Port and Network Scanner - Project Documentation

Introduction

We set out to develop a port and network scanner using Python, a common tool for probing network security and troubleshooting connectivity issues. The primary goal of this project is to create a tool that can scan open ports on a given host.

Our initial inspiration for this project came from a script we found on GeeksforGeeks which provided a simple yet functional implementation of a port scanner. After finding this script via a Google search, we decided to use it as a base and make significant modifications to improve its functionality, scalability, and user experience. The changes we made include adding network scanning, parallelizing port scanning, and improving overall performance.

System Overview

The port and network scanner provides two primary functionalities:
1. **Port Scanning** : Scans a target host for open ports by attempting connections on a range of ports (1-65,535).
2. **Network Scanning** : Pings each IP address in a given subnet to identify active hosts on the network.

Scope and Context

This tool is designed for individuals or teams that need a basic network utility for discovering open ports on hosts or detecting active devices on a network. It is a straightforward utility meant to offer insights into network activity and is an excellent starting point for anyone learning about network security or network engineering.

While functional, this project is not intended for large-scale or high-performance scanning tasks typically required in enterprise-grade tools. Instead, it offers a simplified version that provides essential functionality with a focus on educational value.

Architectural Design

The tool is built using Python's standard libraries, such as `*socket*` for network communication, `*ipaddress*` for managing IP ranges, and `*subprocess*` for executing system commands like `*ping*`. The overall structure is as follows:

- **Argument Parsing** : We use the `*argparse*` library to handle command-line input and distinguish between the two main modes of operation.
- **Port Scanning** : We utilize the `*socket*` library to attempt connections to a target host's ports, checking if they are open.
- **Network Scanning** : For network scans, we use `*subprocess*` to call the `*ping*` command and detect active devices in a local subnet.
- **Concurrency** : To improve performance, especially for the port scan, we implemented multithreading using `*ThreadPoolExecutor*` from the `*concurrent.futures*` library, which speeds up the scan by running multiple port checks in parallel.

Detailed Design

- Port Scanning

In the port scanning mode, the user specifies a target IP address. The script then iterates through a range of ports (1 to 65,535), attempting to open a connection to each port using the `socket` module. If a port is open (i.e., the connection is successful), the tool prints the port number as open. To speed up the process, we introduced multithreading, allowing multiple port checks to run concurrently.

- Network Scanning

For network scanning, the user provides a subnet (e.g., `192.168.1.0/24`). The script generates a list of possible IP addresses within this range and uses the `ping` command to check if each host is active. If a host responds to the ping, it is marked as active. Given the sequential nature of the `ping` command, this process can be slow for large subnets, so we also implemented multithreading here to ping multiple hosts simultaneously.

User Interface Design

The scanner operates entirely via the command line. Users interact with it by passing arguments for the desired mode (`host` or `network`) and the target (an IP address or a network range). The scanner provides real-time feedback on the progress of the scan, such as which ports are open or which hosts are active.

Example command to scan a single host:

```
python scanner.py host 192.168.1.1
```

Example command to scan a network:

```
python scanner.py network 192.168.1.0/24
```

Error Handling and Recovery

We have implemented basic error handling for various potential issues:
- **Invalid IP Addresses** : If the user provides an invalid IP address or network format, the program will exit gracefully with an error message.
- **Network Unavailability** : If the host or network is unreachable, appropriate error messages will be shown (e.g., "Hostname could not be resolved" or "Server not responding").
- **User Interruptions** : The script catches keyboard interrupts (`Ctrl + C`) and exits cleanly without leaving open socket connections.

Dependencies

This project relies only on Python's standard libraries, ensuring that no external dependencies are required. The key libraries used are:
- `*socket*` for network communication.
- `*ipaddress*` for IP and subnet handling.
- `*subprocess*` for calling system commands.
- `*argparse*` for parsing command-line arguments.
- `*concurrent.futures*` for multithreading.

Ethical Considerations

Port scanning and network scanning, while useful for network troubleshooting and security testing, can have ethical and legal implications. Scanning a network or host without permission is often considered a violation of network policies and, in many cases, illegal.

Compliance with Ethical Guidelines

For this project, we have adhered to the following ethical considerations:

- **Educational Purpose**: The scanner was developed with the explicit goal of understanding network scanning techniques and educating ourselves on network security.

- **Responsible Use**: We do not encourage or condone the use of this tool for unauthorized scanning of public or private networks. Users of this scanner must ensure they have explicit permission from network owners or administrators before performing scans.

Key Changes and Improvements

- Original Script

The original script we started with from GeeksforGeeks was a basic port scanner that:
- Took a hostname as input and translated it to an IP address.
- Scanned all ports in a sequential manner (one at a time).
- Used simple error handling but lacked flexibility and performance optimizations.

- Our Modifications

We extended this initial script significantly to include:
1. **Network Scanning** : Added the ability to scan an entire subnet to discover active hosts using the `*ping*` command.
2. **Multithreading** : Enhanced performance by adding concurrent port scanning and network pinging using `ThreadPoolExecutor`.
3. **Improved User Input Handling** : We implemented `argparse` to handle command-line arguments more effectively, allowing users to switch between host and network scanning modes.
4. **Timeout Adjustments** : Optimized the connection timeout to make the scanner faster and more responsive.
5. **Error Handling** : Introduced more robust error handling, covering cases such as invalid IP addresses, unreachable hosts, and keyboard interrupts.

Conclusion

Our port and network scanner provides essential functionality for identifying open ports on hosts and detecting active devices on local networks. Starting from a basic port scanner, we extended its capabilities, improved performance, and made it more user-friendly. This project has been a valuable exercise in network programming, and while it remains a simple utility, it offers flexibility for further enhancement and serves as a strong foundation for learning more about network security.

We hope this tool can be a useful resource for anyone looking to understand the basics of port scanning and network discovery. Going forward, possible improvements could include more advanced error handling, the ability to scan specific port ranges, and better support for scanning over different network protocols.