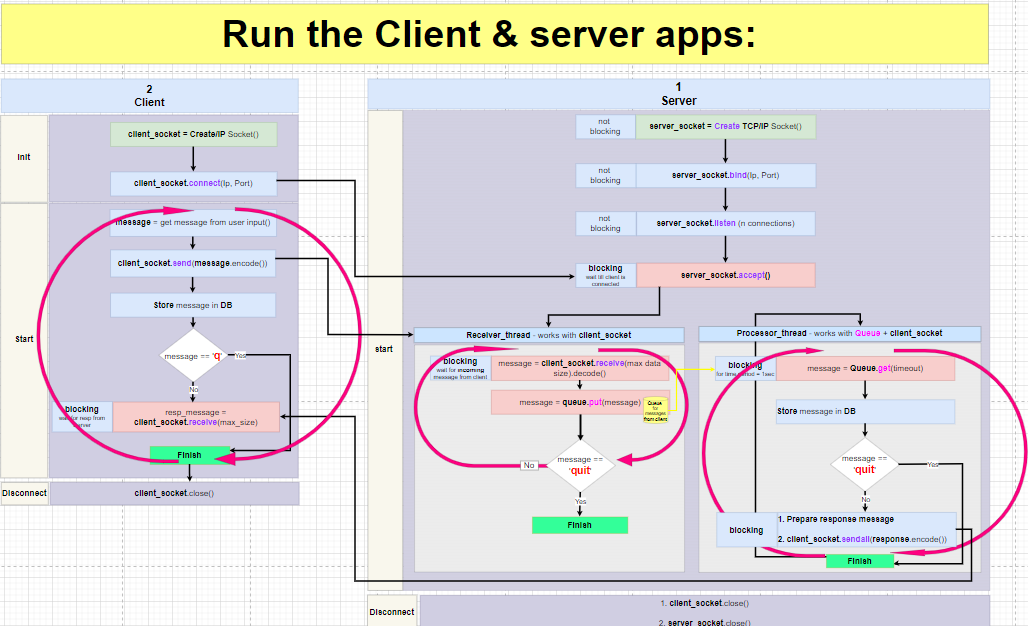
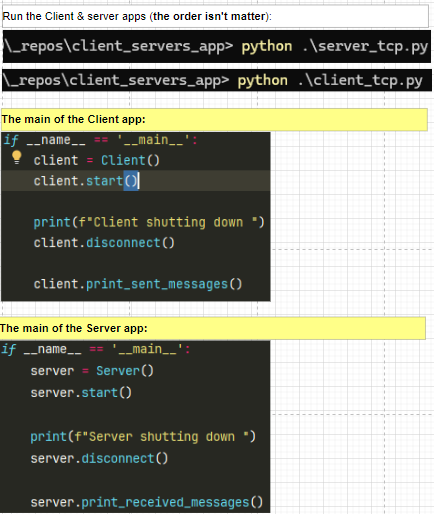
**Single Client Server application – based on TCP / IP Sockets**

This is a block diagram of the program:



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.

**Secured Client- Server app, using SSL socket (secured by protocol: TLS/ SSL)**

Simple story:

Alice and Bob plan to end each other messages. Before they start they need to agree on a reliable, verified, secured communication channel. But how Alice can be sure that on the other side of the channel is Bob that answering? For this, Bob presents himself with something like ID. Alice verifies the ID and if it is valid the channel is good and Alice can trust Server – the only thing that is left is to secure the messages between them. For this Bob can come up with some cipher.

ID and cipher need to be existing before Alis and Bob starting the connection. Server ID will be used to establish a connection between Alice and Bob. Cipher will be used to send/receive messages between them.

Let tell the same story but use professional terms:

1. Channel = socket
2. Alice = Client
3. Bob = Server
4. ID = certificate
5. Cipher = encryption

A screenshot of a computer program

AI-generated content may be incorrect.

Client wishes to send messages to Server securely. Security will be based on preliminary prepared the following:

1. Public **Certificates** – for authentication of the Server
2. Private **Key** – for encryption of the messages

Client and Server perform several steps.

Steps that Server does:

1. Create TCP/IP **socket**
2. Create **context obj**, this means:

Context obj holds different SSL / TLS settings, all security setups required for secured connection, such as:

* cipher suites
* Certificates Authority (CA)

1. Load **Certificates + Key**
2. **Wrap** TCP/IP **socket** with SSL using the context obj - this will make TCP/IP socket be **SSL socket** (secured socket).

* Here Server defines to the Python SSL module that he is a Server side by a flag: server\_side=*True*

The next steps are done on the SSL socket:

1. **Bind** (to IP and PORT)
2. **Listen** (set number of allowed connections)
3. **Accept** – here Server will expect a Client to connect – while Client just like a Server must use **SSL socket**

Steps that Client does:

1. Create TCP/IP **socket**
2. Create **context obj** with definition of what certificate Server side must present – means:

Client side should verify the Server by checking it’s **Certificate**.

Client can define **specific** certificate to verify or can define not to verify certificate at all if Server uses **self-signed certificate\***. This can be done by setting a flag: **check\_hostname = *False***

1. **Wrap** TCP/IP **socket** with SSL using the context obj - this will make TCP/IP **socket** be **SSL socket** (secured socket)
2. **Connect** to Server:

Both Client and Server must use SSL socket (if any side uses regular socket and the other SSL socket the connect will fail). At this point will be performed a **Hand-Shake** between Client and Server, means:

Certificates will be exchanged and verified by Client, if valid the secured SSL socket will be established and all messages sent through it will be encrypted (using **Key**)

\* **self-signed certificate**  - what does it mean? When we will prefer to use it?

Usually, there is **Certificate Authority** (CA) is the one that prepares a certificates. And there is another one that signs on the certificate. In our case there is same CA that created the certificate and signed it. Therefor it is called – self signed certificate.

We will prefer such certificate in the next cases:

1. We not at production, when we test our application, when we wish to work in the isolated environment, when we don’t care about trusting issues.
2. We do not want to pay for the certificate.
3. We don’t care about the authentication but wish the messages be encrypted only.

Python SSL module will by default **reject** self-signed certificate as it doesn’t have **required trust level** and browsers or any other clients will not trust such certificate – so we need to suppress certification check by: **check\_hostname = *False***

**Prerequisites**:

You should have Git Bash installed on the computer:



(after installation, write *Git Bash* in ’Search’)

Enter Git Bash, run command (see later):

to create 2 things for the (this operation is done 1 time):

1. P**ublic** **certificate** (self signed) – will be loaded by Server, sent to Client, verified by Client.
2. **Private key** - will be used to encrypt messages

Run the command from the project folder (I suggest creating a folder in the project called certs/ and run this command from it.

Also it is very important to add into **.gitignore** file the:

cert/

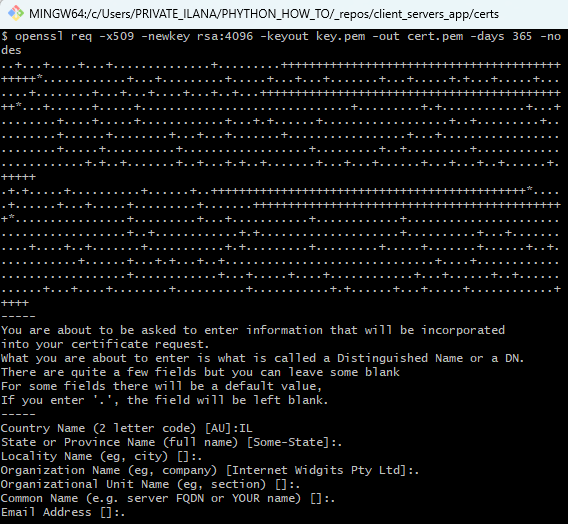
\*.pem

As you mustn’t upload certificates/keys to Git !!!

Command:

openssl req -x509 -newkey rsa:4096 -keyout **ilana\_key\_1**.pem -out **ilana\_cert\_1**.pem -days 365 -nodes

* **ilana\_key\_01**.pem & **ilana\_cert\_01**.pem got names I gave, I could give any names



Will be created:

1. **ilana\_cert\_01**.**pem**
2. **ilana\_key\_01.pem**

Manual check:

Run Server app and simulate the Client by ssl command:

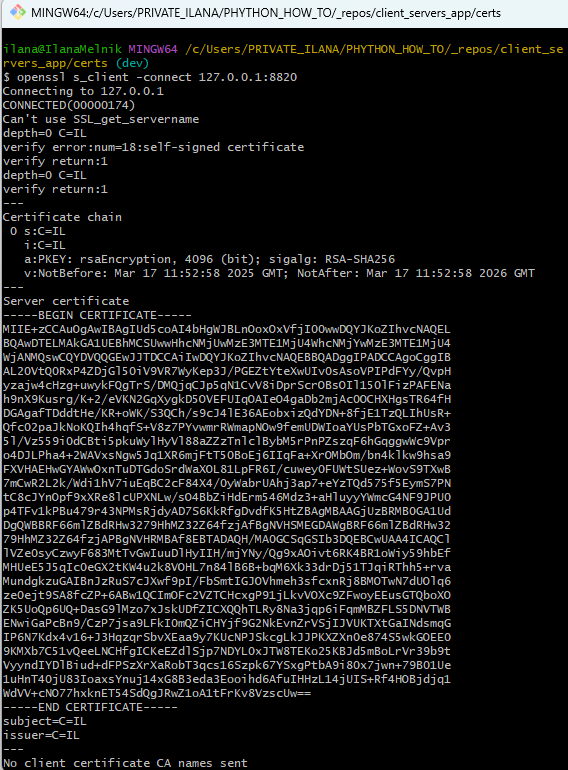
For simulation of the SSL command use same tool: **Git Bash**

Cmd:

openssl s\_client -connect 127.0.0.1:8820

If secured connection was correctly done for the Server side, the command will success and ssl client will connect the server.

Expected to get this response:



A screenshot of a computer screen

AI-generated content may be incorrect.A screenshot of a computer screen

AI-generated content may be incorrect.

What we see here:

**CONNECTED(00000174)** — connection to 127.0.0.1:8820 was successful.

**TLS Handshake** completed using:

* TLSv1.3
* Cipher: TLS\_AES\_256\_GCM\_SHA384

**certificate is working** (even though it's self-signed)

we got:

Verify return code: 18 (**self-signed certificate**)

\*OpenSSL is warning that the cert is self-signed.

**Multi Client Server app:**

There is no change in the way the client app is implemented

The change is in Server app.

Server app will be added with **scanning mechanism.** Scanning means monitoring of the sockets to be notified if new client socket asks to connect or existing client sent new message to server.

Scanning mechanism will use method:

**select.select**: that takes as a parameter 3 lists, these 3 lists of sockets we wish to **monitor**:

* List of **read\_sokets** - sockets we wish to **read** from
* List of **write\_sokets** - sockets we wish to **write** to
* List of **bad\_sokets** - sockets to check for **errors (!!!)**

This method is by default a **blocking method**, but you can add a **timeout**: select.select(rlist, wlist, xlist, **timeout**)

In case there will be no change in list keep on running (in loop)

This method isnt ideal method – it will work well for few hundreds of clients but for **thousands** you will better check using: selectors, asyncio, or epoll/kqueue

Our server app has few parts:

1. **Main server process** that does:
   1. **Create N working threads** to listen on the queue and once new message arrives – to process message.
   2. **Scan (monitor)** the list of sockets to handle one of the next changes:

* New client connection arrived.

(When new client connection arrives, the server will **accept** it and this socket will be added to the list of monitored sockets)

* 1. New message arrived on the existing (already known) client socket

(When new message arrives, it will be processed by a **working thread**)

1. **N working threads**:

N working threads are listening on the queue and waiting for the new messages/s from the existing client socket/s.

The amount of working threads is set according to the number of cores in the PC. In my project, I simply set N = 2.

What Client app can/will do:

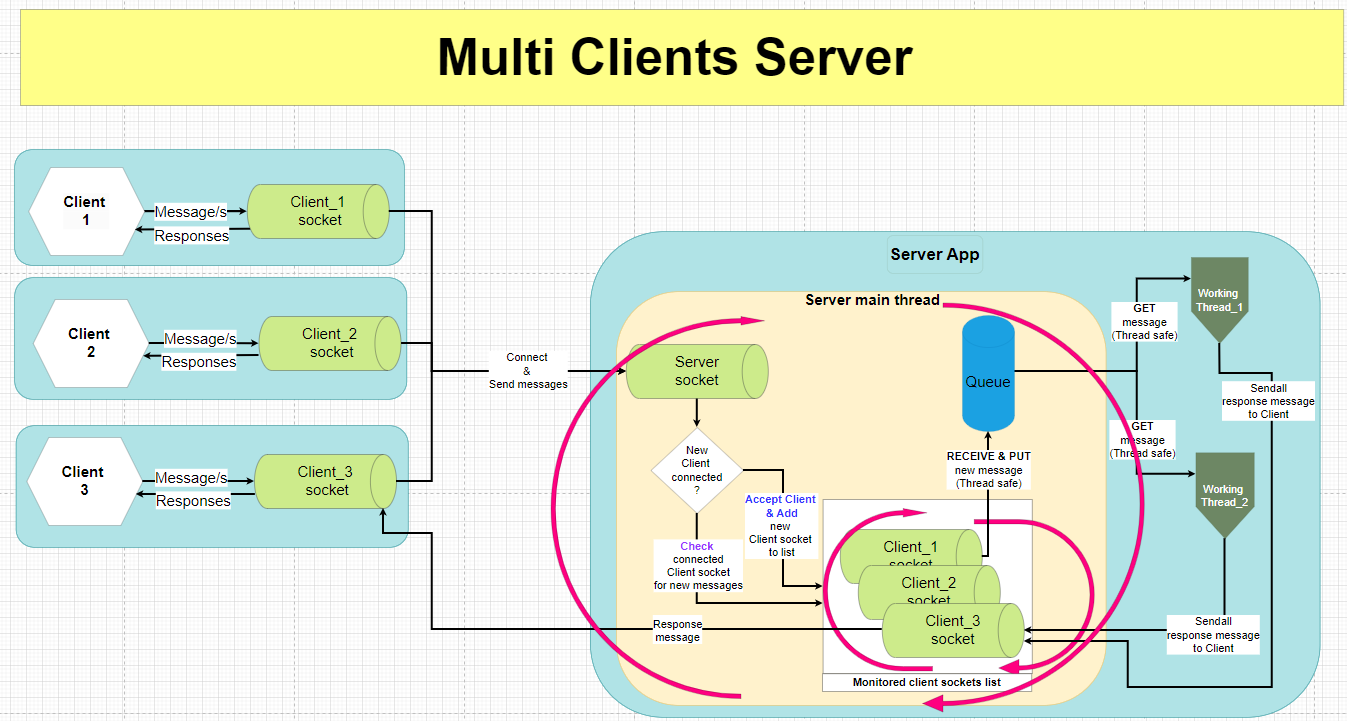
1. Send **message** to the server (any content that is not empty and isn’t ‘q’).
2. Send **terminating connection message** – ‘q’, means client gracefully indicates Server about its termination.
3. To be **killed** (crashed) – this should be indicated by the server as **ungraceful disconnection**.

Also more Client apps can connect same Server app

Some client app can disconnect while other client apps are running well.

Whas Server app must do to be ready to above scenarios:

1. **Scan server socket** – to be notified about new client apps asks to connect.
2. **Scan clients sockets** – to be notified about new messages from known client sockets.
3. Receive and **check messages** from Clients sockets:
   1. If **empty message** arrived, it means that Client was ungracefully killed and disconnected – server should remove such client socket from list of monitored client sockets
   2. If **message with content ‘q’** arrived, it means client app asked gracefully to disconnect Server – server should remove such client socket from list of monitored client sockets.
4. **Put only valid messages from all clients into the queue** for processing by working threads.

Block diagram:

**Comments about the server’s code:**Server extracts messages from the client socket in it’s **main thread**

Each time server extracts **good message**, he puts it in the queue.

* **good message** means:
  + not empty – this tells server that client has disconnected not gracefully (was crashed / killed)
  + not ‘q’ – this tells server that client app disconnected gracefully

To process the messages from queue Server were preliminary created 2 **working threads** that check the queue and once there is a message there the free working thread will get message from queue and **process** it. (OS decides which working thread is free)

* Process the message means:
  + **Get** message from quue
  + Enter this message into the db
  + Prepare an answer to this message and **send** to the client
  + Enter the answer to the db

**Working threads:**

These threads were created with flag: daemon = True.

This means: When the main program finishes, all other threads will finish automatically.

**Responsibilities:**

* **Receive** messages from **client** **socket** - is done in Server’s main thread.
* **Put** messages into **queue** - is done in Server’s main thread.
* **Get** message from **queue** - is done by working thread.
* **Send** back answer to the **client** **socket** - is done by working thread.

**The relationships between main server thread and the working threads:**

Considering the fact: When the main program finishes, all other threads will finish automatically.

As long as main thread monitors the sockets – listens on the socket/s (to find out whether the new clients connected / existing clients sent messages)

and accepts new clients and extracts messages and put messages in queue, the working threads will run.

**When main thread finishes: When / why:**

All connected clients – disconnected either way:

* 1. **Gracefully** by sending the termination message ‘**q**’
  2. **Not gracefully** – by simply crashing or termination that happened at the client side.

**Working threads** will keep running and checking the queue till the main thread finishes.

In my previous post, I explained the basics of creating secure Client - Server application based on TLS/SSL protocol, link:

https://www.linkedin.com/posts/ilana-melnik-82020123b\_in-my-previous-post-i-explained-the-basics-activity-7310708461879648259-aclE?utm\_source=share&utm\_medium=member\_desktop&rcm=ACoAADunE1gBczhsB0Ya6gGLdWtwQCqa7KR25e8