TCP server documentation

The application is a multi-threaded TCP client-server chat application. I implemented both, the server and client with Python 3 and did the networking using the sockets-module and multithreading using the threading-module. The server is currently hard coded to run at localhost on port 6666.

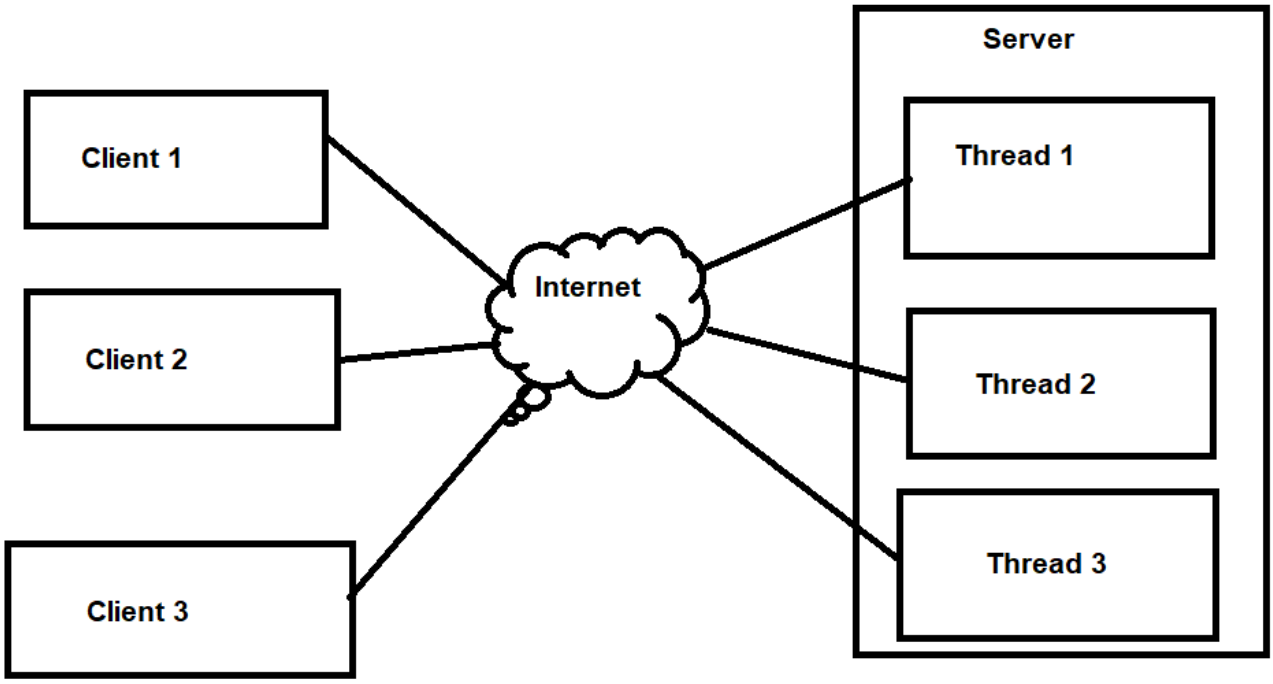

Figure 1: The basic architecture of the system

Usage

The application requires python (version 3.8.3 or newer is guaranteed to work). The server is started by going to the root directory with terminal and typing in **python server.py** and the client is started similarly by typing in **python client.py**.

Establishing connection

The client tries to connect to the server, after which the server responds with a “NAME” message. Then, the client sends the user-defined name to the server. After this, the server checks if the name is valid (doesn’t contain whitespace and doesn’t exist). If the name is correct, the server created a new thread, that receives the client’s messages. The server also created a new Client -object and adds it to the main channel. The client object contains the socket, nickname, and channel, where the client resides.

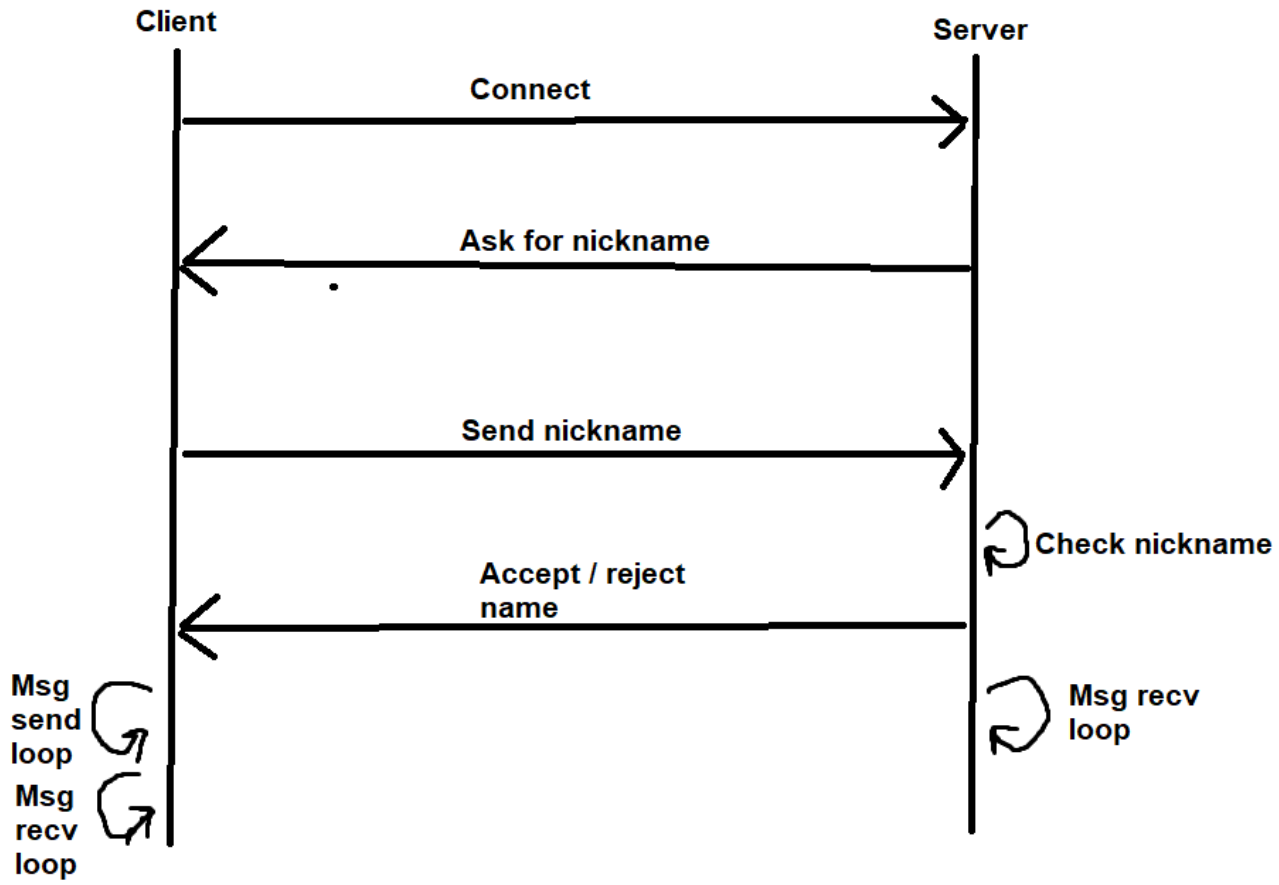

Figure 2: Connection establishing sequence chart

Sending messages:

After the connection has been established, the client starts a thread for sending messages, and starts to receive messages in main thread. In the message-sending thread, the client asks for user input, encodes it to bytes with utf-8 encoding, and sends the bytes to the server. The server then decodes the bytes back to a string in an own thread, and checks if the message starts with a "-", and executes the command if the message starts with a "-", and otherwise broadcasts the message to the clients in the same channel.

Channels

The channels are implemented as a dictionary, where the key is the channel name, and value is a list of clients. The private message, and broadcast commands work across channels. Users can create their own channels with the -ch command.

Scalability/transparency/error handling:

The server scales vertically by assigning a new thread for every incoming client.

The server is transparent since the clients have only one CLI window with what to interact with. The distributed nature of the application doesn't show in any way to the clients.

The source code is littered with try-except -blocks in every networking call. This results in the system functioning, even though connections are lost, some clients crash, or the server faces an exception.

Demo video: https://www.youtube.com/watch?v=9mXsTsOXXuE