

**Đại Học Quốc Gia Thành Phố Hồ Chí Minh**  
**Trường Đại Học Khoa Học Tự Nhiên**



Khoa: Công Nghệ Thông Tin  
Môn: Mạng Máy Tính  
HK I - Năm Học 2024-2025

# **BÁO CÁO ĐỒ ÁN THỰC HÀNH**

## **CHƯƠNG TRÌNH ĐIỀU KHIỂN MÁY TÍNH TỪ XA**

### **Nhóm 5**

Lâm Trọng Nghĩa	22810209
Nguyễn Hải Âu	24880003
Nguyễn Hồ Quỳnh Ngân	24880042
Đặng Thị Phương Nguyên	22880106
Nguyễn Diệu Thiện	22810215

# Nội Dung

<b>Nội Dung</b>	<b>2</b>
<b>Tổng Quan</b>	<b>3</b>
1. Giới Thiệu Nhóm	3
2. Giới Thiệu Chương Trình	3
3. Video Demo Chương Trình	4
4. Yêu Cầu Của Chương Trình	4
5. Các Thư Viện Sử Dụng	5
<b>Chức Năng Chương Trình</b>	<b>6</b>
1. Chức Năng Kết Nối và Giao Tiếp	6
2. Chức Năng Quản Lý Ứng Dụng (Application)	18
3. Chức Năng Quản Lý Process	33
4. Chức Năng Keystroke	48
4. Chức Năng Screenshot	52
5. Chức Năng Quản Lý File	55
5. Chức Năng Shutdown và Reset	70

# Tổng Quan

## 1. Giới Thiệu Nhóm

Nhóm bao gồm 5 thành viên như sau:

Nhóm	MSSV	Họ	Tên	Địa chỉ thư điện tử
5	22810209	Lâm Trọng	Nghĩa	22810209@student.hcmus.edu.vn
	24880003	Nguyễn Hải	Âu	24880003@student.hcmus.edu.vn
	24880042	Nguyễn Hồ Quỳnh	Ngân	24880042@student.hcmus.edu.vn
	22880106	Đặng Thị Phương	Nguyên	22880106@student.hcmus.edu.vn
	22810215	Nguyễn Diệu	Thiện	22810215@student.hcmus.edu.vn

Vai trò và đóng góp của các thành viên trong đồ án:

Thành Viên	MSSV	Vai Trò	Đóng Góp (%)
Lâm Trọng Nghĩa	22810209	- Kiểm thử - Báo cáo (Files)	20
Nguyễn Hải Âu	24880003	- Kiểm thử - Báo cáo (Shutdown/Reset)	20
Nguyễn Hồ Quỳnh Ngân	24880042	- Kiểm thử - Báo cáo (Keystroke, Screenshot) - Làm video	20
Đặng Thị Phương Nguyên	22880106	- Lập trình - Báo cáo (Connection, Tổng Quan, trình bày)	20
Nguyễn Diệu Thiện	22810215	- Lập trình - Báo cáo (Apps, Process)	20

## 2. Giới Thiệu Chương Trình

Chương trình điều khiển từ xa (remote control) được thiết kế nhằm hỗ trợ việc quản lý và điều khiển máy tính từ xa một cách hiệu quả. Chương trình được phát triển trên ngôn ngữ Python. Hệ thống bao gồm hai thành phần chính:

1. Trình **SERVER**: chạy trên máy bị điều khiển
2. Trình **CLIENT**: dùng để điều khiển máy đang chạy trình SERVER

Nhóm đã phát triển được các tính năng bao gồm:

- Quản lý các ứng dụng (application) đang chạy trên máy SERVER (liệt kê, khởi động, hoặc dừng).
- Quản lý các processes trên máy SERVER (liệt kê, khởi động, hoặc dừng).
- Thực hiện tắt nguồn (shutdown) hoặc khởi động lại (reset) máy SERVER.
- Xem màn hình hiện thời của máy SERVER (chụp màn hình và lưu lại).
- Thực hiện khóa bàn phím và bắt phím nhấn (keylogger) trên máy SERVER.
- Xóa hoặc sao chép file từ máy SERVER.

Với các chức năng đa dạng trên, chương trình này là công cụ hữu ích cho việc quản lý từ xa, đặc biệt trong các môi trường cần giám sát và điều hành nhiều máy tính cùng lúc.

### 3. Video Demo Chương Trình

Link của video demo của nhóm có thể coi ở đây:

<https://youtu.be/ERiBeSsTZ4Q>

### 4. Yêu Cầu Của Chương Trình

Để chạy được chương trình, các yêu cầu cần thiết là:

- Cài đặt Python3, pip3
- Cài đặt các thư viện có trong file Requirements.txt
- Máy bị điều khiển (server) chỉ hỗ trợ hệ điều hành Windows. Máy điều khiển (client) và máy bị điều khiển (server) trong cùng một mạng LAN.

Hướng dẫn cách khởi động chương trình:

- Cài đặt các requirements ở terminal (chỉ làm ở *lần chạy đầu tiên*)

```
pip3 install -r requirements.txt
```

- Đặt folder Server trên máy bị điều khiển. Chạy file `main.py`

```
cd path/to/folder/Server
python3 main.py
```

- Đặt folder Client trên máy điều khiển. Chạy file `main.py`

```
cd path/to/folder/Client
python3 main.py
```

- Nhập địa chỉ IP của máy bị điều khiển để kết nối. Port mặc định Server mở là 8080 (có thể cấu hình).

## 5. Các Thư Viện Sử Dụng

Thư viện	Mục đích	Chức năng cụ thể
socket	Kết nối mạng	<ul style="list-style-type: none"> <li>- Tạo và quản lý kết nối TCP</li> <li>- Truyền nhận dữ liệu giữa client-server</li> <li>- Xử lý timeout và đóng kết nối</li> </ul>
json	Xử lý dữ liệu	<ul style="list-style-type: none"> <li>- Chuyển đổi dữ liệu thành JSON</li> <li>- Format dữ liệu gửi/nhận</li> </ul>
threading	Xử lý đa luồng	<ul style="list-style-type: none"> <li>- Tạo thread riêng cho server</li> <li>- Xử lý nhiều kết nối đồng thời</li> </ul>
PyQt5	Giao diện người dùng	<ul style="list-style-type: none"> <li>- Tạo cửa sổ và dialog</li> <li>- Hiển thị thông báo và cảnh báo</li> <li>- Xử lý sự kiện người dùng</li> <li>- Tạo bảng và hiển thị dữ liệu</li> </ul>
os	Thao tác hệ thống	<ul style="list-style-type: none"> <li>- Xử lý file và thư mục</li> <li>- Thực thi lệnh hệ thống</li> <li>- Quản lý path</li> </ul>
subprocess	Xử lý process	<ul style="list-style-type: none"> <li>- Chạy và quản lý process</li> <li>- Thực thi lệnh shell</li> <li>- Lấy output từ command</li> </ul>

Thư viện	Mục đích	Chức năng cụ thể
PIL (Pillow)	Xử lý ảnh	- Chụp màn hình - Xử lý và lưu ảnh
pynput	Ghi nhận phím	- Theo dõi keyboard input - Keylogger - Xử lý keyboard events
logging	Ghi log	- Log thông tin hoạt động - Log lỗi và cảnh báo - Debug và troubleshoot
datetime	Xử lý thời gian	- Tính toán và ghi nhận thời gian - Format thời gian
typing	Type hints	- Khai báo kiểu dữ liệu - Kiểm tra typing
time	Xử lý thời gian	- Thực hiện delay và timeout - Ghi nhận thời gian
sys	Xử lý lệnh hệ thống	- Xử lý command line
collections	Cấu trúc dữ liệu	- Sử dụng cho buffer - Quản lý dữ liệu
platform	Thông tin hệ thống	- Kiểm tra hệ điều hành - Thực hiện lệnh của hệ điều hành
signal	Xử lý signal	- Xử lý giao tiếp giữa system - Tắt máy tính

## Chức Năng Chương Trình

### 1. Chức Năng Kết Nối và Giao Tiếp

#### a. Tổng Quan Về Cơ Chế

##### 1. Các Class Chính

Class	Thiết bị	Chức năng chính
RemoteControlServer	Server	- Quản lý hoạt động của server - Xử lý kết nối từ client (chỉ dùng TCP)

Class	Thiết bị	Chức năng chính
		- Điều phối các lệnh và phản hồi
NetworkManager	Client	- Quản lý kết nối tới server (chỉ dùng TCP) - Xử lý gửi/nhận dữ liệu - Quản lý các socket connection

## 2. Các Phương Thức của Server

Phương thức	Loại	Chức năng
<code>__init__(host, port)</code>	Khởi tạo	- Khởi tạo server với host và port - Thiết lập môi trường
<code>main()</code>	Kết nối	- Tạo và quản lý main/basic socket - Lắng nghe kết nối từ client
<code>handle_connection(conn, addr, is_main)</code>	Xử lý kết nối	- Xử lý từng kết nối client - Phân luồng xử lý main/basic socket
<code>toggle_server():</code>	Xử lý kết nối	- Kiểm soát vòng đời của server - Quản lý tài nguyên hệ thống - Kiểm tra thread kết nối
<code>cleanup()</code>	Đóng kết nối	- Xóa tài nguyên - Đóng các socket
<code>shutdown()</code>	Đóng kết nối	- Tắt server - Giải phóng tài nguyên

## 3. Các Phương Thức của Client

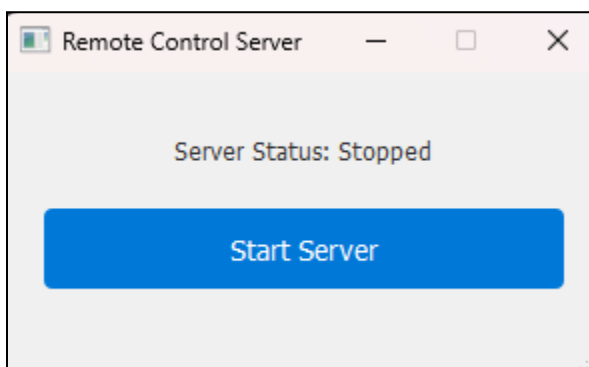
Phương thức	Loại	Chức năng
<code>__init__()</code>	Khởi tạo	- Khởi tạo socket và biến môi trường - Thiết lập timeout (300s) và buffer (4096 bytes, có thể thay đổi tùy nhu cầu của chức năng)

Phương thức	Loại	Chức năng
connect(host, port)	Kết nối	- Tạo kết nối tới server - Thiết lập main và basic socket
send_message(msg)	Gửi/Nhận	- Gửi lệnh tới server - Tự động chọn socket phù hợp
_send_main_message(msg)	Gửi/Nhận	- Gửi lệnh qua main socket (dùng cho process, app)
_send_basic_message(msg)	Gửi/Nhận	- Gửi lệnh qua basic socket (dùng cho file, keylog)
receive_file(filename)	File	- Nhận và lưu file từ server
disconnect()	Đóng kết nối	- Ngắt kết nối với server - Gửi lệnh quit
cleanup()	Đóng kết nối	- Dọn dẹp tài nguyên - Đóng các socket

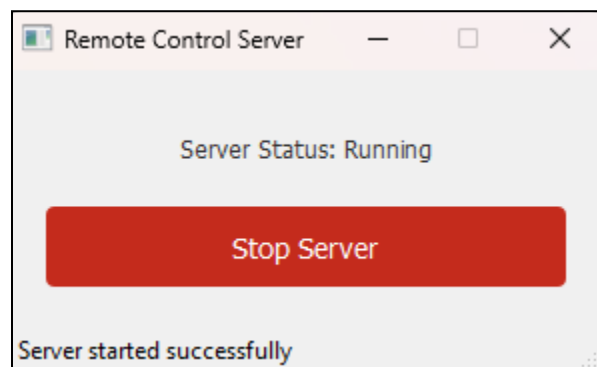
## b. Giao Diện

Ở phía Server:

*Trạng thái chưa khởi động server*

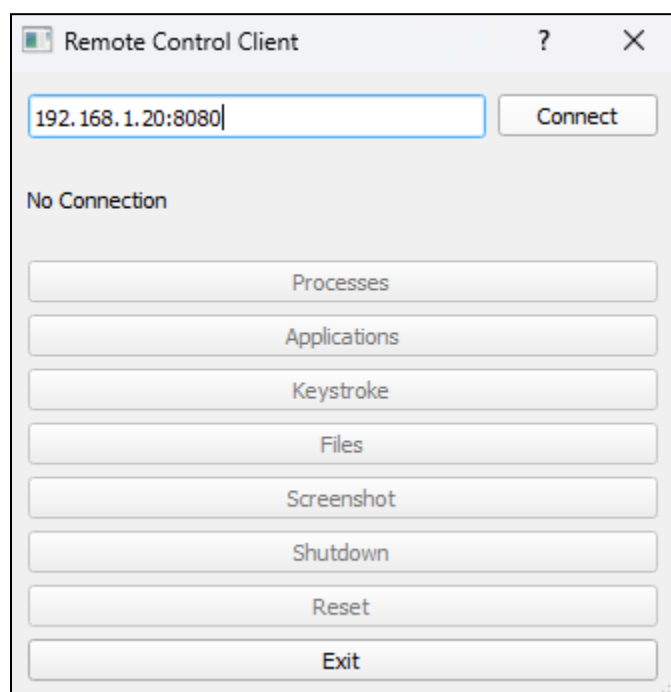


*Trạng thái đã khởi động server*



Ở phía Client:





## c. Cách Hoạt Động Chính

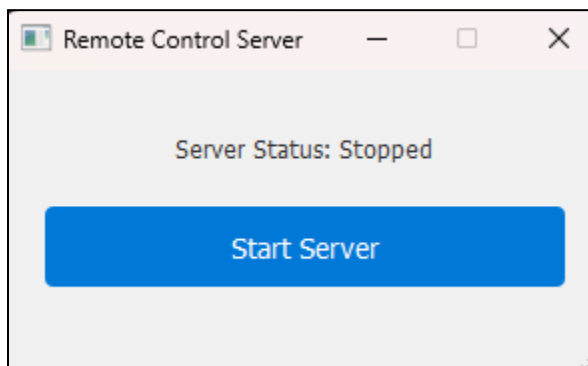
### Chức Năng 1. Khởi động Server

1. Khi người dùng click nút *Start Server*, hàm *toggle\_server* được gọi và khởi động server. Hàm *toggle\_server()* sẽ thực hiện các lệnh sau:

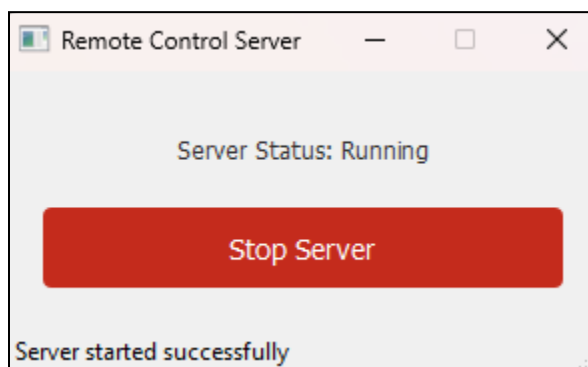
```
def toggle_server(self):
    if not self.server:
        try:
            self.server = RemoteControlServer()
            self.server_thread = threading.Thread(target=self.server.main)
            self.server_thread.daemon = True
            self.server_thread.start()
            self.updateStatus(True)
            self.statusbar.showMessage("Server started successfully")
        except Exception as e:
            QMessageBox.critical(self, "Error", f"Failed to start server: {str(e)}")
            self.updateStatus(False)
    else:
        try:
            self.server.cleanup()
            self.server = None
            self.updateStatus(False)
            self.statusbar.showMessage("Server stopped successfully")
        except Exception as e:
            QMessageBox.critical(self, "Error", f"Failed to stop server: {str(e)}")
```

- Kiểm tra nếu server chưa chạy (if not self.server)
- Tạo instance mới của RemoteControlServer
- Tạo thread mới để chạy server.main()
- Set thread là daemon (sẽ tự động kết thúc khi chương trình chính kết thúc)
- Khởi động thread
- Cập nhật UI (chuyển nút Start Server thành Stop Server)

*Trạng thái chưa khởi động server:*



*Trạng thái đã khởi động server:*



2. Server tạo và bind hai socket:
  - Main socket trên port 8080
  - Basic socket trên port 8081 (8080 + 1)
3. Cả hai socket đều được cấu hình để lắng nghe kết nối, sử dụng protocol TCP (`socket.SOCK_STREAM`):

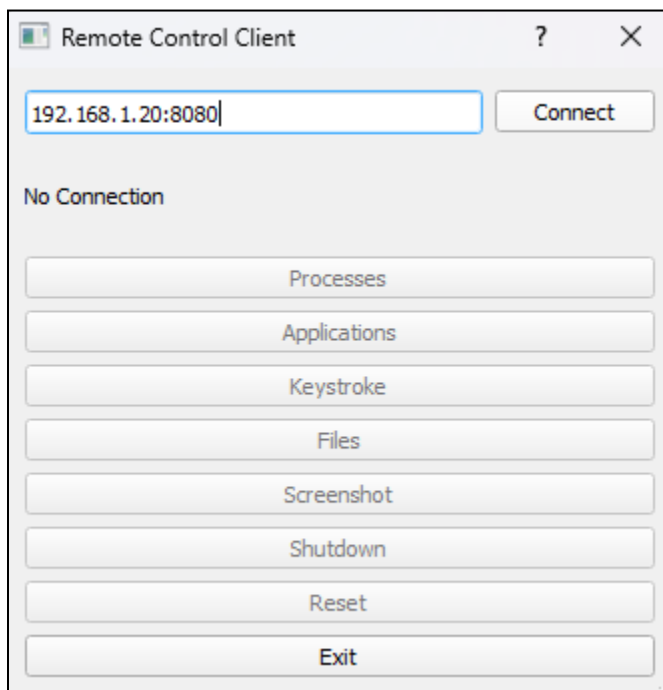
```
# Create main socket for process/app features
self.main_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.main_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
self.main_socket.bind((self.HOST, self.PORT))
self.main_socket.listen(1)

# Create basic socket for other features
self.basic_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.basic_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
self.basic_socket.bind((self.HOST, self.PORT + 1))
self.basic_socket.listen(1)
```

## Chức Năng 2. Kết nối từ Client

1. Khi người dùng nhập địa chỉ và click nút *Connect*, client thực hiện hàm *connect()* trong class *NetworkManager*:

*Trạng thái chưa kết nối:*



Hàm *connect()* thực hiện các lệnh chính sau:

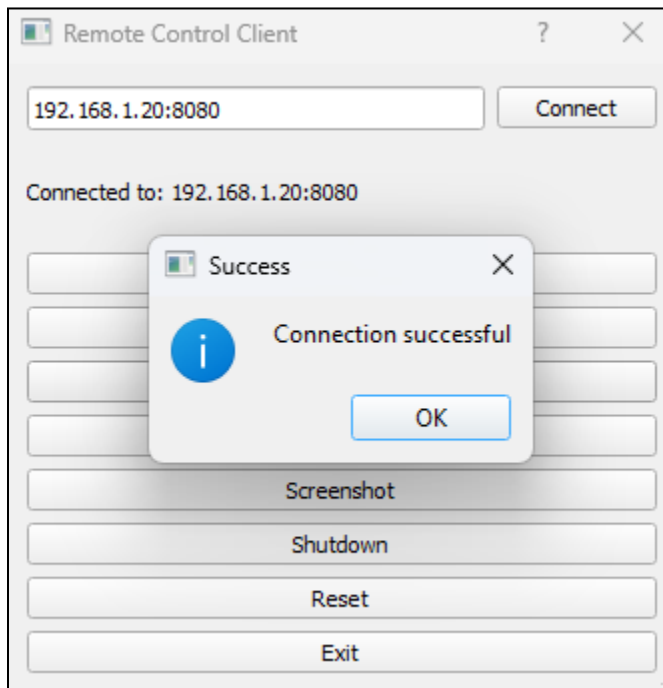
```
def connect(self, host: str, port: int) -> bool:
    try:
        if self.socket:
            self.socket.close()

        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.settimeout(self.timeout) # Set timeout for connection
        self.socket.connect((host, port))
        self.connected = True
        return True
    except Exception as e:
        self.connected = False
        if self.socket:
            self.socket.close()
            self.socket = None
        raise ConnectionError(f"Connection failed: {str(e)}")
```

- Tạo kết nối main socket đến port chính

- Tạo kết nối basic socket đến port + 1
- UI được cập nhật (enable các nút chức năng, hiển thị thông báo trạng thái kết nối)

*Trạng thái kết nối thành công:*



### Chức Năng 3. Thiết lập giao tiếp và đóng kết nối

#### 1. Định dạng message

##### 1.1. Main socket (Process/App commands):

- Format: command//action//parameters
- Ví dụ: process//list, app//kill//123
- Phản hồi dạng JSON với status (trạng thái thành công của kết quả) và data (dữ liệu)

```
response = {
    "status": status,
    "data": data
}
response_str = json.dumps(response) + "\n"
conn.sendall(response_str.encode('utf-8'))
```

### 1.2. Basic socket:

- Format đơn giản hơn: command//parameters
- Ví dụ: capture, key//hook
- Phản hồi dạng binary hoặc text đơn giản

## 2. Xử lý lỗi và timeout

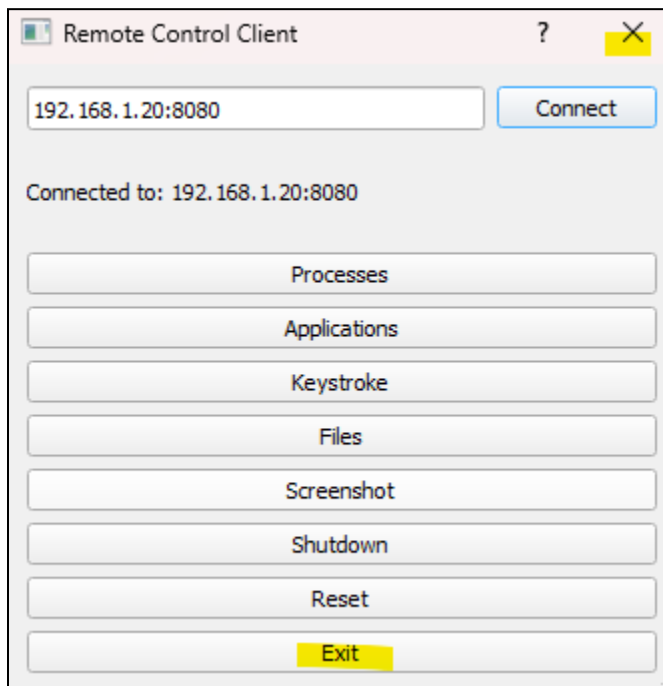
- Timeout mặc định: 300 giây
- Buffer size: 4096 bytes (mặc định, có thể thay đổi theo nhu cầu của từng chức năng)
- Tự động đóng kết nối khi gặp lỗi
- Logging chi tiết các lỗi phát sinh để hỗ trợ debug

## Chức Năng 4. Ngắt kết nối

### 1. Ngắt kết nối từ Client

1.1. Khi người dùng đóng ứng dụng hoặc click Exit.

*Yêu cầu ngắt kết nối bằng cách nhấn X ở góc phải hoặc nhấn Exit:*



Hàm `disconnect()` được gọi:

```
def disconnect(self):  
    """Disconnect from both sockets"""  
    if self.connected:  
        try:  
            if self.main_socket:  
                self.main_socket.sendall(b'quit')  
            if self.basic_socket:  
                self.basic_socket.sendall(b'quit')  
        except:  
            pass  
    finally:  
        self.cleanup()
```

1.2. Các bước thực hiện:

- Gửi lệnh 'quit' đến cả hai socket
- Đóng các socket
- Xóa trạng thái kết nối đã thiết lập

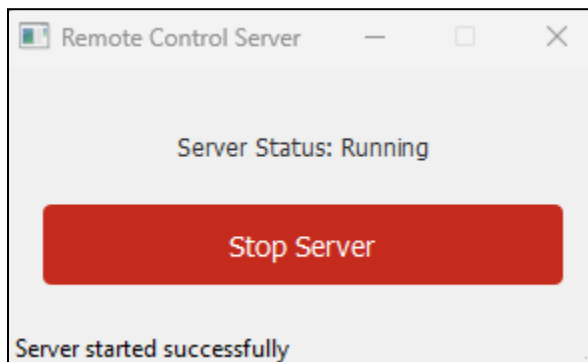
## 2. Ngắt kết nối từ Server

2.1. Nếu server nhận lệnh *quit* từ client thì sẽ phản hồi xác nhận lại cho client:

```
if command == 'quit':  
    logging.info(f"Received quit command from {addr}")  
    # Send message before closing  
    try:  
        if is_main:  
            self.send_formatted_response(conn, "success", "quit")  
        else:  
            conn.sendall(b'ok')  
    except:  
        pass  
    break
```

2.2. Server sẽ dừng bằng cách chạy hàm *cleanup* sau khi nhận lệnh *quit* hoặc sau khi người dùng bấm *Stop Server* hoặc tắt ứng dụng server.

Nhấn *Stop Server* ở server để ngắt kết nối:



Hàm *cleanup()* thực hiện các lệnh chính như sau:



```
def cleanup(self):
    """Clean up all resources"""
    self.running = False

    # Stop keylogger if active
    if self.listener:
        try:
            self.listener.stop()
        except:
            pass
        self.listener = None

    # Clean up main socket
    if self.main_socket:
        try:
            self.main_socket.shutdown(socket.SHUT_RDWR)
        except:
            pass
        finally:
            self.main_socket.close()
            self.main_socket = None

    # Clean up basic socket
    if self.basic_socket:
        try:
            self.basic_socket.shutdown(socket.SHUT_RDWR)
        except:
            pass
        finally:
            self.basic_socket.close()
            self.basic_socket = None
```

- Dừng keylogger nếu đang chạy
- Đóng tất cả các socket
- Giải phóng tài nguyên

## 2. Chức Năng Quản Lý Ứng Dụng (Application)

### a. Tổng Quan Về Cơ Chế

#### 1. Các Phương Thức của Client

Nằm trong Class *Dialong\_app*(*QDialog*, *Ui\_dialog\_app*) trong file *main.py*.

Phương thức	Chức năng
<code>def setup_connections(self)</code>	Các nút thực hiện các mục lớn (Application, Process,...)
<code>def app(self):</code>	Kiểm tra kết nối và mở cửa sổ làm việc cho mục Application
<code>class Dialog_app(QDialog, Ui_dialog_app)</code>	Hiển thị cửa sổ làm việc cho mục Application
<code>def view_apps(self)</code>	Thực hiện request list các applications trên Server và hiển thị các list applications trả về từ phía Server trong cửa sổ làm việc của mục Application
<code>def send_message(self, msg: str) -&gt; str</code>	Chọn socket phù hợp (với application thì chọn port 8080)
<code>def _send_main_message(self, msg: str) -&gt; str</code>	Gửi lệnh đến Server qua socket (có port 8080)
<code>def _receive_main_response(self) -&gt; str</code>	Nhận dữ liệu truyền về từ Server đến Client
<code>def parse_server_response(self, response: str) -&gt; tuple[bool, any]</code>	Kiểm tra dữ liệu truyền về từ Server đến Client
<code>def start_app(self)</code>	Mở cửa sổ làm việc cho mục nhập tên application
<code>class Dialog_start(QDialog, Ui_dialog_start)</code>	Hiển thị cửa sổ làm việc cho mục nhập tên application
<code>def start(self)</code>	Thực hiện request start application trên Server và refresh các list applications

Phương thức	Chức năng
	trong cửa sổ làm việc của mục Application
def kill_app(self)	Mở cửa sổ làm việc cho mục nhập id application
class Dialog_kill(QDialog, Ui_dialog_kill)	Hiển thị cửa sổ làm việc cho mục nhập id application
def kill(self)	Thực hiện request stop application trên Server và refresh các list applications trong cửa sổ làm việc của mục Application

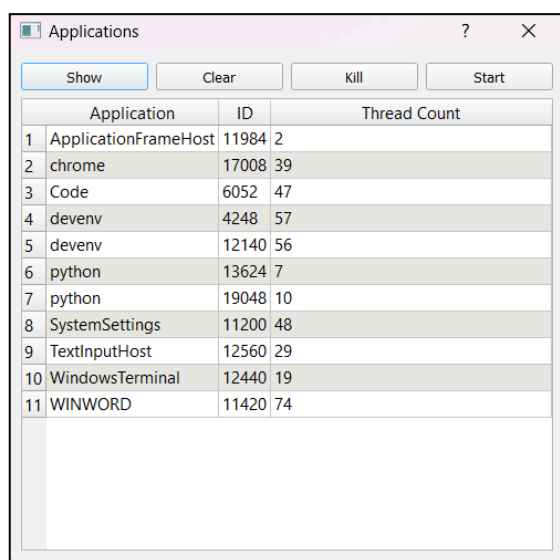
## 2. Các Phương Thức của Server

Nằm trong def *handle\_app\_commands(self, conn: socket.socket, command: str, args: list)* trong file *ps.py*.

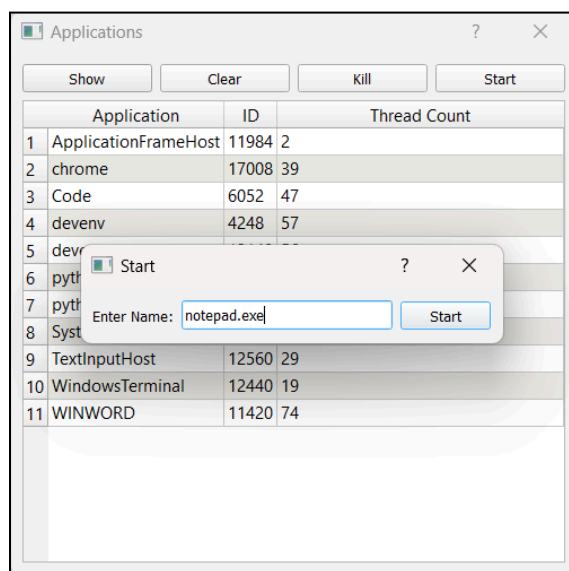
Phương thức	Chức năng
def handle_connection(self, conn: socket.socket, addr: str, is_main: bool)	Xử lý các request từ Client để gọi hàm thực hiện các chức năng tương ứng
def handle_app_commands(self, conn: socket.socket, command: str, args: list)	Thực hiện một trong các thao tác (list, start, stop) applications ở phía Server mà người dùng truyền lệnh thực hiện
def list_apps(self):	Thực hiện list các applications đang chạy ở Server
def send_formatted_response(self, conn: socket.socket, status: str, data: any = None)	Thực hiện format lại response và trả response đã được format về cho Client

## b. Giao Diện

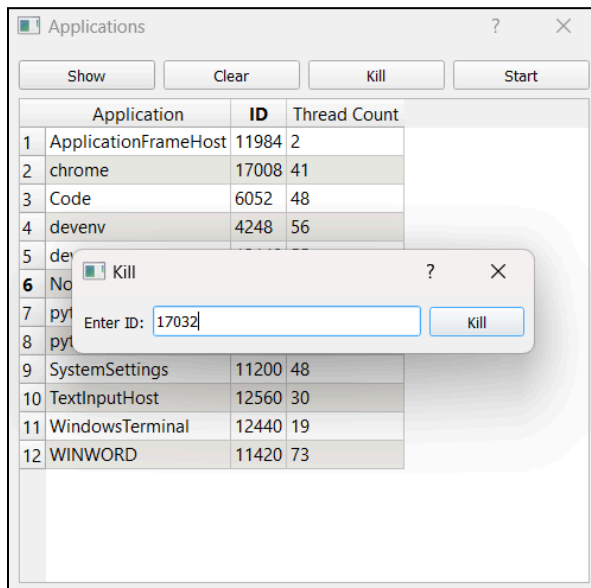
Hiển thị các ứng dụng đang chạy ở Server:



*Bật (start) một ứng dụng nào đó ở Server:*



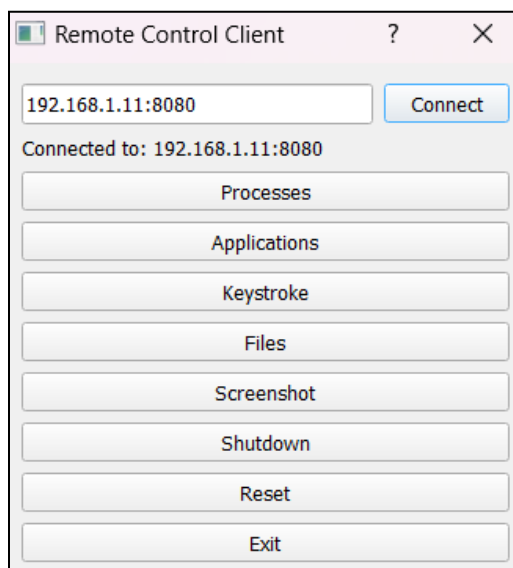
*Tắt (kill) một ứng dụng theo ID ở Server:*



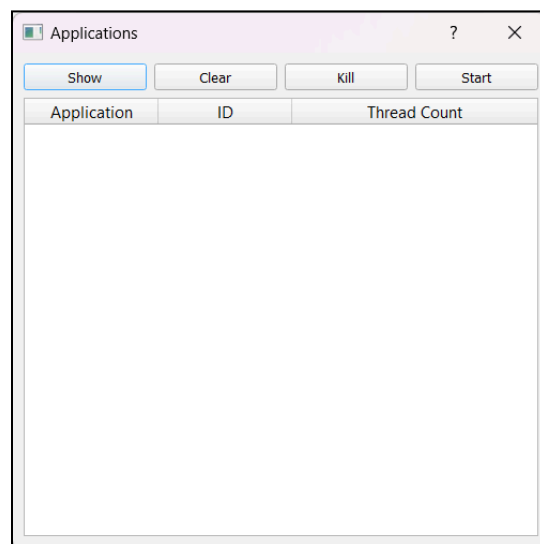
## c. Cách Hoạt Động Chính

### Chức Năng 1. Khởi động Applications

1. Chọn nút bấm **Applications** ở màn hình chính, hàm `def setup_connections(self)` thực hiện lệnh `self.btn_app.clicked.connect(self.app)`, tức khi chọn Applications thì `btn_app` được chỉ thị nhấn chọn và sau khi được nhấn chọn sẽ thực hiện hàm `def app(self)`.



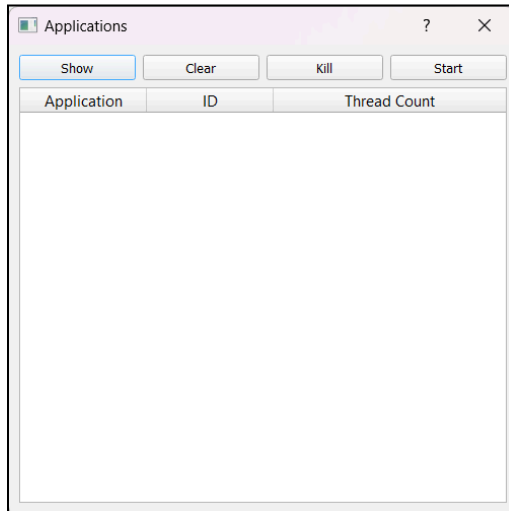
2. Hàm `def app(self)` được thực hiện: Gọi hàm `check_connection()` để kiểm tra hiện trạng kết nối giữa máy Client và Server, nếu kết nối không ổn định thì trả về kết quả báo lỗi bên phía Client, nếu kết nối ổn định thì thực hiện `Dialog_app(self.network, self)` với tham số truyền vào là đối tượng quản lý kết nối mạng giữa Client và Server đang kết nối với nhau.
3. Class `Dialog_app(QDialog, Ui_dialog_app)` được thực hiện với hàm chạy chính là `def __init__(self, network: NetworkManager, parent=None)` bao gồm:
  - `self.setupUi(self)`: Khai báo các đối tượng (nút nhấn, layout,...) giao diện người dùng từ lớp `Ui_dialog_app` trong file `dialog_ui.py`
  - `self.network = network`: Lưu đối tượng `network`
  - `self.setup_ui()`: Thiết lập giao diện hiện sau khi chọn nút Applications
  - `self.connect_signals()`: Khai báo các nút nếu được nhấp chọn sẽ thực hiện các hàm tương ứng để thực hiện list hoặc start hoặc stop các applications đang chạy ở Server



## Chức Năng 2. Hiện (list) các ứng dụng đang chạy ở Server

1. Chọn nút **Show**, hàm `def connect_signals(self)` thực hiện lệnh `self.btn_show.clicked.connect(self.view_apps)`, tức sau khi chọn Show nút

btn\_show được chỉ thị nhấp chọn và sau khi được nhấp chọn sẽ thực hiện hàm `def view_apps(self)`.



2. Hàm `def view_apps(self)` ở phía Client được thực hiện:

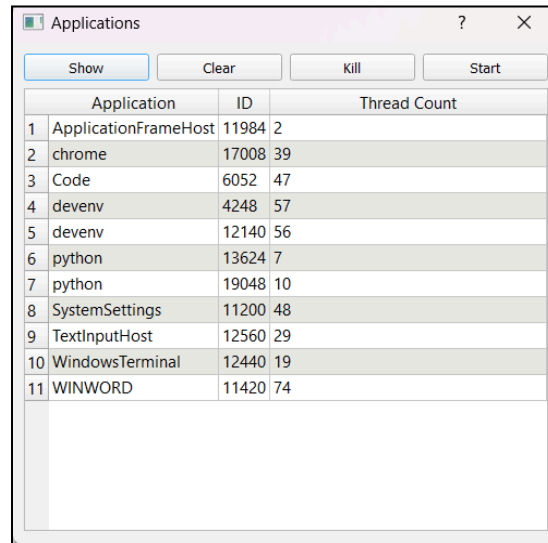
- Thiết lập con trỏ chuột xoay vòng khi đang chờ response từ Server (`QtWidgets.QApplication.setOverrideCursor(Qt.WaitCursor)`)
- Thiết lập thời gian chờ response là 10 giây để tránh tình trạng client ui bị đứng máy quá lâu (`timer = QTimer()` và `timer.singleShot(10000, lambda: QtWidgets.QApplication.restoreOverrideCursor())`)
- Request lên server và chờ nhận response từ server về việc thực hiện chức năng List applications đang chạy ở Server bằng cách gọi hàm `send_message('app//list')` với biến truyền vào "app//list" để nhận diện đang gọi đến server thực hiện chức năng list các applications.
- Hàm `def send_message(self, msg: str) -> str` thực hiện với msg được truyền vào là 'app//list': Kiểm tra kết nối giữa Client và Server, nếu kết nối không ổn định thì báo lỗi, nếu kết nối ổn định thì sẽ phân chia tùy theo chức năng mà sẽ có port thực hiện khác nhau (như process và app sẽ thực hiện ở port 8080 ở hàm `def _send_main_message(self, msg: str) -> str`)
- Hàm `def _send_main_message(self, msg: str) -> str` được thực hiện với msg được truyền vào là 'app//list': Kiểm tra kết nối của socket (`main_socket`), nếu kết nối không còn thì báo lỗi, nếu còn kết nối thì thiết

lập thời gian để request và nhận response từ phía server là 300 (giây) (`self.main_socket.sendall(msg.encode("utf-8"))`). Sau khi thiết lập thời gian thực hiện thì gửi yêu cầu List các applications đang chạy trên Server bằng socket (`self.main_socket.sendall(msg.encode("utf-8"))`) với msg được truyền vào là 'app//list' (Phần Server thực hiện sẽ được đề cập ở phần bên dưới) và nhận các đoạn dữ liệu trả về từ Server qua việc thực hiện hàm `def _receive_main_response(self) -> str` (với các đoạn dữ liệu trả về có size là 4096). Hàm `def _receive_main_response(self) -> str` thực hiện vòng lặp while cho đến khi hết các đoạn dữ liệu (có size 4096) được trả về từ Server, mỗi lần thực hiện vòng lặp thì các đoạn dữ liệu trả về được nối lại với nhau, sau đó hàm trả về dữ liệu hoàn chỉnh và nhận kết quả trả về ở đích đến cuối cùng là response của hàm `def view_apps(self)` (`response = self.network.send_message('app//list')`). Sau khi quay trở lại hàm `def view_apps(self)` và nhận response từ Server, thực hiện hàm `def parse_server_response(self, response: str) -> tuple[bool, any]` (với tham số truyền vào là response của Server) để kiểm tra format response trả về và hàm trả về với 2 giá trị (keyword) là status và data.

- Sau khi nhận kết quả trả về từ hàm `def parse_server_response(self, response: str)` ở hàm `def view_apps(self)` thì gán vào 2 biến success và data (`success, data = self.network.parse_server_response(response)`). Kiểm tra nếu thời gian đã được thiết lập để chờ response trước đó còn hoạt động thì dừng thời gian và cho con trỏ chuột quay về trạng thái ban đầu (`if timer.isActive(): / timer.stop()` / `QtWidgets.QApplication.restoreOverrideCursor()`); Kiểm tra trạng thái thành công (dựa vào biến success), kiểu dữ liệu của data (data phải có dạng dict), và 'app' phải có trong data. Nếu lỗi thì thông báo đến người dùng.
- Khai báo biến apps và gán giá trị là các biến trong data có từ khóa là 'app'.



- In các giá trị (name, pid, threads) trong biến app và hiện ở các hàng với các cột tương ứng (Application, ID, Thread Count).



	Application	ID	Thread Count
1	ApplicationFrameHost	11984	2
2	chrome	17008	39
3	Code	6052	47
4	devenv	4248	57
5	devenv	12140	56
6	python	13624	7
7	python	19048	10
8	SystemSettings	11200	48
9	TextInputHost	12560	29
10	WindowsTerminal	12440	19
11	WINWORD	11420	74

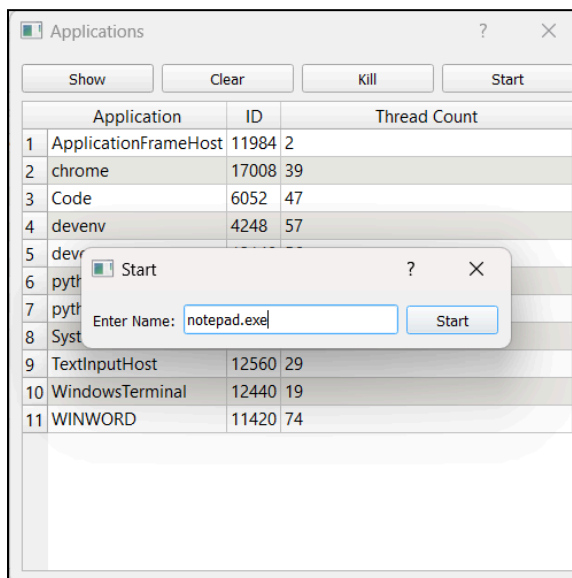
### 3. Phần thực hiện ở Server:

- Sau khi thực hiện hàm `def handle_connection(self, conn: socket.socket, addr: str, is_main: bool)` sẽ điều hướng request được gửi từ Client tương ứng với các hàm thực hiện, ở lệnh truyền vào từ Client là List các applications ('app//list') thì được điều hướng đến thực hiện hàm `def handle_app_commands(self, conn: socket.socket, command: str, args: list)` (với các tham số truyền vào là `conn` : đường truyền (gửi nhận dữ liệu) giữa Client và Server thông qua socket, `command` : chức năng yêu cầu thực hiện) và thực hiện trong mục `command == list` (thực hiện hàm `def list_apps(self)`).
- Hàm `def list_apps(self)` được thực hiện: Sử dụng thư viện `subprocess` để thực hiện lấy các danh sách application có format như csv, thiết lập thời gian thực hiện lệnh là 5 giây và gán kết quả vào biến `output` và `error`. Nếu có error thì ghi vào lịch sử log (`remote_control.log`) và trả về biến dict keyword "app" có danh sách rỗng. Nếu không có error thì push từng dòng giá trị được gán ở biến `output` vào biến dict có keyword là "app" và trả về biến dict đó ở hàm `def handle_app_commands(self, conn: socket.socket, command: str, args: list)` và gán vào biến `app_data`, sau đó thực hiện

hàm `def send_formatted_response(self, conn: socket.socket, status: str, data: any = None)` để format lại dữ liệu trả về và thực hiện trả response về Client qua socket.

### Chức Năng 3. Bật (start) một ứng dụng ở Server

1. Chọn nút **Start**, hàm `def connect_signals(self)` ở class `Dialog_app(QDialog, Ui_dialog_app)` thực hiện lệnh `self.btn_start.clicked.connect(self.start_app)`, tức sau khi chọn Start nút `btn_start` được chỉ thị nhấp chọn và sau khi được nhấp chọn sẽ thực hiện hàm `def start_app(self)` và mở ra hộp cửa sổ làm việc Start (class `Dialog_start(QDialog, Ui_dialog_start)`)

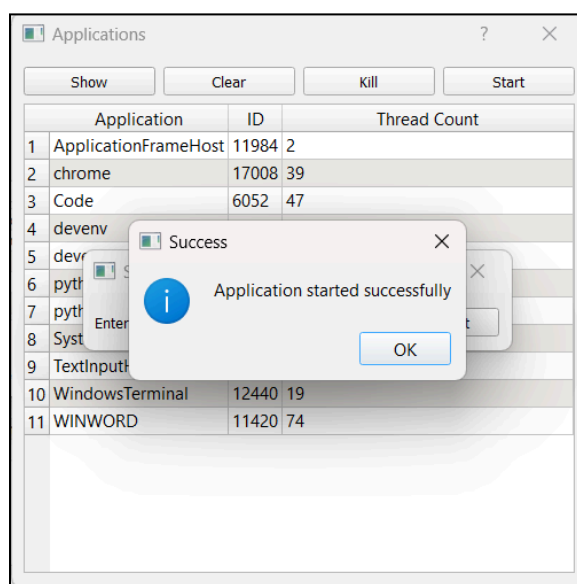


2. Class `Dialog_start(QDialog, Ui_dialog_start)` được thực hiện với hàm chạy chính là `def __init__(self, status: str, network: NetworkManager, parent=None)` bao gồm:
  - `self.setupUi(self)` : Khai báo các đối tượng (nút nhấn, layout,...) giao diện người dùng từ lớp class `Ui_dialog_start(object)` trong file `dialog_ui.py`
  - `self.status = status` : loại chức năng được truyền vào ('app' hoặc 'process')
  - `self.network = network` : Lưu đối tượng network

- `self.btn_start.clicked.connect(self.start)` : sau khi `btn_start` được nhấp chọn thì thực hiện hàm `def start(self)`
3. Hàm `def start(self)` ở class `Dialog_start(QDialog, Ui_dialog_start)` được thực hiện :
- Kiểm tra tên application nhập ở textbox là hợp lệ.
  - Request lên server và chờ nhận response từ server về việc thực hiện chức năng Start application đang chạy ở Server bằng cách gọi hàm `send_message(f'{self.status}//start://{self.lineEdit.text().strip()}')` với biến truyền vào "app//start//tên application" để nhận diện đang gọi đến server thực hiện chức năng start application có tên được Client nhập vào trong textbox.
  - Hàm `def send_message(self, msg: str) -> str` thực hiện với msg được truyền vào là 'app//start//tên application' : Kiểm tra kết nối giữa Client và Server, nếu kết nối không ổn định thì báo lỗi, nếu kết nối ổn định thì sẽ phân chia tùy theo chức năng mà sẽ có port thực hiện khác nhau (như process và app sẽ thực hiện ở port 8080 ở hàm `def _send_main_message(self, msg: str) -> str`.
  - Hàm `def _send_main_message(self, msg: str) -> str` được thực hiện với msg được truyền vào là 'app//start//tên application' : Kiểm tra kết nối của socket (`main_socket`), nếu kết nối không còn thì báo lỗi, nếu còn kết nối thì thiết lập thời gian để request và nhận response từ phía server là 300 (giây). Sau khi thiết lập thời gian thực hiện thì gửi yêu cầu Start application đang chạy trên Server bằng socket (`self.main_socket.sendall(msg.encode("utf-8"))`) với msg được truyền vào là 'app//start//tên application' (Phần Server thực hiện sẽ được đề cập ở phần bên dưới) và nhận các đoạn dữ liệu trả về từ Server qua việc thực hiện hàm `def _receive_main_response(self) -> str` (với các đoạn dữ liệu trả về có size là 4096). Hàm `def _receive_main_response(self) -> str` thực hiện vòng lặp while cho đến khi hết các đoạn dữ liệu (có size 4096) được trả về từ Server, mỗi lần thực hiện vòng lặp thì các đoạn dữ liệu trả

về được nối lại với nhau, sau đó hàm trả về dữ liệu hoàn chỉnh và nhận kết quả trả về ở đích đến cuối cùng là response của hàm `def start(self)` ở class `Dialog_start(QDialog, Ui_dialog_start)`. Sau khi quay trở lại hàm `def start(self)` và nhận response từ Server, thực hiện hàm `def parse_server_response(self, response: str) -> tuple[bool, any]` (với tham số truyền vào là response của Server) để kiểm tra format response trả về và hàm trả về với 2 giá trị (keyword) là status và data.

- Sau khi nhận kết quả trả về từ hàm `def parse_server_response(self, response: str) -> tuple[bool, any]` ở hàm `def start(self)` thì gán vào 2 biến `success` và `data` (`success, data = self.network.parse_server_response(response)`). Kiểm tra trạng thái thành công (dựa vào biến `success`). Nếu không `success` thì thông báo lỗi đến người dùng, nếu `success` thì hiển thị hộp thoại thông báo start thành công và refresh lại parent dialog (gọi lại hàm view tương ứng).



#### 4. Phần thực hiện ở Server

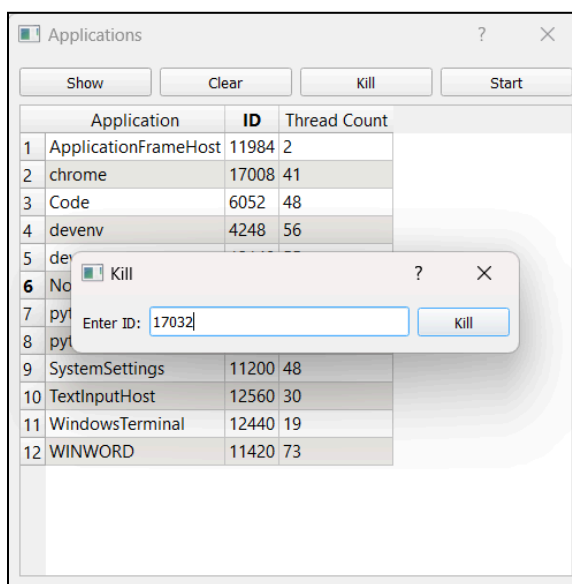
- Sau khi thực hiện hàm `def handle_connection(self, conn: socket.socket, addr: str, is_main: bool)` sẽ điều hướng request được gửi từ Client tương ứng với các hàm thực hiện, ở lệnh truyền vào từ Client là `Start application` ('app//start//tên application') thì được điều hướng đến thực hiện hàm `def`

`handle_app_commands(self, conn: socket.socket, command: str, args: list)` (với các tham số truyền vào là `conn` : đường truyền (gửi nhận dữ liệu) giữa Client và Server thông qua socket, `command` : chức năng yêu cầu thực hiện), `args` : tên application cần start và thực hiện trong mục `command == start`.

- Sử dụng thư viện `subprocess` để thực hiện start application. Sau đó thực hiện hàm `def send_formatted_response(self, conn: socket.socket, status: str, data: any = None)` để format lại dữ liệu trả về và thực hiện trả response về Client qua socket.

#### Chức Năng 4. Dừng (stop) ứng dụng theo ID ở Server

1. Chọn nút **Kill**, hàm `def connect_signals(self)` ở class `Dialog_app(QDialog, Ui_dialog_app)` thực hiện lệnh `self.btn_kill.clicked.connect(self.kill_app)`, tức sau khi chọn Kill nút `btn_kill` được chỉ thị nhấp chọn và sau khi được nhấp chọn sẽ thực hiện hàm `def kill_app(self)` và mở ra hộp cửa sổ làm việc Kill (class `Dialog_kill(QDialog, Ui_dialog_kill)`).



2. Lớp class Dialog\_kill(QDialog, Ui\_dialog\_kill) được thực hiện với hàm chạy tiên khởi là `def __init__(self, status: str, network: NetworkManager, parent=None)` bao gồm:

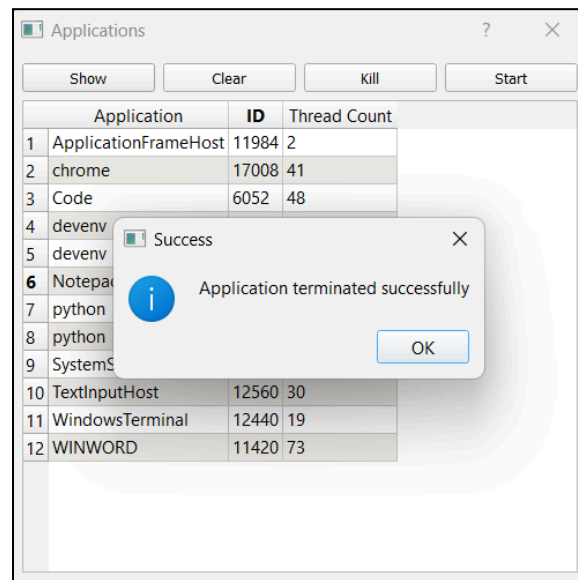
- `self.setupUi(self)` : Khai báo các đối tượng (nút nhấn, layout,..) giao diện người dùng từ lớp class Ui\_dialog\_kill(object) trong file dialog\_ui.py.
- `self.status = status` : loại chức năng được truyền vào ('app' hoặc 'process').
- `self.network = network` : Lưu đối tượng network.
- `self.btn_kill.clicked.connect(self.kill)` : sau khi btn\_kill được nhấp chọn thì thực hiện hàm `def kill(self)`.

3. Hàm `def kill(self)` được thực hiện:

- Kiểm tra id application nhập ở textbox là hợp lệ.
- Request lên server và chờ nhận response từ server về việc thực hiện chức năng Stop application đang chạy ở Server bằng cách gọi hàm `send_message(f'{self.status}//kill//{pid}')` với biến truyền vào "app//kill//id application" để nhận diện đang gọi đến server thực hiện chức năng stop application có id được Client nhập vào trong textbox.
- Hàm `def send_message(self, msg: str) -> str` thực hiện với msg được truyền vào là 'app//kill//id application' : Kiểm tra kết nối giữa Client và Server, nếu kết nối không ổn định thì báo lỗi, nếu kết nối ổn định thì sẽ phân chia tùy theo chức năng mà sẽ có port thực hiện khác nhau (như process và app sẽ thực hiện ở port 8080 ở hàm `def _send_main_message(self, msg: str) -> str`.
- Hàm `def _send_main_message(self, msg: str) -> str` được thực hiện với msg được truyền vào là 'app//kill//id application' : Kiểm tra kết nối của socket (main\_socket), nếu kết nối không còn thì báo lỗi, nếu còn kết nối thì thiết lập thời gian để request và nhận response từ phía server là 300 giây. Sau khi thiết lập thời gian thực hiện thì gửi yêu cầu Stop application đang chạy trên Server bằng socket (`self.main_socket.sendall(msg.encode("utf-8"))`) với msg được truyền vào

là 'app//kill//id application' (Phần Server thực hiện sẽ được đề cập ở phần bên dưới) và nhận các đoạn dữ liệu trả về từ Server qua việc thực hiện hàm `def _receive_main_response(self) -> str` (với các đoạn dữ liệu trả về có size là 4096). Hàm `def _receive_main_response(self) -> str` thực hiện vòng lặp `while` cho đến khi hết các đoạn dữ liệu (có size 4096) được trả về từ Server, mỗi lần thực hiện vòng lặp thì các đoạn dữ liệu trả về được nối lại với nhau, sau đó hàm trả về dữ liệu hoàn chỉnh và nhận kết quả trả về ở đích đến cuối cùng là `response` của hàm `def kill(self)` ở class `Dialog_kill(QDialog, Ui_dialog_start)`. Sau khi quay trở lại hàm `def kill(self)` và nhận `response` từ Server, thực hiện hàm `def parse_server_response(self, response: str) -> tuple[bool, any]` (với tham số truyền vào là `response` của Server) để kiểm tra format `response` trả về và hàm trả về với 2 giá trị (keyword) là `status` và `data`.

- Sau khi nhận kết quả trả về từ hàm `def parse_server_response(self, response: str) -> tuple[bool, any]` ở hàm `def kill(self)` thì gán vào 2 biến `success` và `data` (`success, data = self.network.parse_server_response(response)`). Kiểm tra trạng thái thành công (dựa vào biến `success`). Nếu không `success` thì thông báo lỗi đến người dùng, nếu `success` thì hiển thị hộp thoại thông báo start thành công và refresh lại parent dialog (gọi lại hàm `view` tương ứng).



#### 4. Phần thực hiện ở Server:

- Sau khi thực hiện hàm `def handle_connection(self, conn: socket.socket, addr: str, is_main: bool)` sẽ điều hướng request được gửi từ Client tương ứng với các hàm thực hiện, ở lệnh truyền vào từ Client là Stop application ('app//kill//id application') thì được điều hướng đến thực hiện hàm `def handle_app_commands(self, conn: socket.socket, command: str, args: list)` (với các tham số truyền vào là `conn`: đường truyền (gửi nhận dữ liệu) giữa Client và Server thông qua socket, `command`: chức năng yêu cầu thực hiện), `args`: id application cần start và thực hiện trong mục `command == kill`.
- Sử dụng thư viện `subprocess` để kiểm tra id application có đang trong danh sách các application đang chạy không và thực hiện kill application. Sau đó thực hiện hàm `def send_formatted_response(self, conn: socket.socket, status: str, data: any = None)` để format lại dữ liệu trả về và thực hiện trả response về Client qua socket.



### 3. Chức Năng Quản Lý Process

#### a. Tổng Quan Về Cơ Chế

##### 1. Các Phương Thức của Client

Nằm trong Class *Dialog\_process*(*QDialog*, *Ui\_dialog\_process*) trong file *main.py*.

Phương thức	Chức năng
<code>def setup_connections(self)</code>	Các nút thực hiện các mục lớn (Application, Process,...)
<code>def process(self)</code>	Kiểm tra kết nối và mở cửa sổ làm việc cho mục Process
<code>class Dialog_process(QDialog, Ui_dialog_process)</code>	Hiển thị cửa sổ làm việc cho mục Process
<code>def view_processes(self)</code>	Thực hiện request list các processes trên Server và hiển thị các list processes trả về từ phía Server trong cửa sổ làm việc của mục Process
<code>def send_message(self, msg: str) -&gt; str</code>	Chọn socket phù hợp (với process thì chọn port 8080)
<code>def _send_main_message(self, msg: str) -&gt; str</code>	Gửi lệnh đến Server qua socket (có port 8080)
<code>def _receive_main_response(self) -&gt; str</code>	Nhận dữ liệu truyền về từ Server đến Client
<code>def parse_server_response(self, response: str) -&gt; tuple[bool, any]</code>	Kiểm tra dữ liệu truyền về từ Server đến Client
<code>def start_process(self)</code>	Mở cửa sổ làm việc cho mục nhập tên process
<code>class Dialog_start(QDialog, Ui_dialog_start)</code>	Hiển thị cửa sổ làm việc cho mục nhập tên process
<code>def start(self)</code>	Thực hiện request start process trên Server và refresh các list processes trong cửa sổ làm việc của mục Process
<code>def kill_process(self)</code>	Mở cửa sổ làm việc cho mục nhập id process
<code>class Dialog_kill(QDialog, Ui_dialog_kill)</code>	Hiển thị cửa sổ làm việc cho mục nhập id process

Phương thức	Chức năng
def kill(self)	Thực hiện request stop process trên Server và refresh các list processes trong cửa sổ làm việc của mục Process

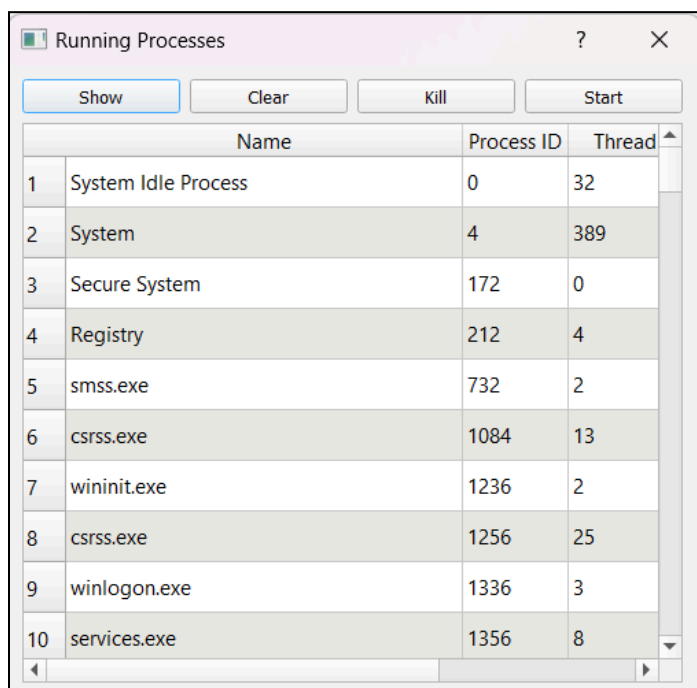
## 2. Các Phương Thức của Server

Nằm trong def *handle\_process\_commands*(self, conn: socket.socket, command: str, args: list) trong file ps.py.

Phương thức	Chức năng
def handle_connection(self, conn: socket.socket, addr: str, is_main: bool)	Xử lý các request từ Client để gọi hàm thực hiện các chức năng tương ứng
def handle_process_commands(self, conn: socket.socket, command: str, args: list)	Thực hiện một trong các thao tác (list, start, stop) process ở phía Server mà người dùng truyền lệnh thực hiện
def list_process(self)	Thực hiện list các processes đang chạy ở Server
def send_formatted_response(self, conn: socket.socket, status: str, data: any = None)	Thực hiện format lại response và trả response đã được format về cho Client

## b. Giao Diện

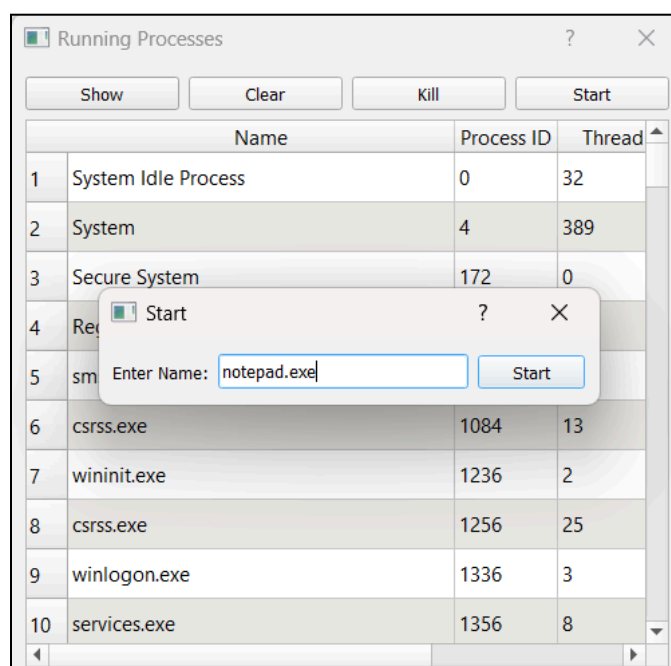
*Hiển thị các process đang chạy ở Server:*



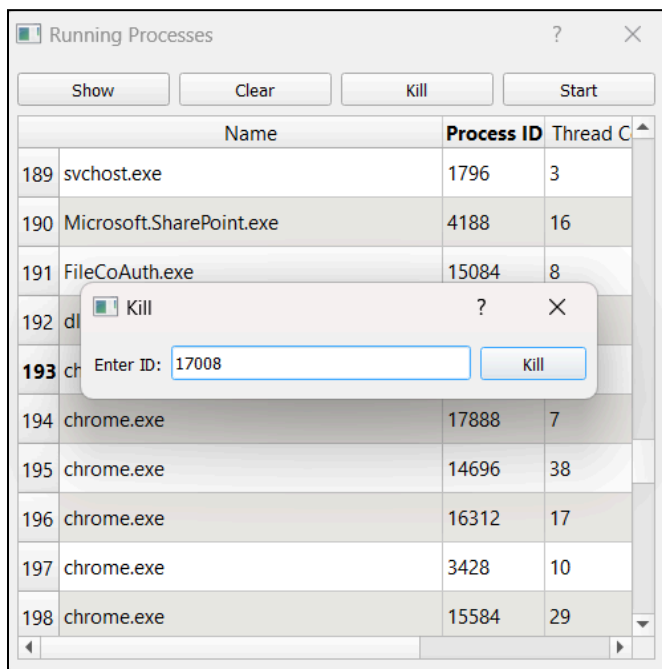
The screenshot shows the 'Running Processes' window with a table of active processes. The table has columns for an index, Name, Process ID, and Thread. The processes listed are System Idle Process, System, Secure System, Registry, smss.exe, csrss.exe, wininit.exe, csrss.exe, winlogon.exe, and services.exe.

	Name	Process ID	Thread
1	System Idle Process	0	32
2	System	4	389
3	Secure System	172	0
4	Registry	212	4
5	smss.exe	732	2
6	csrss.exe	1084	13
7	wininit.exe	1236	2
8	csrss.exe	1256	25
9	winlogon.exe	1336	3
10	services.exe	1356	8

*Bật (start) một process nào đó ở Server:*



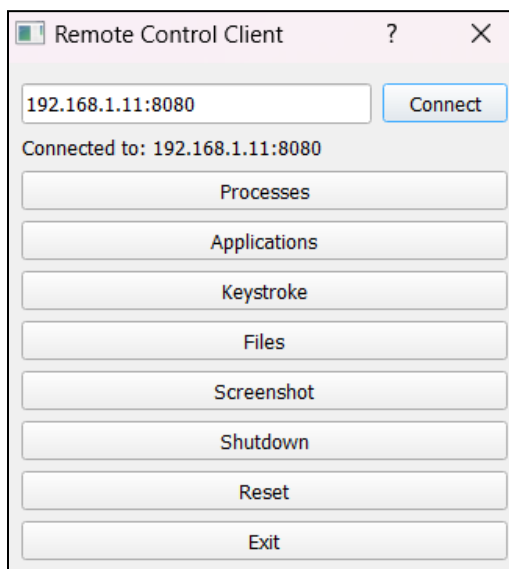
*Tắt (kill) một process theo ID ở Server:*



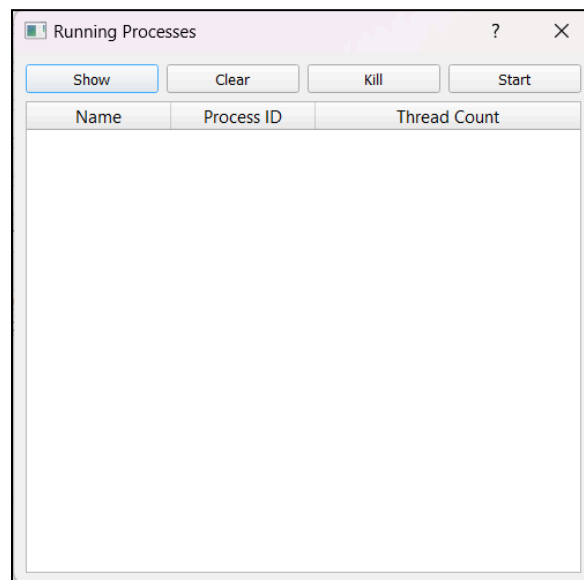
## c. Cách Hoạt Động Chính

### Chức Năng 1. Khởi động Processes

1. Chọn nút bấm **Processes**, hàm `def setup_connections(self)` thực hiện lệnh `self.btn_process.clicked.connect(self.process)`, tức khi chọn Processes thì `btn_process` được chỉ thị nhấn chọn và sau khi được nhấn chọn sẽ thực hiện hàm `def process(self)`.

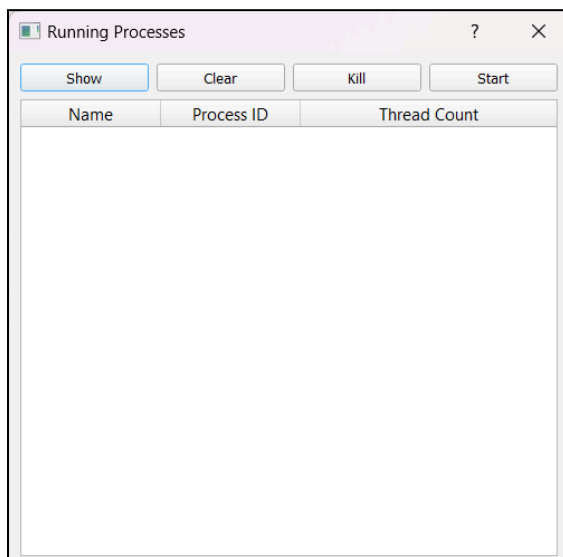


2. Hàm `def process(self)` được thực hiện: Gọi hàm `check_connection()` để kiểm tra hiện trạng kết nối giữa máy Client và Server, nếu kết nối không ổn định thì trả về kết quả báo lỗi bên phía Client, nếu kết nối ổn định thì thực hiện `Dialog_process(self.network, self)` với tham số truyền vào là đối tượng quản lý kết nối mạng giữa Client và Server đang kết nối với nhau.
3. Lớp `class Dialog_process(QDialog, Ui_dialog_process)` được thực hiện với hàm chạy tiên khởi là `def __init__(self, network: NetworkManager, parent=None)` bao gồm:
  - `self.setupUi(self)` : Khai báo các đối tượng (nút nhấn, layout,...) giao diện người dùng từ lớp `class Ui_dialog_process(object)` trong file `dialog_ui.py`.
  - `self.network = network` : Lưu đối tượng `network`.
  - `self.setup_ui()` : Thiết lập giao diện hiện sau khi chọn nút `Processes`.
  - `self.connect_signals()` : Khai báo các nút nếu được nhấp chọn sẽ thực hiện các hàm tương ứng để thực hiện list hoặc start hoặc stop các processes đang chạy ở Server.



## Chức Năng 2. Hiện (list) các process đang chạy ở Server

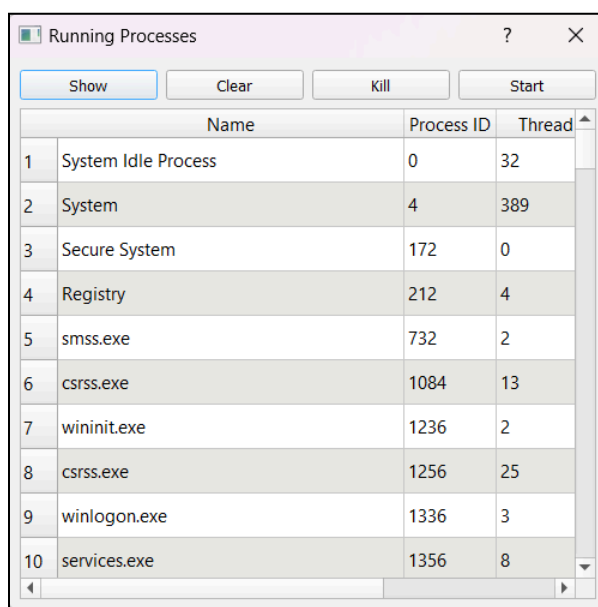
1. Chọn nút **Show**, hàm `def connect_signals(self)` thực hiện lệnh `self.btn_show.clicked.connect(self.view_apps)`, tức sau khi chọn Show nút `btn_show` được chỉ thị nhấp chọn và sau khi được nhấp chọn sẽ thực hiện hàm `def view_processes(self)`.



2. Hàm `def view_processes(self)` được thực hiện:

- Thiết lập con trỏ chuột xoay vòng khi đang chờ response từ Server (`QtWidgets.QApplication.setOverrideCursor(Qt.WaitCursor)`).
- Request lên server và chờ nhận response từ server về việc thực hiện chức năng List processes đang chạy ở Server bằng cách gọi hàm `send_message(process//list)` với biến truyền vào "process//list" để nhận diện đang gọi đến server thực hiện chức năng list các processes.
- Hàm `def send_message(self, msg: str) -> str` thực hiện với msg được truyền vào là `process//list`: Kiểm tra kết nối giữa Client và Server, nếu kết nối không ổn định thì báo lỗi, nếu kết nối ổn định thì sẽ phân chia tùy theo chức năng mà sẽ có port thực hiện khác nhau (như process và app sẽ thực hiện ở port 8080 ở hàm `def _send_main_message(self, msg: str) -> str`).
- Hàm `def _send_main_message(self, msg: str) -> str` được thực hiện với msg được truyền vào là `process//list`: Kiểm tra kết nối của socket (`main_socket`), nếu kết nối không còn thì báo lỗi, nếu còn kết nối thì thiết lập thời gian để request và nhận response từ phía server là 300 (giây). Sau khi thiết lập thời gian thực hiện thì gửi yêu cầu List các processes đang chạy trên Server bằng socket (`self.main_socket.sendall(msg.encode("utf-8"))`) với msg được truyền vào là `process//list` (Phần Server thực hiện sẽ được đề cập ở phần bên dưới) và nhận các đoạn dữ liệu trả về từ Server qua việc thực hiện hàm `def _receive_main_response(self) -> str` (với các đoạn dữ liệu trả về có size là 4096). Hàm `def _receive_main_response(self) -> str` thực hiện vòng lặp while cho đến khi hết các đoạn dữ liệu (có size 4096) được trả về từ Server, mỗi lần thực hiện vòng lặp thì các đoạn dữ liệu trả về được nối lại với nhau, sau đó hàm trả về dữ liệu hoàn chỉnh và nhận kết quả trả về ở đích đến cuối cùng là response của hàm `def view_processes(self)` (`response = self.network.send_message(process//list)`). Sau khi quay trở lại hàm `def view_processes(self)` và nhận response từ Server, thực hiện hàm `def parse_server_response(self, response: str) -> tuple[bool, any]`

- (với tham số truyền vào là response của Server) để kiểm tra format response trả về và hàm trả về với 2 giá trị (keyword) là status và data.
- Sau khi nhận kết quả trả về từ hàm `def parse_server_response(self, response: str) -> tuple[bool, any]` ở hàm `def view_processes(self)` thì gán vào 2 biến `success` và `data` (`success, data = self.network.parse_server_response(response)`). Kiểm tra trạng thái thành công (dựa vào biến `success`), kiểu dữ liệu của `data` (`data` phải có dạng dict), và 'process' phải có trong `data`. Nếu lỗi thì thông báo đến người dùng.
  - Khai báo biến `processes` và gán giá trị là các biến trong `data` có từ khóa là 'process'.
  - In các giá trị (`name`, `pid`, `threads`) trong biến `processes` và hiện ở các hàng với các cột tương ứng (`Name`, `Process ID`, `Thread Count`).



	Name	Process ID	Thread
1	System Idle Process	0	32
2	System	4	389
3	Secure System	172	0
4	Registry	212	4
5	smss.exe	732	2
6	csrss.exe	1084	13
7	wininit.exe	1236	2
8	csrss.exe	1256	25
9	winlogon.exe	1336	3
10	services.exe	1356	8

### 3. Phần thực hiện ở Server

- Sau khi thực hiện hàm `def handle_connection(self, conn: socket.socket, addr: str, is_main: bool)` sẽ điều hướng request được gửi từ Client tương ứng với các hàm thực hiện, ở lệnh truyền vào từ Client là List các processes ('process//list') thì được điều hướng đến thực hiện hàm `def handle_process_commands(self, conn: socket.socket, command: str,`

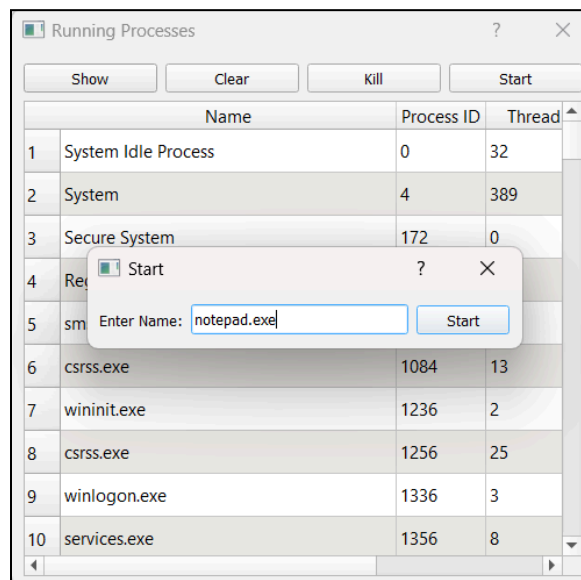


args: list) (với các tham số truyền vào là conn : đường truyền (gửi nhận dữ liệu) giữa Client và Server thông qua socket, command : chức năng yêu cầu thực hiện) và thực hiện trong mục command == list (thực hiện hàm def list\_process(self)).

- Hàm def list\_process(self) được thực hiện : Sử dụng thư viện os để thực hiện lấy các danh sách process bằng câu lệnh wmic. Nếu có lỗi thì ghi vào lịch sử log (remote\_control.log) và trả về biến dict keyword “process” có danh sách rỗng. Nếu không có lỗi thì push từng dòng giá trị được gán ở biến output vào biến kiểu dict có keyword là “process” và trả về biến kiểu dict đó ở hàm def handle\_process\_commands(self, conn: socket.socket, command: str, args: list) và gán vào biến process\_data, sau đó thực hiện hàm def send\_formatted\_response(self, conn: socket.socket, status: str, data: any = None) để format lại dữ liệu trả về và thực hiện trả response về Client qua socket.

### Chức Năng 3. Bật (start) process ở Server

1. Chọn nút Start, hàm def connect\_signals(self) ở class Dialog\_process(QDialog, Ui\_dialog\_process) thực hiện lệnh self.btn\_start.clicked.connect(self.start\_process), tức sau khi chọn Start nút btn\_start được chỉ thị nhấp chọn và sau khi được nhấp chọn sẽ thực hiện hàm def start\_process(self) và mở ra hộp cửa sổ làm việc Start (class Dialog\_start(QDialog, Ui\_dialog\_start)).

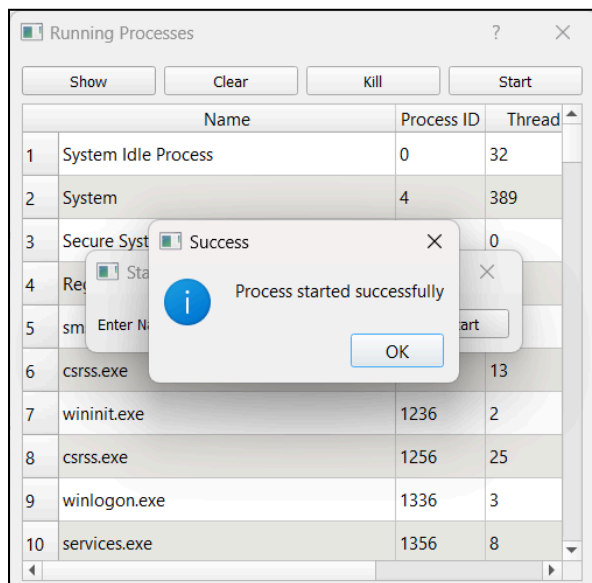


2. Lớp class `Dialog_start(QDialog, Ui_dialog_start)` được thực hiện với hàm chạy tiên khởi là `def __init__(self, status: str, network: NetworkManager, parent=None)` bao gồm:
  - `self.setupUi(self)` : Khai báo các đối tượng (nút nhấn, layout,...) giao diện người dùng từ lớp class `Ui_dialog_start(object)` trong file `dialog_ui.py`.
  - `self.status = status` : loại chức năng được truyền vào ('app' hoặc 'process')
  - `self.network = network` : Lưu đối tượng network.
  - `self.btn_start.clicked.connect(self.start)` : sau khi `btn_start` được nhấp chọn thì thực hiện hàm `def start(self)`.
3. Hàm `def start(self)` ở class `Dialog_start(QDialog, Ui_dialog_start)` được thực hiện :
  - Kiểm tra tên process nhập ở textbox là hợp lệ
  - Request lên server và chờ nhận response từ server về việc thực hiện chức năng Start process đang chạy ở Server bằng cách gọi hàm `send_message(f'{self.status}//start://{self.lineEdit.text().strip()}')` với biến truyền vào "process//start//tên process" để nhận diện đang gọi

đến server thực hiện chức năng start process có tên được Client nhập vào trong textbox.

- Hàm `def send_message(self, msg: str) -> str` thực hiện với msg được truyền vào là 'process//start//tên process': Kiểm tra kết nối giữa Client và Server, nếu kết nối không ổn định thì báo lỗi, nếu kết nối ổn định thì sẽ phân chia tùy theo chức năng mà sẽ có port thực hiện khác nhau (như process và app sẽ thực hiện ở port 8080 ở hàm `def _send_main_message(self, msg: str) -> str`.
- Hàm `def _send_main_message(self, msg: str) -> str` được thực hiện với msg được truyền vào là 'process//start//tên process': Kiểm tra kết nối của socket (`main_socket`), nếu kết nối không còn thì báo lỗi, nếu còn kết nối thì thiết lập thời gian để request và nhận response từ phía server là 300 (giây). Sau khi thiết lập thời gian thực hiện thì gửi yêu cầu Start process đang chạy trên Server bằng socket (`self.main_socket.sendall(msg.encode("utf-8"))`) với msg được truyền vào là 'process//start//tên process (Phần Server thực hiện sẽ được đề cập ở phần bên dưới) và nhận các đoạn dữ liệu trả về từ Server qua việc thực hiện hàm `def _receive_main_response(self) -> str` (với các đoạn dữ liệu trả về có size là 4096). Hàm `def _receive_main_response(self) -> str` thực hiện vòng lặp while cho đến khi hết các đoạn dữ liệu (có size 4096) được trả về từ Server, mỗi lần thực hiện vòng lặp thì các đoạn dữ liệu trả về được nối lại với nhau, sau đó hàm trả về dữ liệu hoàn chỉnh và nhận kết quả trả về ở đích đến cuối cùng là response của hàm `def start(self)` ở class `Dialog_start(QDialog, Ui_dialog_start)`. Sau khi quay trở lại hàm `def start(self)` và nhận response từ Server, thực hiện hàm `def parse_server_response(self, response: str) -> tuple[bool, any]` (với tham số truyền vào là response của Server) để kiểm tra format response trả về và hàm trả về với 2 giá trị (keyword) là status và data.

- Sau khi nhận kết quả trả về từ hàm `def parse_server_response(self, response: str) -> tuple[bool, any]` ở hàm `def start(self)` thì gán vào 2 biến `success` và `data` (`success, data = self.network.parse_server_response(response)`). Kiểm tra trạng thái thành công (dựa vào biến `success`). Nếu không `success` thì thông báo lỗi đến người dùng, nếu `success` thì hiển thị hộp thoại thông báo start thành công và refresh lại parent dialog (gọi lại hàm view tương ứng).



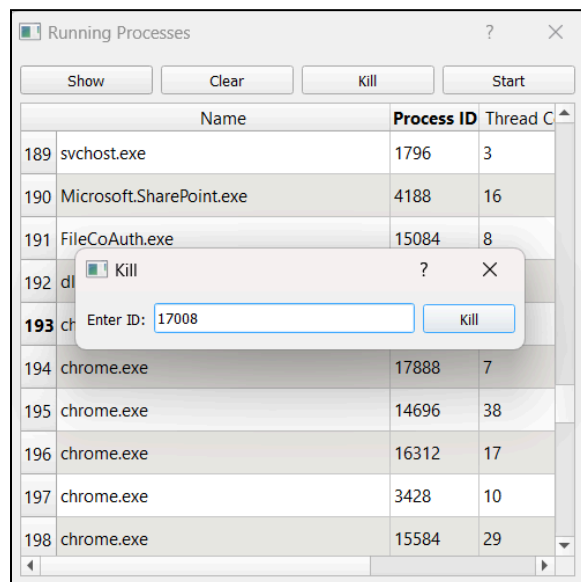
#### 4. Phần thực hiện ở Server (ps.py)

- Sau khi thực hiện hàm `def handle_connection(self, conn: socket.socket, addr: str, is_main: bool)` sẽ điều hướng request được gửi từ Client tương ứng với các hàm thực hiện, ở lệnh truyền vào từ Client là Start process ('process//start//tên process) thì được điều hướng đến thực hiện hàm `def handle_process_commands(self, conn: socket.socket, command: str, args: list)` (với các tham số truyền vào là `conn` : đường truyền (gửi nhận dữ liệu) giữa Client và Server thông qua socket, `command` : chức năng yêu cầu thực hiện), `args` : tên process cần start và thực hiện trong mục `command == start`.
- Sử dụng thư viện `subprocess` để thực hiện start process. Sau đó thực hiện hàm `def send_formatted_response(self, conn: socket.socket,`

status: str, data: any = None) để format lại dữ liệu trả về và thực hiện trả response về Client qua socket.

#### Chức Năng 4. Dừng (stop) process theo ID ở Server

1. Chọn nút Kill, hàm def connect\_signals(self) ở class Dialog\_process(QDialog, Ui\_dialog\_process) thực hiện lệnh self.btn\_kill.clicked.connect(self.kill\_process), tức sau khi chọn Kill nút btn\_kill được chỉ thị nhấp chọn và sau khi được nhấp chọn sẽ thực hiện hàm def kill\_process(self) và mở ra hộp cửa sổ làm việc Kill (class Dialog\_kill(QDialog, Ui\_dialog\_kill)).



2. Lớp class Dialog\_kill(QDialog, Ui\_dialog\_kill) được thực hiện với hàm chạy tiên khởi là def \_\_init\_\_(self, status: str, network: NetworkManager, parent=None) bao gồm:
  - self.setupUi(self) : Khai báo các đối tượng (nút nhấn, layout,..) giao diện người dùng từ lớp class Ui\_dialog\_kill(object) trong file dialog\_ui.py.
  - self.status = status : loại chức năng được truyền vào ('app' hoặc 'process').
  - self.network = network : Lưu đối tượng network.

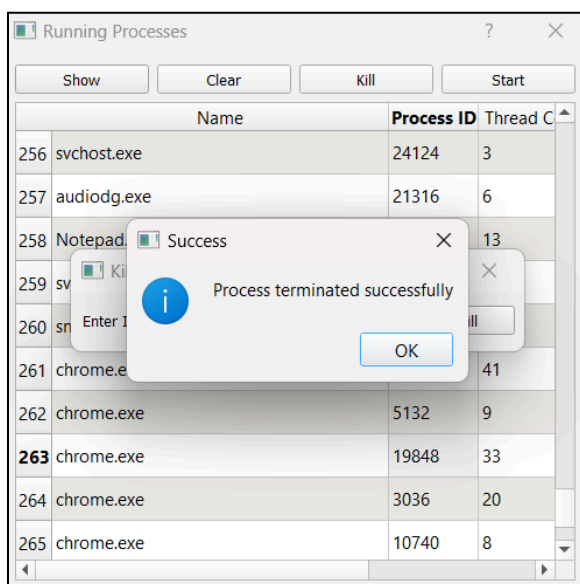
- `self.btn_kill.clicked.connect(self.kill)` : sau khi `btn_kill` được nhấp chọn thì thực hiện hàm `def kill(self)`.

3. Hàm `def kill(self)` được thực hiện :

- Kiểm tra id process nhập ở textbox là hợp lệ.
- Request lên server và chờ nhận response từ server về việc thực hiện chức năng Stop process đang chạy ở Server bằng cách gọi hàm `send_message(f'{self.status}//kill//[pid]')` với biến truyền vào "process//kill//id process" để nhận diện đang gọi đến server thực hiện chức năng stop process có id được Client nhập vào trong textbox.
- Hàm `def send_message(self, msg: str) -> str` thực hiện với msg được truyền vào là ' process//kill//id process ' : Kiểm tra kết nối giữa Client và Server, nếu kết nối không ổn định thì báo lỗi, nếu kết nối ổn định thì sẽ phân chia tùy theo chức năng mà sẽ có port thực hiện khác nhau (như process và app sẽ thực hiện ở port 8080 ở hàm `def _send_main_message(self, msg: str) -> str`.
- Hàm `def _send_main_message(self, msg: str) -> str` được thực hiện với msg được truyền vào là ' process//kill//id process ' : Kiểm tra kết nối của socket (`main_socket`), nếu kết nối không còn thì báo lỗi, nếu còn kết nối thì thiết lập thời gian để request và nhận response từ phía server là 300 (giây). Sau khi thiết lập thời gian thực hiện thì gửi yêu cầu Stop process đang chạy trên Server bằng socket (`self.main_socket.sendall(msg.encode("utf-8"))`) với msg được truyền vào là 'process//kill//id process (Phần Server thực hiện sẽ được đề cập ở phần bên dưới) và nhận các đoạn dữ liệu trả về từ Server qua việc thực hiện hàm `def _receive_main_response(self) -> str` (với các đoạn dữ liệu trả về có size là 4096). Hàm `def _receive_main_response(self) -> str` thực hiện vòng lặp while cho đến khi hết các đoạn dữ liệu (có size 4096) được trả về từ Server, mỗi lần thực hiện vòng lặp thì các đoạn dữ liệu trả về được nối lại với nhau, sau đó hàm trả về dữ liệu hoàn chỉnh và nhận kết quả trả về ở đích đến cuối

cùng là response của hàm `def kill(self)` ở class `Dialog_kill(QDialog, Ui_dialog_start)`. Sau khi quay trở lại hàm `def kill(self)` và nhận response từ Server, thực hiện hàm `def parse_server_response(self, response: str) -> tuple[bool, any]` (với tham số truyền vào là response của Server) để kiểm tra format response trả về và hàm trả về với 2 giá trị (keyword) là status và data.

- Sau khi nhận kết quả trả về từ hàm `def parse_server_response(self, response: str) -> tuple[bool, any]` ở hàm `def kill(self)` thì gán vào 2 biến `success` và `data` (`success, data = self.network.parse_server_response(response)`). Kiểm tra trạng thái thành công (dựa vào biến `success`). Nếu không `success` thì thông báo lỗi đến người dùng, nếu `success` thì hiển thị hộp thoại thông báo start thành công và refresh lại parent dialog (gọi lại hàm `view` tương ứng).



#### 4. Phần thực hiện ở Server (ps.py)

- Sau khi thực hiện hàm `def handle_connection(self, conn: socket.socket, addr: str, is_main: bool)` sẽ điều hướng request được gửi từ Client tương ứng với các hàm thực hiện, ở lệnh truyền vào từ Client là Stop process ('process//kill//id process) thì được điều hướng đến thực hiện hàm `def handle_process_commands(self, conn:`

socket.socket, command: str, args: list) (với các tham số truyền vào là conn : đường truyền (gửi nhận dữ liệu) giữa Client và Server thông qua socket, command : chức năng yêu cầu thực hiện), args : id process cần start và thực hiện trong mục command == kill.

- Sử dụng thư viện subprocess để kiểm tra id process có đang trong danh sách các application đang chạy không và thực hiện kill application. Sau đó thực hiện hàm def send\_formatted\_response(self, conn: socket.socket, status: str, data: any = None) để format lại dữ liệu trả về và thực hiện trả response về Client qua socket.

## 4. Chức Năng Keystroke

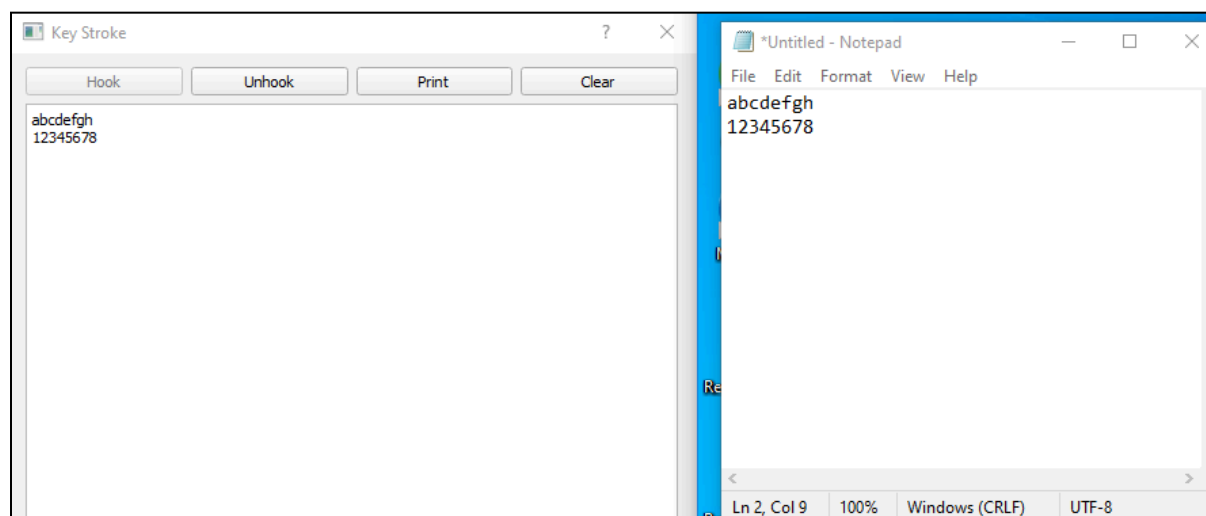
### a. Tổng Quan Về Cơ Chế

**Module:** dialog\_keystroke()

Phương thức	Chức năng
toggle_hook()	Bắt đầu ghi lại các phím nhấn bằng cách gửi yêu cầu 'hook' tới server.
unhook()	Ngừng ghi lại các phím nhấn bằng cách gửi yêu cầu 'unhook' tới server.
get_keystrokes()	Lấy nhật ký các phím nhấn từ server và hiển thị trong giao diện.
clear_log()	Xóa nhật ký các phím nhấn trong giao diện.
setup_ui()	Cài đặt giao diện ban đầu, bao gồm cài đặt chế độ chỉ đọc và trạng thái nút.
connect_signals()	Kết nối các nút giao diện với các chức năng tương ứng.
update_hook_button_state()	Cập nhật trạng thái các nút bắt đầu và ngừng keylogger dựa trên trạng thái hiện tại.



## b. Giao Diện



## c. Cách Hoạt Động Chính

### Chức Năng 1. Khởi động Keystroke

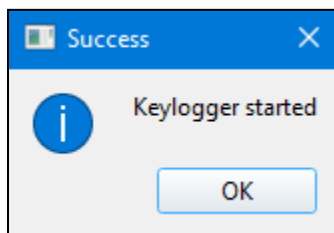
- Khi nhấn nút Keystroke , tức là lớp Dialog\_keystroke được khởi tạo, hàm **\_\_init\_\_** sẽ được gọi.
- Hàm này thực hiện việc thiết lập giao diện người dùng qua **setupUi(self)**.
- Gọi hàm **setup\_ui(self)** để cấu hình các thành phần giao diện như: đặt trạng thái của textBrowser\_2 là chỉ đọc (read-only) và thiết lập **is\_hooked** là False (không khóa phím).
- Gọi hàm **connect\_signals(self)** để kết nối các nút bấm với các hàm xử lý sự kiện:
  - Nút **Hook** sẽ gọi hàm **toggle\_hook(self)** để bắt đầu keylogger.
  - Nút **Unhook** sẽ gọi hàm **unhook(self)** để dừng keylogger.
  - Nút **Print** sẽ gọi hàm **get\_keystrokes(self)** để lấy và in ra các phím đã ghi lại.
  - Nút **Clear** sẽ gọi hàm **clear\_log(self)** để xóa log phím.

## Chức Năng 2. Khóa (hook) phím của Server

- Khi người dùng nhấn nút Hook, hàm `toggle_hook(self)` sẽ được gọi.
- Hàm này gửi một yêu cầu qua mạng tới server để bắt đầu keylogger bằng cách gửi lệnh 'key//hook':

```
response = self.network.send_message('key//hook')
```

- Nút **Hook** sẽ bị vô hiệu hóa (disabled).
- Nút **Unhook** sẽ được kích hoạt (enabled).
- Hiện thị người dùng qua hộp thoại:



- Nếu có lỗi xảy ra, thông báo lỗi sẽ được hiển thị cho người dùng qua hộp thoại `QMessageBox.critical`.

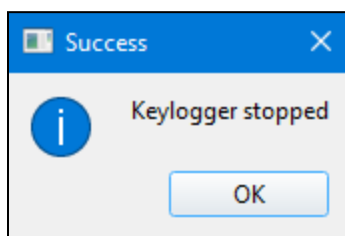
## Chức Năng 3. In (print) phím đã log

- Khi người dùng nhấn nút **Print**, hàm `get_keystrokes(self)` sẽ được gọi.
  - Hàm này gửi một yêu cầu qua mạng tới server để lấy dữ liệu các phím đã gõ thông qua lệnh 'key//getkey':
- ```
response = self.network.send_message('key//getkey')
```
- Nếu server trả về một chuỗi không phải là '404' (tức là có dữ liệu phím), các phím đó sẽ được thêm vào `textBrowser_2`.
    - Nội dung cũ trong `textBrowser_2` sẽ được giữ lại và các phím mới sẽ được thêm vào.
    - Sau khi thêm, thanh cuộn trong `textBrowser_2` sẽ tự động cuộn xuống dưới để hiển thị phím mới.
  - Nếu không có dữ liệu hoặc có lỗi xảy ra, một thông báo lỗi sẽ được hiển thị qua `QMessageBox.critical`.

## Chức Năng 4. Bỏ khóa (unhook) phím của Server

- Khi người dùng nhấn nút Unhook, hàm unhook(self) sẽ được gọi.
- Hàm này gửi yêu cầu qua mạng tới server để dừng keylogger bằng lệnh 'key//unhook':  

```
response = self.network.send_message('key//unhook')
```
- Nếu server trả về 'ok', keylogger sẽ bị dừng, và trạng thái is\_hooked sẽ được cập nhật thành False. Giao diện sẽ thay đổi trạng thái các nút:
  - Nút **Hook** sẽ được kích hoạt lại.
  - Nút **Unhook** sẽ bị vô hiệu hóa.
- Hiển thị người dùng qua hộp thoại:



- Nếu có lỗi xảy ra, thông báo lỗi sẽ được hiển thị cho người dùng qua QMessageBox.critical.

## Chức Năng 5. Phím Clear

Nút **Clear** được kết nối với phương thức clear\_log(), và phương thức này có chức năng:

- Xóa nội dung trong textBrowser\_2, là nơi hiển thị log các phím được ghi lại từ keylogger.
- Cụ thể, nó sử dụng phương thức clear() của QTextBrowser để làm trống toàn bộ nội dung hiển thị.

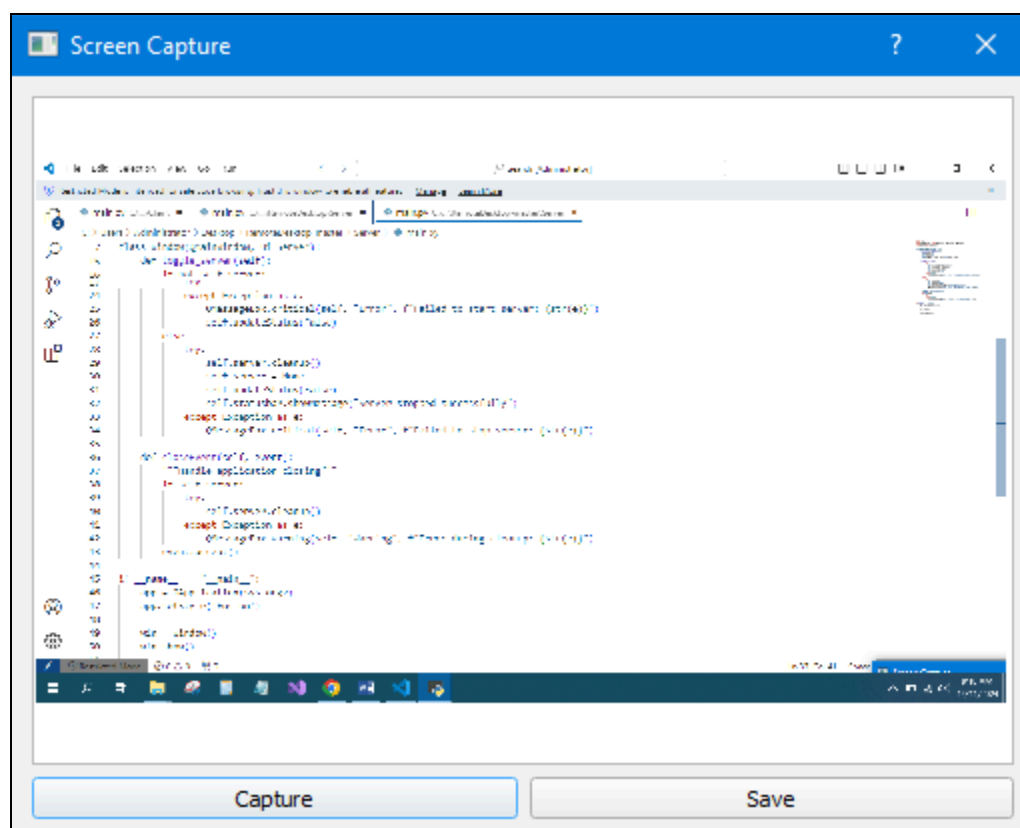
## 4. Chức Năng Screenshot

### a. Tổng Quan Về Cơ Chế

Module: dialog\_capture()

| Phương thức | Chức năng                                                         |
|-------------|-------------------------------------------------------------------|
| capture()   | Chụp ảnh màn hình từ server và hiển thị trực tiếp trên giao diện. |
| save()      | Lưu ảnh chụp màn hình vào tệp tin do người dùng chọn.             |

### b. Giao Diện



## c. Cách Hoạt Động Chính

### Chức Năng 1. Khởi động Screenshot

1. Khi ứng dụng khởi động, phương thức `setup_connections(self)` sẽ được gọi. Đây là nơi thiết lập các kết nối sự kiện giữa các nút bấm trong giao diện người dùng và các hàm xử lý của chương trình.

2. Trong `setup_connections(self)`, lệnh sau được sử dụng để kết nối nút **Screenshot** (nút `pushButton_2`) với hàm `capture()`:

```
self.pushButton_2.clicked.connect(self.capture)
```

Điều này có nghĩa là khi người dùng nhấn nút **Screenshot**, chương trình sẽ gọi hàm `capture()` để bắt đầu quá trình chụp ảnh.

### Chức Năng 2. Chụp màn hình (capture) của Server

Khi người dùng nhấn nút Screenshot, hàm `capture(self)` sẽ được gọi và thực hiện các bước sau:

1. Kiểm tra kết nối mạng:

- Chương trình sẽ kiểm tra xem socket có được kết nối với server hay không. Nếu không có kết nối, chương trình sẽ đưa ra thông báo lỗi và không tiếp tục thực hiện.
- Nếu kết nối không tồn tại, sẽ có thông báo:

```
raise Exception("Not connected to server")
```

2. Gửi lệnh chụp ảnh:

- Nếu kết nối thành công, chương trình sẽ gửi lệnh `capture` đến server thông qua socket để yêu cầu server thực hiện chụp ảnh màn hình.
- Câu lệnh gửi sẽ được thực hiện như sau:

```
self.network.basic_socket.sendall(b'capture')
```

3. Nhận dữ liệu ảnh từ server:

- Sau khi gửi lệnh, chương trình bắt đầu nhận dữ liệu ảnh từ server qua socket.

- Chương trình sử dụng một vòng lặp để nhận dữ liệu theo từng phần nhỏ (chunk). Dữ liệu này được lưu vào bộ đệm `img_data`.
- Vòng lặp sẽ tiếp tục nhận dữ liệu cho đến khi gặp chuỗi kết thúc `<<END>>`, thông báo rằng dữ liệu ảnh đã nhận xong.

```
chunk = self.network.basic_socket.recv(32768)
```

#### 4. Kiểm tra và hiển thị ảnh:

- Khi dữ liệu ảnh đã nhận đầy đủ, chương trình kiểm tra xem dữ liệu có hợp lệ hay không. Nếu không, chương trình sẽ đưa ra thông báo lỗi.
- Nếu ảnh hợp lệ, chương trình sẽ sử dụng `QPixmap` để tải và hiển thị ảnh lên `QGraphicsView`:

```
pixmap = QPixmap()  
pixmap.loadFromData(img_data):
```

#### 5. Xử lý lỗi:

- Nếu có lỗi xảy ra trong quá trình nhận dữ liệu (như socket timeout, kết nối bị mất), chương trình sẽ hiển thị thông báo lỗi cho người dùng:

```
except TimeoutError as e:  
    QMessageBox.warning(self, "Warning",  
        "Connection timed out but the screenshot might have been saved.\n")
```

### Chức Năng 3. Lưu (save) ảnh chụp về Client

Sau khi ảnh được chụp và hiển thị, người dùng có thể lưu ảnh vào máy tính của mình. Khi nhấn nút Save, hàm `save(self)` sẽ được gọi và thực hiện các bước sau:

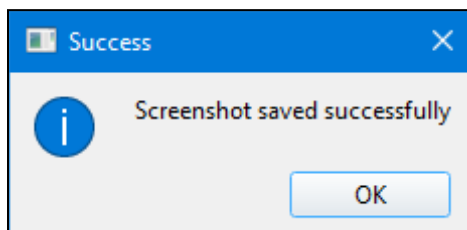
1. Mở hộp thoại lưu tệp:
  - Khi người dùng nhấn nút Save, một hộp thoại `QFileDialog.getSaveFileName()` sẽ mở ra, cho phép người dùng chọn vị trí và tên tệp để lưu ảnh.
  - Dữ liệu của tệp ảnh sẽ được lưu từ `capture.png` (ảnh mặc định trong ứng dụng).
2. Sao chép và lưu ảnh:
  - Sau khi người dùng chọn vị trí và tên tệp, chương trình sẽ mở tệp `capture.png` và sao chép nội dung vào tệp mà người dùng đã chọn.

- Đây là mã thực hiện sao chép dữ liệu ảnh:

```
try:
    file1 = open("capture.png", "rb")
    file2 = open(str(path[0]), "wb")
    l = file1.readline()
    while l:
        file2.write(l)
        l = file1.read()
    file1.close()
    file2.close()
```

### 3. Thông báo kết quả:

- Sau khi ảnh được lưu thành công, chương trình sẽ hiển thị thông báo thành công cho người dùng:



- Nếu có lỗi xảy ra trong quá trình lưu ảnh, chương trình sẽ hiển thị thông báo lỗi:

```
QMessageBox.critical(self, "Error", f"Failed to save screenshot: {str(e)}")
```

## 5. Chức Năng Quản Lý File

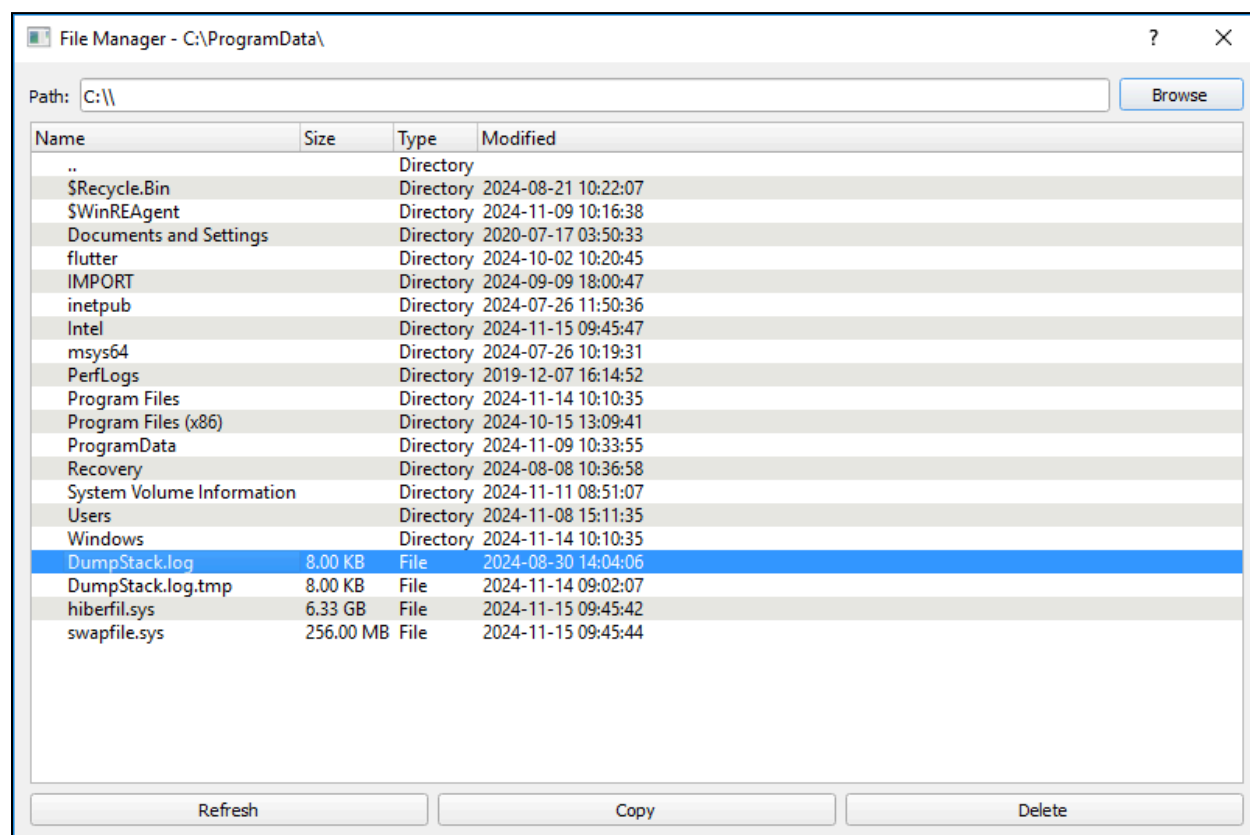
### a. Tổng Quan Về Cơ Chế

**Feature:** Files

| Phương thức         |                        | Chức năng                                      |
|---------------------|------------------------|------------------------------------------------|
| Client              | Server                 |                                                |
| show_files()        |                        | Check connect tới server và gọi dialog_files() |
| dialog_files()      | handle_file_commands() | Khởi tạo giao diện và các sự kiện cần thiết.   |
| browse_typed_path() | list_files()           | Duyệt đường dẫn thư mục từ textbox.            |

| Phương thức                            |               | Chức năng                                        |
|----------------------------------------|---------------|--------------------------------------------------|
| Client                                 | Server        |                                                  |
| refresh_files(),<br>update_file_list() | list_files()  | Duyệt lại đường dẫn và cập nhật DS file          |
| on_item_double_clicked()               | list_files()  | Duyệt và show tất cả file, folder nếu là thư mục |
| go_up_directory()                      | list_files()  | Trở về thư mục trước đó                          |
| download_file()                        | send_file()   | Tải file từ server về máy client                 |
| delete_file()                          | delete_file() | Xóa file trên máy server                         |

## b. Giao Diện





## c. Cách Hoạt Động Chính

### Chức Năng 1. Khởi động File Manager và Duyệt Thư Mục

1. Nhấn vào nút Files từ giao diện client sẽ call function show\_files

```
def setup_connections(self):
    self.btn_cap.clicked.connect(self.capture)
    self.btn_process.clicked.connect(self.process)
    self.btn_app.clicked.connect(self.app)
    self.btn_key.clicked.connect(self.key)
    self.btn_connect.clicked.connect(self.connect)
    self.btn_shutdown.clicked.connect(self.shutdown)
    self.btn_reset.clicked.connect(self.reset)
    self.btn_files.clicked.connect(self.show_files)
    self.btn_exit.clicked.connect(self.exit)
```

2. Trong function này sẽ check connection tới server, nếu có thì gọi function Dialog\_files để khởi tạo và thực thi (\*) các nghiệp vụ và show form File Manager với ổ đĩa đặt mặc định là C. Ngược lại sẽ hiện “not connect to server”.

```
def show_files(self):
    if not self.check_connection():
        self.show_error("Not connected to server")
        return
    try:
        dialog = Dialog_files(self.network, self)
        dialog.exec()
    except Exception as e:
        self.show_error(f"Operation failed: {str(e)}")
```

- Các nghiệp vụ ở phần (\*) là: Khởi tạo sự kiện cho các button, key Backspace và đọc tất cả folder, file (cấp 1) bên trong ổ C trên Server hiện lên form File Manager thông qua các function \_init\_, setup\_connections, setup\_shortcuts, refresh\_files.

### 3. Xử lý phía Client:

```
class Dialog_files(QDialog, Ui_dialog_files):
    def __init__(self, network: NetworkManager, parent=None):
        super().__init__(parent)
        self.setupUi(self)
        self.network = network
        self.current_path = "C:\\\\" # Default path
        self.path_history = ["C:\\\\" # Add path history
        self.setup_shortcuts()
        self.setup_connections()
        self.pathEdit.setText(self.current_path)
        self.refresh_files()

    def setup_connections(self):
        self.btn_browse.clicked.connect(self.browse_typed_path)
        self.btn_refresh.clicked.connect(self.refresh_files)
        self.btn_download.clicked.connect(self.download_file)
        self.btn_delete.clicked.connect(self.delete_file)
        self.pathEdit.returnPressed.connect(self.browse_typed_path)
        self.treeWidget.itemDoubleClicked.connect(self.on_item_double_clicked)

    def setup_shortcuts(self):
        self.shortcut_back = QtWidgets.QShortcut(QtGui.QKeySequence("Backspace"),
        self.shortcut_back.activated.connect(self.go_up_directory)
```

- Client sẽ gửi message “files//list//{path}” qua Server xử lý và nhận lại kết quả.

```
def refresh_files(self):
    try:
        QtWidgets.QApplication.setOverrideCursor(Qt.WaitCursor)

        # Update window title to show current path
        self.setWindowTitle(f"File Manager - {self.current_path}")

        response = self.network.send_message(f'files//list//{self.current_path}')

    except:
        # Check if response is empty or invalid
        if not response or not response.strip():
            raise Exception("Empty response from server")

        data = json.loads(response)

        if data.get('status') == 'error':
            # If the path doesn't exist, try going up one level
            if 'Path does not exist' in data.get('error', ''):
                parent_path = os.path.dirname(os.path.dirname(self.current_path.rstrip('\'))) + '\ '
                if parent_path and parent_path != self.current_path:
                    self.current_path = parent_path
                    self.pathEdit.setText(self.current_path)
                    return self.refresh_files()
            raise Exception(data.get('error', 'Unknown error'))

        if 'items' not in data:
            raise Exception("Invalid response format: missing items")

        self.update_file_list(data.get('items', []))

    except json.JSONDecodeError as e:
        # Log the problematic response for debugging
        logging.error(f"Invalid JSON response: {response[:1000]}...") # Log first 1000 chars
        raise Exception(f"Invalid server response format: {str(e)}")

    except Exception as e:
        QMessageBox.critical(self, "Error", f"Failed to list files: {str(e)}")
    finally:
        QtWidgets.QApplication.restoreOverrideCursor()
```

#### 4. Xử lý phía Server:

- Server sẽ xử lý cắt chuỗi message để biết command cần thực hiện là files và action là list folder như hình qua 2 function (handle\_connection, handle\_file\_commands) và list\_files.

```

def handle_connection(self, conn: socket.socket, addr: str, is_main: bool):
    """Handle a single connection"""
    connection_type = "Main" if is_main else "Basic"
    logging.info(f"Started {connection_type} connection handler for {addr}")

    try:
        while self.running:
            try:
                data = conn.recv(1024)
                if not data: ...

                # Handle ping for basic connection
                if not is_main and data == b'ping': ...

                command = data.decode().strip()
                if command == 'quit': ...

                parts = command.split('/')
                command = parts[0]

                if is_main: ...
            else:
                # Handle basic commands
                try:
                    if command == "key":
                        self.handle_keylogger_commands(conn, parts[1])
                    elif command == "files":
                        self.handle_file_commands(conn, parts[1], parts[2:])
                    elif command == "capture":
                        self.capture_screen(conn)
                    elif command == "shutdown":
                        conn.sendall(b'ok')

def handle_file_commands(self, conn: socket.socket, command: str, args: list):
    """Handle file-related commands"""
    try:
        if not args:
            response = json.dumps({
                "status": "error",
                "error": "Missing arguments"
            })
        elif command == "list":
            response = self.list_files(args[0])
        elif command == "download":
            self.send_file(conn, args[0])
            return # send_file handles its own response
        elif command == "delete":
            response = self.delete_file(args[0])
        else:
            response = json.dumps({
                "status": "error",
                "error": "Invalid command"
            })
    }

```

- Bên trong **list\_files** sẽ dùng thư viện **os** đọc tất cả files và folders bên trong gồm các thông tin **tên, size, type** và **DateModified**. Sau đó trả ra 1 danh sách **items** và trạng thái **success** hoặc **error** và thông báo lỗi

```
def list_files(self, path: str) -> str:
    try:
        # Clean and normalize the path
        path = os.path.normpath(path)

        # Validate path exists before proceeding
        if not os.path.exists(path): ...

        if not os.path.isdir(path): ...

        # Test directory access before proceeding
        try: ...
        except PermissionError: ...
        except Exception as e: ...

        items = []
        with os.scandir(path) as entries:
            for entry in entries:
                try:
                    stat = entry.stat()
                    size = ""
                    if entry.is_file():
                        size_bytes = stat.st_size
                        for unit in ['B', 'KB', 'MB', 'GB', 'TB']:
                            if size_bytes < 1024:
                                size = f"{size_bytes:.2f} {unit}"
                                break
                            size_bytes /= 1024

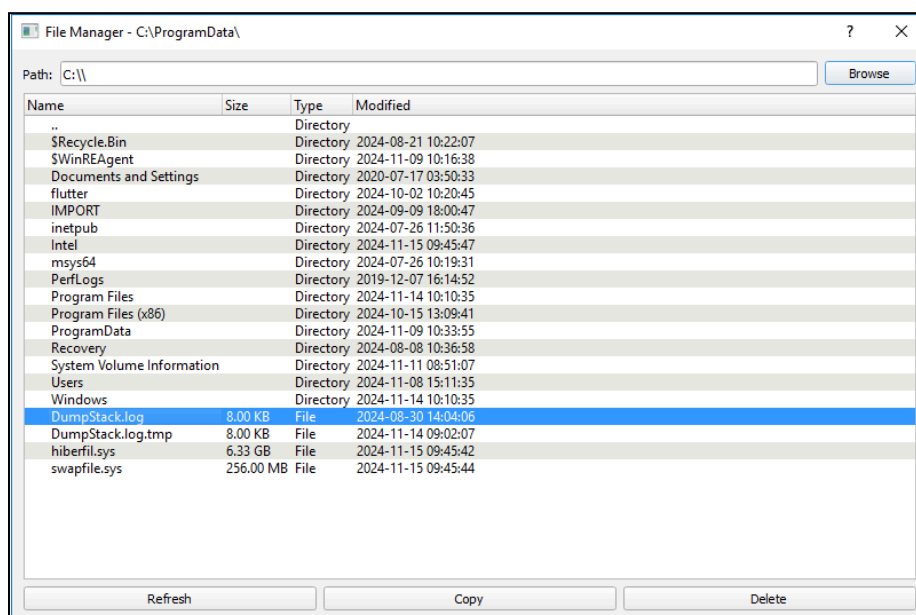
                    # Clean the filename to prevent JSON encoding issues
                    name = entry.name
                    # Remove or replace problematic characters
                    name = name.replace('\\', '\\\\').replace("'", '\\\\')
                    # Ensure the name is valid UTF-8
                    name = name.encode('utf-8', errors='replace').decode('utf-8')

                    items.append({
                        "name": name,
                        "size": size,
                        "type": "File" if entry.is_file() else "Directory",
                        "modified": datetime.fromtimestamp(stat.st_mtime).strftime("%Y-%m-%d %H:%M:%S")
                    })
                except Exception as e:
                    logging.error(f"Error processing entry {entry.name}: {str(e)}")
                    continue

        # Sort items: directories first, then files
        items.sort(key=lambda x: (x['type'] == 'File', x['name'].lower()))

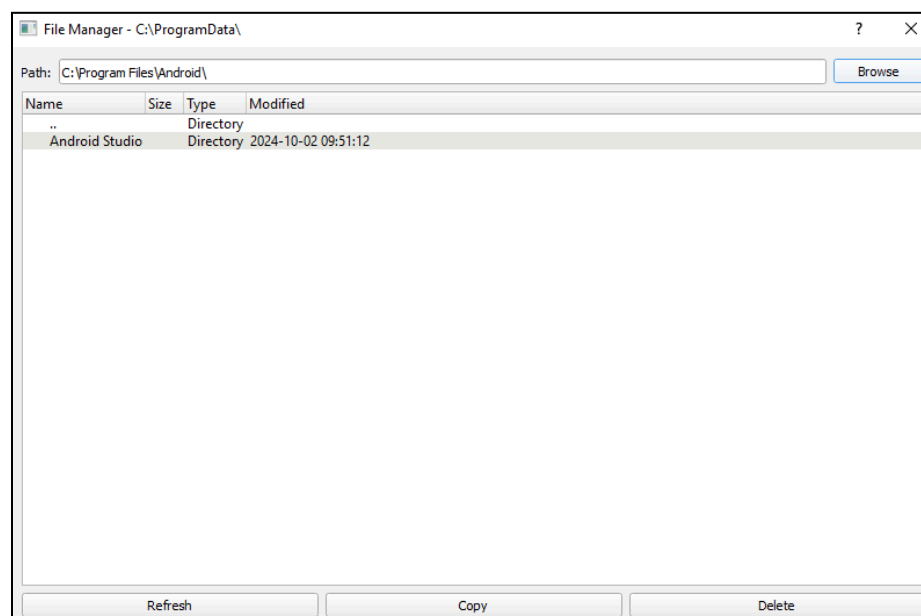
        # Use json.dumps with proper encoding and escaping
        return json.dumps({
            "status": "success",
            "items": items
        }, ensure_ascii=False, default=str)

    except Exception as e:
        error_msg = str(e).encode('utf-8', errors='replace').decode('utf-8')
        return json.dumps({
            "status": "error",
            "error": error_msg
        }, ensure_ascii=False)
```



5. Trên giao diện File Manager các tính năng như:

- Duyệt đường dẫn từ ô textbox bằng function `browse_typed_path`



- Tải lại danh sách file và folder trong ổ C sau đó trình bày lên giao diện bằng function `refresh_files` và `update_file_list`:

```
def update_file_list(self, items):
    """Update the file list with received data"""
    self.treeWidget.clear()

    # Add parent directory entry if not at root
    if self.current_path.upper() != "C:\\":
        parent_item = QtWidgets.QTreeWidgetItem(["..", "", "Directory", ""])
        parent_item.setIcon(0, self.get_icon("Directory"))
        self.treeWidget.addTopLevelItem(parent_item)

    # Add received items
    for item in items:
        try:
            tree_item = QtWidgets.QTreeWidgetItem([
                item['name'],
                item.get('size', ''),
                item.get('type', ''),
                item.get('modified', '')
            ])
            tree_item.setIcon(0, self.get_icon(item.get('type', '')))
            self.treeWidget.addTopLevelItem(tree_item)
        except Exception as e:
            logging.error(f"Error creating tree item: {str(e)}")
            continue

    for i in range(self.treeWidget.columnCount()):
        self.treeWidget.resizeColumnToContents(i)
```

- Khi double click vào 1 item trong danh sách nếu là folder thì sẽ duyệt các thư mục con và file bên trong folder vừa click và show ra form bằng function `on_item_double_clicked`:

```
def on_item_double_clicked(self, item, column):
    if item.text(2) == 'Directory':
        try:
            old_path = self.current_path

            if item.text(0) == "..":
                # Go up one directory
                new_path = os.path.dirname(os.path.dirname(self.current_path.rstrip('\\'))) + os.path.sep
            else:
                # Enter selected directory
                new_path = os.path.join(self.current_path, item.text(0))

            # Normalize path and ensure it ends with separator
            new_path = os.path.normpath(new_path) + os.path.sep

            # Try to list files in new path before updating UI
            response = self.network.send_message(f'files//list/{new_path}')
            data = json.loads(response)

            if data.get('status') == 'error':
                QMessageBox.warning(self, "Warning", data.get('error', 'Failed to access directory'))
                return

            # Only update path if listing was successful
            self.current_path = new_path
            self.pathEdit.setText(new_path)
            self.path_history.append(new_path)

            # Update the display with the received data
            self.update_file_list(data.get('items', []))
```

- Khi nhấn phím Backspace trên bàn phím thì sẽ trở ra folder trước và show danh sách file và folder của folder trước bằng function `go_up_directory`: bên trong sẽ xử lý lưu lại đường dẫn hiện tại, sau đó thực hiện lấy đường dẫn cha bằng cách gọi lồng func **`os.path.dirname`**, nếu đường dẫn cha hợp lệ thì call qua server để lấy danh sách files và folders con. Nếu không hợp lệ thì giữ nguyên hiện trạng.

```
def go_up_directory(self):
    try:
        if self.current_path.upper() == "C:\\":
            return

        old_path = self.current_path
        parent_path = os.path.dirname(os.path.dirname(self.current_path.rstrip('\\'))) + os.path.sep

        if not parent_path:
            return

        # Try to list files in parent path before updating UI
        response = self.network.send_message(f'files//list//{parent_path}')
        data = json.loads(response)

        if data.get('status') == 'error':
            QMessageBox.warning(self, "Warning", data.get('error', 'Failed to access directory'))
            return

        # Only update if listing was successful
        self.current_path = parent_path
        self.pathEdit.setText(parent_path)
        self.path_history.append(parent_path)

        # Update the display with the received data
        self.update_file_list(data.get('items', []))

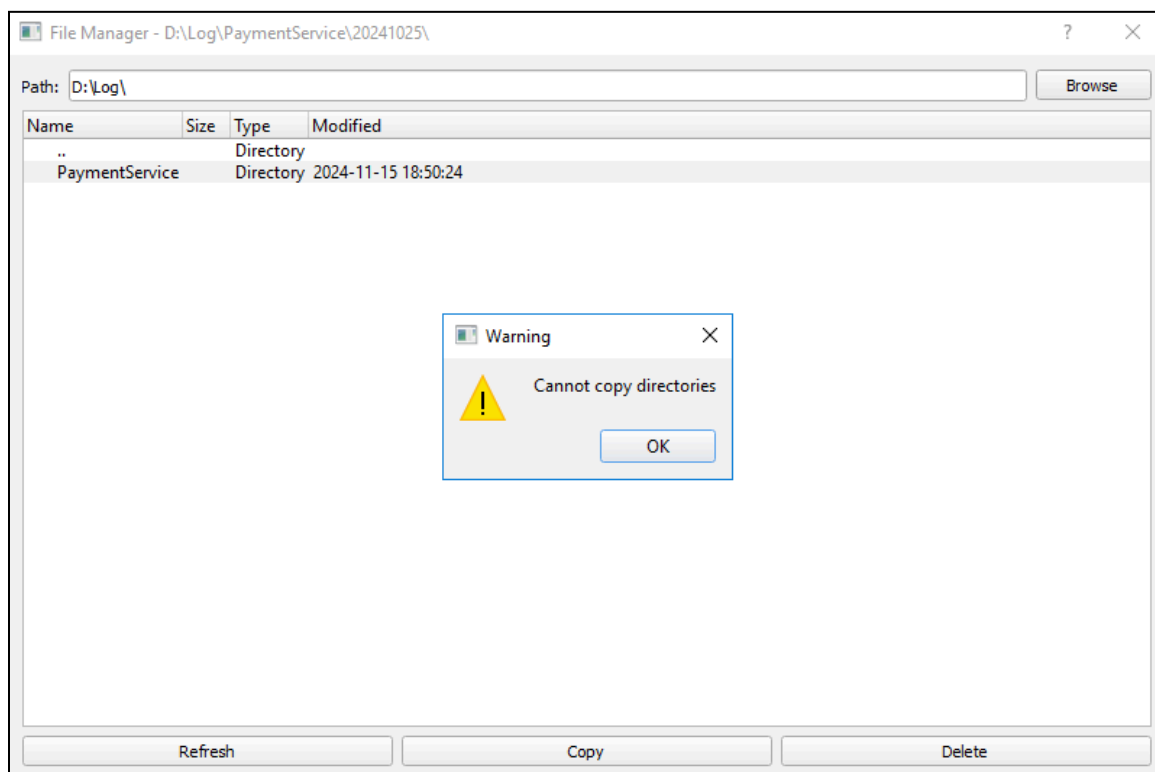
    except json.JSONDecodeError as e:
        QMessageBox.critical(self, "Error", f"Failed to parse server response: {str(e)}")
        self.pathEdit.setText(old_path)
    except Exception as e:
        QMessageBox.critical(self, "Error", f"Failed to go up directory: {str(e)}")
        self.pathEdit.setText(old_path)
```

## Chức Năng 2. Sao chép (copy) file từ Server về Client

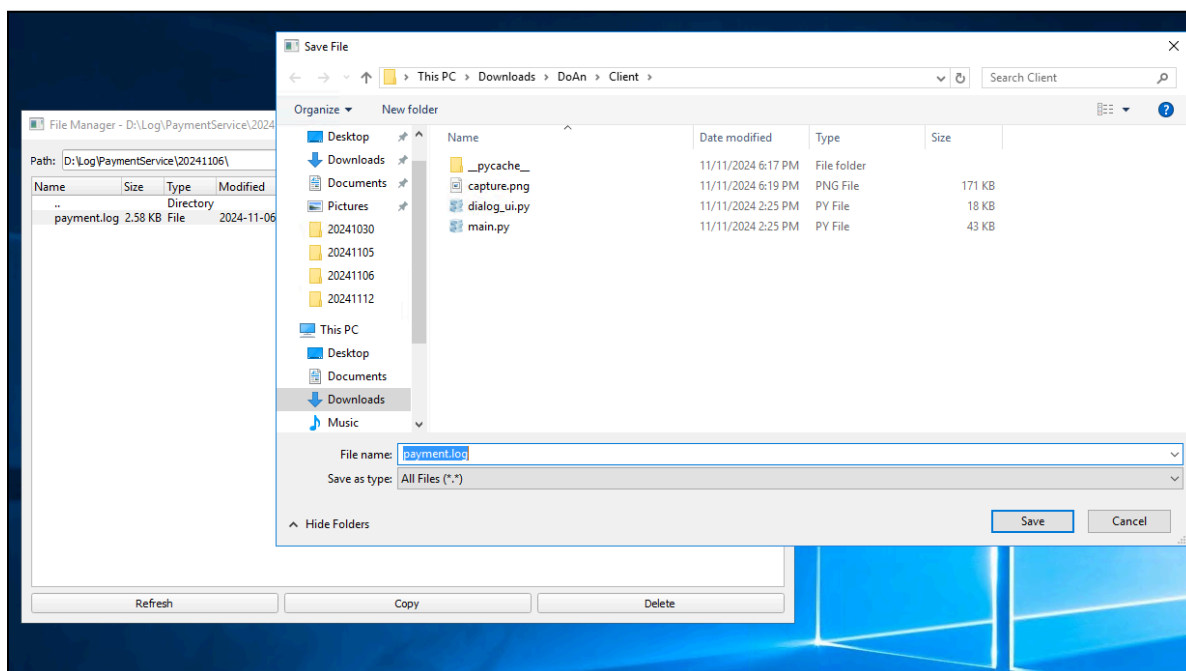
1. Khi click vào 1 item trong danh sách và ấn nút **Copy**, nếu là file thì sẽ tải về máy client ngược lại là thư mục thì thông báo là "Cannot copy directories" bằng



function download\_file.



2. Nếu là file, chọn đường dẫn để lưu lại ở Client.



### 3. Xử lý phía Client:

```
def download_file(self):
    try:
        current_item = self.treeWidget.currentItem()
        if not current_item:
            QMessageBox.warning(self, "Warning", "Please select a file to copy")
            return

        file_name = current_item.text(0)
        file_type = current_item.text(2)

        if file_type == 'Directory':
            QMessageBox.warning(self, "Warning", "Cannot copy directories")
            return

        full_path = os.path.join(self.current_path, file_name)
        save_path, _ = QFileDialog.getSaveFileName(
            self, "Save File", file_name, "All Files (*.*)"
        )

        if save_path:
            QtWidgets.QApplication.setOverrideCursor(Qt.WaitCursor)
            try:
                if self.network.send_file_command(f'files//download://{full_path}'):
                    # Use shutil.copy2 instead of os.rename
                    import shutil
                    if os.path.exists("temp_file"):
                        if os.path.exists(save_path):
                            os.remove(save_path)
                        shutil.copy2("temp_file", save_path)
                        os.remove("temp_file") # Clean up temp file after copying
                        QMessageBox.information(self, "Success", "File copied successfully")
                    else:
                        raise Exception("Failed to receive file")
            finally:
                QtWidgets.QApplication.restoreOverrideCursor()
                # Ensure temp file is cleaned up
                if os.path.exists("temp_file"):
                    try:
                        os.remove("temp_file")
                    except:
                        pass

        except Exception as e:
            QMessageBox.critical(self, "Error", f"Failed to copy file: {str(e)}")
```

Khi gọi hàm `download_file` thì sẽ lấy thông tin tên và loại của dòng được chọn, nếu là file thì tạo full path, và gửi command `download/fullpath` lên server, Server xử lý và trả về trong hàm `send_file_command` với tên file là `temp_file` (quá trình này sẽ chạy ngầm). Sau đó sẽ copy từ `temp_file` vào cái `savePath` mà người dùng chọn từ `FileDialog`

```
def send_file_command(self, msg: str) -> bool:
    """Send file-related command using basic socket"""
    if not self.connected or not self.basic_socket:
        raise ConnectionError("Not connected to server")

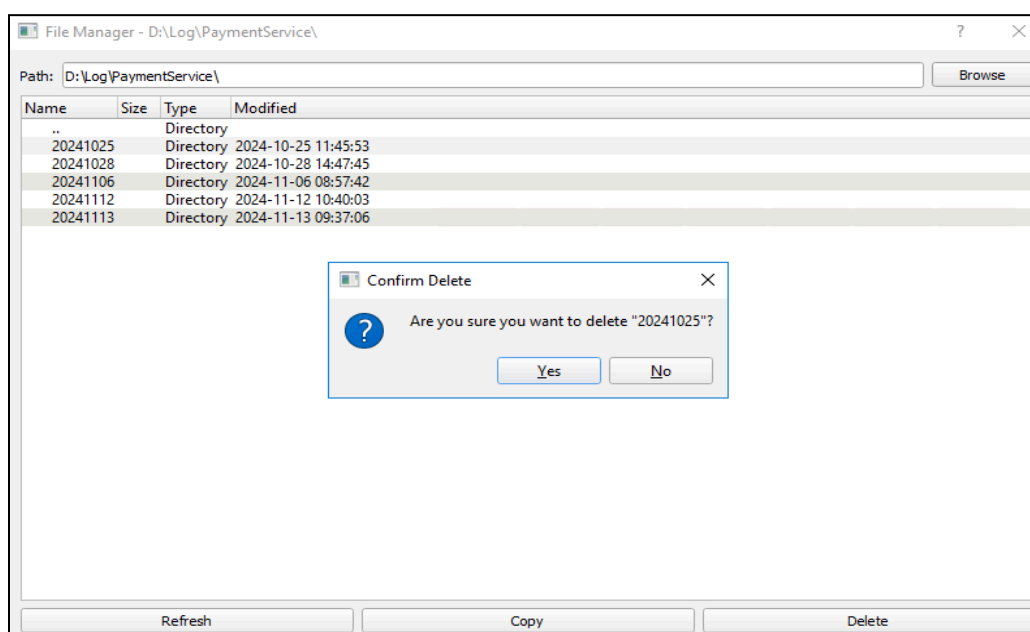
    try:
        self.basic_socket.sendall(msg.encode("utf-8"))
        with open("temp_file", 'wb') as f:
            while True:
                chunk = self.basic_socket.recv(8192)
                if chunk.endswith(b'<<END>>'):
                    f.write(chunk[:-7])
                    break
                if not chunk:
                    break
                f.write(chunk)
            return True
    except Exception as e:
        raise ConnectionError(f"File command error: {str(e)}")
```

#### 4. Xử lý phía Server:

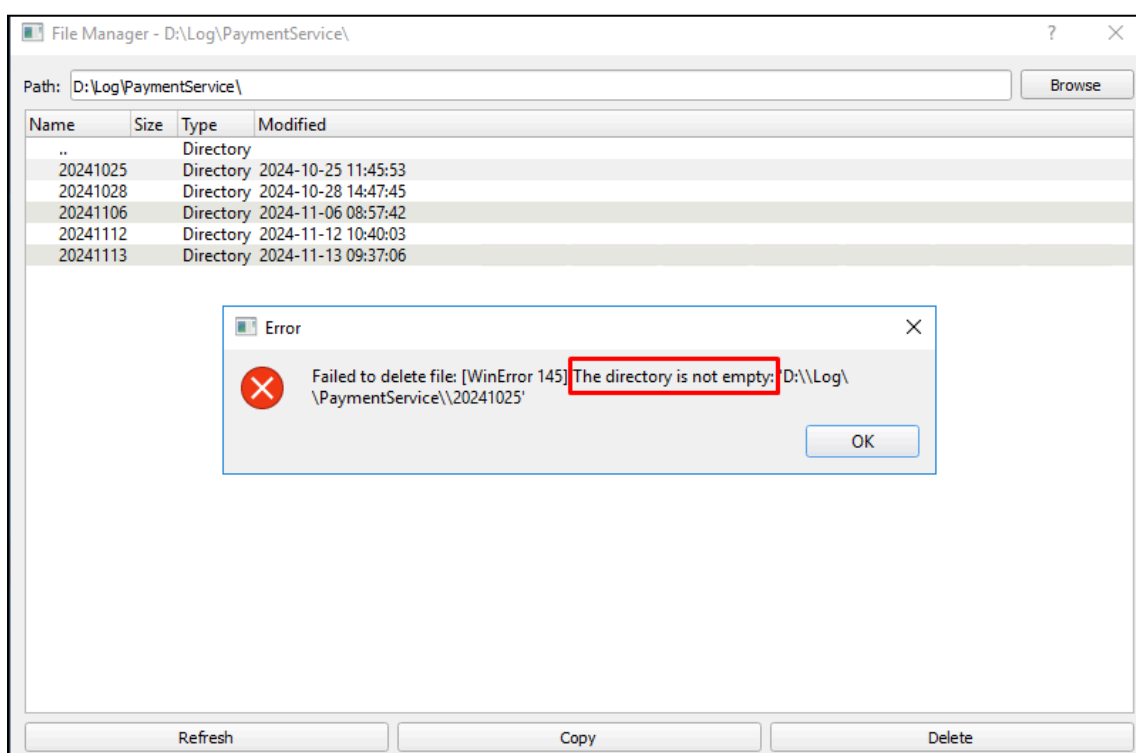
```
def send_file(self, conn: socket.socket, file_path: str):
    try:
        with open(file_path, 'rb') as f:
            while True:
                chunk = f.read(8192)
                if not chunk:
                    break
                conn.sendall(chunk)
            conn.sendall(b'<<END>>')
    except Exception as e:
        try:
            conn.sendall(b'ERROR: ' + str(e).encode())
        except:
            pass
```

### Chức Năng 3. Xóa (delete) file ở Server

1. Khi click vào 1 item trong danh sách và ấn nút **Delete** thì sẽ hiện popup xác nhận xóa.



- Khi người dùng xác nhận, chương trình kiểm tra nếu folder tồn tại folder hoặc file con sẽ hiển thị lỗi bằng function `delete_file`.



2. Xử lý phía Client: khi call hàm **delete\_file** sẽ check nếu là thư mục cha hoặc k tồn tại thì return, nếu là file thì tạo ra fullpath và show con confirm box cho người

dùng xác nhận, nếu Yes thì gửi command **delete/fulpath** lên server. Nếu server trả về thành công thì call hàm **refresh\_files** để cập nhật lại danh sách.

```
def delete_file(self):
    try:
        current_item = self.treeWidget.currentItem()
        if not current_item:
            QMessageBox.warning(self, "Warning", "Please select a file or folder to delete")
            return

        file_name = current_item.text(0)
        if file_name == "..":
            return

        full_path = os.path.join(self.current_path, file_name)

        if QMessageBox.question(
            self, 'Confirm Delete',
            f'Are you sure you want to delete "{file_name}"?',
            QMessageBox.Yes | QMessageBox.No
        ) == QMessageBox.Yes:
            try:
                response = self.network.send_message(f'files//delete//{full_path}')
                data = json.loads(response)

                if data.get('status') == 'success':
                    QMessageBox.information(self, "Success", data.get('message', 'File deleted successfully'))
                    self.refresh_files()
                else:
                    raise Exception(data.get('error', 'Failed to delete file'))

            except json.JSONDecodeError:
                raise Exception("Invalid server response")

    except Exception as e:
        QMessageBox.critical(self, "Error", f"Failed to delete file: {str(e)}")
```

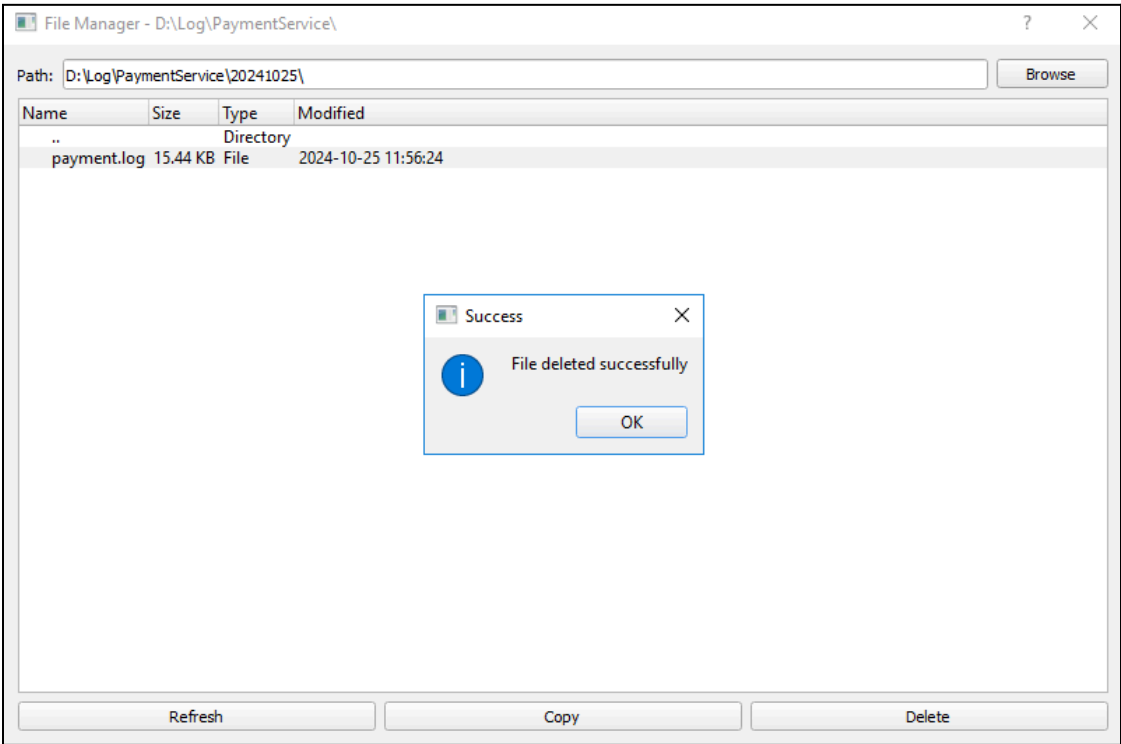
### 3. Xử lý phía Server:

```
def delete_file(self, file_path: str) -> str:
    """Delete file or directory and return JSON response"""
    try:
        if not os.path.exists(file_path):
            return json.dumps({
                "status": "error",
                "error": "File not found"
            })

        if os.path.isfile(file_path):
            os.remove(file_path)
        elif os.path.isdir(file_path):
            os.rmdir(file_path)

        return json.dumps({
            "status": "success",
            "message": "File deleted successfully"
        })
```

4. Nếu thoả hết điều kiện thì thực hiện xóa file và hiển thị kết quả:



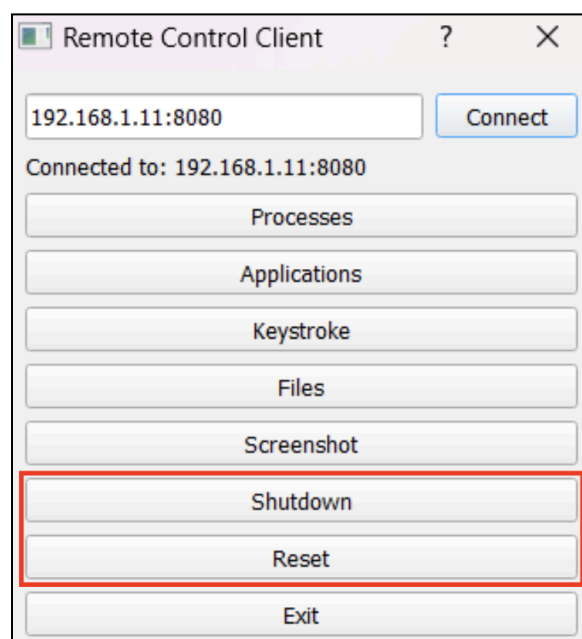
5. Chức Năng Shutdown và Reset

a. Tổng Quan Về Cơ Chế

| Hàm                   | Thiết bị | Chức năng                                            |
|-----------------------|----------|------------------------------------------------------|
| reset()               | client   | Thực hiện các nhiệm vụ gửi lệnh reset cho server.    |
| shutdown()            | client   | Thực hiện các nhiệm vụ gửi lệnh shutdown cho server. |
| QMessageBox.question  | client   | Tạo bảng thông báo để xác nhận lệnh.                 |
| send_message()        | client   | Chọn socket phù hợp với lệnh của người dùng          |
| _send_basic_message() | client   | Gửi nội dung lệnh reset (hoặc shutdown) cho socket.  |

| Hàm                 | Thiết bị | Chức năng                                                                                                          |
|---------------------|----------|--------------------------------------------------------------------------------------------------------------------|
| setTimeout()        | client   | Thiết lập thời gian chờ cho socket (được cấu hình là 300ms).                                                       |
| sendall()           | client   | Gửi lệnh reset (hoặc shutdown) vào socket.                                                                         |
| time.time()         | client   | Lấy giá trị thời gian hiện tại.                                                                                    |
| handle_connection() | server   | Điều hành và thực hiện các chức năng nhận và thực thi từ socket.                                                   |
| recv()              | server   | Hàm của socket để nhận thông tin từ socket đồng thời giải tin hiệu từ bit thành byte.                              |
| decode().strip()    | server   | chuyển mã từ byte sang bảng chữ cái và cắt bỏ những khoảng trống ở đầu và cuối chuỗi để tạo 1 câu lệnh hoàn chỉnh. |
| os.system()         | server   | Truy cập vào hệ thống mà server đang hoạt động thông qua thư viện os.                                              |

## b. Giao Diện



## c. Cách Hoạt Động Chính

### Chức Năng 1. Reset Server

1. Nhấn nút **Reset** ở màn hình chính.
2. Xử lý phía Client:
  - Cấu hình phím chọn “reset” ở client sẽ gọi đến hàm reset khi người dùng nhấn vào phím “reset”

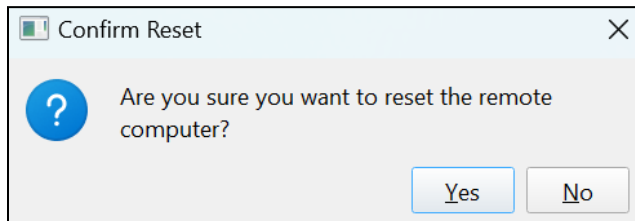
```
def setup_connections(self):  
    self.btn_cap.clicked.connect(self.capture)  
    self.btn_process.clicked.connect(self.process)  
    self.btn_app.clicked.connect(self.app)  
    self.btn_key.clicked.connect(self.key)  
    self.btn_connect.clicked.connect(self.connect)  
    self.btn_shutdown.clicked.connect(self.shutdown)  
    self.btn_reset.clicked.connect(self.reset)  
    self.btn_files.clicked.connect(self.show_files)  
    self.btn_exit.clicked.connect(self.exit)
```

- Hàm reset sau đó sẽ được thực hiện:

```
def reset(self):  
    if not self.check_connection():  
        self.show_error("Not connected to server")  
        return  
  
    if QMessageBox.question(self, 'Confirm Reset',  
        'Are you sure you want to reset the remote computer?',  
        QMessageBox.Yes | QMessageBox.No) == QMessageBox.Yes:  
        try:  
            self.network.send_message('reset')  
        except Exception as e:  
            self.show_error(f"Reset failed: {str(e)}")
```



- Sau khi kiểm tra sự kết nối giữa client và server vẫn sẵn sàng, hàm `QMessageBox.question()` gọi ra bảng thông báo để xác nhận rằng người dùng muốn thực hiện chức năng reset.



- Khi người dùng chọn “Yes”, khi này đối tượng network sẽ thực hiện hàm `send_message()` với thông điệp ‘reset’ đính kèm.

```
def send_message(self, msg: str) -> str:
    if not self.connected:
        raise ConnectionError("Not connected to server")
    try:
        # Special handling for capture command (screenshot)
        if msg == 'capture':
            self.basic_socket.settimeout(self.timeout)
            self.basic_socket.sendall(b'capture')
            return 'ok'

        # Use main socket for process and app commands
        if msg.startswith(('process//', 'app//')):
            return self._send_main_message(msg)

        # Use basic socket for everything else
        else:
            return self._send_basic_message(msg)
    except Exception as e:
        # Don't set connected to False here
        # Just raise the error and let the calling method handle it
        raise ConnectionError(f"Communication error: {str(e)}")
```

- Lúc này hàm `send_message()` kiểm tra thông điệp 'reset' với các điều kiện được thiết lập sẵn và hàm `_send_basic_message()` cùng với message 'reset'.

```
def _send_basic_message(self, msg: str) -> str:
    try:
        self.basic_socket.settimeout(self.timeout)
        self.basic_socket.sendall(msg.encode("utf-8"))
        data = ""
        start_time = time.time()
        while True:
            try:
                chunk = self.basic_socket.recv(self.buffer_size)
                if not chunk:
                    break
                data += chunk.decode("utf-8")
                if len(chunk) < self.buffer_size:
                    break
                if time.time() - start_time > self.timeout: # Check timeout
                    raise socket.timeout("Operation timed out")
            except socket.timeout:
                if data: # If we have some data, consider it complete
                    break
                raise # Re-raise the timeout if no data received
        return data
```

- Sau khi kiểm tra, tự động thiết lập thời gian chờ timeout cho `basic_socket`. Lúc này sẽ kiểm tra xem thời gian chờ timeout có bị vượt quá hay không, nếu có thì đưa ra ngoại lệ `socket.timeout`

```
except socket.timeout:
    if data: # If we have some data, consider it complete
        break
    raise # Re-raise the timeout if no data received
return data
```

- Nếu có `socket.timeout` thì kiểm tra kết quả nhận từ server, nếu không, sẽ ném lại ngoại lệ timeout.
- Nhận kết quả từ server, nếu thất bại thì hiện mã lỗi.

### 3. Xử lý phía Server:

```
def handle_connection(self, conn: socket.socket, addr: str, is_main: bool):
    try:
        while self.running:
            try:
                data = conn.recv(1024)
                if not data:
                    logging.info(f"{connection_type} connection closed by client {addr}")
                    break

                # Handle ping for basic connection
                if not is_main and data == b'ping':
                    conn.sendall(b'pong')
                    continue

                command = data.decode().strip()
```

- Ở phía Server, dữ liệu từ socket sau khi nhận từ client sẽ được gán vào biến data. Và tiếp tục đc giải mã đồng thời cắt bỏ những khoảng trắng ở đầu và cuối chuỗi thông qua hàm strip() để có được dữ liệu rõ ràng lưu vào biến command.

```
try:
    if command == "key":
        self.handle_keylogger_commands(conn, parts[1])
    elif command == "files":
        self.handle_file_commands(conn, parts[1], parts[2:])
    elif command == "capture":
        self.capture_screen(conn)
    elif command == "shutdown":
        conn.sendall(b'ok')
        os.system("shutdown /s /t 1")
    elif command == "reset":
        os.system("shutdown /r /t 1")
    else:
        conn.sendall(b'404')
except Exception as e:
    logging.error(f"Error handling basic command {command}: {str(e)}")
    conn.sendall(b'404')
```

- Ứng với điều kiện 'reset' phù hợp với command sẽ thực hiện reset. Với sự hỗ trợ của hàm system trong thư viện os, ta có thể truy cập chức năng reset hoặc shutdown của máy tính.
- Nếu command không trùng với bất cứ điều kiện nào thì server sẽ gửi lỗi 404 về cho client.

## Chức Năng 2. Shutdown Server

Giống như reset, nhưng phía Client sẽ gửi thông điệp 'shutdown' cho Server và hoạt động còn lại tuân tự như chức năng 'reset'.