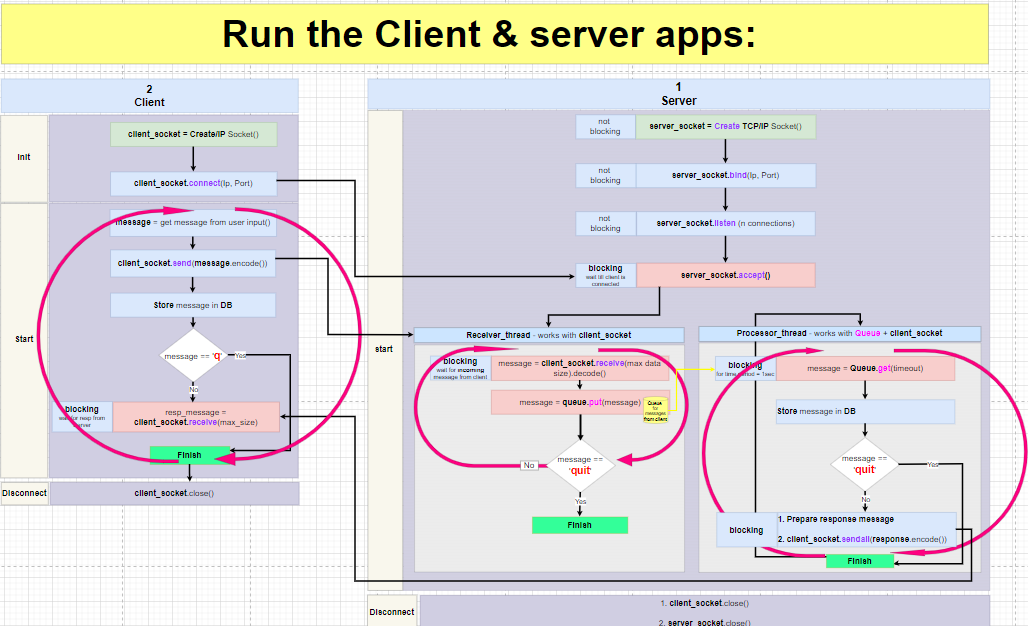
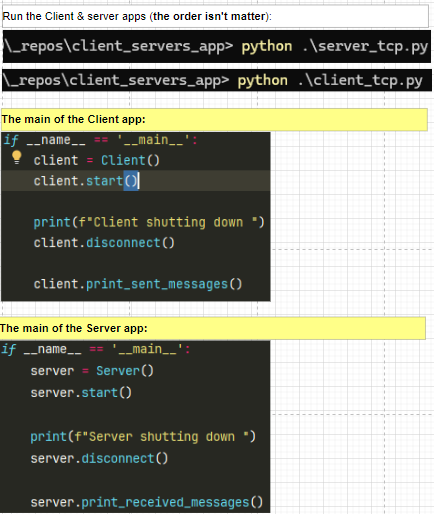
Simple Client Server application – based on **TCP** **/ IP** Sockets

This is a block diagram of the task:



These files we have in the project:

A screenshot of a computer

AI-generated content may be incorrect.

**TCP sockets:**

TCP – Transfer Control Protocol - Reliable network protocol:

1. Will be created a **socket** between Client and Server
2. This **socket** will be **opened** and **used** during the entire connection session with the **server.** Both Client and the Server must create a socket and must make a connection.
3. Client uses api: **connect()**
4. Server uses api: **accept()** which is blocking till the client side is connected. Server sets number of clients it is ready to serv.
5. All the IP packets will ride on this same **socket**
6. Client will expect to receive response messages – will be sure that these messages are sent (on this same **socket**) from Server that he knows and that he connected at the beginning of the session.
7. Client and server are 2 stable (same) partners during the entire session, Client can be sure that Server will not be replaced.
8. That is why when **socket** is of type TCP both Client and server must create a connection between them and then use this connection for sending / receiving messages. TCP is a **reliable** protocol and session and more operations ensure all TCP messages will arrive correctly, in the correct order, with correct content (will not be lost)

**UDP Sockets:**

**UDP – Universal Datagram Protocol**. It is fast but not reliable network protocol.

1. If we used socket of type **UDP** (Universal Datagram Protocol)

Client creates UDP socket, Client **doesn’t need** to perform **connect**! It starts **sending** messages right away towards Server. Client doesn’t even check that Server exists and that Server is listening to him. Every message is **independently** sent to the server. Every message then holds information about the server (**IP**, **PORT**).

When Client comes to extract the server’s response it gets 2 things:

1. Response message
2. Server’s address (as Client doesn’t know who responds to Client’s messages)

Lets here talk only about **TCP socket**:

Has 2 parts:

1. Server
2. Client

In this exp, both Client and the Server run on same PC (local host) therefore the use same IP and Port.

Ip: **127.0.0.1**

Port: **8820**

**Remember – run the Server first !!**

**Server:**

It is implemented by **2 threads**, these 2 threads will be created right after Client connects to the server:

1. **Receive** **& Store**:

* **Receive** – from client’s socket
* **Store** - in the server’s Q

This thread **listens on the socket** and once client message arrives, it does:

* 1. Retrieves the message
  2. Check to see it is not ‘q’ message: session ending message with content ‘q’.
  3. Any message is put into **Q**
  4. Check if it is ‘q’ message the thread is ended

1. **Retrieve & Process:**

* **Retrieve** – from **Q**
* **Process** – check message and put in **Dict**

This thread **listens on the Q** (each time interval = n sec) it checks if messages arrived the Q, if yes it does:

* 1. Extracts the message from the Q, by api get().
  2. It checks content of the message.
  3. Every message (whether it ‘q’ or not) will be located in the **Dict**.
  4. If the message isn’t ‘q’, will be created response message and put in **Dict**.
  5. If the message is ‘q’ the thread will be ended.

**Client:**

Run client after Server is created and waiting to get connection from the Client side.

Client implemented in same process

Using this sequential manner:

1. Client asks User for the message
2. Empty messages are ignored by Client (are not sent to Server side)
3. For not empty messages, Client does:
   1. Send input message to Server side
   2. Insert input message to local **Dict**
   3. If user’s input **==** ‘q’:
      1. Client socket will be closed
   4. If user’s input **!=** ‘q’:
      1. Client will wait to receive a response message from Server
      2. Client will add the received response message (to the message it just sent) to the **Dict.**
   5. Client will keep asking user to enter new input message
   6. Upon ‘q’ message, the Client socket will be closed.