# Burak Erdilli 04.11.2024

Github Link For the Code: https://github.com/BurakErdilli/Rakort

# Question 1: Network Scanner and SSH Connector

## Overview

The Network Scanner and SSH Connector is a FastAPI application designed to scan the `172.29.0.0/16` network to identify active IP addresses and establish SSH connections to them. The application operates asynchronously, enabling the simultaneous processing of multiple IP addresses, and provides endpoints to retrieve information about unreachable IPs.

## Requirements

To run this application, the following libraries are required:

- `fastapi`
- `httpx`
- `paramiko`
- `uvicorn` (for running the server)

**Install the required packages using:**

```
pip install fastapi httpx paramiko uvicorn
```

## Code Breakdown

### Importing Libraries

```
from fastapi import FastAPI
import asyncio
import httpx
import paramiko
```

These libraries are essential for creating the FastAPI application, performing asynchronous HTTP requests, and handling SSH connections.

**Application Initialization**

```
app = FastAPI()
```

This line initializes the FastAPI application.

**Global Variables**

```
unreachable_ips = []  # List to store unreachable IP addresses
network_prefix = "172.29."  # Network prefix for scanning
running = True  # Global variable to control the running state
```

- `unreachable_ips`: Stores unreachable IP addresses.
- `network_prefix`: Defines the base of the IP addresses to scan.
- `running`: Indicates whether the scanning operation is ongoing.

**IP Checking and SSH Connection Functions**
**Checking IP Address**

```
async def check_ip(ip: str):
    """Checks if the specified IP address is reachable and attempts to
establish an SSH connection."""
    ...
```

**Establishing SSH Connection**

```
async def ssh_connect(ip: str):
    """Attempts to establish an SSH connection to the specified IP
address."""
    …
```

**Network Scanning Function**

```
async def scan_network():
    """Scan the specified network for active IP addresses."""
    ...
```

**Application Event Handlers**
**Startup Event**

```
@app.on_event("startup")
async def startup_event():
    """Starts the scanning process when the application starts."""
    asyncio.create_task(scan_network())
```

**Shutdown Event**

```
@app.on_event("shutdown")
async def shutdown_event():
    """Handles the application shutdown event."""
    global running
    running = False
```

**API Endpoints**
**Unreachable IPs Endpoint**

```
@app.get("/unreachable/")
async def get_unreachable_ips():
    """Retrieves the list of unreachable IP addresses."""
    ...
```

**Scan Status Endpoint**

```
@app.get("/scan/")
async def scan():
    """Initiates the network scan."""
    ...
```

**Main Execution Block**

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

This block runs the FastAPI application using Uvicorn.

## Instructions for Usage

Save the code as `app.py`.

Install the required dependencies:

```
pip install fastapi httpx paramiko uvicorn
```

Run the application:

```
uvicorn app:app --reload
```

## Endpoints

To retrieve unreachable IPs:

```
http://127.0.0.1:8000/unreachable/
```

To check the scan status:

```
http://127.0.0.1:8000/scan/
```

---

# Question 2: IP Ping App Documentation

## Overview

This FastAPI application, `ip_ping_app.py`, continuously pings a predefined range of IP addresses (from `192.168.1.1` to `192.168.1.255`) to check their reachability. It collects the IP addresses that are unreachable and provides an endpoint to retrieve this information. The application runs asynchronously and handles user interruptions gracefully.

## Key Features

- **Asynchronous Ping**: Utilizes `httpx` for non-blocking HTTP requests to check the reachability of IP addresses.
- **Continuous Monitoring**: Pings all specified IPs every 5 seconds while the application is running.
- **Graceful Shutdown**: Supports safe quitting via keyboard interruption (Ctrl+C).
- **Unreachable IP Reporting**: Maintains a list of unreachable IP addresses and serves this list via an HTTP endpoint.

## Usage

**Install Required Packages**: Ensure you have FastAPI and httpx installed.

```
pip install fastapi httpx uvicorn
```

**Run the application**:

```
uvicorn ip_ping_app:app --reload
```

**Accessing the Endpoint**: After starting the server, access the list of unreachable IPs:

```
http://127.0.0.1:8000/unreachable/
```

**Stopping the Application**: To safely stop the application, press Ctrl+C in the terminal running the server.

## Functionality

- **IP Address Ping**: The application pings each IP address using an asynchronous GET request. If the response is successful (status code 200), it is considered reachable; otherwise, the IP is marked as unreachable and stored in a list.
- **Asynchronous Operations**: The `ping_ip` function performs asynchronous pings, allowing multiple IP addresses to be checked simultaneously without blocking the application.
- **Scheduled Pinging**: The `ping_all_ips` function orchestrates continuous pinging of all specified IP addresses in a loop.
- **Graceful Shutdown**: The application registers a signal handler for the SIGINT signal (triggered by Ctrl+C). Upon receiving this signal, it invokes the shutdown procedure to halt the pinging operations cleanly.

## Data Structure

**Unreachable IP List**: The application maintains a list of unreachable IP addresses, accessible via the `/unreachable/` endpoint, returning data in JSON format:

json

```
{
    "unreachable_ips": ["192.168.1.2", "192.168.1.3", ...]
}
```

## Conclusion

The `ip_ping_app.py` FastAPI application effectively demonstrates asynchronous programming and HTTP handling in Python. It serves as a practical tool for network monitoring and provides a foundation for future enhancements and features.

---

# Question 3: Database Operations with SQLAlchemy

## Overview

In this Python script, I demonstrate how to perform bulk Insert, Update, and Delete operations on a SQLite database using SQLAlchemy. The database consists of two tables: users and products.

## Requirements

- Python 3.x
- SQLAlchemy

**I can install SQLAlchemy using pip:**

```
pip install sqlalchemy
```

## Database Schema

The script creates the following two tables:

**Users Table**

- `id`: Integer (Primary Key)
- `name`: String
- `email`: String
- `created_at`: DateTime (Default: Current Timestamp)
- `updated_at`: DateTime (Default: Current Timestamp, Automatically updated on modification)

**Products Table**

- `id`: Integer (Primary Key)
- `name`: String
- `price`: Float
- `user_id`: Integer (Foreign Key referencing Users)
- `created_at`: DateTime (Default: Current Timestamp)

## Functionality

### Database Connection

The script connects to a SQLite database named `example.db`. If this database does not exist, it is created automatically.

### Model Definitions

I define two models using SQLAlchemy's ORM:

- **User**: Represents the users in the database.
- **Product**: Represents the products linked to users.

### Inserting Data

The `insert_data` function inserts a specified number of users and their associated products into the database. It can handle:

- `num_users`: The total number of users to insert (default: 100,000).
- `num_products_per_user`: The number of products per user (default: 2).

I print progress updates every 1,000 users inserted.

### Updating Data

The `update_data` function randomly updates the names of 1,000 users and the prices of 1,000 products in the database, ensuring a mix of updates across both tables.

### Deleting Data

The `delete_data` function randomly deletes 1,000 users and products from the database.

## Main Execution

The script's execution begins with timing the insert, update, and delete operations, printing the elapsed time for each operation.

```
In [1]: runfile('C:/git push/Rakort/q3/sql.py', wdir='C:/git push/Rakort/q3')
Insert operation completed. Time: 1.23 seconds.
Update operation completed. Time: 4.94 seconds.
Delete operation completed. Time: 8.32 seconds.

In [2]:
```

Timings of the requested three operation

### Conclusion

This script efficiently manages user and product data in a SQLite database, showcasing the basic capabilities of SQLAlchemy for performing bulk operations.

---

# Question 4: Process Management in Python

### Overview

I utilize the following libraries and classes in my process management implementation:

- `time`: For handling time-related tasks.
- `random`: To simulate varying task durations.
- `Process`: Class from multiprocessing to create and manage processes.
- `create_engine`, `Column`, `Integer`, `String`, `DateTime`: Components from SQLAlchemy for database operations.
- `declarative_base`: To define a base class for SQLAlchemy models.
- `sessionmaker`: For creating a new session for database transactions.
- `func`: Provides access to SQL functions for use in queries.

### Database Connection

I establish a connection to a SQLite database named `process_states.db`:

```
DATABASE_URL = "sqlite:///process_states.db"
engine = create_engine(DATABASE_URL, echo=True)
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
```

### Process Class Definition

I define a class named `ProcessState` to track the state of different processes in the database:

```
class ProcessState(Base):
    __tablename__ = "process_states"

    id = Column(Integer, primary_key=True)
    name = Column(String)
    state = Column(String)
    start_time = Column(DateTime)
    end_time = Column(DateTime)
```

This class includes fields for process identification, state, start time, and end time.

### Asynchronous Task Management

I implement a function `run_process` to simulate a process. It sleeps for a random duration, updates the process state in the database, and handles transitions between states:

```
def run_process(name):
    start_time = datetime.now()
    # Create a new session
    session = Session()

    # Record the process state as 'running'
    process_state = ProcessState(name=name, state='running',
start_time=start_time)
    session.add(process_state)
    session.commit()

    # Simulate the process doing work
    time.sleep(random.uniform(1, 5))

    # Update the process state to 'completed'
    process_state.state = 'completed'
    process_state.end_time = datetime.now()
```

```
    session.commit()
    session.close()
```

This function records the process state in the database, simulates work using `time.sleep`, and finally updates the process state to 'completed'.

### Executing Processes

I create and start multiple processes to demonstrate the functionality:

```
if __name__ == '__main__':
    processes = []
    for i in range(5):
        process = Process(target=run_process,
args=(f'Process-{i+1}',))
        processes.append(process)
        process.start()

    for process in processes:
        process.join()
```

This code block initializes and runs five processes, each managed by the `run_process` function.

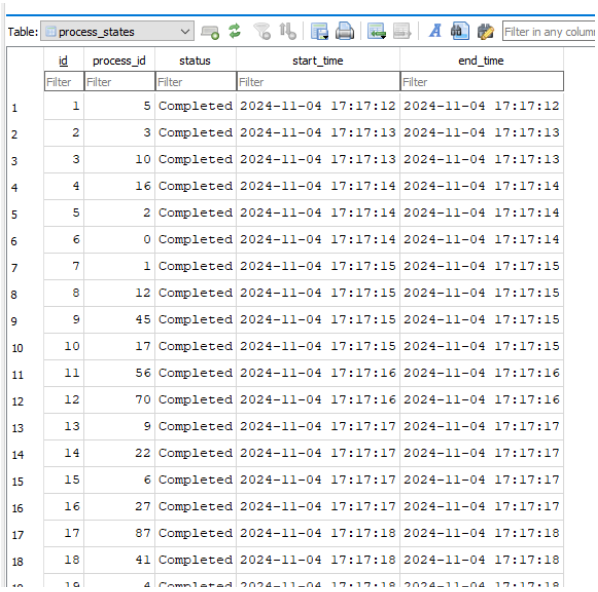

Table: process_states

| | id | process_id | status | start_time | end_time |
|---|---|---|---|---|---|
| | Filter | Filter | Filter | Filter | Filter |
| 1 | 1 | 5 | Completed | 2024-11-04 17:17:12 | 2024-11-04 17:17:12 |
| 2 | 2 | 3 | Completed | 2024-11-04 17:17:13 | 2024-11-04 17:17:13 |
| 3 | 3 | 10 | Completed | 2024-11-04 17:17:13 | 2024-11-04 17:17:13 |
| 4 | 4 | 16 | Completed | 2024-11-04 17:17:14 | 2024-11-04 17:17:14 |
| 5 | 5 | 2 | Completed | 2024-11-04 17:17:14 | 2024-11-04 17:17:14 |
| 6 | 6 | 0 | Completed | 2024-11-04 17:17:14 | 2024-11-04 17:17:14 |
| 7 | 7 | 1 | Completed | 2024-11-04 17:17:15 | 2024-11-04 17:17:15 |
| 8 | 8 | 12 | Completed | 2024-11-04 17:17:15 | 2024-11-04 17:17:15 |
| 9 | 9 | 45 | Completed | 2024-11-04 17:17:15 | 2024-11-04 17:17:15 |
| 10 | 10 | 17 | Completed | 2024-11-04 17:17:15 | 2024-11-04 17:17:15 |
| 11 | 11 | 56 | Completed | 2024-11-04 17:17:16 | 2024-11-04 17:17:16 |
| 12 | 12 | 70 | Completed | 2024-11-04 17:17:16 | 2024-11-04 17:17:16 |
| 13 | 13 | 9 | Completed | 2024-11-04 17:17:17 | 2024-11-04 17:17:17 |
| 14 | 14 | 22 | Completed | 2024-11-04 17:17:17 | 2024-11-04 17:17:17 |
| 15 | 15 | 6 | Completed | 2024-11-04 17:17:17 | 2024-11-04 17:17:17 |
| 16 | 16 | 27 | Completed | 2024-11-04 17:17:17 | 2024-11-04 17:17:17 |
| 17 | 17 | 87 | Completed | 2024-11-04 17:17:18 | 2024-11-04 17:17:18 |
| 18 | 18 | 41 | Completed | 2024-11-04 17:17:18 | 2024-11-04 17:17:18 |

Output Database Example

### Conclusion

The implementation showcases basic process management and state tracking using Python's multiprocessing capabilities and SQLAlchemy for database integration. The use of asynchronous tasks and database operations allows for effective monitoring and management of multiple processes concurrently.

---

# Question 5: SSH Connection Management

In this section, I introduce a Python class designed to establish SSH connections to remote Linux servers, allowing for the execution of commands or configurations using the Paramiko library. This class also supports handling multiple SSH connections simultaneously for a specified network block.

### Prerequisites

Before running the code, ensure that the Paramiko library is installed in your environment by executing the following command:

```
pip install paramiko
```

## Code Explanation

### Imports

- **paramiko**: Used for establishing SSH connections to remote servers.
- **threading**: Facilitates the concurrent execution of commands across multiple servers.
- **ipaddress**: Assists in managing IP addresses and network blocks.

### SSHManager Class

- `__init__` **Method**: Initializes the `SSHManager` class with parameters such as:
  - **network block**: Defines the range of IP addresses to connect to.
  - **username**: The username for SSH authentication.
  - **password**: The password for SSH authentication.
- `execute_command` **Method**: Connects to a specified server and executes the provided command, capturing and printing both the output and any errors encountered.
- `run_commands` **Method**: Iterates through each host in the defined network block, creating a separate thread for each server to initiate command execution concurrently. This allows for efficient management of multiple connections.

## Main Execution

To demonstrate the functionality of the `SSHManager` class, the user must specify the network block, along with the username, password, and the command to be executed. It is essential to replace the placeholders with valid credentials and the appropriate command for each server. Additionally, care should be taken to verify that the specified network block is valid to prevent any unintended connections.

# Question 6: Code Documentation for DHCPParser

## Imports

The code imports the following libraries:

- **paramiko**: Used for establishing SSH connections to remote servers.
- **time**: Facilitates time-related functions, such as handling durations for command execution.
- **re**: Provides support for regular expressions, which are essential for parsing text data.

## DHCPParser Class

The `DHCPParser` class is designed to handle the parsing of DHCP requests efficiently.

### `__init__` Method

This constructor method initializes the object with essential parameters, including:

- **server details**: Information required to connect to the SSH server.
- **command**: The specific command to execute on the server for retrieving DHCP data.

### `connect_ssh` Method

This method establishes an SSH connection to the server using the parameters provided during initialization. It manages authentication and ensures a secure connection.

### `run_command` Method

This method executes the specified command on the server and collects DHCP data for a defined duration. It handles the command execution process and retrieves output for further analysis.

### `parse_line` Method

The `parse_line` method processes each line of output from the command execution, extracting DHCP request details using regular expressions. This parsing is crucial for obtaining structured data from the raw command output.

### `save_results` Method

This method saves the parsed DHCP request results into a specified text file, allowing for easy access and analysis of the data later.

## Main Block

In the main block of the code, I instantiate the `DHCPParser` class with the specified server parameters and commands. After execution, the parsed results are saved to a text file. Make sure to replace the placeholders in the **HOST**, **USERNAME**, **PASSWORD**, and **COMMAND** variables with actual connection details to ensure proper functionality.