instance of each group's tournament client and 2 seeds. This setup will be run 256 times. Points will be awarded according to rank by average time to complete the download of the entire file. I.e. if 10 clients particpate, the fastest client will recieve 10 points, the second 9, and so on, and so on...
      Likely parameters for the competition are

```
python sim.py --loglevel=info  --num-pieces=128 --blocks-per-piece=32
  --min-bw=16 --max-bw=64 --max-round=1000 --iters=256 Seed,2 Client1,1
  Client2,1 ...
```

but they may vary depending on runtime of the clients.
Your client can access the simulation configuration via `self.conf` (see the bottom of `sim.py` for the available parameters), so you shouldn't hard-code any of these values.

2. Analysis: Provide your answers based on test runs with the clients you programmed. When asked for comparative performance results, provide some evidence (e.g., simple statistics) for the results you are reporting/describing.

    (a) For the standard client explain what assumptions or decisions you had to make beyond those specified in Chapter 5.

    (b) Write a concise summary of the strategies you used for the tournament client, and why you chose them.

    (c) Outperforming the standard client:

    i. How does the *BitTyrant* client do in a population of standard clients?
    ii. How does the *Tourney* client do in a population of standard clients?
    iii. How does the *PropShare* client do in a population of standard clients?

    Look at the relative ranking of the clients and the percentage improvement (or impairment) in the number of rounds it takes the client to get the complete file.

    (d) Overall performance of populations:

    i. How does a population of only *BitTyrant* clients perform? What about a population of only *Tourney* clients?
    ii. How does a population of only *PropShare* clients do?

    Look at the time it takes to get the file out to all clients (i.e., when does the last client complete downloading the whole file), as well as the average download time for the individual clients.

    (e) Write a paragraph about what you learned from these exercises about BitTorrent, game theory, and programming strategic clients? (We aren't looking for any particular answers here, but are looking for evidence of real reflection.)

3. Theory:

    (a) State three ways in which the peer-to-peer file sharing game of the BitTorrent network is different from a repeated Prisoner's dilemma.

    (b) State three ways in which the BitTorrent reference client is different from the tit-for-tat strategy in a repeated Prisoner's Dilemma.

    (c) Explain two reasons why just having a BitTorrent client that is a best response to itself is insufficient for this client to form an equilibrium in a peer-to-peer system.

Comments and Hints

Python features: Useful python features / functions / modules that you may want to google and maybe use:

- random: shuffle, choice, sample
- min, max, map, filter, zip
- set
- list and dictionary comprehensions

Programming in Python: For general information on Python and tutorials, check out `http://docs.python.org/tutorial/`.

Design:

- Make sure you sort things correctly (you'll often want decreasing instead of the default increasing)
- Beware of symmetry. All peers know about all the others, and there's no lying about available pieces modeled, so silly things can happen; e.g., if all your tit-for-tat peers use the same deterministic algorithm for choosing which piece to request next, you may have a situation where everyone always has the same pieces, there's nothing to trade, and the system devolves to very slowing downloading of everything from the seed.
- Debug using small numbers of peers, pieces, and blocks so you can see what's happening. Realize that the relative performance of different strategies is different at very small scales (e.g. tit-for-tat will not kick in if you only have 2 pieces).
- Look at more than one round of history when making peering decisions. Otherwise thrashing will occur (I upload to you in round 1. You didn't upload to me. So in round 2 you upload to me to reciprocate, but I've already moved on to someone else).

Debugging: Make sure your clients work with the unmodified simulator. If you changed anything, re-download the simulator and double check that your clients work with it.

Manipulations: The code is designed so that it's very hard to mess up the main simulation accidentally, but because everything is in the same process, it is still possible to cheat by directly modifying the simulation data structures and such. **Don't!**
Also don't use the specific ids of peers in any of your decision making, e.g.
`if peer.id.startswith("MyClient"): special-case-code`.

Bugs: If you find bugs in the code, let us know.