SAÉ3.02
Developer documentation

Louis DESVERNOIS

16th December 2022

Contents

List of Listings

1 Introduction

This document is the developer documentation for the remote control program made for the SAÉ3.04. On both the server and client, setting the `DEBUG` constant in the beginning of each file to `True` will change some default behaviour

2 Server

The server consists of two Python files, `main.py` which contains the main server class and `action.py` which is used to obtain information about the machine and execute commands.

2.1 Server class

The server is implemented using a Python class and is relatively simple, as it only accepts one client at a time.

```
class Server:
    def __init__(self, host: tuple):
        self.host = host
        self.killed = False
```

Listing 1: Server constructor

The server class only takes a tuple `(host, port)` as an argument. It does not connect automatically, instead the `start()` method has to be used, this allows for a "clean" shutdown of the server if we except for a `KeyboardInterrupt`.

```
if __name__ == "__main__":
    server = Server((host, port))
    try:
        server.start()
    except KeyboardInterrupt:
        logging.info("KeyboardInterrupt: killing server...")
        server.kill()
```

Listing 2: Starting the server

Once the `start()` method is called, the server creates the socket, listens on the specified port. Once a client is connected, it will wait for incoming messages.

```python
def start(self):
    while not self.killed:
        self.server = socket.socket()
        # While True loop
        self.__bind(self.host)
        self.server.listen(1)
        message = ""
        while not self.killed and message != "reset":
            self.client, addr = self.server.accept()
            message = ""  # reset so we can reconnect
            while (
                not self.killed and message != "reset" and message != "disconnect"
            ):
                # Here we wait for a message
                self.__handle(message, addr)
            # Close connection to client
        # Close the server
    # Kill the process
```

Listing 3: start() method simplified

The server will try to indefinitely bind itself to the specified port (at line 5 in Listing 3), this is done to ensure that the server can rebind to the port after a reset.
Once a message is received it is sent to the `__handle(message, addr)` (line 15), this method serves no purposes other than code readability and maintainability. *This is where new features would be added.*

```python
def __handle(self, message: str, addr: tuple):
    if message == "kill":
        logging.info(f"Kill requested by {addr}...")
        self.killed = True  # avoid adding a condition to while loops
    elif message == "reset":
        logging.info(f"Client at {addr} requested a reset.")
    elif message == "info":
        self.client.send(("info" + json.dumps(actions.get_all())).encode())
    elif message[:7] == "command":
        command = json.loads(message[7:])
        rep = "cmmd"
        if command["shell"] == "dos":
            if sys.platform == "win32":
                rep += actions.send_command(command["com"], "dos")
            else:
                rep += "Cannot execute a DOS command on this operating system."
        # ... More elif to handle other OSs
        self.client.send(rep.encode())
```

Listing 4: Handle method simplified

The first two conditions do not do much except print a log in the console. However, if the server receive "info" from a client, it replies with a JSON encoded dict object containing information about the machine (the info is gathered using `action.py`).
If a message starts with the word "command", the server will try executing the given command if the shell selected by the user is available and send the output back to the client.

3 Client

The client consists of two classes the spans across two python files, `main.py` contains the GUI, and the file `connection.py` is the server connection, which is a class that allows the client to connect to multiple servers.

3.1 Connection class

This class handles all the communication to the server, including sockets and all actions. The connection object takes an IP address, a port as well as two GUI objects to write information to. This makes showing data to the user as soon the message is received.

3.1.1 Initialization

```python
def __init__(
    self, host: str, port: int, label_info: QLabel, label_command: QTextBrowser
) -> None:
    self.client = socket.socket()
    self.msgsrv = ""
    self.addr = (host, port)
    self.info = {}

    self.label_info = label_info
    self.label_command = label_command

    self.__connect()
    self.send("info")
```

Listing 5: Connection init method

`__connect()` is used to connect to the server, having a separate method allows reconnecting to the server after a disconnect. Once we are connected to the server, an "info" request is automatically sent[1].

```python
def __connect(self) -> None:
    self.client.connect(self.addr)
    # Flag to kill the handle thread
    self.__killed = False
    self.msgsrv = ""
    # Starting handle thread for incoming messages
    client_handler = threading.Thread(target=self.__handle, args=[self.client])
    client_handler.start()
```

Listing 6: __connect method

This method connects to the server and start the receive thread, the `self.killed` variable is used as a condition to keep the reception running, allowing it to be killed easily.

[1] This can cause the client to hang at startup since the connection is not threaded

3.1.2 Threaded code

```python
def __handle(self, conn) -> None:
    while self.msgsrv != "kill" and self.msgsrv != "reset" and not self.__killed:
        try:
            self.msgsrv = conn.recv(4096)
        except Exception as e:
            logging.error(f"Receive failed: {e}")
            break
        logging.debug(f"Size of the recieved message is {len(self.msgsrv)}")
        if not self.msgsrv:
            break  # prevents infinite loop on disconnect, auto disconnect clients
        self.msgsrv = self.msgsrv.decode()
        logging.info(f"Message from {self.addr}: {self.msgsrv}")

        if self.msgsrv[:4] == "info":
            self.info = json.loads(self.msgsrv[4:])
            logging.info("Got the server information.")
            self.label_info.setText(self._info_string())
        elif self.msgsrv[:4] == "cmmd":
            logging.info("Got a command output from the server.")
            self.label_command.append(self.msgsrv[4:])

    logging.debug(f"Closing handle thread for {self.addr}")
    self.client.close()
    self.__killed = True
```

Listing 7: Threaded code

The client receives as long as the server is not killed, reset or the `self.killed` variable is not set to `True`. Once the type of message is detected, we log the message and set the text in one of the GUI element the class has access to[2]. The `send` method checks if the connection is available before transmitting data. Additional methods are available to the client such as `disconnect`, `reconnect`, `kill` or `reset`.

3.2 GUI

This application is using tabs easily access servers with one window. Each tab is its own `QHBoxLayout` that contains two `QGridLayout`. The widgets and the connection are stored in a `dict` that is appended to a list (of tabs). The `_create_tab()` is handling all the actual widget placement.

[2]The _info_string() method returns a formatted string of the raw JSON data received from the server

3.2.1 Threaded code

```python
def _create_tab(self, name: str, ip: str, port: int):
    Label_info = QLabel("Placeholder\nPlaceholder\nPlaceholder\nPlaceholder\nPlaceholder")
    TextBrowser_resultcommand = QTextBrowser()
    # Never crash when connectiong to a server, instead send notification to user
    try:
        logging.info(f"Connecting to {name}, {ip}:{port}...")
        conn = connection.Connection(
            ip, port, Label_info, TextBrowser_resultcommand
        )
    except Exception as e:
        logging.error(f"Connection to {name}, {ip}:{port} failed! ({e})")
        self.error_box(
            e, f"Connection to {name} ({ip}:{port}) failed!"
        )
    else:
        self.tabs.append(
            {
                "widget": QWidget(),
                "widget_left": QWidget(),
                "widget_right": QWidget(),
                "Button_info": QPushButton("Refresh information"),
                "Label_info": Label_info,
                "ComboBox_shell": QComboBox(),
                "LineEdit_sendcommand": QLineEdit(),
                "Button_clear": QPushButton("Clear"),
                "TextBrowser_resultcommand": TextBrowser_resultcommand,
                "Button_disconnect": QPushButton("Disconnect"),
                "Button_kill": QPushButton("Kill"),
                "Button_reset": QPushButton("Reset"),
                "Button_reco": QPushButton("Reconnect"),
            }
        )
        tab = self.tabs[-1]
        # We then use the tab variable to access all of the tab's elements
        ...
```

Listing 8: Beginning of the _create_tab() method

First we create a `QLabel` and a `QTextBrowser` to use for the creation of the connection. For the actual connection we except all exceptions and store the message is en `e` variable, so we can notify the user of the error without crashing the application, the tab is only created if not except are encountered while connecting. Other methods are briefly explained in the source code.