If Comic Sans font is not working, just switch to whatever font you like :3

//----------------------------------------------------------------------------------------//

***Issue #1.*** Even a small TCP packet loss (over the WAN) will cause inadequeate behaviour of the image (sent over the network), because there is no control of lost pieces, for example, of image.

So, if server send **100** pieces of one image, and client receives only **90**, client will continue waiting and reading **10** pieces from the ANOTHER image... This issue coudn't be fixed in runtime, because for receiver there is NO DIFFERENCE between the first image and the second one. After some lost parts of image, screen on client's device will be full of "dead" data and the only way to fix this - REBOOT THE SYSTEM

We should detect that the image doesn't containt some part of it's initial form, so we can draw it even in case of data loss (with "dead" bytes), and then begin to accept a new image or just drop the image.

Consequences : not scalable for Internet or other huge networks

Confirmation of issue :

https://www.codeproject.com/Articles/37496/TCP-IP-Protocol-Design-Message-Framing

https://www.codeproject.com/Articles/11922/Solution-for-TCP-IP-client-socket-message-boundary

Notice : links don’t give the solutions based on the data loss.

//----------------------------------------------------------------------------------------//

//----------------------------------------------------------------------------------------//

***Issue #2.*** Actually, sending the image via TCP isn't a good idea, especially when we talking about the Internet.

Sometimes waiting for image isn't acceptable, i.e. if we're monitoring some processes on remote desktop or even playing online video games through remote desktop from our slow phone.

https://stackoverflow.com/questions/6187456/tcp-vs-udp-on-video-stream

So UDP is much more acceptable for **image** **streaming**, but it need some attachments like controling the **right** **order** of datagrams on the client side, and, of course, we should take in count possible HUGE data loss, and we should just don't draw images half-filled with junk from previous images...or at least we should will them with solid green/write/... color.

In addition, we can **switch** between TCP and UDP by auto-detecting distance to server/latency. If we're in local network, we can freely use TCP, but when we're sending data through the Internet, we switch on UDP. It allows us to create application that can be well-scaled in the near future!

//----------------------------------------------------------------------------------------//

//----------------------------------------------------------------------------------------//

***Issue #3***. But TCP is very good for sequenced data like user input. So we'll use TCP for this, but there is one problem :

if we're using Reactor pattern (which is so genious and effective because we can use only one(or some) thread(-s) for receiving data sequenty from each client, and not one thread per each client!)

, it works smthg like this :

for each\_socket in sockets :

if(each\_socket.poll()) :

ReadBytes(...)

... and while we processing this loop, some new data from other client will become available.

Example :

socket\_1 socket\_2 socket\_3 socket\_4

NOT READY READY(2nd) READY(3rd) READY (1st)

Expected order : socket\_4 , socket\_2, socket\_3

But :

1) We'll try to read data from socket\_1. It's not ready. Move ahead.

2) By the time we have been cheked socket\_1, socket\_2 may become available, just BEFORE socket\_4 (which THE FIRST!)

4) So we'll read data from socket\_2, and the order of receiving will be BROKEN even if we are using TCP !

//----------------------------------------------------------------------------------------//