Explication code

In this document we will explain how my program works.

At first the code starts with the imports as seen in the document "installation.md" which will allow to get the libraries previously installed.

We will start by explaining the libraries used.

- PyQt5

PyQt5 is a library for developing graphical user interface (GUI) applications in Python. It allows you to create graphical applications with windows, menus, toolbars, and other elements of the user interface using the Python programming language.

- socket

In Python, a socket is a low-level network communication endpoint. It is used to send and receive data across a network using the network's sockets (TCP/IP protocol). A socket is composed of an IP address and a port number, which identifies a specific process running on a host (computer).

- time

The "time" library allows you to manipulate dates and times. It provides functions that allow you to get the current time, measure periods of time, convert dates and times to strings and vice versa, and perform other time-related operations.

- psutil

psutil (process and system utilities) is a Python library that allows you to collect information about running processes and system resource usage, such as memory, CPU, disks, and networks.

- platform

The Python "platform" library provides information about the operating system currently running, such as the platform name, the operating system version, and other technical details.

Voilà pour les informations sur les librairies passons maintenant au serveur.

```
serveur_socket = socket.socket()
serveur_socket.bind((host, port))
serveur_socket.listen(5)
```
This code creates a socket object and sets it up to listen for incoming connections at the specified IP address and port.

The "bind()" method associates the socket object with the specified IP address and port.

The "listen()" method tells the socket to start listening for incoming connections.

The "5" parameter indicates the maximum number of pending connections that the socket can accept.

```
while msg != 'arret':
```
If the message being sent is not "arret", this code continues with...

```
 msg = conn.recv(1024).decode()
    message = msg + time.strftime("  reçu à %H:%M")
    print('from Client: ' + message)
```
This code reads messages sent by the client once a connection has been established. The "recv()" method of the "conn" socket object reads the data sent by the client. The "1024" parameter indicates the maximum size of data to read in bytes.

The data is returned as bytes, so the "decode()" method is used to decode it into a string.

```
 if msg == 'cpu':
        reply = str(cpu)
        conn.send(str(cpu).encode())
        print("CPU envoyé" + time.strftime("  à %H:%M"))
```
In this part, if the message is "cpu", the server uses the "cpu_percent()" function from the psutil library to get the current CPU usage.

After the data has been sent, the server prints a message indicating that the data was sent with the current time (obtained using the "strftime()" function from the "time" library).

```
    elif msg == 'ram':
        reply = str(ram)
        conn.send(str(ram).encode())
        print("RAM envoyé" + time.strftime("  à %H:%M"))
```
Similarly for ram.

```
    elif msg == 'os':
        reply = str(os)
        conn.send(str(os).encode())
        print("OS envoyé" + time.strftime("  à %H:%M"))
```
Similarly for l'os.

```
    elif msg == 'name':
        reply = str(name)
        conn.send(str(name).encode())
        print("Name envoyé" + time.strftime("  à %H:%M"))
```
Similarly for name.

```
    elif msg == 'ip':
        reply = str(ip)
        conn.send(str(ip).encode())
        print("IP envoyé" + time.strftime("  à %H:%M"))
```
Similarly for Ip.

```
    elif msg == 'ping':
        temp = msg.split()[1]
        result = os.system("ping -c 1" + temp)
        if result == 0:
            conn.send("{} atteint".format(temp).encode())
        else:
            conn.send("inconnu".encode())
```
This code sends a message to the client when the message received from the client is "ping". The first part of the message is supposed to be the word "ping" and the second part is supposed to be the IP address or hostname to "ping".

The server then uses the "system()" function from the "os" library to execute the "ping" command with the "-c 1" option and the specified IP address or hostname.

The "system()" function returns a result indicating whether the command was successful or not. If the result is "0", it means that the command was successful and the IP address or hostname was reached.

```
conn.close()
```
The "close()" method of the "conn" socket object ends the connection with the client.

Now, let's move on to explaining the client.

```
class client(QMainWindow):

    def __init__(self):
        super().__init__()
```

This line of code defines a class named client which inherits from the QMainWindow class from the Qt library.

```
self.setWindowTitle("SAE32")
```

It sets the title of the main window of the object upon its creation.

```
widget = QWidget()
        self.setCentralWidget(widget)

        grid = QGridLayout()
        widget.setLayout(grid)
```

These three lines of code create a base widget. By using a central widget, you can add other widgets to the main window.

```
        os = QPushButton("OS")
        ram = QPushButton("RAM")
        cpu = QPushButton("CPU")
        ip = QPushButton("IP")
        name = QPushButton("NAME")
```

These five lines of code create five command buttons with the labels "OS", "RAM", "CPU", "IP", and "NAME".

The command buttons are created using the QPushButton class from the Qt library.

```
disconnect = QPushButton("DISCONNECT")
        disconnect.clicked.connect(self.close)
        arret = QPushButton("ARRET")
        arret.clicked.connect(self.close)
        self.__ping = QLineEdit("")
        ping1 = QPushButton("PING")
        clear = QPushButton("CLEAR")
```

A "DISCONNECT" command button that closes the main window of the object when clicked.

A editable text field "QLineEdit" which can be used to enter text.

A "PING" and "CLEAR" command button that performs an action when clicked.

The clicked.connect method is used to connect the "clicked" event (a click on the button) to a function.

In summary, these lines of code create six widgets that will be added to the user interface of the object of the client class later in the code.

```
        self.lab1 = QLabel("")
        self.lab2 = QLabel("")
        self.lab3 = QLabel("")
        self.lab4 = QLabel("")
        self.lab5 = QLabel("")
        self.lab6 = QLabel("")
```

Creates a new QLabel object and assigns it to a variable called labx. The empty string within parentheses is the text that will be displayed by the label.

```
        grid.addWidget(os, 0, 1, 1, 2)
        grid.addWidget(ram, 2, 1, 1, 2)
        grid.addWidget(cpu, 4, 1, 1, 2)
        grid.addWidget(ip, 6, 1, 1, 2)
        grid.addWidget(name, 8, 1, 1, 2)
        grid.addWidget(disconnect, 10, 1, 1, 2)
        grid.addWidget(arret, 11, 1, 1, 2)
        grid.addWidget(self.__ping, 12, 1, 1, 2)
        grid.addWidget(ping1, 13, 1, 1, 2)
        grid.addWidget(clear, 15, 1, 1, 2)

        grid.addWidget(self.lab1, 1, 2)
        grid.addWidget(self.lab2, 3, 2)
        grid.addWidget(self.lab3, 5, 2)
```

```
        grid.addWidget(self.lab4, 7, 2)
        grid.addWidget(self.lab5, 9, 2)
        grid.addWidget(self.lab6, 14, 1, 1, 2)
```

The implementation of the buttons and the arrangement of the widgets in a specific order.

```
        os.clicked.connect(self.__actionOS)
        ram.clicked.connect(self.__actionRAM)
        cpu.clicked.connect(self.__actionCPU)
        ip.clicked.connect(self.__actionIP)
        name.clicked.connect(self.__actionNAME)
        arret.clicked.connect(self.__actionARRET)
        ping1.clicked.connect(self.__actionPING)
        clear.clicked.connect(self.__actionCLEAR)
```

For example, when the os widget is clicked, it emits the clicked signal which is connected to the self.__actionOS slot.

This means that when os is clicked, the __actionOS function will be executed.

Similarly, when the cpu widget is clicked, it emits the clicked signal which is connected to the self.__actionCPU slot, which executes the __actionCPU function.

```
    def __actionOS(self):
        message = "os"
        client_socket.send(message.encode())
        print("requête os envoyée")
        data = client_socket.recv(1024).decode()
        dtime = data + time.strftime(" reçu à %H:%M", time.localtime())
        self.lab1.setText(dtime)
```

This function is executed when the os widget is clicked. It does the following:

It sends a message to the server using a client socket (client_socket). The message sent is simply the string "os".

It uses the setText method of the lab1 object to display the contents of the dtime variable in the label.

```
    def __actionRAM(self):
        message = "ram"
        client_socket.send(message.encode())
        print ('requête ram envoyée')
        data = client_socket.recv(1024).decode()
        dtime = data + time.strftime("  reçu à %H:%M")
        self.lab2.setText(dtime)


    def __actionCPU(self):
        message = "cpu"
        client_socket.send(message.encode())
        print ('requête cpu envoyée')
        data = client_socket.recv(1024).decode()
        dtime = data + time.strftime("  reçu à %H:%M")
        self.lab3.setText(dtime)


    def __actionIP(self):
        message = "ip"
        client_socket.send(message.encode())
        print ('requête ip envoyé')
        data = client_socket.recv(1024).decode()
        dtime = data + time.strftime("  reçu à %H:%M")
        self.lab4.setText(dtime)


    def __actionNAME(self):
        message = "name"
        client_socket.send(message.encode())
        print ('requête name envoyé')
        data = client_socket.recv(1024).decode()
        dtime = data + time.strftime("  reçu à %H:%M")
        self.lab5.setText(dtime)


    def __actionARRET(self):
        message = "arret"
        client_socket.send(message.encode())
        print ('requête arret envoyé')
        data = client_socket.recv(1024).decode()
        dtime = data + time.strftime("  reçu à %H:%M")
```

The functioning is the same for these functions.

```
    def __actionPING(self):
        message = "ping " + self.__ping.text()
        print (message)
        client_socket.send(message.encode())
        print ('ping envoyé')
        data = client_socket.recv(1024).decode()
        dtime = data + time.strftime("  reçu à %H:%M")
        self.lab6.setText(dtime)
```

The __actionPING function is executed when the ping1 widget is clicked. It does the following:

It creates a message string by concatenating the "ping" string with the text entered in the __ping widget.

It sends the message to the server by encoding the string as bytes and using a client socket (client_socket).

It displays a message in the console indicating that the ping has been sent.

It concatenates the server's response with the current time and stores it in the dtime variable.

It uses the setText method of the lab6 object to display the contents of the dtime variable.

```
    def __actionCLEAR(self):
        self.lab1.setText("")
        self.lab2.setText("")
        self.lab3.setText("")
        self.lab4.setText("")
        self.lab5.setText("")
        self.lab6.setText("")
```

The __actionCLEAR function seems to be executed when the clear widget is clicked.

When this happens, the function uses the setText method of each QLabel object to replace the text of each label with an empty string. This clears the text from the labels and makes them empty.

```
if __name__ == "__main__":

    host = socket.gethostname()
    port = 6000

    print("client se connecte au serveur")
    client_socket = socket.socket()

    client_socket.connect((host, port))
    print("Client connecté au serveur")

    app = QApplication(sys.argv)

    window = client()
    window.show()
    app.exec()
```

When the script is run, it performs the following actions:

It defines the host and port variables using the gethostname and socket functions.

It creates a client socket and connects to the server using the host variable and the port variable.

It creates an instance of the client object (which is defined elsewhere in the code).

It displays the user interface window by calling the show method of the window object.

It runs the application by calling the exec method of the app object.