Cloud Computing Assignment

Aritra Das Ray (s423160)

December 2023

Contents

1 Introduction

1.1 Aims

The aim of the work described in this report is to explore cloud computing, design and deploy a cloud based system that processes and stores data at scale and serves data to clients on request.

1.2 Objective

In order to achieve the aim we will be carrying out following tasks:

- Download sensor data from online source to a S3 bucket.
- Load processed data from S3 to DynamoDb by running a python script on AWS CloudShell.
- Setup a Client and Server on EC2 instances using the Python sockets library.
- Servicing client request for AQI information through the Server.

2 Methodology

2.1 Theory - Cloud Computing

What is Cloud Computing?
As per the AWS Documentation, "Cloud computing is the on-demand delivery of compute power, database, storage, applications, and other IT resources via the internet with pay-as-you-go pricing." [1]

Advantages of Cloud Computing

- Cost Flexibility: Transition from fixed capital expenses to variable expenses, paying only for consumed resources, reducing upfront investments, and adapting quickly to new applications.[1]
- Economies of Scale: Leverage massive economies of scale in cloud computing, benefiting from lower pay-as-you-go prices due to aggregated usage from a diverse customer base.[1]
- Capacity Precision: Eliminate capacity guessing by accessing resources on-demand, scaling up or down with minimal notice, avoiding costly idle resources or limited capacity issues.[1]
- Speed and Agility: Accelerate resource availability from weeks to minutes in a cloud environment, enhancing organizational agility, lowering costs, and expediting experimentation and development.[1]

- Infrastructure Focus: Shift focus from running and maintaining data centers to customer-centric projects, allowing businesses to concentrate on differentiation rather than infrastructure management.[1]
- Global Deployment: Deploy applications globally in minutes across multiple AWS Regions, ensuring lower latency, improved customer experiences, and cost-effective expansion worldwide.[1]

2.2 Environmental Sensor Data

Sensor.Community represents a groundbreaking global initiative in the realm of environmental data acquisition, fostering a community-driven network of contributors dedicated to the creation of Open Environmental Data. The platform's overarching mission is to cultivate a shared passion for nature that is not only genuine but also joyous and positive, thereby enriching lives through collective curiosity.[6]

This dynamic repository boasts a diverse array of environmental sensors strategically positioned worldwide, capturing a spectrum of crucial parameters such as temperature, humidity, pressure, and particulate matter. These sensors, meticulously deployed by enthusiastic community members, form a comprehensive global sensor network, providing an unparalleled dataset for scientific exploration and inquiry.[6]

The data shared on Sensor.Community is made available through the Database Contents License (DbCL) v1.0, a legal framework that grants users a worldwide, royalty-free, non-exclusive, perpetual, and irrevocable copyright license. This license explicitly permits a broad spectrum of activities, including commercial use, fostering a collaborative and inclusive approach to data utilization.[5]

Contributors to Sensor.Community adhere to the Open Database License (ODbL) 1.0, ensuring the responsible and ethical use of the shared data. The DbCL emphasizes the non-assertion of copyright over factual information, acknowledging the importance of free access to valuable scientific insights.[5]

It is crucial to note that the contents provided by Sensor.Community are shared "as is," with no warranty of any kind. However, the commitment to open data extends to a disclaimer of liability, with the license carefully crafted to limit the licensor's responsibility to the extent permitted by law. This ensures that contributors and users alike engage with the data responsibly, fostering a collaborative spirit in the pursuit of scientific knowledge. Sensor.Community stands as a testament to the power of community-driven initiatives in democratizing environmental data, contributing to a more informed and empowered global society. [6] [5]

2.3 AWS Services Used

1. **AWS EC2** - Amazon Elastic Compute Cloud (Amazon EC2) represents a cornerstone in cloud computing, offering resizable virtual machines as a fundamental infrastructure-as-a-service (IaaS) solution. This service provides unparalleled flexibility, allowing users to independently manage server configurations, select operating systems, and tailor the size and resource capabilities of launched servers. Leveraging the concept of virtual machines, EC2 caters to IT professionals familiar with on-premises computing, offering a seamless transition to the cloud environment. The acronym EC2 encapsulates its key attributes: Elasticity, signifying the effortless scalability of servers; Compute, emphasizing its role in hosting applications and processing data; and Cloud, highlighting the cloud-hosted nature of EC2 instances. As one of the pioneering AWS services, Amazon EC2 remains a paramount choice for users seeking dynamic and efficient compute resources to power diverse applications and computational workloads in the cloud. [4]

2. **AWS S3** - Amazon Simple Storage Service (S3) stands as a pinnacle in managed cloud storage, offering a seamlessly scalable solution designed with a remarkable durability. S3 enables the storage of an unlimited number of objects within universal and uniquely named buckets, supporting read, write, and delete operations. With each object capable of reaching up to 5 TB in size, S3 ensures redundant storage across multiple facilities and devices by default. Significantly, the service dissociates stored data from specific servers, alleviating users from infrastructure management. Facilitating low-latency access over HTTP or HTTPS, S3 accommodates a vast array of data types, including images, videos, and large database snapshots. Privacy and security are paramount, with fine-grained access control through IAM policies and bucket policies. [3]

3. **AWS DynamoDb** - DynamoDB is a dynamic and efficient NoSQL database service within the AWS ecosystem, designed to meet the demands of diverse applications requiring consistent, sub-millisecond latency at any scale. Amazon seamlessly manages the underlying data infrastructure and employs fault-tolerant architecture by redundantly storing data across multiple facilities in a native US Region. DynamoDB offers unparalleled flexibility, allowing users to create tables and items, with the system automatically handling data partitioning to meet varying workload requirements. Notably, the absence of a practical limit on the number of items in a table accommodates extensive storage needs, with some users managing tables containing billions of items. A notable feature lies in the capability of items within the same table to possess different attributes, facilitating schema evolution without the need for migration. DynamoDB's solid state drive (SSD) storage, coupled with its straightforward query language, ensures consistent low-latency query performance. Additionally, the service allows manual or automatic provisioning of read and write throughput,

ensuring scalability as application popularity grows. Key features include global tables for automatic replication across AWS Regions, encryption at rest, and item Time-to-Live (TTL), solidifying DynamoDB as a versatile and robust solution for scalable and low-latency data management.[2]

3 System and Code Architecture

This section outlines the development and implementation of a robust system leveraging Amazon Web Services (AWS) to process and retrieve environmental data sourced from the Sensor.Community platform. The system seamlessly integrates AWS services, including S3 for data storage, DynamoDB for efficient data management, and EC2 instances for client-server communication. A Python script, executed in the Cloud Shell, orchestrates the data processing pipeline, contributing to the generation of valuable insights. [2] [3] [4]

3.1 Data Ingestion

In the data ingestion process, environmental sensor data is seamlessly retrieved from the Sensor.Community website and efficiently transferred to an AWS S3 bucket using a combination of Linux commands.

```
curl -o data_dd_MM_yy_hh_mm.json https://data.sensor.
    community/static/v2/data.24h.json

aws s3 mv data_dd_MM_yy_hh_mm.json s3://
    mysensordatainjson/
```

The initial step involves leveraging the 'curl' command to download the latest sensor data in JSON format from the specified Sensor.Community data endpoint. As a standard the data files are named in the format "$data\_dd\_MM\_yy\_hh\_mm.json$. Subsequently, the 'aws s3 mv' command facilitates the swift movement of the acquired data to a designated AWS S3 bucket named 'mysensordatainjson.' This streamlined process not only ensures the timely acquisition of up-to-date environmental information but also establishes a seamless data flow, laying the foundation for further analysis, storage, and utilization within the AWS ecosystem.

3.2 Data Processing Pipeline

The Python script "LoadingData.py", executed within the AWS Cloud Shell, constitutes the core of the data processing pipeline. This script undertakes essential tasks such as data cleansing, transformation, and calculations before loading the refined data into an AWS DynamoDB table. The DynamoDB setup ensures quick and efficient access to processed environmental data.

The script utilizes the Boto3 library to interact with Amazon Web Services

(AWS) and employs JSON and datetime libraries for data manipulation. Key functionalities include:

S3 Data Retreival
The script connects to an S3 bucket named 'mysensordatainjson' and downloads a specific JSON file available in the S3 bucket containing sensor data.

DynamoDB Data Loading
Boto3 is employed to interact with DynamoDB, AWS's NoSQL database service. The script connects with a pre-configured DynamoDB table named $air\_quality\_data$. Two functions, $has\_p1\_p2\_data$ and AQI_cal, validate the presence of relevant sensor data and calculate Air Quality Index (AQI) values based on particulate matter (P2.5 and P10) concentrations. The $extract\_fields$ function extracts essential fields from each sensor data record, including latitude, longitude, altitude, country, timestamp, date, P2.5, P10, AQI, and AQI_range. The main loop iterates through the retrieved data, processes records with P1 and P2 data, extracts relevant fields, and inserts them into the DynamoDB table.

Python Dependencies
Boto3: AWS SDK for Python, facilitating interaction with AWS services.
JSON: Used for reading and parsing JSON data.
Datetime: Enables the manipulation of timestamp data for conversion and formatting.

3.3 Server Setup

The "Server.py" Python script configures a server on an EC2 instance, managing client connections, DynamoDB interactions, and CSV file operations. Key components and functionalities include [7]:

Libraries and Dependencies
socket: Facilitates the creation of a server socket to handle client connections.
threading: Implements multi-threading to concurrently handle multiple client connections.
json, boto3, os, csv: Support various data manipulation and AWS DynamoDB interactions.

Server Initialization
We Declare and Establish server-related global constants such as HEADER, PORT, ADDR, FORMAT, and $DISCONNECT\_MESSAGE$.
Then Create a socket, bind it to the IP address of the EC2 instance, and listen for incoming connections.

DynamoDB Connection
The EC2 instance utilizes the Boto3 library to connect to DynamoDB, specifically to the table named '$air\_quality\_data$'.

CSV Operations
Two functions *write_to_csv* and *send_csv* are implemented to create and handle csv files and send it to the clients respectively. [8]

Client Handling
The *handle_client* function is threaded for concurrent client processing. Upon connection, retrieves a message from the client, queries DynamoDB for the corresponding item, and prepares data for CSV file creation. Writes the retrieved data to a CSV file, sends it to the client, and removes the temporary CSV file afterward.

Server Start up and Execution
The "start" function initiates the server, listens for incoming connections, and spawns a new thread to handle each client connection concurrently.
The script continuously listens for incoming connections, and upon connection, handles client requests by querying DynamoDB, creating CSV files, and sending data back to clients.

Thus the "server.py" script is a fundamental part of a larger AWS system, providing a server-side mechanism for interacting with DynamoDB and facilitating data exchange with clients through CSV files. Its modular design ensures scalability and efficient handling of multiple client requests concurrently.

3.4 Client Setup

This Python script configures a client on an EC2 instance, facilitating communication with a server. Key components and functionalities include [7]:

Libraries and Dependencies
socket: Enables the creation of a client socket for connecting to the server.

Client Initialization
We have declared and established client-related global constants such as HEADER, PORT, SERVER, ADDR, FORMAT, and $DISCONNECT\_MESSAGE$.
Then we create a socket and connect to the server address.

CSV Operations
We implement functions *receive_csv* to receive CSV data from the server and store it in a local file. [8]

Client Send Operation
The *send* function sends messages to the server, including a specific message for disconnecting ($DISCONNECT\_MESSAGE$). For each non-disconnect message, the client also receives a CSV file from the server, stores it locally, and prints its content.

Execution
The script connects to the server using a predefined address (SERVER, PORT). Sends a message specifying the particular record "id" to the server, receives and stores the corresponding CSV file, and prints its content.
Sends a disconnect message to the server, closing the connection.

Thus the "client.py" script serves as a client-side interface for interacting with a server by sending messages, receiving CSV data, and managing the connection state. The modular design ensures flexibility and facilitates the integration of this client into a larger, distributed system.

3.5 DynamoDB Schema

The schema of the data stored in DynamoDB follows a comprehensive structure capturing vital information from environmental sensor records. Each record is uniquely identified by an 'id' and includes geographical details such as latitude, longitude, altitude, and country from the sensor's location. The timestamp is stored in its original format and is also transformed into a date format for ease of analysis. The concentrations of particulate matter, denoted as 'P2.5' and 'P10', are recorded, enabling a detailed understanding of air quality. Additionally, the calculated Air Quality Index (AQI) and its corresponding range provide insights into the overall air quality classification. This structured schema ensures that essential parameters related to location, time, and air quality metrics are readily available for analysis and interpretation, forming a robust foundation for comprehensive environmental monitoring.

The table, named '*air_quality_data*' consists of the following columns:

1. **id (Primary Key)**: Unique identifier for each sensor data record. Data Type: String.
2. **latitude**: Latitude coordinates of the sensor location. Data Type: Number.
3. **longitude**: Longitude coordinates of the sensor location. Data Type: Number.
4. **altitude**: Altitude of the sensor location. Data Type: Number.
5. **country**: Country information associated with the sensor location. Data Type: String.
6. **timestamp**: Timestamp indicating when the sensor data was recorded. Data Type: String.
7. **date**: Date extracted from the timestamp for easy reference. Data Type: String.

8. **P2.5**: Particulate matter (PM2.5) concentration recorded by the sensor. Data Type: Number.

9. **P10**: Particulate matter (PM10) concentration recorded by the sensor. Data Type: Number.

10. **AQI**: Air Quality Index calculated based on PM2.5 and PM10 concentrations. Data Type: Number.

11. *AQI_range*: Categorization of AQI into specific ranges such as 'Low,' 'Medium,' 'High,' and 'Very High.' Data Type: String.

4 Discussion and Conclusion

4.1 Elasticity and Auto scaling

The system demonstrates elasticity and auto-scaling capabilities across multiple components, enhancing its responsiveness to varying workloads and ensuring optimal resource utilization. Firstly, the data ingestion process from Sensor.Community to AWS S3 showcases elasticity by efficiently handling dynamic data volumes. Leveraging the 'curl' command for data retrieval and subsequent 'aws s3 mv' command for storage, this process adapts seamlessly to fluctuating data sizes, automatically scaling its performance based on the incoming environmental sensor information. The inherent elasticity of AWS S3 further ensures that the storage capacity scales effortlessly with the growing influx of data, providing a robust foundation for accommodating diverse data sets.

Secondly, the data processing and transformation stage, where information is loaded from S3 to DynamoDB, also exhibits elasticity. The script intelligently manages data variations, employing the dynamic nature of DynamoDB to handle varying object sizes and types. The DynamoDB's ability to scale throughput capacity in response to changing demands ensures optimal performance during data loading. This adaptability allows the system to efficiently process and store sensor data in real-time, meeting the demands of the dynamic environmental monitoring landscape.

In the server provisioning and communication component, the EC2 instance-based server deployment and the server-client interaction highlight remarkable elasticity. The server script, hosted on an EC2 instance, automatically scales to accommodate incoming client requests, enabling concurrent connections through multi-threading. Additionally, the server's connection to DynamoDB benefits from DynamoDB's auto-scaling feature, adjusting read and write capacity based on workload fluctuations. This elasticity ensures that the server seamlessly accommodates varying communication loads, guaranteeing efficient and responsive data retrieval and transmission.

In conclusion, the auto-scaling and elasticity embedded in each component of the system contribute to its overall resilience and efficiency. The design ensures that the system can gracefully handle fluctuations in data volume, processing demands, and communication requirements, making it well-suited for the dynamic and evolving nature of environmental sensor data management.

4.2 Implication of Data Security and Sovereignty w.r.t. sensor data

The data obtained from Sensor Community are governed by Database Contents License (DbCL) v1.0 issued by The Open Knowledge Foundation. Based on this license certain Data Security and Data Sovereignty issues arise.

- **Limited Warranty** The disclaimer stated in the License states that the data provided "as is" poses challenges in ensuring the accuracy, reliability, and security of the data. Lack of assurance impacts reliability of scientific findings derived from this data
- **Global Nature of Sensor data** The data originates from diverse geographical locations and jurisdictions. Navigating international data protection laws and ensuring compliance will be a challenge.
- **Uncertain Jurisdictional Control** Absence of explicit statements regarding jurisdictional control and legal obligations in the licensing documentation can create uncertainties about which country's regulations govern the data, posing challenges in establishing a clear framework for data sovereignty.

In order to work around these concerns we must take the following steps:

- Ensure that the data is cleaned and validated before being used for any scientific endeavor
- Get a better understanding of the International laws govering the use of this data.
- Strive for community driven better jurisdictional control on the data use.

References

[1] Amazon Web Services. AWS Academy Cloud Foundations Module 01, Student Guide, Version 2.0.13[Internet]. Seattle: AWS; 2022 [cited 20 Dec 2023] Available from: https://awsacademy.instructure.com/courses/63756/modules/items/5639824

[2] Amazon Web Services. AWS Academy Cloud Foundations Module 08 Student Guide Version 2.0.13[Internet]. Seattle: AWS; 2022 [cited 20 Dec 2023] Available from: https://awsacademy.instructure.com/courses/63756/modules/items/5640737

[3] Amazon Web Services. AWS Academy Cloud Foundations Module 07 Student Guide Version 2.0.13[Internet]. Seattle: AWS; 2022 [cited 20 Dec 2023] Available from: https://awsacademy.instructure.com/courses/63756/modules/items/5640648

[4] Amazon Web Services. AWS Academy Cloud Foundations Module 06 Student Guide Version 2.0.13[Internet]. Seattle: AWS; 2022 [cited 20 Dec 2023] Available from: https://awsacademy.instructure.com/courses/63756/modules/items/5640517

[5] The Open Knowledge Foundation. Database Contents License (DbCL) v1.0[Internet]. London, UK; 2020 Feb 20 [cited 20 Dec 2023] Available From: https://opendatacommons.org/licenses/dbcl/1-0/

[6] Sensor.Community. Home Page[Internet]. Stuttgart, Germany;2020 Feb 20 [cited 20 Dec 2023] Available from: https://sensor.community/en/

[7] Tech with Tim. Python Socket Programming Tutorial[video on the Internet]. San Bruno (CA): Youtube; 2020 Apr 05[cited 2023 Dec 20]. Available from: https://www.youtube.com/watch?v=3QiPPX-KeSc

[8] File Transfer via Sockets in Python - NeuralNine NeuralNine. File Transfer via Sockets in Python[video on the Internet]. San Bruno (CA): Youtube; 2022 Sep 12[cited 2023 Dec 20]. Available from: https://www.youtube.com/watch?v=qFVoMo6OMsQ

A Scripts

A.1 LoadingData.py

```
import boto3
import json
from datetime import datetime

#accessing s3 bucket

bucket_name = 'mysensordatainjson'
file_key = 'data_20_12_23_18_55.json'

s3 = boto3.client('s3')

with open('data_20_12_23_18_55.json','wb') as file:
    s3.download_fileobj(bucket_name, file_key, file)

with open('data_20_12_23_18_55.json','r') as file:
    data = json.load(file)

#loading data dynamodb
dynamodb = boto3.resource('dynamodb')
table_name = 'air_quality_data'
table = dynamodb.Table(table_name)

def has_p1_p2_data(record):
    for sdv in record.get('sensordatavalues',[]):
        if sdv.get('value_type') in ['P1','P2']:
            return True
    return False

def AQI_cal(P1, P2):
    P1 = float(P1) if P1 is not None else 0
    P2 = float(P2) if P2 is not None else 0

    if 0<=P1<=11 or 0<=P2<=16:
        AQI_value = 1
        AQI_range = "Low"
    elif 11<P1<=23 or 16<P2<=33:
        AQI_value = 2
        AQI_range = "Low"
    elif 23<P1<=35 or 33<P2<=50:
        AQI_value = 3
        AQI_range = "Low"
    elif 35<P1<=41 or 50<P2<=58:
```

```
            AQI_value = 4
            AQI_range = "Medium"
        elif 41<P1<=47 or 58<P2<=66:
            AQI_value = 5
            AQI_range = "Medium"
        elif 47<P1<=53 or 66<P2<=75:
            AQI_value = 6
            AQI_range = "Medium"
        elif 53<P1<=58 or 75<P2<=83:
            AQI_value = 7
            AQI_range = "High"
        elif 58<P1<=64 or 83<P2<=91:
            AQI_value = 8
            AQI_range = "High"
        elif 64<P1<=70 or 91<P2<=100:
            AQI_value = 9
            AQI_range = "High"
        elif P1>70 or P2>100:
            AQI_value = 10
            AQI_range = "Very␣High"
        else:
            AQI_value = 0
            AQI_range = "Very␣Low"

        return AQI_value, AQI_range

    def extract_fields(record):
        P1 = next((sdv['value'] for sdv in record.get('
            sensordatavalues', []) if sdv['value_type'] ==
            'P1'), None)
        P2 = next((sdv['value'] for sdv in record.get('
            sensordatavalues', []) if sdv['value_type'] ==
            'P2'), None)

        aqi, range_value = AQI_cal(P1,P2)
        timestamp_str = record.get('timestamp')
        datetime_obj = datetime.strptime(timestamp_str, '%
            Y-%m-%d␣%H:%M:%S')
        date_str = datetime_obj.strftime('%Y-%m-%d')
        return {
        'id' : record.get('id'),
        'latitude' : record.get('location',{}).get('
            latitude'),
        'longitude' : record.get('location',{}).get('
            longitude'),
        'altitude' : record.get('location',{}).get('
```

```
        altitude'),
    'country' : record.get('location',{}).get('country
        '),
    'timestamp' : record.get('timestamp'),
    'date' : date_str,
    'P2.5' : P1,
    'P10' : P2,
    'AQI' : aqi,
    'AQI_range' : range_value
    }

for record in data:
    if has_p1_p2_data(record):
        fields = extract_fields(record)
        table.put_item(Item = fields)
        print(f"Inserted record with id: {fields.get('
            id')}")

print('Data insertion complete')
```

A.2 Server.py

This is my script which I used to create a server on an EC2 instance :

```
import socket
import threading
import json
import boto3
import os
import csv

#server global variables

HEADER = 64
PORT = 12000
SERVER = socket.gethostbyname(socket.gethostname())
ADDR = (SERVER,PORT)
FORMAT = 'utf-8'
DISCONNECT_MESSAGE = "!DISCONNECT"

server = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
server.bind(ADDR)

#connect to dynamodb
```

```
dynamodb = boto3.resource('dynamodb', region_name = '
   us-east-1')
table = dynamodb.Table('sensor_data_raw')

#create csv
def write_to_csv(data, filename):
    with open(filename, 'w', newline='') as csvfile:
        fieldnames = data[0].keys()
        writer = csv.DictWriter(csvfile, fieldnames=
           fieldnames)

        writer.writeheader()
        for row in data:
            writer.writerow(row)

#send csv
def send_csv(conn, filename):
    with open(filename, 'rb') as file:
        file_data = file.read()
        #message = file_data.encode(FORMAT)
        message_length = len(file_data)
        send_length = str(message_length).encode(
           FORMAT)
        send_length += b' '*(HEADER - len(send_length)
           )
        conn.send(send_length)
        conn.send(file_data)

#server functions

def handle_client(conn, addr):
    print(f"[NEW CONNECTION]{addr} connected.")

    connected = True
    while connected:
        msg_length = conn.recv(HEADER).decode(FORMAT)
        if msg_length:
            msg_length = int(msg_length)
            msg = conn.recv(msg_length).decode(FORMAT)
            if msg == DISCONNECT_MESSAGE:
                connected = False
                print(f"[{addr}] Disconnecting...")
            else:
                retrieve = table.get_item(
                    Key={
                        "id": int(msg)
```

```
                    }
                )
                item = retrieve.get('Item')
                if item:
                    data = [item]
                    filename = f"data_{msg}.csv"

                    # Write data to CSV
                    write_to_csv(data, filename)

                    # Send the CSV file to the client
                    send_csv(conn, filename)

                    # Remove the CSV file after
                        sending
                    os.remove(filename)

                else:
                    item_str = "Item not found"
                    print(f"[{addr}] {msg}")
                    conn.send(item_str.encode())


    conn.close()

def start():
    server.listen()
    print(f"Server is listening on {SERVER}")
    while True:
        conn, addr = server.accept()
        thread = threading.Thread(target =
            handle_client, args = (conn,addr))
        thread.start()
        print(f"[ACTIVE CONNECTIONS] {threading.
            activeCount()-1}")

print('[STARTING] server is starting')
start()
```

A.3 Client.py

```
import socket

#client global variables
HEADER = 64
```

```
PORT = 12000
SERVER = "44.201.134.149"
ADDR = (SERVER,PORT)
FORMAT = 'utf-8'
DISCONNECT_MESSAGE = "!DISCONNECT"

client = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
client.connect(ADDR)

#receive csv
def receive_csv(conn, filename):
    msg_length = conn.recv(HEADER).decode(FORMAT)
    if msg_length:
        msg_length = int(msg_length)
        msg = conn.recv(msg_length)
        print(msg.decode(FORMAT))
        with open(filename, 'wb') as file:
            file.write(msg)

#client send
def send(msg):
    message = msg.encode(FORMAT)
    msg_length = len(message)
    send_length = str(msg_length).encode(FORMAT)
    send_length += b' ' * (HEADER - len(send_length))
    client.send(send_length)
    client.send(message)
    if msg != DISCONNECT_MESSAGE:
        filename = f"data_{msg}.csv"
        receive_csv(client, filename)
        with open(filename, 'rb') as file:
            file_data = file.read()
            print(file_data)

    #data = client.recv(2048).decode(FORMAT)
    #print(data)

send("18456691289")
send(DISCONNECT_MESSAGE)
```