Week 8 Discussion

CS 131 Section 1B
20 May 2022
Danning Yu

Announcements

- HW5 released, due 5/20
- Project released, due 5/31
- Homeworks should be submitted on BruinLearn, under Assignments
- Before submitting
  - Make sure your code compiles on SEASnet server
  - Make sure your function signatures are correct
  - Follow all instructions and specifications
  - Do not submit files in a .zip unless told to do so
- Help and starter code from past TAs
  - https://github.com/CS131-TA-team
  - https://github.com/CS131-TA-team/hw5-grading-script

Python and `asyncio`

Python

- General-purpose, interpreted language
  - Very popular, easy to write code in
  - Rich collection of libraries for nearly every purpose
- Use Python 3.10 for this class
  - Python 2 is deprecated but still can be seen sometimes
- Download from https://www.python.org
  - SEASnet: `python3`
- Resources
  - https://www.learnpython.org/
    - Interactive tutorial, fast and easy to get started
  - https://docs.python.org/3/tutorial/
    - Official tutorial
  - https://docs.python.org/3/
    - Reference material for the language and the official libraries

Introduction

- Hello world

```
print("Hello World")
```

Hello World!

- Dynamic typing, no special syntax for declaring a new variable

```
x = 123                  # integer
x = 3.14                 # double float
x = "hello"              # string
x = [0,1,2]              # list
x = (0,1,2)              # tuple
```

- Does not have a main function by default; code executed line-by-line
- The following structure can be used to emulate a "main function"

```
def main():
    print("Hello World!")

if __name__ == '__main__':
    main()
```

Functions

- Declared using the `def` keyword:
  - No semicolons, indentation matters

```
def my_function(name):
    print("Hello, " + name)

my_function("Steve")
```

Hello, Steve

- Both positional and keyword parameters allowed

```
def my_func(a, b=1, c=1):
    print(a + b + c)

my_func(2)          # "4"
my_func(2, 2, 2)    # "6"
my_func(1, c=2)     # "4"
```

Modules

- Every file defines a module
- Use import to refer to other modules
  - Export a module by putting an `__init__.py` file in the same directory

mymodule.py

```
def print_hello(name):
    print("Hello, " + name)
```

hello.py

```
import mymodule

def main():
    mymodule.print_hello("Steve")

if __name__ == '__main__':
    main()
```

Variable Scope

- If a variable is assigned in a function, that variable is local unless annotated with the `global` keyword

```
x = 5

def my_function():
    print(x)

my_function()
```

5

```
x = 5

def my_function():
    x = 10
    print(x)

my_function()
print(x)
```

10
5

Variable Scope

- If a variable is assigned in a function, that variable is local unless annotated with the `global` keyword

```
x = 5

def my_function():
    print(x)
    x = 10

my_function()
print(x)
```

NameError: name 'x' is not defined

```
x = 5

def my_function():
    global x
    print(x)
    x = 10

my_function()
print(x)
```

5
10

Classes

- Use the `class` keyword to define a class

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def print_greeting(self, greeting):
        print(greeting + self.name)

p = Person("John", 36)
p.print_greeting("Hello, ")
```

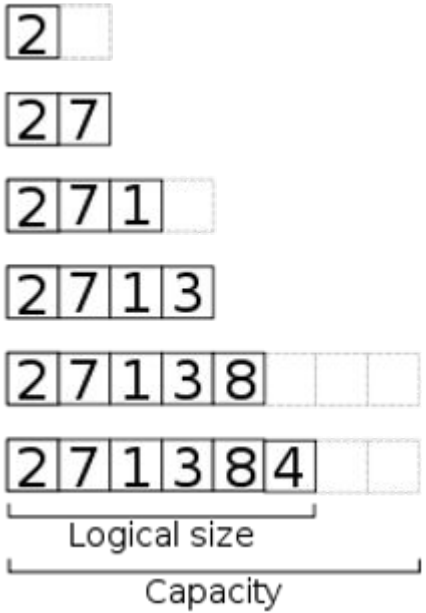
Hello, John

Lists

- Lists are variable size arrays
  - Fast random access, easy to add/remove elements
  - Uses a bit more memory

```
my_list = [1, 2, 3]

print(my_list[2])
3

print(my_list[1:])
[2, 3]

print(my_list[0:2])
[1, 2]
```

```
for item in my_list:
    print(item)
1
2
3
```



Lambda Functions

- Use `lambda` keyword to define a lambda function
  - Suitable for simple one-liners in most cases
- Example: usage in map function

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
print(squared)
```

`[1, 4, 9, 16, 25]`

List Comprehensions

- Alternative syntax for map and filter operations on a list

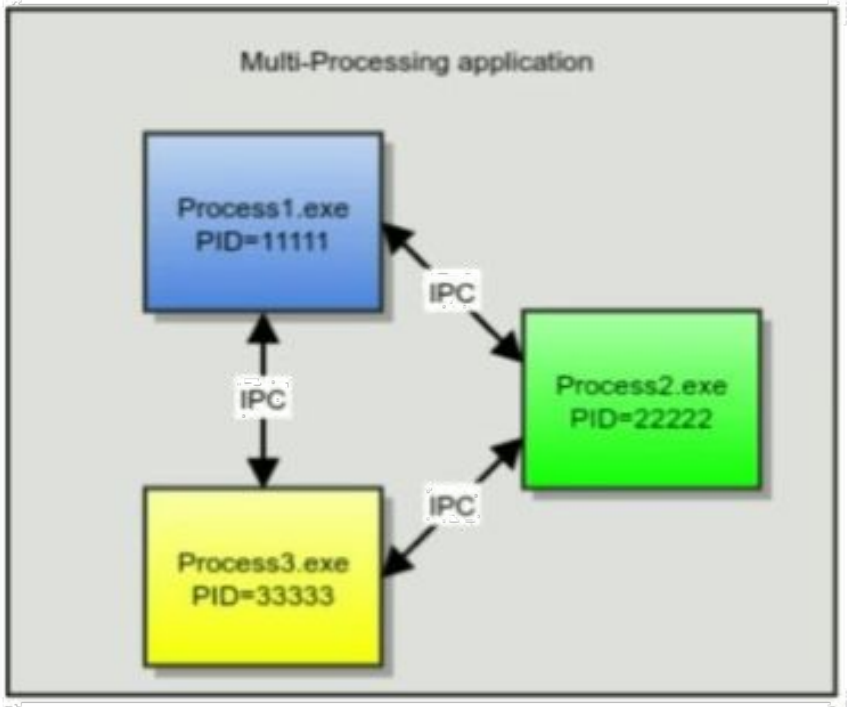
```
my_list = [1, 2, 3, 4, 5]

new_list = [x*x for x in my_list]
print(new_list)
[1, 4, 9, 16, 25]

new_list2 = [x for x in my_list if x%2 == 0]
print(new_list2)
[2, 4]
```

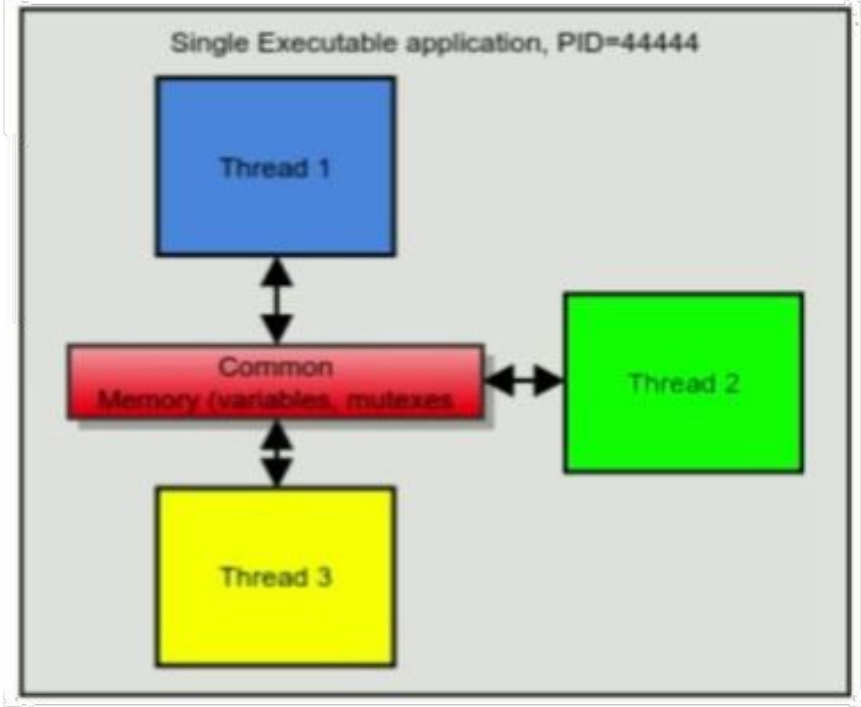
Dictionary

- The hash table for Python: unordered collection of key, value pairs
- Keys must be immutable, so lists not allowed as keys
  - Tuples are allowed instead

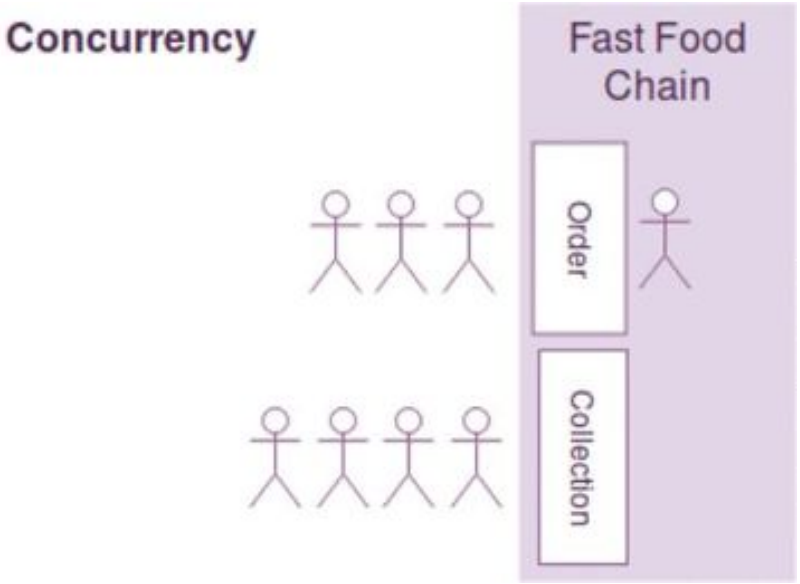
```
my_dict = { "a": 1, "b": "f", 42: 3 }

print(my_dict["a"])
1

print(my_dict[42])
3

my_dict["something new"] = 4
print(my_dict["something new"])
4
```

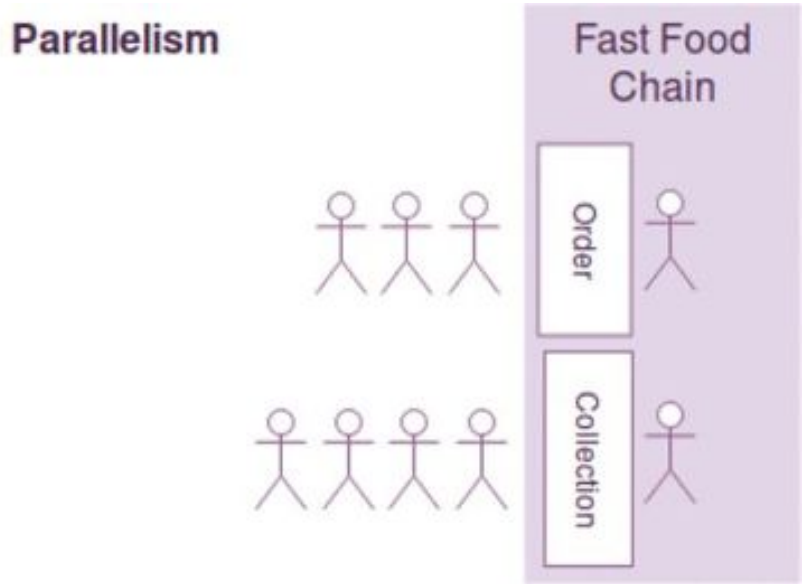
Multithreading vs. Multiprocessing





Concurrency vs. Parallelism





Python's Global Interpreter Lock (GIL)

- A mutex lock that prevents multiple threads from executing Python bytecodes at once
  - Exists in the default C implementation of Python, CPython
- Python memory management depends on reference counting (possible race conditions with multithreading)
- Python also uses C libraries that are not thread-safe
- Result is fast single-threaded code, simple memory management compared to other types of garbage collectors
- Downside: multithreading does not improve the performance of CPU intensive tasks
  - In fact, might be even slower due to thread contention
- When can we benefit from threads in Python?

How To Utilize Multiple CPUs with Python

- Python's `multiprocessing` module
- Libraries
  - Many numerical computation and machine learning libraries support parallel processing
  - Typically implemented in C or other low-level language

Python's `asyncio`

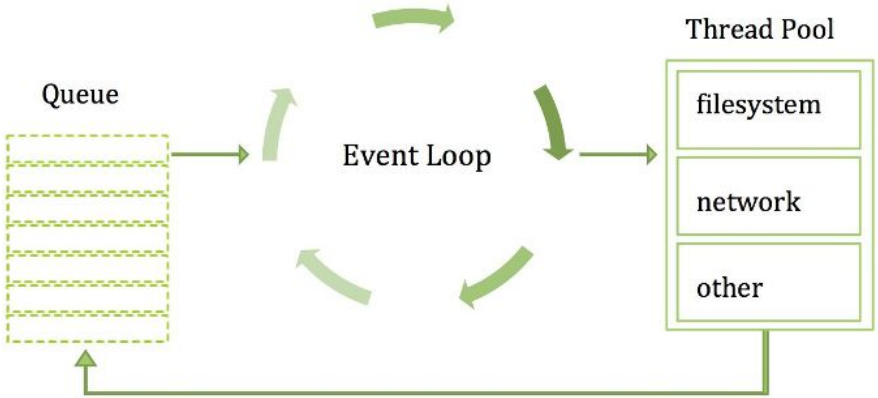
- Single-threaded approach for concurrent programming
  - Well-suited for I/O intensive applications
  - Similar to NodeJS
- Cooperative multitasking library
  - Tasks can voluntarily take breaks and let other tasks run, as opposed to preemptive multitasking
- Introduced in Python 3.4 in 2014
  - Relatively new, so changes often with new versions

Basic `asyncio` Concepts

- `async` keyword defines that a function is a coroutine
  - A function that can suspend its execution and give control to another coroutine
- `await` keyword suspends the execution of the current coroutine until the awaited function is finished

```
async def do_something():
    result = await io_operation()
    return result
```

- Event loop runs tasks that are waiting to be executed



Example 1: Basic Counter

```
import asyncio

async def count():
    print("One")
    # Any IO-intensive task here
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())       # Add to an event loop
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in "
          f"{elapsed:0.2f} seconds.")
```

```
$ python3 countasync.py
One
One
One
Two
Two
Two
countasync.py executed
in 1.01 seconds.
```

Example 2: Server

```
import asyncio

async def main():
    server = await
asyncio.start_server(handle_connection,
host='127.0.0.1', port=12345)
    await server.serve_forever()

async def handle_connection(reader, writer):
    data = await reader.readline()
    name = data.decode()
    greeting = "Hello, " + name
    writer.write(greeting.encode())
    await writer.drain()
    writer.close()

if __name__ == '__main__':
    asyncio.run(main())
```

```
$ nc localhost 12345
John
Hello, John
```

Example 2: Client

```
import asyncio

async def main():
    reader, writer = await
asyncio.open_connection('127.0.0.1', 12345)
    writer.write("John\n".encode())
    data = await reader.readline()
    print('Received: {}'.format(data.decode()))
    writer.close()

if __name__ == '__main__':
    asyncio.run(main())
```

```
$ python3 client.py
Received: 'Hello, John\n'
```

`asyncio` Reources

- Async IO in Python: A Complete Walkthrough
  - https://realpython.com/async-io-python/
- Asyncio Documentation
  - https://asyncio.readthedocs.io/en/latest/

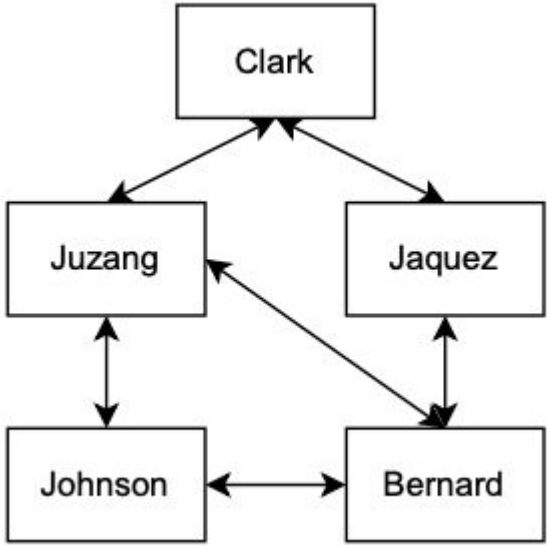
Project

Background: Client-Server Communication

- Server and client
  - Definition and differences
- How do servers and clients communicate?
  - From the low level: through sockets
  - Many programming languages have libraries and frameworks that provides more friendly APIs
  - We will practice using `asyncio` from Python for the project
  - Protocol: the common language between clients and server

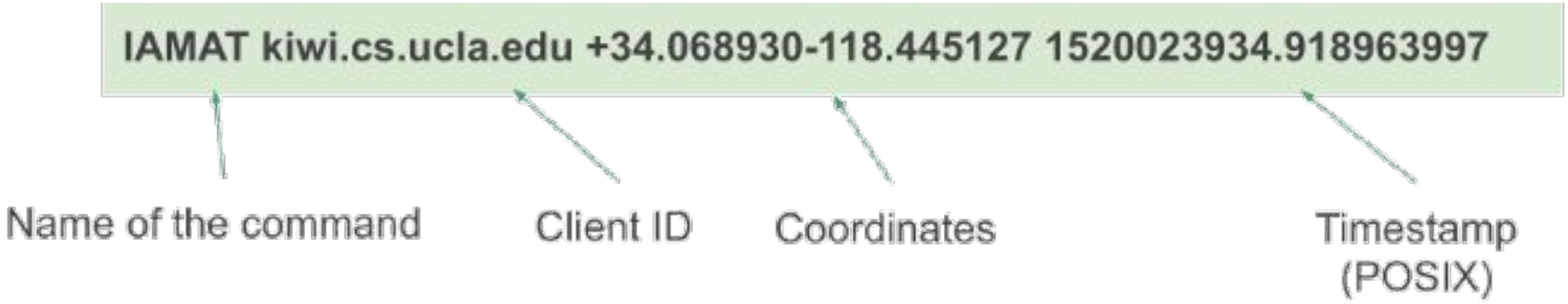
Project Overview

- Task: Build a server herd that can synchronize data and communicate with client applications
- Each server is a separate process, to launch one of the servers:
  - `python3 server.py <server_name>`
  - You need to do it 5 times to launch all the servers
- Port assignments have been posted on BruinLearn
  - Use your assigned port on SEASnet to avoid conflicting with each other
  - Make sure the requests/responses look exactly the same as instruction, as we will use fully automated tests to grade the submissions

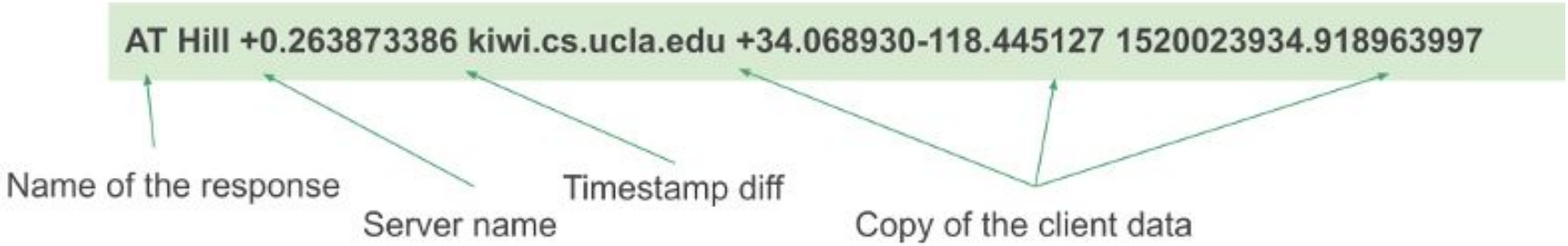


Client-Server Communication: IAMAT Message

- Client can send their current location to any server using text-based TCP protocol

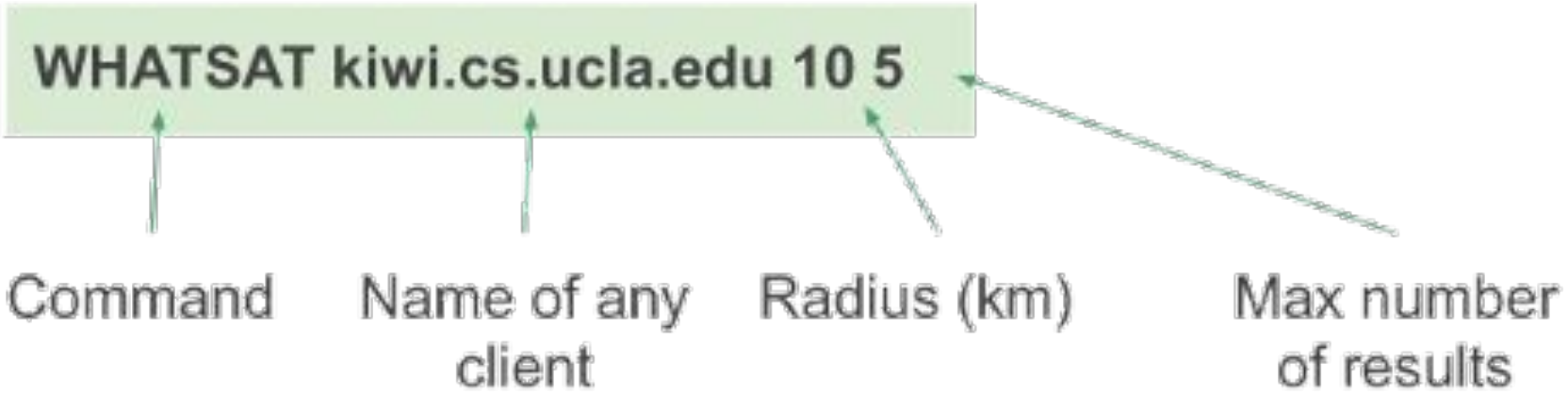


- Response from server



Client-Server Communication: WHATSAT Message

- Clients can ask what is near one of the clients



- Server uses Google places API to find nearby locations
- Google Places API gives results in JSON format, return it to the client in the same format, just remove duplicate newlines
  - See project instructions for details

Client-Server Communication: WHATSAT Message

- Response from server

```
AT Hill +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997
{
   "html_attributions" : [],
   "next_page_token" : "CvQ...L2E",
   "results" : [
     {
          "geometry" : {
          "location" : {
               "lat" : 34.068921,
               "lng" : -118.445181
          }
          },
          "icon" : "http://maps.gstatic.com/mapfiles/place_api/icons/university-71.png",
          "id" : "4d56f16ad3d8976d49143fa4fdfffbc0a7ce8e39",
          "name" : "University of California, Los Angeles",
          "photos" : ....
```

Server-Server Communication

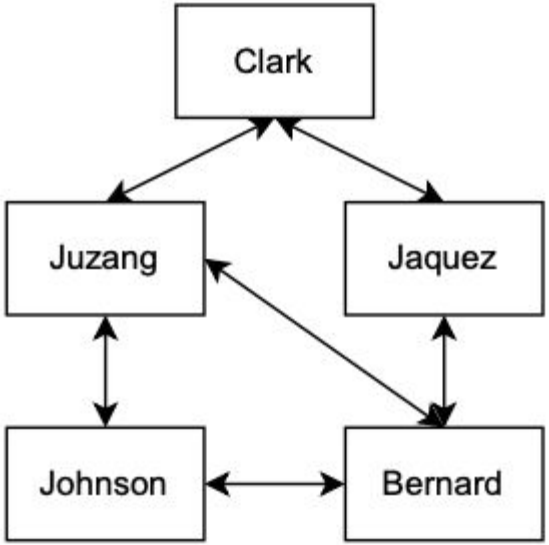
- After one server receiving a location, the location should be synchronized among servers
- Implement a flooding algorithm so that every server receives the message, even if it is not directly connected to the original server
  - Need to prevent messages from looping infinitely
  - Up to you to decide server-server message protocol
- If a server goes down, all other servers should still function normally
- No need to propagate old messages when the server is restarted

Notes

- One message/response pair per TCP connection
  - Can close the connection after responding to a message
- Messages can contain any number of spaces/newlines between the words
  - Can wait for end-of-file (EOF) character by using `reader.read()`
- Messages can be invalid
  - Example: `WHATSAT kiwi` (missing parts)
  - Example: `WHATAT some_client`, where `some_client` is a client that server does not know the location of
  - Respond with `? <received message>`, e.g. `? WHATSAT kiwi`

Request Types Summary

- Client requests
  - IAMAT
  - WHATSAT
- IAMAT
  - Server saves the location and propagates it among the herd
- WHATSAT
  - Server calls Google Places API to check what is near the given client, and sends result back to caller
- Unknown request or WHATSAT for unknown client: respond with the same command with ? appended in front



Testing the Server

- Use `telnet` or `nc`
  - Much easier to use or control than writing a program yourself

```
nc [options] localhost port
telnet [options] localhost port
```

Google Places API

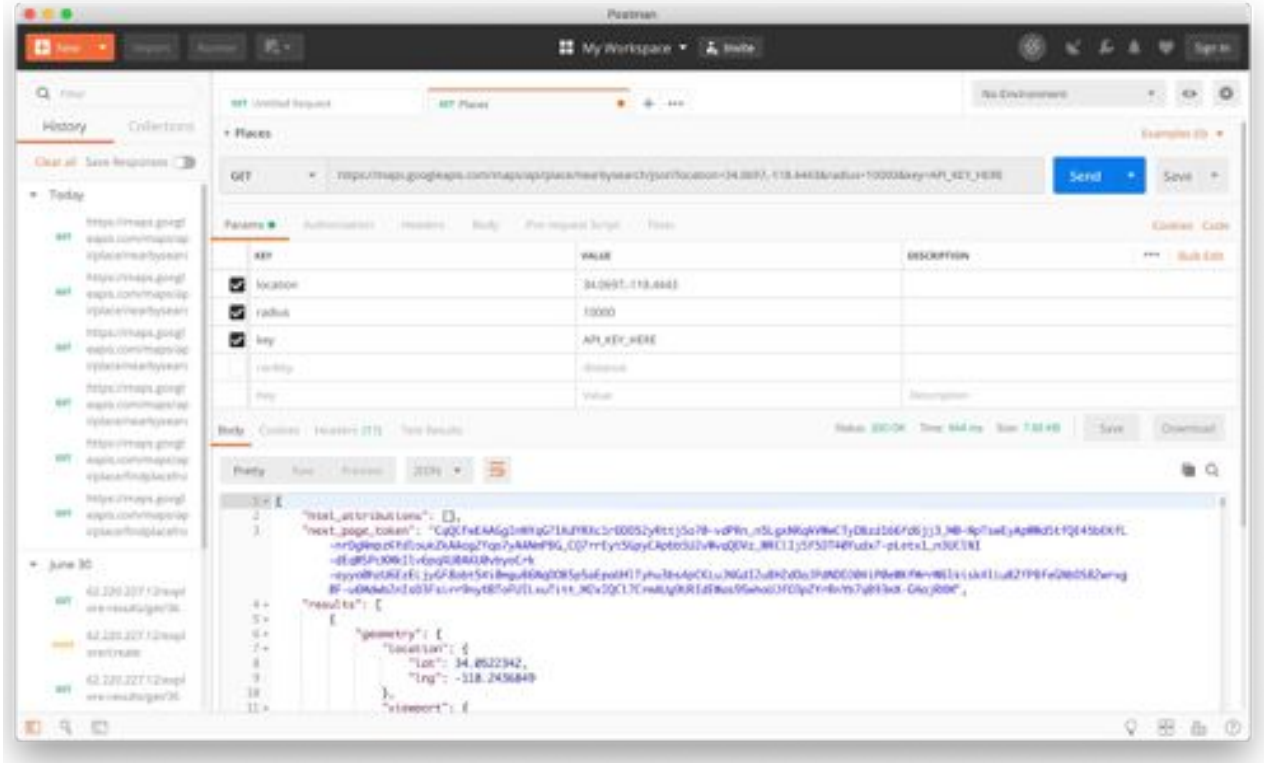
- Gives you information on what is around a given location
  - Can also be used to find an address for given coordinates, get details on specific locations, etc.
- You need to create a developer account to access the API
  - Free trial is enough for this project
  - Do not share your key with anyone! (e.g. don’t post the key on Github)
- This project uses the "Nearby Search” request API

```
https://maps.googleapis.com/maps/api/place/nearbysearch/json
?location=-33.8670522,151.1957362&radius=1500&key=YOUR_API_KEY
```

  - Search places near coordinates -33.8670522, 151.1957362
  - Limits radius to 1500 meters
- Documentation: https://developers.google.com/places/web-service/search

Testing Google Places API Requests

- You can use `curl` or developer tools like Postman to try out the HTTP requests



Making HTTP Requests in Python

- Use the `aiohttp` library
  - Only for making requests to Google Places API; do not use it for server functionality
  - Can reuse the same session for all the requests

```
async with aiohttp.ClientSession() as session:
    params = [('param-name1', 'some value'), ('param-name2', '100')]

    async with session.get('https://ucla.edu', params=params) as resp:
            print(await resp.text())
```

Project Report

- Max 5 pages
- Discuss pros/cons of `asyncio`
- Is it suitable for this kind of application?
  - What problems did you run into?
  - Any problems regarding type, memory management, multithreading; could compare with Java for this section
  - How does `asyncio` compare to NodeJS?
  - Performance implications of `asyncio`?
  - How easy is it to write a server using `asyncio`?
  - How important are the `asyncio` features introduced in Python 3.9?

Thank You