This code demonstrates a basic client-server communication using Python's `socket` module, where a server listens for a connection and exchanges messages with a client. The use of threading allows both server and client functions to run almost simultaneously in the same script. Here's a line-by-line explanation:

```
import socket
import threading
import time
```

- **Imports `socket`, `threading`, and `time` modules:**
    - `socket`: Provides low-level networking interface for communication between computers.
    - `threading`: Allows concurrent execution of threads, which helps in running server and client simultaneously.
    - `time`: Used here to introduce delays, ensuring the server starts before the client tries to connect.

```
PORT=8080
```

- **Defines `PORT` variable**: Specifies the port number (8080) on which both the server and client will communicate. This is a common way to specify a port number, and `8080` is often used for web servers in testing.

Server Function

```
def run_server():
```

- **Defines the server function**: This function will handle the server-side operations.

```
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- **Creates a server socket**:
    - `socket.AF_INET` specifies IPv4 addressing.
    - `socket.SOCK_STREAM` specifies TCP protocol, a connection-oriented protocol that ensures reliable data transmission.

```
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

- **Sets socket options**:
    - `SO_REUSEADDR` allows the socket to reuse the same address without waiting for the socket to time out, which is useful when running the server multiple times in a short period.

```
server_socket.bind(('0.0.0.0', PORT))
```

- **Binds the socket to an address and port**:
    - `'0.0.0.0'` allows the server to accept connections on all available network interfaces.
    - `PORT` is the port number defined earlier, making the server listen on port 8080.

```
server_socket.listen(1)
 print("Server listening on port", PORT)
```

- **Listens for incoming connections**:
    - `listen(1)` tells the server to allow only one connection in the queue.
    - Prints a message to confirm that the server is listening for connections on the

```
client_socket, addr = server_socket.accept()
 print("Connection from", addr)
```

- **Accepts a client connection**:
    - `accept()` blocks and waits until a client connects to the server.
    - `client_socket` is a new socket object to communicate with the client.
    - `addr` is the address of the client, which is printed for confirmation.

```
buffer = client_socket.recv(1024).decode()
```

```
 print(f"Server: {buffer}")
```

- **Receives data from the client**:
    - `recv(1024)` receives up to 1024 bytes of data from the client.
    - `decode()` converts the received byte data to a string.
    - Prints the received message to the console.

```
hello = "Hello from server"
 client_socket.send(hello.encode())
 print("Hello message sent from server")
```

- **Sends a response to the client**:
    - Prepares a response message `"Hello from server"`.
    - `send()` transmits the encoded message back to the client.
    - Prints a confirmation message that the server sent a response.

```
client_socket.close()
 server_socket.close()
```

- **Closes the sockets**:
    - `client_socket.close()`: Closes the client connection after communication ends.
    - `server_socket.close()`: Shuts down the server completely.

Client Function

```
def run_client():
    time.sleep(1)  # Wait for the server to start
```

- **Defines the client function**:
    - Adds a delay with `time.sleep(1)` to give the server enough time to start listening before the client tries to connect.

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- **Creates a client socket**:
    - Similar to the server, this creates a TCP/IP socket for the client.

```
try:
     client_socket.connect(('127.0.0.1', PORT))
 except ConnectionRefusedError:
     print("Connection failed. Is the server running?")
     return
```

- **Connects to the server**:
    - `connect(('127.0.0.1', PORT))` attempts to connect to the server at `127.0.0.1` (localhost) on port `8080`.
    - If the connection fails (for example, if the server is not running), it raises a `ConnectionRefusedError`, which is caught by the `except` block, and prints an error message.

```
hello = "Hello from client"
 client_socket.send(hello.encode())
 print("Client: Hello message sent")
```

- **Sends a message to the server**:
    - Prepares a message `"Hello from client"`.
    - `send()` transmits the encoded message to the server.
    - Prints a confirmation that the client has sent the message.

```
buffer = client_socket.recv(1024).decode()
 print(f"Client: {buffer}")
```

- **Receives a response from the server**:
    - `recv(1024)` waits to receive a response from the server, up to 1024 bytes.
    - `decode()` converts the byte data to a string.
    - Prints the received message, which should be `"Hello from server"`.

```
client_socket.close()
```

- **Closes the client socket**: Closes the client's connection after the message exchange.

Starting Server and Client Threads

```
server_thread = threading.Thread(target=run_server)
server_thread.start()
```

- **Creates and starts the server thread**:
  - `threading.Thread(target=run_server)`: Creates a new thread to run the `run_server` function. This allows the server to run in parallel with the client in the same script.
  - `start()` begins the execution of `run_server` in a separate thread.

```
time.sleep(1)  # Allow time for the server to start
```

- **Waits for the server to start**:
  - A brief delay (`time.sleep(1)`) is added to make sure the server is ready before running the client.

```
run_client()
```

- **Runs the client function**: After waiting, the client attempts to connect to the server and exchange messages.

```
server_thread.join()
```

- **Waits for the server thread to finish**:
  - `join()` ensures the main program waits until the `server_thread` completes before ending, so that the server shuts down gracefully after the client disconnects.

Summary

- This script creates a simple client-server interaction where:
  - The server listens on port 8080, waits for a client, receives a message, and responds.
  - The client connects to the server, sends a message, and waits for a response.
  - Threading allows both the server and client to run in the same script.