

Lab 4

Purpose:

- To implement multithreaded client handling on a TCP server
- To practice using Python classes

Due: Friday, June 3, 11:59pm

Handin:

Follow the steps on the Handin Instructions page to submit this lab. You are to hand in the following file:

- `server.py`

Specification:

You are to modify your `server.py` TCP-based file transfer server from Lab 2 so that it can accept multiple client connections.

The ClientHandler class

You must create a subclass of `threading.Thread` to handle all communication with a single client. Since you already have the code that does this handling, this requirement is really about setting up a new python class, creating a constructor, and copying your client-handling code into the `run()` method.

Your server, when it receives a new client connection with the `sock.accept()` method, creates a client handler thread **but does not start it!** It simply hands the waiting client thread to the “manager”.

The Manager class

This class will maintain two data structures: a queue, and a set.

The queue will hold all the waiting client connections; they haven't started yet, they've just been added to the queue, waiting to be executed. The main program calls a method of this class to add new client threads to the queue.

When the manager decides it is time to start one of the waiting client threads, it issues the `t.start()` command, and then adds it to the "running" set.

(Sets are kind of like queues, except they don't maintain any particular order; you add items, remove items, and iterate efficiently, but you can't expect any particular order when you iterate.)

We will impose a limit on how big the "running" set may be, so that the clients are limited to how much of the server's resources they can collectively use. The server will now take an additional commandline parameter (`sys.argv[2]`) that says how many clients may actually be actively running concurrently (e.g. 5 -- don't make it too big!). Note that `sys.argv[1]` will still be the server's binding port. You can implement a `-v` flag if you like, but it's not required and will not be graded; your server should not have *any* output as the default option.

The manager basically sits in an infinite loop, executing the following pseudocode:

- check the "running" threads; if any of them have stopped, remove them from the set.
- check the waiting queue:
 - if empty, sleep for 1 second and return to the top of the loop;
 - if it has an item:
 - check the size of the running set:
 - if it is full, sleep for 1 second and return to the top of the loop;
 - if it has space:
 - remove the next client thread from the queue
 - start the thread
 - add the thread to the running set

The manager thread just keeps running forever; it will stop when the main thread is terminated externally.

Hints:

- Try not to write *too* much code!! You have to create the two classes, but the client is 99% copy-and-paste. The Manager is a little bit trickier, but follow the pseudocode above as closely as you can!
- To create a queue object, import the `queue` module, and instantiate:

```
self.q = queue.queue()
```

- To create a set object, you just instantiate the built-in `set` data type:

```
self.running = set()
```

- Note that you can't remove an item from a set *while you are iterating through it*. This is quite normal for many data structures. When you are looking for threads to kick out of the "running" set, you have to actually temporarily append them to a list:

```
kick = []
for t in self.running:
    if not t.isAlive(): kick.append(t)
for t in kick:
    self.running.remove(t)
```

- The main program thread won't terminate while it has a child threading still running. Specifically, your server's main `while True:` loop *can't* be broken using CTRL-C, because the Manager thread is running! To force all the threads to stop, you will have to use an operating-system kill. On linux/mac, you can do the following:

- `ps aux | grep python` - gives a list of python processes; find the `server.py` process, and note the pid (second column) number (e.g. 12573)
- `kill <pid>` - forces all threads under that process id to stop.

- On Windows, use the Task Manager to kill the process.