SAE 3.02

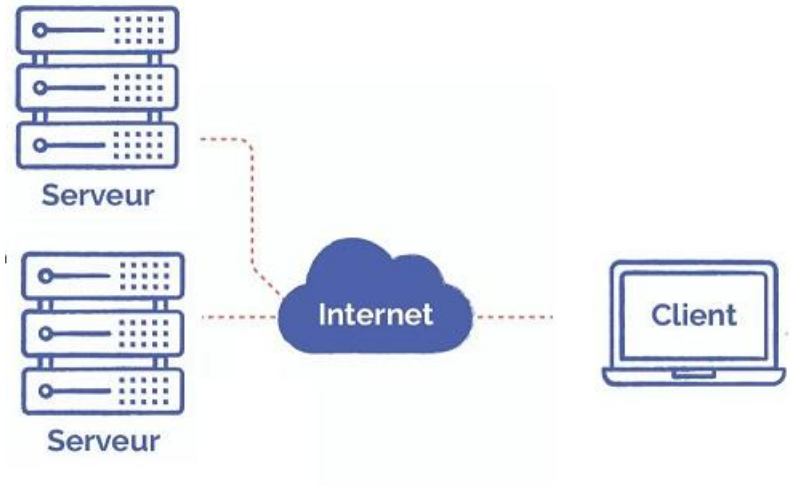
SHELL
Technical Document

DAOUDI KHALIL

In this project we want to create a client-server system to send system commands to monitor and diagnose machines or servers remotely.

I will therefore present in this document the technical side of the project.

I will first present the architecture of my project:

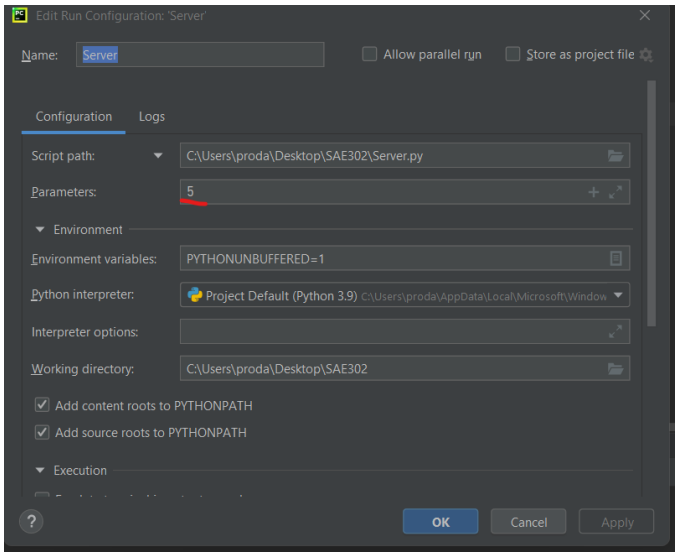


Here is a server on the right of the diagram it will be launched using the script Server.py then will be waiting for a connection from a client such as for example on the left of the diagram a client will connect to the server launch using its graphical interface. It will therefore be necessary to indicate the port and the IP address of the server. Then it will be possible to retrieve information from the server using certain commands that we will type in the graphical interface.

1) Starting the server

```
PORT = int(sys.argv[1])
ADDRESS = '0.0.0.0'
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind((ADDRESS, PORT))
server.setblocking(False)
clients = []
arret = False
reset = False
```

For the server to start, you must indicate its IP address and its port as you see above, for the IP address I put 0.0.0.0 to indicate all the IP addresses then the port a default argument that I chose on PyCharm for example 5.



Then I use a bind because it must be imposed on which port and IP address it will listen to otherwise it will choose those by default.

Then I use a setBlocking because it will no longer wait for a connection but rather question who will connect to its server, so it is more interesting than waiting.

```
while True:
    while not arret:
        try:
            server.listen(1)
            clientsock, clientAddress = server.accept()
            clients.append(clientsock)
            newthread = ClientThread(clientAddress, clientsock)
            newthread.start()
        except:
            pass
```

Then as long as the server is not stopped, we will use exceptions to avoid errors, so we try to listen to at least one client then create a thread that will run in a loop when starting the server which will constantly listen to a customer.

2) *Kill, Disconnect, Reset*

```
def run(self):
    global reset
    global arret
    print ("Connection from : ", clientAddress)
    #self.csocket.send(bytes("Hi, This is from Server..",'utf-8'))
    while True:
        try:
            data = self.csocket.recv(1024)
            msg = data.decode()
            if len(msg) > 0:
                if msg.lower() == 'disconnect':
                    self.csocket.send('disconnect'.encode())
                    self.csocket.close()
                    break
                elif msg.lower() == 'kill':
                    self.csocket.send("kill".encode())
                    arret = True
                    server.close()
                    break
                elif msg.lower() == 'reset':
                    self.csocket.send("reset".encode())
                    server.close()
                    arret = True
                    reset = True
                else:
                    self.csocket.send(cmd(msg).encode())
        except:
            pass
    print ("Client at ", clientAddress, " disconnected...")
```

I am using a loop that will execute an exception, so it will try to decode the message received by the client and then check if it is equal to kill, disconnect or reset.

If the message is equal to kill then the server is completely shut down, if the message is equal to disconnect then we disconnect the connection or if the message is equal to reset then we close all connections then restart the socket.

3) How Orders Work

First, I will show you my command help because it displays all the possible commands for the server.

```
    if cmd.lower() == 'help':
        return ("""CMD HELP:
- IP
- NAME
- RAM
- OS
- CPU
- KILL
- RESET
- DISCONNECT
- PING
- CLEAR
- PYHTON --VERSION
- DOS:
- Linux:
- PowerShell:""")
```

To execute all this command, I used the subprocess library.

Here is an example:

```
if cmd.lower() == 'python --version':
    x = str(subprocess.check_output('python --version', shell=True))
```

The subprocess module allows you to start new processes, connect them to input/output/error pipes, and get their return codes.