CS 3251: Computer Networking I (Section A and B) **Spring 2022**

Programming Assignment 2

Handed Out: Feb 23^{rd}*,* 2022 *Due: 1:59pm, April* 13^{th}*,* 2022

1 Introduction

BitTorrent is a peer-to-peer (P2P) file sharing service that transfers files between hosts without the involvement of a central server. This provides scalability benefits compared to client-server architectures, where the server can be a performance bottleneck. Peer-to-peer systems are also popular for an unethical reason, namely that they may help evade law enforcement when distributing copyrighted material. You can read an overview of P2P systems in general and BitTorrent in particular in the Kurose and Ross textbook, 8th Edition, Section 2.5.

In this assignment you will be creating a mini-version of a file sharing service that has some characteristics in common with BitTorrent. There are two primary components that you will code: (1) a **P2PTracker** responsible for keeping track of *registered* P2PClients, and (2) a **P2PClient** that stores files (or data objects), answers requests about which files it has, and (in a full system) delivers files to peers (other P2PClients) upon request. You will not have to deliver the actual files. The P2PClient can also make requests about which files a peer has, and register/deregister with the P2PTracker. As you can see from this description, the P2PClient acts as both a client (requesting file information) and a server (delivering file information). You may notice that the P2PTracker is a server, thus the system is not purely peer-to-peer. The file transfers, however, only happen between peers. More detail on the required functionality of the P2PTracker (**also called the Bootstrapper**) and the P2PClient (also called the Client or peer) you need to program is below.

We will grade the assignment using an autograder. It is to your advantage to read the assignment in its entirety, before commencing work on it. Though not intentional, there will undoubtedly be some details that are not clear from the English language description. This is part of why finite state machines are useful to specify protocols! You should expect some clarifications to emerge as students start to work on the assignment. Piazza posts will be useful at that stage.

2 Overview of the System

Each data object to be shared is identified by a 32 hex-character key that is the MD5 hash of the data object's content. We will give you the keys; you will not need to compute them. For our purposes, we will assume that these hash value keys are unique. You will also not need to keep actual files, since the point of this assignment is to model the tracking and finding of data objects, but not really transmit data files.

Each P2PClient keeps two data structures, a *Content List (CL)*, which is simply a list of the keys for the data objects the client currently holds, and a *Content Originator List (COL)* which is a list of the keys for data objects that this Client has received from other Clients. For each key in the COL, the IP address and port number of the sending client is kept, as illustrated in the figure below (Figure 1).

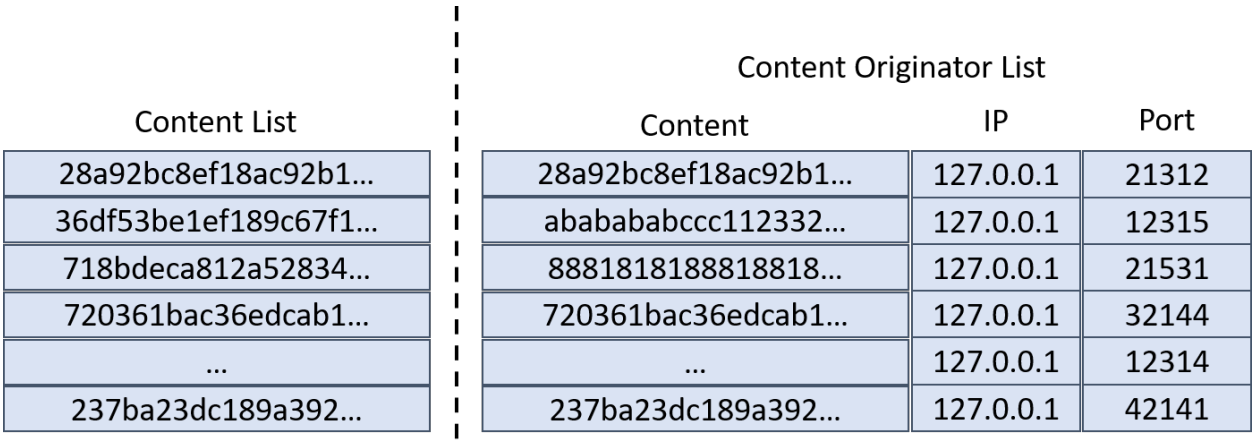

Figure 1: Data-structures at each client

Clients may delete data objects from their content list at any time. When they do so, the corresponding key is immediately removed from the CL (but not from COL). News of the deletion is not propagated any further; other clients are not told that an object has been deleted. As a result, the COL list at other peers may be out of date, i.e., a COL list may indicate that a particular peer has a data object when, in fact, that peer has already deleted the content. This leads to a searching and routing problem, checking to see if a peer has a data object, and if not, discovering other peers to ask.

The P2PTracker is the mechanism that allows a P2PClient to find out about currently registered peers in the system. The P2PTracker keeps a list of all currently registered peers (via a Client ID and the IP address and port number for the client) and provides that list upon request. The P2PTracker also sends a start command to all peers who are registered, in order to trigger the start of peer action, and to enforce some command ordering across peers, which makes it easier to test.

3 P2PClient Behavior

A P2PClient (p2pclient.py) starts by reading its command line parameters, which specify its globally unique ClientID, the P2PTracker IP address and port number, the filename for the local file containing the Content List, and filename for the local input-script file containing commands for the client to execute. p2pclient.py file should have the functionality to handle client behaviour described below. client.py instantiates the class defined in p2pclient.py based on the command-line arguments given. Both files have more explicit instructions as TODO handles in them. Starter code files are provided to you.

After reading the command line parameters, the client registers with the P2PTracker using the IP address of the host machine it is running on and the port number on which other peers can contact this P2PClient. It then waits for the start message from the P2PTracker. The start message is a single character 'S' received from the P2PTracker. After receiving the start message, the client executes the commands in the input-script in order, pausing as required by the script to respect the command timing. All the possible commands are contained in the table below.

Command Label	Command Meaning
R,<P2PTracker IP>,<port>	Register itself with the P2PTracker. Send ClientID
U,<P2PTracker IP>,<port>	Unregister itself from the P2PTracker
L,<P2PTracker IP>,<port>	Obtain a list of all clients from bootstrapper. Get ClientID, ClientIP, ClientPort
Q,<data object hashcode>	reQuest a data object
P,<data object hashcode>	Purge (remove) data object from own list (but retain it in the content originator list)
O,<clientID>	Obtain a list of all other clients known to a particular client (clientID).
M,<clientID>	Obtain a list of all data object hashes from a particular client (clientID).

Suppose a particular client has a ClientID 1. The structure of the input-script is `<time, command label, parameters>`. Here is an example input-script:

```
2,Q,898119abdc27abd898119abdc27abd13
3,Q,aaaaaaaaaaaaaaaaaaaaaabbbbbbbbb
3,Q,77777777777777777777778888888888
3,Q,cccccbbbbccccccccccccbbbbbbbcbcb
4,P,aaaaaaaaaaaaaaaaaaaaaabbbbbbbbb
5,Q,ddddddddddddddddddddddddddddddd
6,U,127.0.0.1,5001
7,L,127.0.0.1,5001
7,O,4
8,M,7
```

This script first requests fetching the content for hashcode `898119abdc27abd898119abdc27abd13`. The internal processing on this command can be quite complex. The program will first look at its own Content list and if its present, it doesn't look further(Since this is very trivial, we will not be testing this); if its not present, it looks at its own COL to check if this content was previously obtained from any other peer. If yes, it asks the peer for the content. If the peer doesn't have the content, it begins an extensive search as described next. The program will then query the P2PTracker for a set of all clients registered with the P2PTracker. It will then start sending requests for the data objects to each of these clients in a serial manner. From any P2PClient, it might either receive confirmation that the peer has the data object, or should receive a hint about where the content might be located. On receiving a hint, the client attempts to contact that P2Pclient before progressing ahead with its current list. **Please look at the next**

paragraph on how the hint is processed. If the content is not found by asking all clients in the Bootstrapper list and all the hints that came from the clients in this list, then it proceeds to request every P2PClient from the list obtained from the Bootstrapper for a list of all other clients that client knows about (this is the set of unique clients from the COL of the that client), again in a serial manner and creates a **longer list** for trying various P2PClients. Note that the client handles each request in a stateless-way, i.e, it does not cache information received from the Bootstrapper or hints from one request and use it for a subsequent one.

Processing hints: There are 3 things that could happen when a hint is received: (1) "hinted" client has content, in this case the search terminates; (2) "hinted" client doesn't have the content and doesn't have a mention of it in its COL either, then this hint is no longer useful and the client goes back to its original list (or, if original list is exhausted then it proceeds to make a longer list as described above). (3) A hint can result in another hint. In this case, the program must process a maximum of 2 recursive hints before getting back to the original list. (This is done to avoid a loop of hints.) Note that this is a suggestion on how to handle corner cases like these, but we will not be testing it.

At time 3 (seconds), the data object with hashcode `aaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbb` is requested. After that is complete, the next line is executed (same time 3), and so on. Then at time 4, the client is purging its own (or newly obtained) content. It just removes that entry from the "Content List" datastructure. At time 6, it unregisters from the P2PTracker. At time 7, it asks for a list of all clients currently connected to the P2PTracker (this operation is allowed even if this client is not currently registered). The client can also request data and respond to other clients' requests for data.

The output of your program will follow a specific format. After completing every action, the program must log the output of that action, by maintaining a list and appending to it. The exact format of how the log must be generated can be seen at the top of the p2pclient.py file. Here is a snapshot of the same (Figure 2):

```
"""
Appending to log: every time you have to add a log entry, create a new dictionary and append it to self.log. The dictionary formats for diff. cases are given be
Registraion: (R)
{
    "time": <time>,
    "text": "Client ID <client_id> registered"
}
Unregister: (U)
{
    "time": <time>,
    "text": "Unregistered"
}
Fetch content: (Q)
{
    "time": <time>,
    "text": "Obtained <content_id> from <IP>#<Port>
}
Purge: (P)
{
    "time": <time>,
    "text": "Removed <content_id>"
}
Obtain list of clients known to a client: (O)
{
    "time": <time>,
    "text": "Client <client_id>: <<client_id>, <IP>, <Port>>, <<client_id>, <IP>, <Port>>, ..., <<client_id>, <IP>, <Port>>"
}
Obtain list of content with a client: (M)
{
    "time": <time>,
    "text": "Client <client_id>: <content_id>, <content_id>, ..., <content_id>"
}
Obtain list of clients from Bootstrapper: (L)
{
    "time": <time>,
    "text": "Bootstrapper: <<client_id>, <IP>, <Port>>, <<client_id>, <IP>, <Port>>, ..., <<client_id>, <IP>, <Port>>"
}
"""
```

Figure 2: Client Logging format

4 P2PTracker Behavior

As described earlier, the P2PTracker is a simple program that keeps a single list of all P2PClients it knows about. The list contains the ClientID, IP address, and port number, along with a status, either registered or deregistered. A P2PClient registers itself upon startup. It can then issue a deregistration or a registration in the future. (To be completely robust, the P2PTracker could only allow registration for currently deregistered Clients, and vice versa, but you do not need to check for this.) A P2PClient may continue to exist and serve requests even after it deregisters itself from the P2PTracker. The P2PTracker must send a start command to all clients to signal the beginning of time (t=0). The P2PTracker issues the start command by sending an 'S' to all P2PClients that have registered with the P2PTracker. All P2PClients will start executing their own input-scripts from this time. The P2PTracker is started by simply executing the command: python3 bootstrapper.py. The bootstrapper log structure is simple: similar to clients, it maintains a list to which, at each time step (after all clients have finished their actions), it appends a log entry with the registered clients. See the sample below (Figure 3):

```
{
    "time": <time>,
    "text": "Clients registered: <<client_id>, <IP>, <Port>>, <<client_id>, <IP>, <Port>>, ..., <<client_id>, <IP>, <Port>>"
}
"""
```

Figure 3: Bootstrapper Logging format

5 Coding Instructions

1. We will be testing your code with mainly 8 clients and a bootstrapper, however, to test for scale, we will instantiate 20 clients and check for registration status on the bootstrapper.
2. Each client has some content (MD5 hashes) to begin with, the same can be read by your program in JSON format. For example, data of client 5 will be present in 5.json. Each json file contains 3 items: client_id, content(the content this client has at the start of the program) and actions (what the client has to do and when it needs to do it) Fun fact: all MD5 hashes are obtained by encrypting different locations on campus!
3. Complete the code snippets in these files: **p2pclient.py, p2pbootstrapper.py, client.py and bootstapper.py**
4. Ensure that you are logging the output of clients' actions (appending to self.log in p2pclient class). At the end of your actions, "export" self.log to a file: client_x.json, this is what the autograder is looking for.
5. Similarly, the bootstrapper logs needs to be exported into bootstrapper.json.
6. Python's json package should come handy.
7. Note that because each entity in system has to act as both a client and a server during the execution of the program, you will have to make use of threads to handle blocking functions. Carefully go through the TODO handles in p2pclient.py, p2pbootstrapper.py and client.py and boostrapper.py. Python's threading library should come handy.
8. **Randomness in client port** : While the bootstrapper starts on a fixed, known-to-client port 8888, the clients start listening on a random port picked by OS, which it then communicates to the bootstrapper. But to make Gradescope testing deterministic, we have to ensure that the ports are chosen in a pseudo-random fashion using seeds. The **random_port_instructions.txt** file tells you how to do it. Use the code in this file in your p2pclient.py file, There's a TODO to help you with this.
9. **Timing considerations** : Since all clients and bootstrapper are individual programs running, to enforce order in execution so that results are deterministic with respect to the autograder, you will have to implement a clocking system. The requirement is that all clients must complete actions at timestep t before any of the clients move onto actions in timestep t + 1. However, you don't have to enforce ordering within a timestep, i.e, client 5 can do its action at timestep 5 before client 8 at timestep 5

or the other way round, and the results will be the same. As an example, all clients should have completed their tasks for timestep 5 before any of the clients can do their actions at timestep 6. One way to enforce this is via the Bootstrapper, with every client communicating to the Bootstrapper when it completes an action, and when the Bootstrapper gets a 'complete' from all clients it knows (registered or not), it proceeds to increment the time and informs the clients to continue with the next action. The clients on the other hand, finish the task, inform Bootstrapper and wait for Bootstrapper to tell them about it. Note that, this is not a part of the P2P architecture, and is only done here to get deterministic results. Here is an FSM for the same:

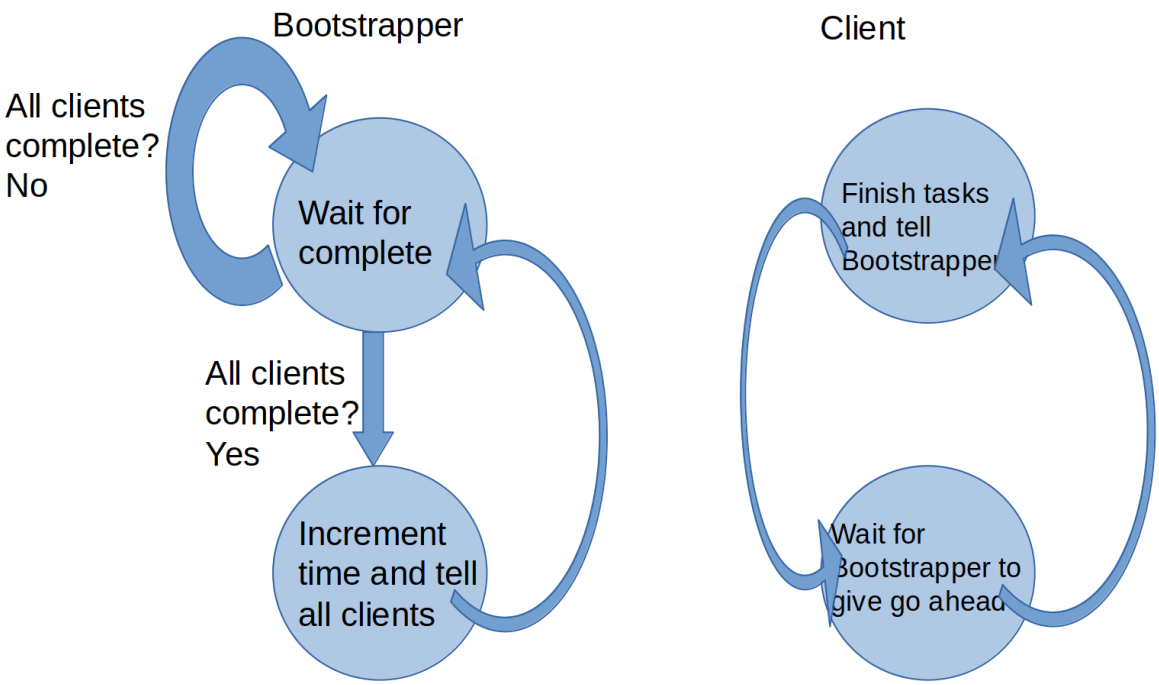

Figure 4: FSM for handling timing for bootstrapper and clients

10. **Autograder format clarifications:**

    (a) In case a command results in an empty output, say no clients or no content, your code should give a blank instead of "<>".

    (b) In the bootstrapper, you will need to order clients by client id in increasing order (i.e., store client 1, then client 2 in self.clients). This will ensure that when the clients query the Bootstrapper, they get the clients in pre-determined order.

    (c) In the client, you will have to complete a method that returns a list of known clients. This list should be ordered by decreasing client id to ensure outputs are

consistent with the autograder.

11. **Python version:** 3.6.7(recommended), versions 3.5 and 3.7 should also work. You will not need to install any new libraries. You can use libraries that come pre-installed with these versions. Do not use asynchio. Libraries which you need like sockets, json, time, random all come pre-installed.

6 Helpful Resources

1. `https://docs.python.org/3/howto/sockets.html`
2. `https://docs.python.org/3/library/threading.html`
3. `https://www.programiz.com/python-programming/json`

7 Sequence of Execution

1. P2PTracker is started by executing the command python3 bootstrapper.py.
2. All P2PClients are started by executing their respective commands. For example, to instantiate client 3: python3 client.py –file 3.json
3. P2PTracker waits for 10 seconds for all clients to get instantiated and send register messages before sending the 'S' signal to allow all P2PClients to execute their own input-scripts.
4. P2PClient input-scripts will not exceed t=120.

The following diagram shows the sequence of execution:

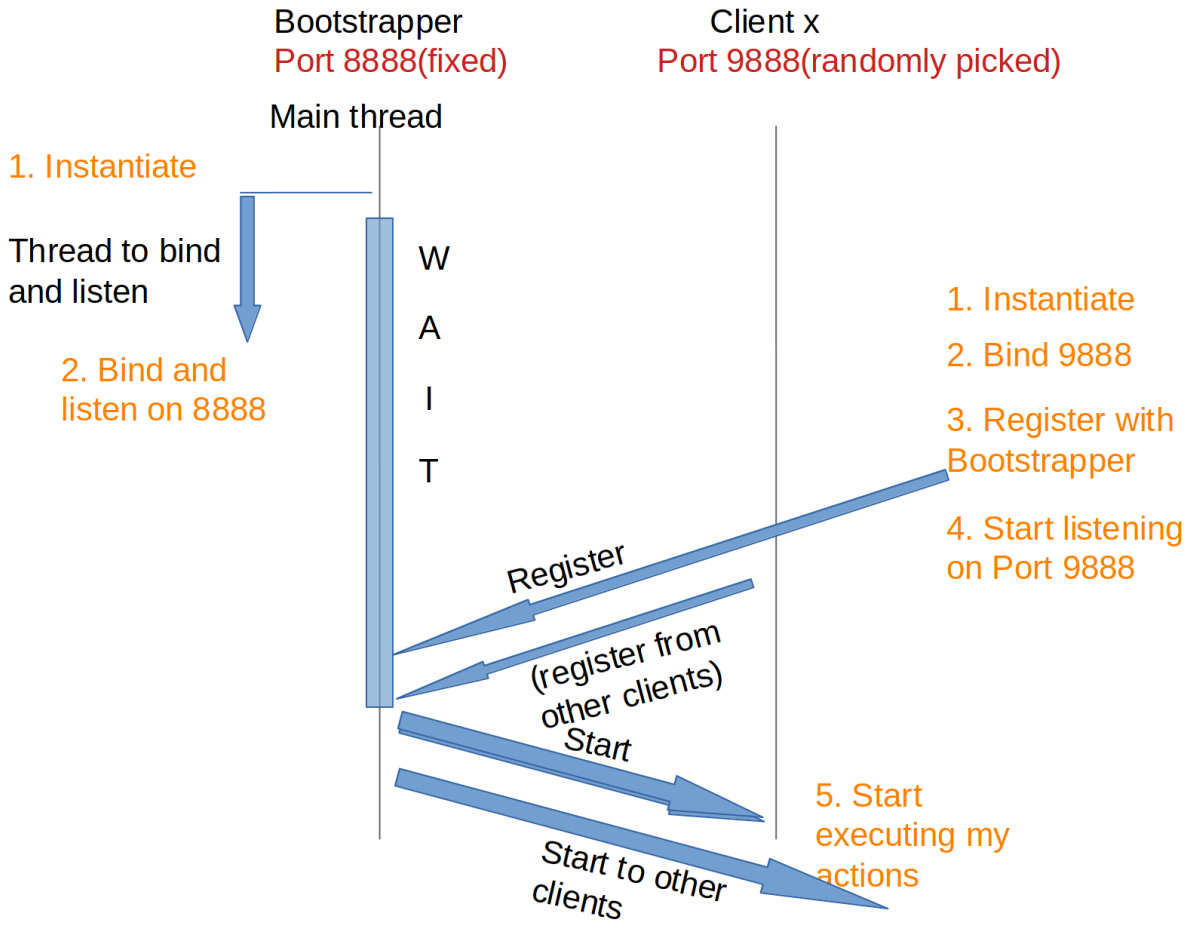

Figure 5: Timeline

8 Submission Guidelines and Grading

When you are ready to submit to Gradescope:

- Run the collect_submission script : python collect_submission.py
- This will generate a zip file: pa2_submission.zip which you can then upload on Gradescope.
- You are allowed to use the autograder as many times as you want until the last due date.

Your code will be checked for the following:

- P2PTracker maintains a list and supports register and un-register commands (10 points)
- P2Pclient can obtain a list of clients from the P2PTracker (5 points)

- P2PClient can check for data objects at the Clients listed by the P2PTracker (10 points)
- P2PClient can provide hints about data objects it knows about (10 points)
- P2PClient can use hints and does so before iterating the list provided by the P2PTracker. (10 points)
- P2PClient purges data objects from its own list (5 points).
- P2PClient still provides hints about purged data objects. (5 points)
- P2PClient can fetch data from a client unreachable from the P2PTracker for which no hint was obtained (by obtaining a list of all clients known to all known clients). (10 points)
- P2PClient can obtain all data objects hashes from a particular peer (10 points).
- P2PClient can re-register with the P2PTracker (5 points).
- More than 20 clients are supported (10 points).
- P2PClient can un-register but keep running and serving direct requests for data objects. (10 points).

9 Test Scripts

You are provided with the test examples that will exercise all of the options described above. You may execute them locally. These are provided for your convenience and to reduce the load on the Gradescope. However, the score on the Gradescope is considered final. The test examples on the Gradescope are different from those provided to you, so do not overfit your code just to pass the test-cases.

The test examples can be found in the **'toy'** folder. You are also provided the expected outcomes for these to verify the accuracy of the code.
To test code locally:

- Linux : Run the runner script given, this will create an identical setup as Gradescope : **./runner.sh**
- Windows : Open separate terminal windows and run bootstrapper.py, client.py <arguments> with each client on a different terminal, with the bootstrapper coming up first.

10 Logistics

1. **Start early! :)** and test in stages, locally and on Gradescope.
2. Try to make use of existing threads set up by TAs to post questions, helps keep things organized

3. Your question may already be answered: try to go through existing questions before posting new ones.
4. For further clarifications, please see piazza note 170.