CS 131
PROGRAMMING LANGUAGES
(WEEK 8)

UCLA WINTER 2019

TA: SHRUTI SHARAN

DISCUSSION SECTION: 1D

ADMINISTRATIVE INTRODUCTION

TA: Shruti Sharan

Email: shruti5596@g.ucla.edu
(Will reply by EOD)

Office Hours: Mondays 1.30PM – 3.30PM
Location: Eng. VI 3rd Floor

Discussion Section: Friday 4.00-5.50PM
Location: 2214 Public Affairs

TODAY'S AGENDA

PYTHON: INTRODUCTION

ITERATORS AND GENERATORS

ASYCIO LIBRARY

HOMEWORK #5

PYTHON

PYTHON - INSTALLATION

- We'll be using Python 3.7.2 (latest)
- https://www.python.org/downloads/release/python-372/
- Eclipse has PyDev plugin
- http://pydev.org
- Can also use python IDLE, or your favorite text editor.
  - The file should have a .py extension
  - Run it on the terminal using python3 command. (Default is Python 2)

PYTHON 2 VS PYTHON 3 - MAIN CHANGES

- Syntax changes - print “hello” in 2 to print(“hello”) in 3
- Integer division - $3/2 = 1$ in 2 to $3/2 = 1.5$ in 3
- Strings are stored as ASCII in 2 and Unicode in 3
- xrange() doesn’t exist in python 3 where as range() and xrange() are both present in python 2.
  - range() in 3 works like xrange() in 2
- Many more syntax changes and functionality changes for example round() now rounds to nearest even integer.
- Many older libraries are not forward compatible

PYTHON - BASICS

- What type of language is Python?
- What is Python used for?
- Is Python interpreted or compiled?

PYTHON - BASICS

- Is Python Dynamically or Statically typed?
- How does Python handle memory?
- What's important about Python syntax?
  - (As compared to other languages like Java)

if-else STATEMENTS

```
if x < 2:
    print('less than 2')
elif x > 3:
    print('greater than 3')
elif x is 3:
    print('its 3!')
else:
    print('I have no idea what it is')
```

LISTS

- Lists are dynamic length arrays (compare to Scheme/OCaml)
  - Fast random access, easy to add/remove elements (with a slight performance overhead)
  - Uses a bit more memory

```
>>> my_list=[1,2,3,4,5]
>>> print(my_list[2])
3
>>> new_list=[x**2 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]
```

OCaml map:
List.map (fun x -> x*x) [1; 2; 3; 4; 5];;

OCaml filter:
List.filter (fun x -> x mod 2 = 0) [1; 2; 3; 4; 5];;

DICTIONARIES

- A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

```
>>> my_dict={"a":1,"b":2,42:3}
>>>
>>> print(my_dict["a"])
1
>>> print(my_dict[42])
3
>>> my_dict["something new"]=4
>>> my_dict
{'a': 1, 'b': 2, 42: 3, 'something new': 4}
>>>
>>> my_dict["a"]=5
>>> print(my_dict["a"])
5
```

```
>>> for keys in my_dict:
...     print(my_dict[keys])
...
5
2
3
4
```

EXERCISE

- How could we use a dictionary to count item occurrences in a list

```
>>> x=[1,2,1,2,3,1,2,1,1]
>>> d={}
>>> for i in x:
...     if i not in d:
...             d[i]=0
...     d[i]+=1
...
>>> d
{1: 5, 2: 3, 3: 1}
```

EXERCISE

- Write a function ‘isogram(word)’ that returns True if word is an isogram and False if not.
  - An isogram is a word that does not repeat any letters.
  - Ex: isogram, lumberjacks, background

SOLUTIONS

```
>>> def isogram(word):
...     for letter in word:
...             if word.count(letter)>1:
...                     return False
...     return True
...
>>> isogram("hello")
False
>>> isogram("hipster")
True
```

```
>>> def isogram(word):
...     letter_list=[]
...     for letter in word:
...             if letter in letter_list:
...                     return False
...             letter_list.append(letter)
...     return True
```

- What's a recursive solution to this problem?

```
>>> def isogram(word):
...     return (word,True) if word and len(set(word)) == len(word) else (word,False)
```

CLASSES

- A Class is like an object constructor, or a "blueprint" for creating objects.

```
>>> class Sloth():
... #### Represent a sloth ####
...  def __init__(self,name):
...   self.name=name
...  def climb(self):
...   print(self.name + ' climbs so very slowly')
...  def __add__(self,newSloth):
...   print('Now we have a baby sloth!')
```

```
>>> s1=Sloth('Barry')
>>> s2=Sloth('Harry')
>>> s1+s2
Now we have a baby sloth!
```

```
>>> Sloth.climb(s1)
Barry climbs so very slowly
>>>
>>> Sloth.climb(s2)
Harry climbs so very slowly
```

- All classes have a function called __init__(), which is always executed when the class is being initiated.

VARIADIC ARGUMENTS

```
>>> def printArguments(*args, **kwargs):
...   print('Args: ', args)
...   print('Keyword Args: ', kwargs)
```

Non Keyworded

```
>>> printArguments(1,2,"hi", r=3, f=4)
Args:  (1, 2, 'hi')
Keyword Args:  {'r': 3, 'f': 4}
```

Keyworded

PYTHON ITERATORS

- What's going on in this code?

```
>>> A=[1,2,3,4,5,]
>>> for i in A:
...  print(i)
...
1
2
3
4
5
```

PYTHON ITERATORS

- Required methods for Iterators
- __iter__
  - Initializes iterator, returns object that has a __next__ method
- __next__
  - Returns next value in the iterable
  - If no more values, raise StopIteration

PYTHON GENERATORS

- Easier way of making iterators
- Uses **'yield'** instead of **'return'**
  - Means, return this thing and then wait until called again
- Saves state of function after **'yield'** for subsequent calls

PYTHON GENERATORS – EXAMPLE 1

```
>>> def my_gen():
...   n=1
...   print('This is printed first')
...   #Generator function contains yeild statements
...   yield n
...   n+=1
...   print('This is printed second')
...   yield n
...
>>> my_gen()
<generator object my_gen at 0x10e0fd2a0>
```

```
>>> for item in my_gen():
...   print(item)
...
This is printed first
1
This is printed second
2
```

PYTHON GENERATORS – EXAMPLE 2

```
>>> def gen():
...   n=1
...   while True:
...    n+=1
...    yield n
...    n+=10
...    yield n
```

```
>>> x=gen()
>>> x
<generator object gen at 0x10e0fd2a0>
>>> x.__next__()
2
>>> x.__next__()
12
>>> x.__next__()
13
>>> x.__next__()
23
```

GENERATORS

```
>>> class PowTwo:
...     def __init__(self,max=0):
...             self.max=max
...     def __iter__(self):
...             self.n=0
...             return self
...     def __next__(self):
...             if self.n>self.max:
...                     raise StopIteration
...             result= 2**self.n
...             self.n+=1
...             return result
```

```
>>> def PowTwoGen(max=2):
...     n=0
...     while n<max:
...             yield 2**n
...             n+=1
```

Output:

```
>>> x=PowTwo()
>>> x.__iter__()
<__main__.PowTwo object at 0x10e164fd0>
>>> x.__next__()
1
>>> x.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in __next__
StopIteration
```

```
>>> x=PowTwoGen()
>>> x
<generator object PowTwoGen at 0x10e0fd840>
>>> x.__next__()
1
>>> x.__next__()
2
>>> x.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

ASYNC

- Asynchronous code allows the program to execute other code while it waits for slow operations to finish (e.g. file I/O, network requests, etc)
- Consider regular, non-asynchronous code:

```
>>> def say_after(delay,what):
...  time.sleep(delay)
...  print(what)
...
>>> def main():
...  print("Started at {}".format(time.strftime('%X')))
...  say_after(3,"Hello")
...  say_after(5,"World!")
...  print("Finished at {} ".format(time.strftime('%X')))
...
>>> if __name__=='__main__':
...  main()
```

```
Started at 17:20:42
Hello
World!
Finished at 17:20:50
```

- Same code asynchronously:

```
>>> import asyncio
>>> async def say_after(delay,what):
...  await asyncio.sleep(delay)
...  print(what)
...
>>> async def main():
...  print("Started at {}".format(time.strftime('%X')))
...  task1=asyncio.create_task(say_after(3,'Hello'))
...  task2=asyncio.create_task(say_after(5,'World!'))
...  await task1
...  await task2
...  print("Finished at {}".format(time.strftime('%X')))
...
...
>>> if __name__=='__main__':
...  asyncio.run(main())
```

```
Started at 17:32:49
Hello
World!
Finished at 17:32:54
```

asyncio LIBRARY

- **asyncio** is a library to write **concurrent** code using the **async/await** syntax.
- asyncio provides a set of **high-level** APIs to:
  - Run Python coroutines concurrently and have full control over their execution;
  - perform network IO and IPC;
  - Control subprocesses ;
  - distribute tasks via queues;
  - synchronize concurrent code;
- Additionally, there are **low-level** APIs for *library and framework developers* to:
  - create and manage event loops, which provide asynchronous APIs for networking, running subprocesses, handling OS Signals, etc;
  - implement efficient protocols using transports;
  - bridge callback-based libraries and code with async/await syntax.

PROJECT

- **Basics**
  - Studying the LAMP Wikipedia System.
  - Creating an asynchronous server system.
  - Client-server interaction, with servers also interacting with servers.
  - Servers interacting with an outside source (Google Places) to get information which is then given back to client.

CLIENT / SERVER

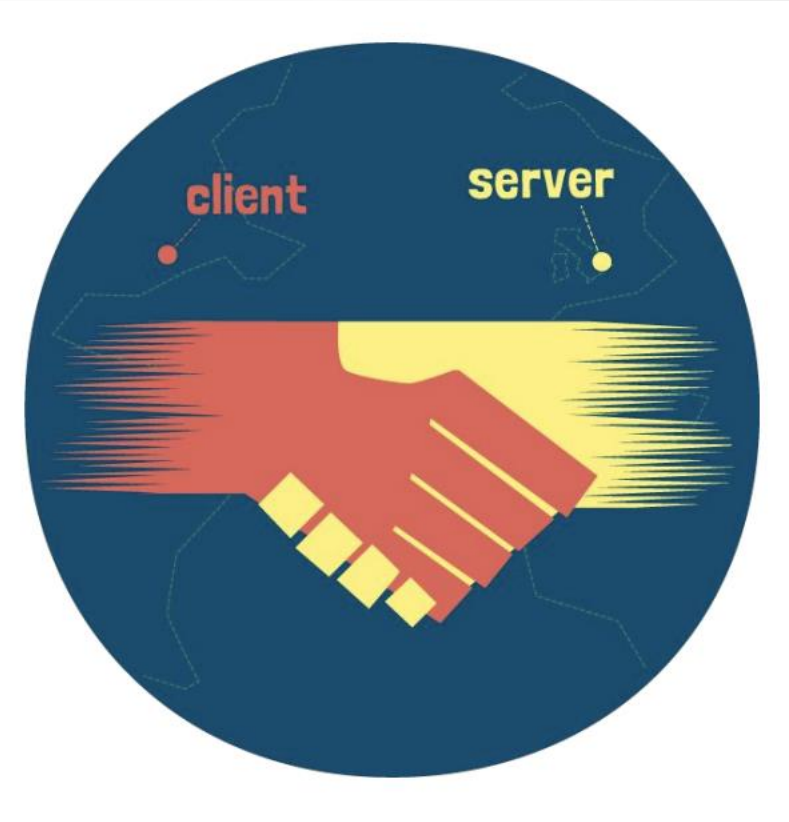
- **1 Client - 1 Server**
  - Interact using TCP
    - Think of it like a text messaging app
  - Client sends server messages (requests)
  - Server sends messages in reply (responses)

SERVER / SERVER

- Not every server can communicate with each other
- Need a way of spreading relevant information so all are on same page.
- **Flooding Algorithm**
  - When a server gets new info, propagate it to the other servers it talks to
  - If a server goes down, then comes back up:
  - Doesn't need old info, just new info as usual.

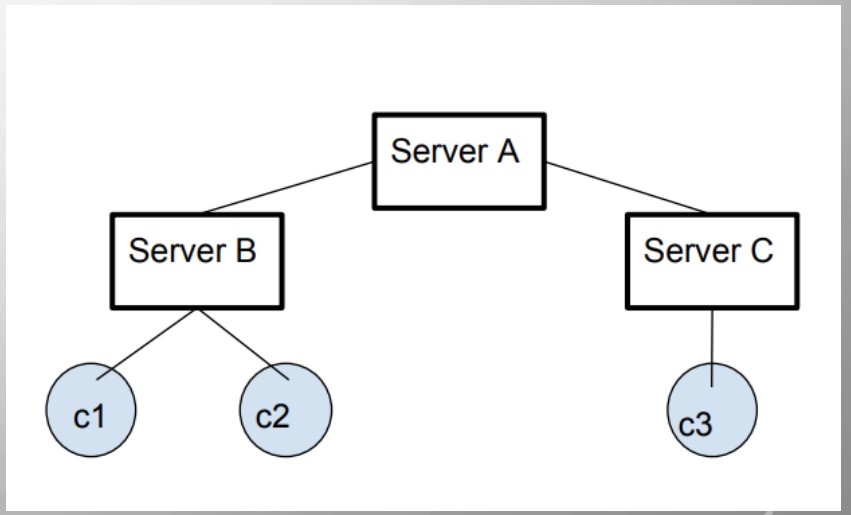
TCP

- Transmission Control Protocol
- Used in conjunction with IP
- It seeks to ensure:
  - No data is lost (retransmission)
  - In-order delivery
  - Congestion control / avoidance
  - Data integrity



APPLICATION SERVER HEARD

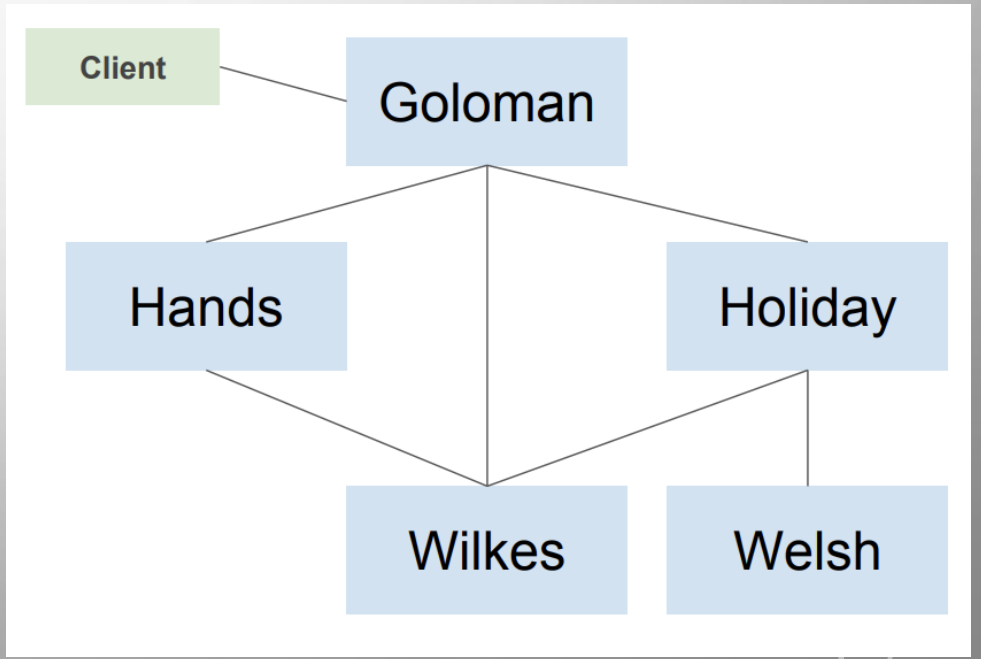
- The multiple application servers communicate directly to each other as well as via the core database and caches.
- For example, you might have three application servers A, B, C such that A talks with B and C but B and C do not talk to each other.
- However, the idea is that if a user's cell phone posts its GPS location to any one of the application servers then the other servers will learn of the location after one or two inter-server transmissions, without having to talk to the database.



PROJECT

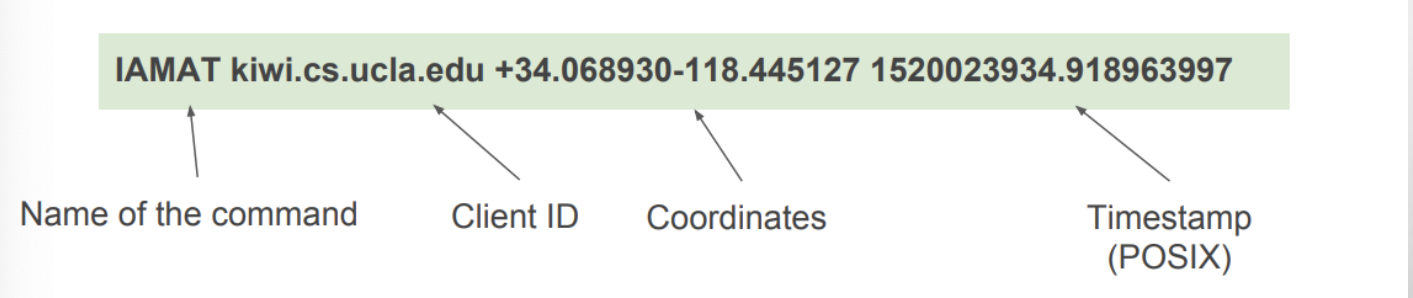
Task: Build a server herd that can synchronize data and communicate with client applications.

- Prototype consists of 5 servers that communicate with each other based on certain criteria:
  - Server ids: 'Goloman', 'Hands', 'Holiday', 'Welsh', 'Wilkes'
  - Goloman talks with Hands, Holiday and Wilkes.
  - Hands talks with Wilkes.
  - Holiday talks with Welsh and Wilkes.
- Each server should accept TCP connects from clients.

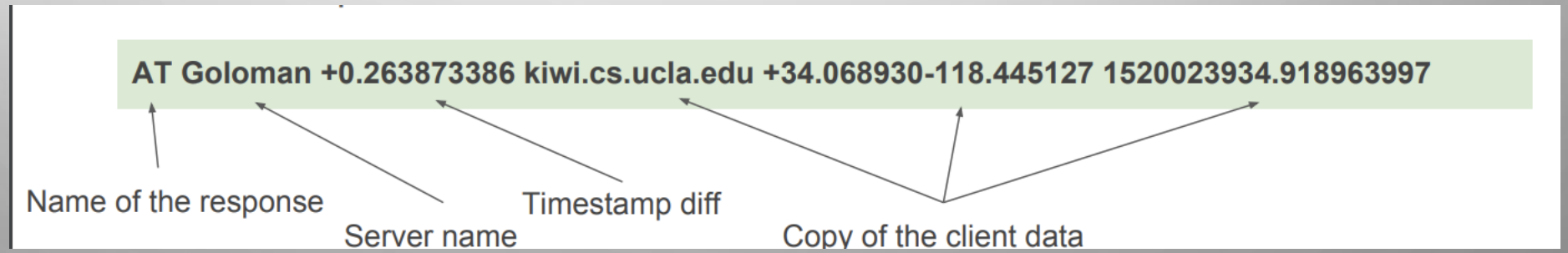


CLIENT-SERVER COMMUNICATION

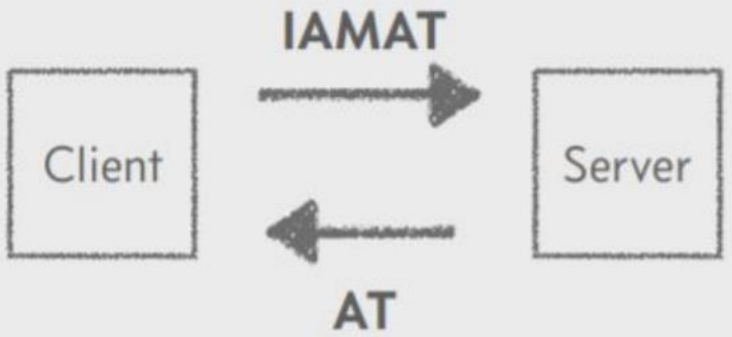
- Client sends a location to the server by sending a message:



- Server response to clients:



PROJECT - IAMAT



client sends server:

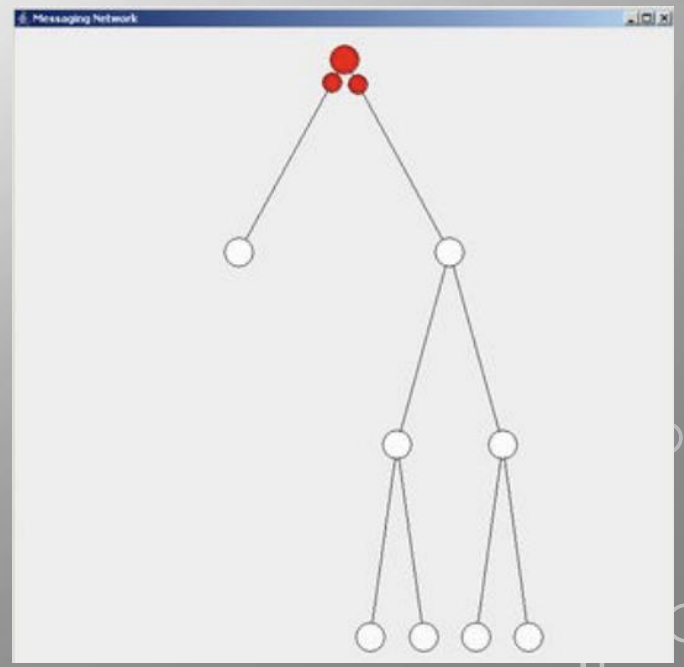
	ISO 6709	POSIX time
IAMAT kiwi.cs.ucla.edu	+34.068930-118.445127	1479413884.392014450

server responds with:

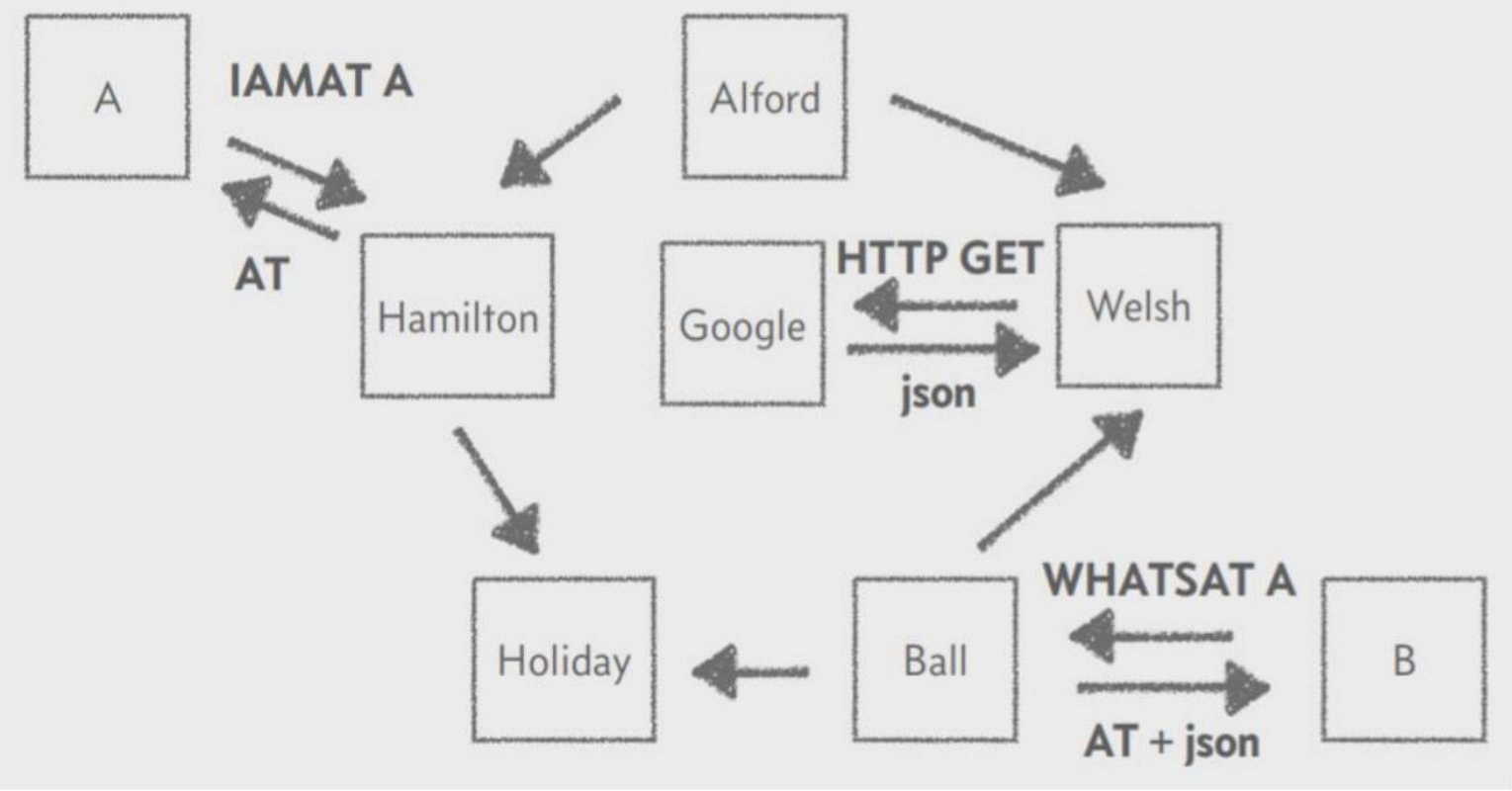
		ISO 6709	POSIX time
AT	Alford +0.263873386 kiwi.cs.ucla.edu	+34.068930-118.445127	1479413884.392014450

SERVER-SERVER COMMUNICATION

- After a server receives a location, it must inform the other servers about the updated location
  - Implement a flooding algorithm so that every server receives the message, even if it is not directly connected to the original server
- Challenge: Must prevent infinite loops
  - You can decide what type of messages servers use to communicate
  - If a server goes down, all the other servers should still function normally
  - No need to propagate old messages when the server is restarted



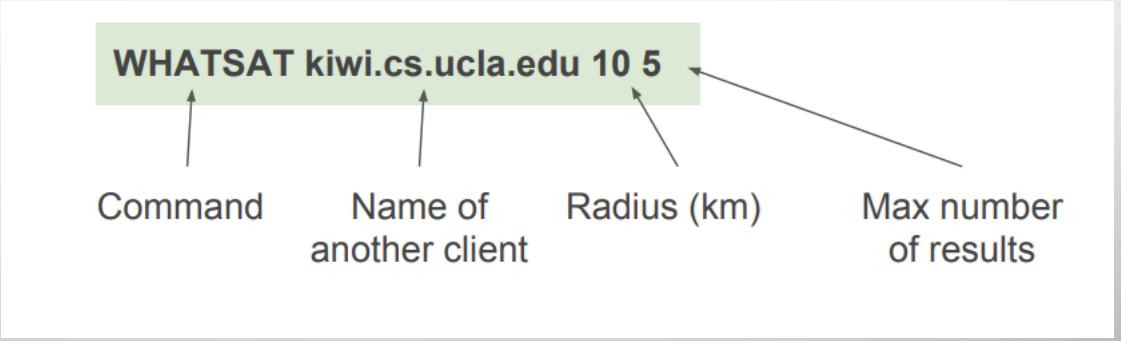
PROJECT - FLOODING ALGORITHM



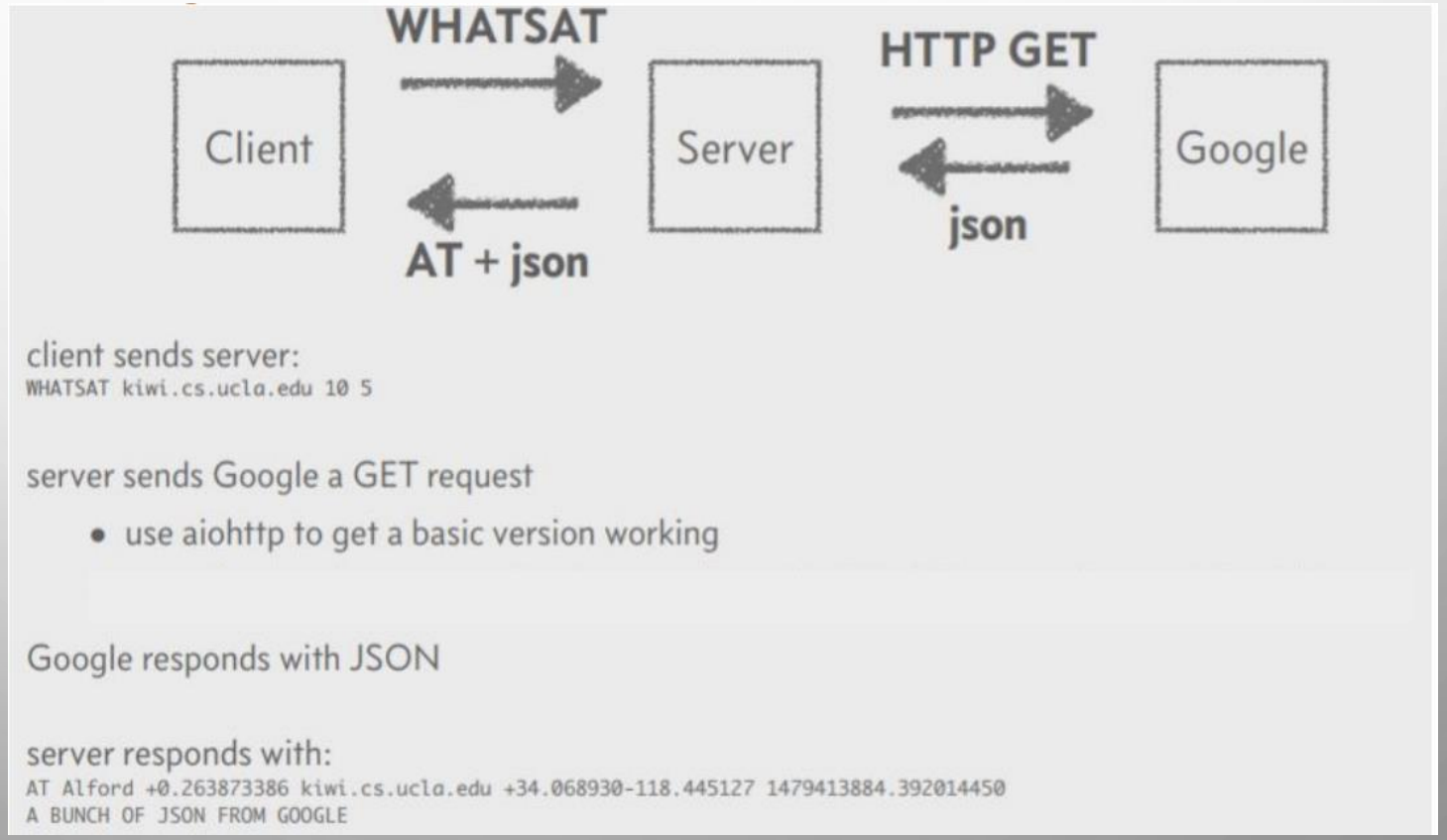
'talks to' relationship is bidirectional.

CLIENT-SERVER COMMUNICATION - WHATSAT

- Clients can ask what is near one of the clients:



PROJECT - WHATSAT



CLIENT-SERVER COMMUNICATION - WHATSAT

- Server uses Google Places API to find the 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)
- Append the usual “AT …” line before the Google’s response

CLIENT-SERVER COMMUNICATION - WHATSAT

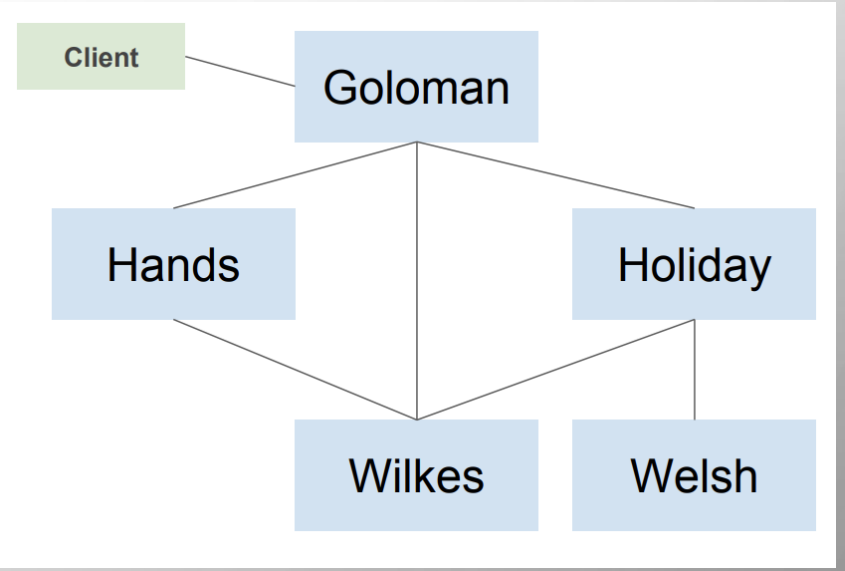
- Server responds:

(JSON FORAMT)

```
AT Goloman +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" : [
            {
               "height" : 1200,
               "html_attributions" : [ "From a Google User" ],
               "photo_reference" : "CnR...4dY",
               "width" : 1600
            }
         ],
         "rating" : 4.5,
         "reference" : "CpQ...r5Y",
         "types" : [ "university", "establishment" ],
         "vicinity" : "Los Angeles"
      },
      ...
   ],
   "status" : "OK"
}
```

RECAP

- Client can send:
  - IAMAT
  - WHATSAT
- IAMAT: - Server saves the location and propagates it
- WHATSAT: - Server calls Google Places API to check what is near the given user, sends results to caller.



HOW TO IMPLEMENT A SERVER IN PYTHON?

- We will use **asyncio** library

  (Your report should discuss pros/cons of using this library, so read the documentation carefully)

```python
import asyncio

async def main():
 server = await asyncio.start_server(handle_connection,host='127.0.0.1', port=12349)
 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())
```

```
[Shruti@Python$nc localhost 12349
John
Hello, John
```

HOW TO CONNECT TO A SERVER IN PYTHON?

```python
import asyncio

async def main():
    reader, writer = await asyncio.open_connection('127.0.0.1', 12349)
    writer.write("John\n".encode())

    data = await reader.readline()
    print('Received: {}'.format(data.decode()))

    writer.close()

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

```
[Shruti@Python$python3 connect.py
Received: Hello, John
```

GOOGLE PLACES API

- https://cloud.google.com/maps-platform/places/
- Gives you information on what is around a given location
- Also can be used to find an address for given coordinates, get details on specific locations
- You need to create a developer account to access the API
  - Free trial is enough for this project

GOOGLE PLACES API

- Example request:

Request

```
location: -33.8670,151.1957
radius: 500
types: food
name: cruise
key: API_KEY
```

Url

```
https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=-33.8670,151.1957&radius=500&types=food&name=cruise&key=YOUR_API_KEY
```

- Searches places near coordinates -33.8670522, 151.1957362
- Limits radius to 500 meters
- Limits searched places to food
- Searches for keyword “cruise” from any information related to that place
- See documentation for description of all the search parameters

HOW TO MAKE HTTP REQUESTS IN PYTHON?

- To query the Google Places API, you will need to send Google servers an HTTP request. However, asyncio only supports the TCP and SSL protocols. Your server will need to manually create and send an HTTP GET.
- Use **aiohttp** library
  - This is only for making requests to Google Places API, do not use it for server functionality!

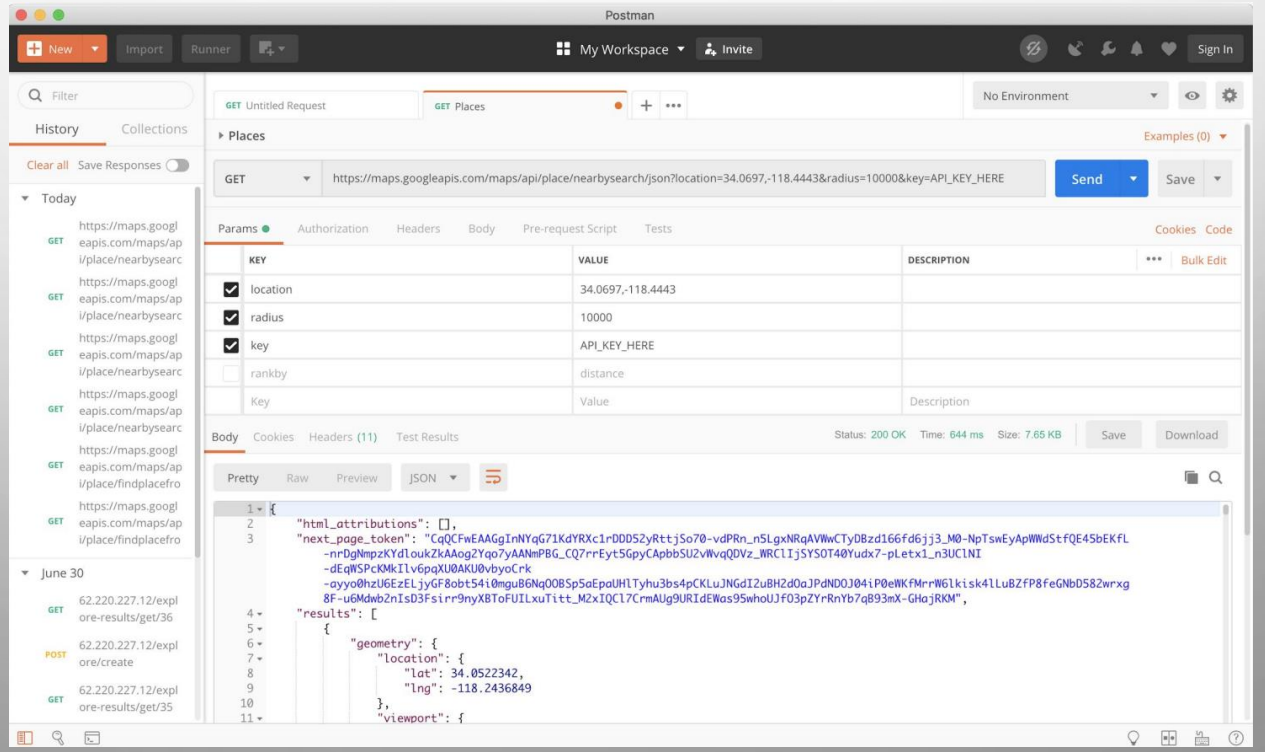
```
async with aiohttp.ClientSession() as session:
    params = [('param-name1', 'some value'), ('param-name2', '100')]

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

- Note: you can re-use the same session for all the requests.

TESTING GOOGLE PLACES API REQUESTS

- Postman is a free tool for testing HTTP requests



PROJECT - GENERAL TIPS

- Use telnet or nc to test your protocol.
- Telnet generates TCP messages
- Make sure your server works properly when sending requests through asyncio instead of telnet
- Servers should use the most recent location of a client in their places requests.
- Check the timestamp of different AT messages.

REPORT

- Maximum 5 pages.
- Discuss pros/cons of asyncio
  - Is it suitable for this kind of an application?
  - What problems did you run into?
  - Any problems regarding type checking, memory management, multithreading?
  - Compare to Java
  - How does asyncio compare to Node.js?

PROJECT

- If you are using SEASnet servers, request ports from us.
- A Google sheet will be shared soon. We'll let you know which ports will be dedicated to you.
- Make sure the requests/responses look exactly the same as instructed, as we'll use automated tests to grade the submissions.
- Start early as the project is time consuming and the due date is Next Friday (03/08/2019).
- The weightage of this project is 10% of your grade.
- All the best!

USEFUL READING

- https://hackernoon.com/asyncio-for-the-working-python-developer-5c468e6e2e8e
- https://www.programiz.com/python-programming/generator
- https://www.saltycrane.com/blog/2008/01/how-to-use-args-and-kwargs-in-python/
- https://github.com/python/cpython/blob/master/Lib/asyncio/
- https://searchnetworking.techtarget.com/definition/TCP
- https://www.geeksforgeeks.org/iterators-in-python/
- https://developers.google.com/places/webservice/search
- https://console.developers.google.com/