```
import asyncio
import aiohttp
import json
import sys
import time
import re

# Google Places API key
API_KEY = "AIzaSyAe6BULCewKb1OtTSSThVAhZ_TgRERb2Tw"

# Hardcode dictionary with key as servername and value as its connections
flood_sequence = {
    "Goloman": ["Hands", "Holiday", "Wilkes"],
    "Hands": ["Goloman", "Wilkes"],
    "Holiday": ["Goloman", "Welsh", "Wilkes"],
    "Welsh": ["Holiday"],
    "Wilkes": ["Goloman", "Hands", "Holiday"]
}

# My ports: 12048 (START) - 12056 (END)
ports = {
    "Goloman": 12048,
    "Hands": 12049,
    "Holiday": 12050,
    "Welsh": 12051,
    "Wilkes": 12052
}

# Save client information
clients = {}

#####################################################################
#                      Process & Verify Requests                    #
#####################################################################

# Break request up into a list of words
def tokenize_request(command):
    return command.strip().split()

# Test if something is a number
def is_number(x):
    try:
        float(x)
        return 1
    except ValueError:
        return -1

# Separate and verify latitude and longitude
def get_geo_coord(lat_long):
    # Get all signs in the string
    signs = []
    for i in lat_long:
        if i == '+' or i == '-':
            signs.append(i)

    # Check there are only two signs
    if len(signs) != 2:
        return None

    # Get the floats
    coordinates = re.split("[\+\-]", lat_long)
    coordinates.pop(0)

    # Check there's only two floats
    if len(coordinates) != 2:
        return None

    # Check that the floats are in fact floats
    for i in coordinates:
        if is_number(i) == -1:
            return None

    # Returns a list of the separated latitude and longitude
    final_coords = list(x+str(y) for x, y in zip(signs, coordinates))
    return final_coords

# Verify IAMAT operands
def verify_IAMAT(request):
    # Check request is length 4
    if len(request) != 4:
        return -1

    # Check coordinates
    coords = get_geo_coord(request[2])
    if coords is None:
        return -1

    # Check timestamp
    if is_number(request[3]) == -1:
        return -1

    return 1

# Verify WHATSAT operands
def verify_WHATSAT(request):
    # Check request is length 4
    if len(request) != 4:
        return -1

    # Check radius is a float
    if is_number(request[2]) == -1:
        return -1

    # Check radius is in range
    if float(request[2]) <= 0.0 or float(request[2]) > 50.0:
        return -1

    # Check upperbound is in range and an integer
    try:
        int(request[3])
        if int(request[3]) <= 0 or int(request[3]) > 20:
            return -1
    except ValueError:
        return -1

    return 2

# Verify AT has all contents
def verify_AT(request):
    if len(request) != 6:
        return -1

    return 3

# Verify a request
def process_request(request):
    if request[0] == "IAMAT":
        return verify_IAMAT(request)
    elif request[0] == "WHATSAT":
        return verify_WHATSAT(request)
    elif request[0] == "AT":
        return verify_AT(request)
    else:
        return -1

    return -1

####################################################################
#                        Requests Handlers                         #
####################################################################

# Returns difference between received and sent time; else for exhaustion
def get_time_diff(sent_timestamp, received_timestamp):
    time_diff = float(received_timestamp) - float(sent_timestamp)
    final_diff = None
    if time_diff > 0:
        final_diff = "+" + str(time_diff)
    else:
        final_diff = str(time_diff)

    return final_diff

# Check if data in server is more recent than data sent to server; TA
said no older timestamps will be sent by client,
# so this should always return -1 on first try
def check_time(new_timestamp, old_timestamp):
    if float(new_timestamp) <= float(old_timestamp):
        return 1

    return -1

# Flooding algorithm; have server act like client and propagate info
async def flood(client_info, server_name):
    for server in flood_sequence[server_name]:
        server_log.write("Connecting to server {0} from server
{1}...\n".format(server, server_name))
        try:
            reader, writer = await asyncio.open_connection('127.0.0.1',
ports[server])
            server_log.write("Success.\n")
            writer.write(client_info.encode())
            writer.write_eof()
            await writer.drain()
            writer.close()
        except:
            server_log.write("Failed to connect to server
{0}.\n".format(server))
            pass

# Handle IAMAT request
```

```
async def do_IAMAT(msg_list, received_timestamp):
    # client_info[0] = servername
    client_info = []
    client_info.append(sys.argv[1])

    # client_info[1] = time diff
    time_diff = get_time_diff(msg_list[3], received_timestamp)
    client_info.append(time_diff)

    # client_info[2:4] = client id, coordinates, time sent
    client_info.extend(msg_list[1:])

    # Only update client info if client is nonexistent or timestamp is
more recent
    if msg_list[1] not in clients:
        clients[msg_list[1]] = client_info
    else:
        old_info = clients[msg_list[1]]
        if check_time(msg_list[3], old_info[4]) == -1:
            clients[msg_list[1]] = client_info

    # Format response
    response = ("AT {0} {1} {2}\n".format(sys.argv[1], time_diff, '
'.join(msg_list[1:])))
    asyncio.ensure_future(flood(response, sys.argv[1]))
    return response

# Handle WHATSAT request
async def do_WHATSAT(msg_list):
    # Check if the info exists or not
    if msg_list[1] not in clients:
        return ("? {0}".format(' '.join(msg_list)))
    else:
        # API usage: https://developers.google.com/places/web-
service/search
        client_info = clients[msg_list[1]]
        lat_long = get_geo_coord(client_info[3])
        radius = float(msg_list[2]) * 1000.0
        HTTP_URL =
"https://maps.googleapis.com/maps/api/place/nearbysearch/json?key={0}&loc
ation={1},{2}&radius={3}".format(API_KEY, lat_long[0], lat_long[1],
radius)
        response = ("AT {0}\n".format(' '.join(client_info)))

        # aiohttp usage:
https://docs.aiohttp.org/en/stable/client_quickstart.html#passing-
parameters-in-urls
        async with aiohttp.ClientSession() as session:
            async with session.get(HTTP_URL) as resp:
                json_out = await resp.json()
                json_out['results'] =
json_out['results'][:int(msg_list[3])]
                response += json.dumps(json_out, indent=3)
                response += "\n\n"

        return response

# Handle AT(flood)
async def do_AT(msg_list):
    # Only update and propagate info if client is nonexistent or
timestamp is more recent
    if msg_list[3] not in clients:
        clients[msg_list[3]] = msg_list[1:]
        flood_msg = ("{0}".format(' '.join(msg_list)))
        asyncio.ensure_future(flood(flood_msg, sys.argv[1]))
    else:
        client_info = clients[msg_list[3]]
        if check_time(msg_list[5], client_info[4]) == -1:
            clients[msg_list[3]] = msg_list[1:]
            flood_msg = ("{0}".format(' '.join(msg_list)))
            asyncio.ensure_future(flood(flood_msg, sys.argv[1]))

    return None

# Handle either IAMAT, WHATSAT, or AT(flood)
async def handle_request(protocol_number, msg_list, received_timestamp):
    # If protocol is 1: do IAMAT, 2: WHATSAT, 3: AT(flood)
    if(protocol_number == 1):
        return await do_IAMAT(msg_list, received_timestamp)
    elif(protocol_number == 2):
        return await do_WHATSAT(msg_list)
    else:
        return await do_AT(msg_list)

#####################################################################
#                         Event Loop Handler                        #
#####################################################################

async def handle_echo(reader, writer):
    in_data = await reader.read(-1)
    timestamp = time.time()
    message = in_data.decode()
    server_log.write("IN: " + message + "\n")

    # Process and verify received data
    msg_list = tokenize_request(message)
    protocol_number = process_request(msg_list)

    # Proceed with given protocol
    response = ""
    if protocol_number > 0:
        # Handle either IAMAT or WHATSAT or UPDATE
        response = await handle_request(protocol_number, msg_list,
timestamp)
    else:
        # If request verification sent back an error, handle it
        response = ("? {0}".format(message))

    # Send back response
    if response is not None:
        out_data = response.encode()
        writer.write(out_data)
        writer.write_eof()
        await writer.drain()
        # AT's are terminated by newline, so redundant ATs may have blank
line before them
        server_log.write("OUT: " + response + "\n")

#####################################################################
```

```
#                                Main                                 #
#####################################################################

# Starts server connection loop
async def server_setup():
    server = await asyncio.start_server(handle_echo, '127.0.0.1',
ports[sys.argv[1]])
    async with server:
        await server.serve_forever()

server_log = None
def main():
    # Check valid server name
    if len(sys.argv) != 2:
        print("Error: invalid arguments")
        sys.exit(1)
    if sys.argv[1] not in ports:
        print("Error: servername not found")
        sys.exit(1)

    # Create a server_log
    global server_log
    server_log = open(sys.argv[1] + "_log.txt", "w+")

    # Depending on the passed servername, initialize the port
    server_log.write("Starting server {0} at port
{1}...\n".format(sys.argv[1], ports[sys.argv[1]]))
    asyncio.run(server_setup())

# Execute only when we're in the main
if __name__ == '__main__':
    main()
    server_log.write("Closing server {0} at port {1}.
Goodbye.\n".format(sys.argv[1], ports[sys.argv[1]]))
    server_log.close()
```