```
/**
 * This documentation is about the development of LiveScan3D, including any
 * advancements from the original code.
 *
 * **************
 * Author: Ioannis Selinis 2019-2020 (5GIC, University of Surrey)
 */
```

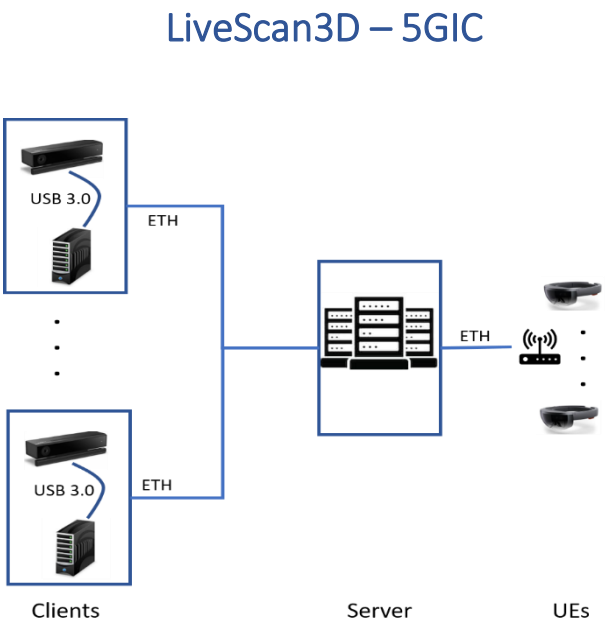


Figure 1: Topology

LiveScan3D is a system developed by M. Kowalski and J. Naruniec that uses multiple Kinect V2 D-RGB sensors (*Clients)* simultaneously to produce a 3D object (hologram) [https://github.com/MarekKowalski/LiveScan3D]. The main novelty in this work is the use of multiple *Clients*, where an object can be captured from multiple angles and each *Client* sends this information to a local *Server,* where the final frame is constructed. In our Lab, we can use up to 4 *Clients* and calibrated them using the white boxes. For convenient, the IDs per marker are written on the top of the white box. Bandwidth requirement (up to 800 Mbps for streaming the whole scene).

Figure 2: Calibration Box & Markers

Calibration: To calibrate the system when multiple *Clients* are used (i.e. >1), we need to place the white box in a distance of ~1.5m and make sure that each *Client* captures only one marker. For more information about the distance and its impact on Kinect V2, the reader can refer to the Microsoft Documentation and to other existing works that show the impact and sensitivity of multiple Kinect V2. For the calibration, go to settings, disable stream only bodies, and based on the number of *Clients* and the marker ID, we set calibration as follows (an example for 3 markers):

ID	Orientation	Translation
0	0 0 0	0 0 0
2	0 180 0	0 0 -0.34
3	0 270 0	-0.17 0 -0.17

Note that 34 cm is the length of the box, hence translation corresponds to the x,y,z [https://pterneas.com/2016/08/11/measuring-distances-kinect/].

The transformation happens locally at the *Client (Server* transmits all settings to the associated *Clients).*

Once calibration finishes, then the *Server* can request per connected *Client* for a frame and waits until the frame has been received by the *Client.* To ensure frame synchronisation, the *Server* blocks all other functionalities/ requests to other *Clients* and waits for the frame. Further, the *Clients* store locally only the latest produced frame, hence any other frames produced between two requests are discarded and can be considered as lost. On the other hand, if the request rate is higher than the production rate (i.e. 30 FPS @ *Clients)* then the *Client* may transmit the same frame multiple times. On top of that, *Clients* may produce frames at a lower rate due to the blocking operations.

Other Settings: The user can tune the settings based on the needs, e.g. reduce “noise” and “flying” pixels by increasing the N neighbours or reduce the Max Distance. However, this puts more stress on the CPU. Furthermore, Compression Level by default is set to 2, however this may not be supported for running the application on Linux, thereafter, setting to 0 may solve this issue. Also, this setting does not always work, i.e. compressing the data.

Finally, once the frames have been received from the *Clients,* the *Server* renders them and produces the final frame, which is the one displayed on the screen and streamed to the Hololens [https://github.com/MarekKowalski/LiveScan3D-Hololens].

Development of new features in the original code to support:

1. High FPS
2. Short/Long RTT
3. Local/cloud-based infrastructure
4. E2E (on Android devices)
5. Reduced complexity

New features:

- Buffers: Queueing system (FIFO) has been developed and integrated in all nodes (i.e. Client, Server, UE), with a default size of 255 frames. A *Client* has one queue where the frames produced are stored. The queue starts filling with frames once the connection is established and transmits all buffered frames upon request. The *Server* has a three level buffering scheme, where the first one is the queueing system for the *Clients*, the second one is for the final frame produced (the one displayed), and the third level is the buffer where the final frames are stored for serving the UEs. The first level comprises multiple buffers that depends on the number of *Clients,* one buffer per *Client,* to avoid any buffer overflow and for synchonisation. The rest two levels comprise one buffer. Note that for the third level buffer, the *Server* monitors the frames served per *UE*, in order to correctly serve the users. The *UE* on the other hand has also 1 queue.
- Non-blocking operations: Most of the operations are non-blocking and have been decoupled in order to maintain high FPS and lower stress on the CPU/GPU. In that way, the system can operate locally or in long-RTT scenarios at 30 FPS.
- Headers concatenation & new signal field: The headers are now attached to the payload and a new signal has been added to inform the *Server* that there is not any frame available in the queue (@ *Client).* Furthermore, a new field to carry information about the *Timestamp* is introduced and corresponds to the time that the frame was produced by the *Client.*
- Multiple parallel TCP Connections: *Client-Server* up to 6 with the ports 48000-48005 and *Server-UE* up to 5 with ports 48006-48010. Connection is based on Asynchronous sockets. By default, the number of TCP connections is set to 1, with the user to be able to increase the number **before initiating the connection**! Note that 1 queue is maintained per *Client (@* the *Server)* irrespectively of the number of TCP connections. Example for the UE: 131.227.61.110:2 (:2 corresponds to 2 parallel TCP connections)
- Threshold: This corresponds to the threshold that is set at the *Server* for the first-level buffering scheme. The *Server* requests for a frame only when the number of stored frames is lower than the threshold defined (values: 1-255).
- Synchronisation: Initial development, dequeue frames that their timestamps are within a time window (20ms default) from the 1st level buffer.
- SNTP protocol: For the time being the timestamp and synchronisation is based on public NTP servers, and some discrepancies have been observed between the timestamps at the client and server.
- RandomDrop: The user can control the % of frames to be dropped at the *Client.* The *Client* randomly discards the frames produced. It can also be controlled at the server (see future work).
- Statistics: All statistics are re-directed on a text file (bin folder), for the *Client* and *Server.*
- Profiling: Measure execution time per function (reduced FPS).
- Support of Android devices: Unity software & then Android Studio were used to develop the Hololens application for Android devices. Also, the GoogleAR suite was used.

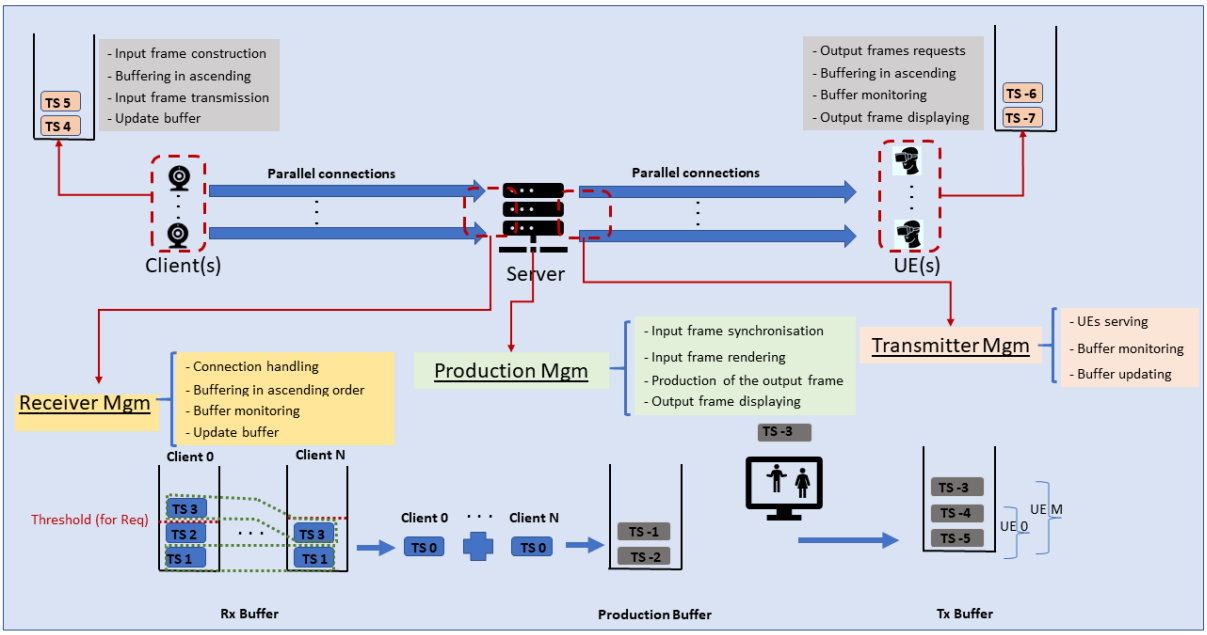


Figure 3: Buffering mechanism

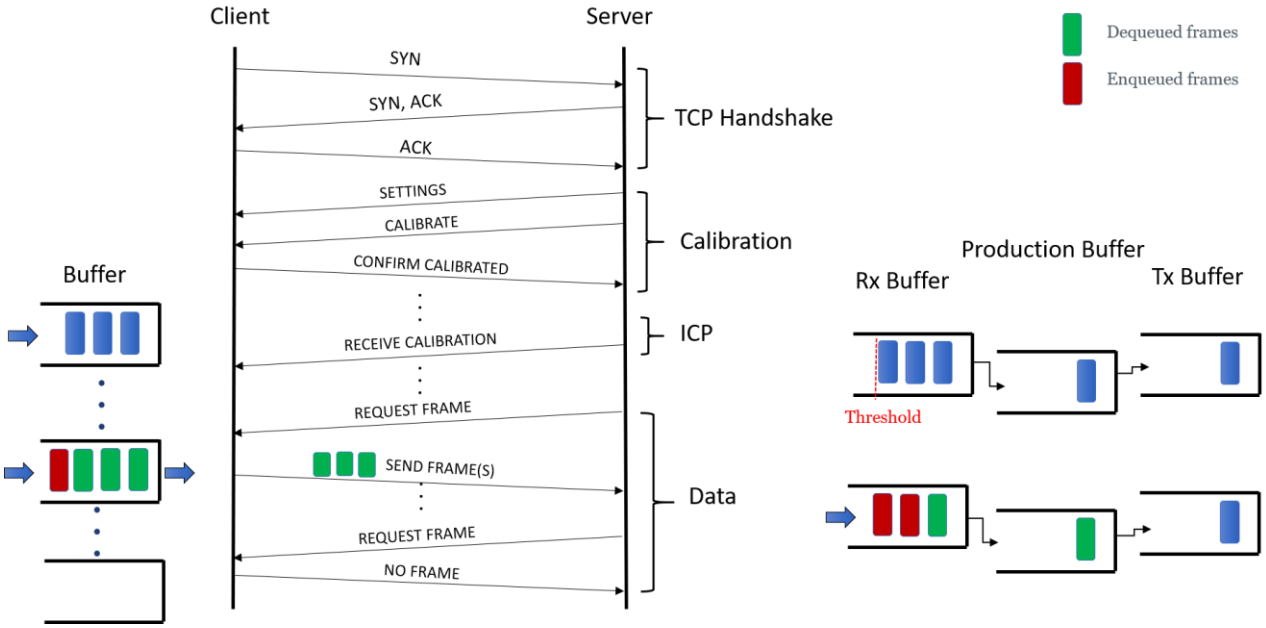


Figure 4: Signaling exchange

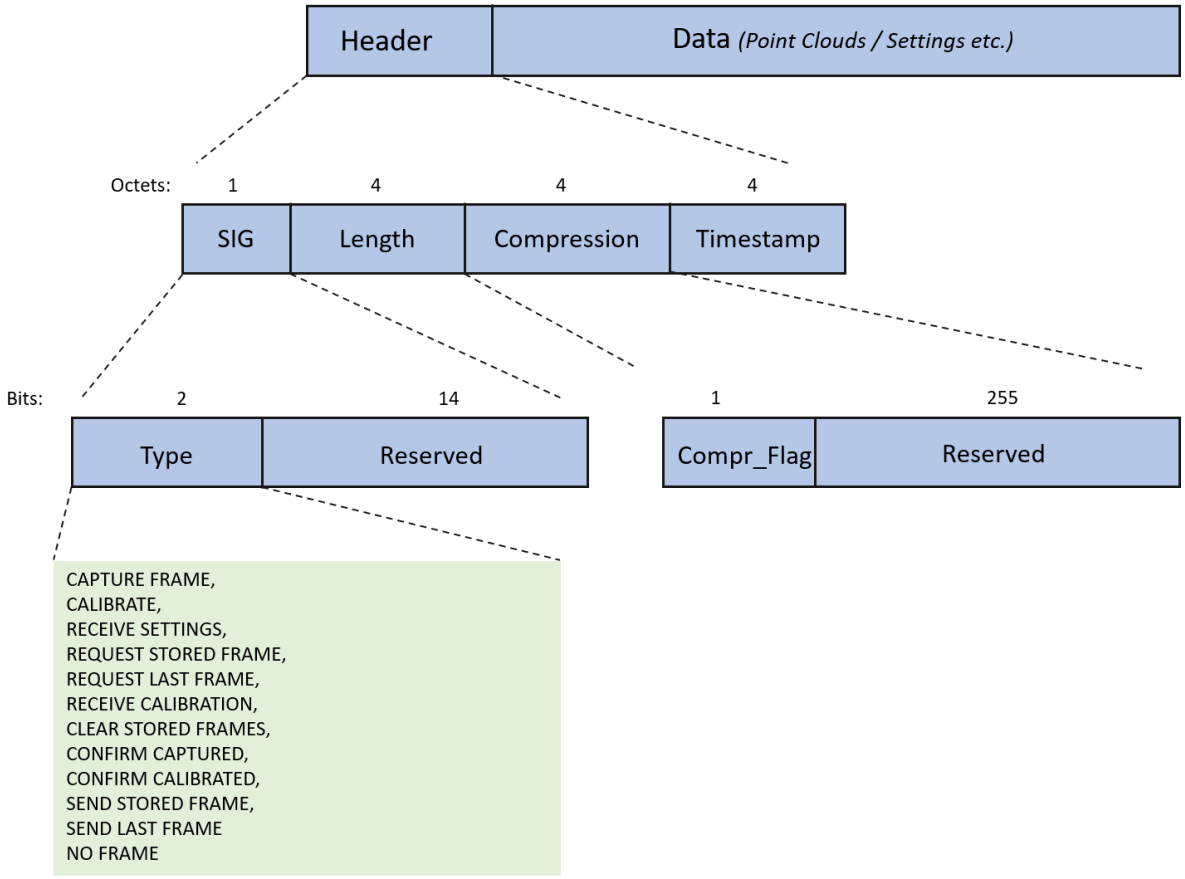


Figure 5: Frame Structure

Optimisation is also possible for the headers, in order to i) reduce the length and ii) include fields required for the continuation of this work. For example, there are 14 bits reserved in the *SIG* field that could be either used or removed and merged with the *Length* field.

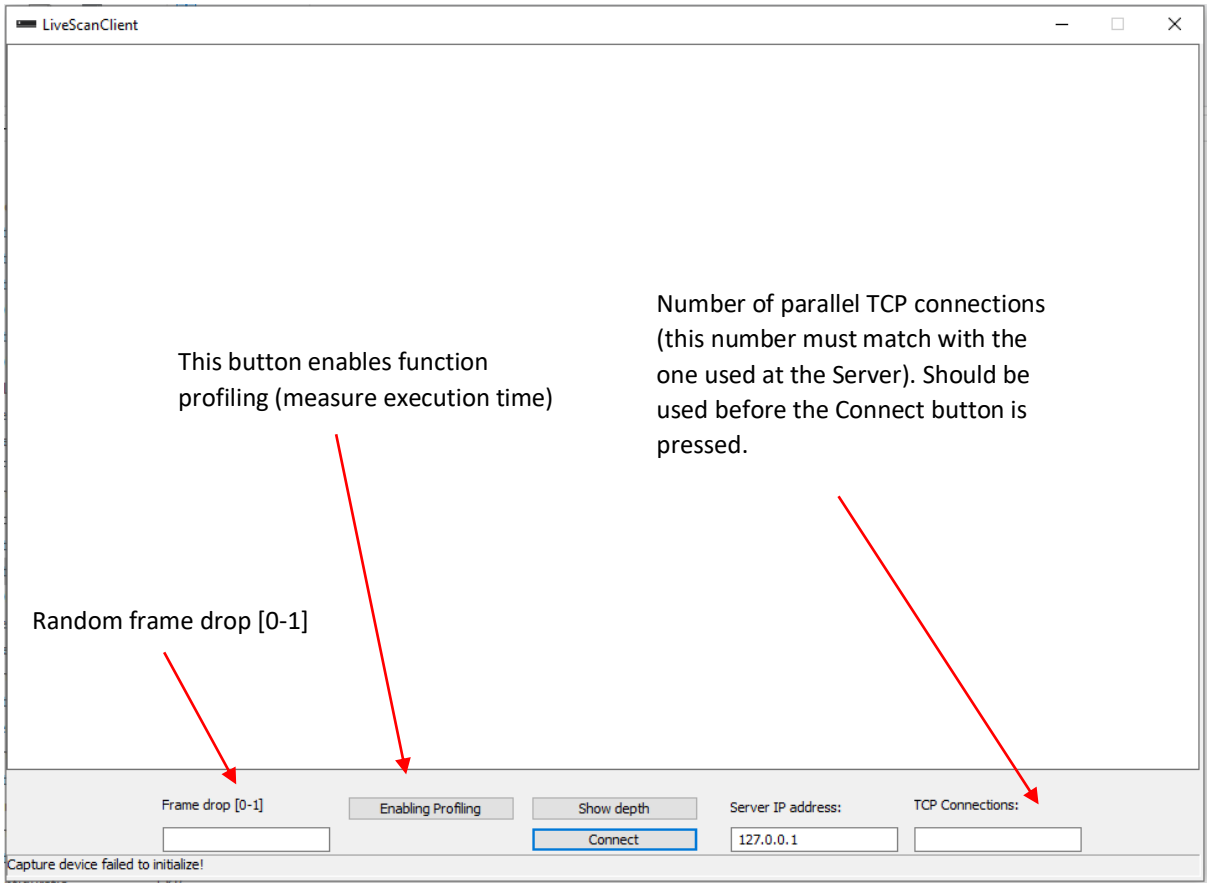


Figure 6: Client GUI

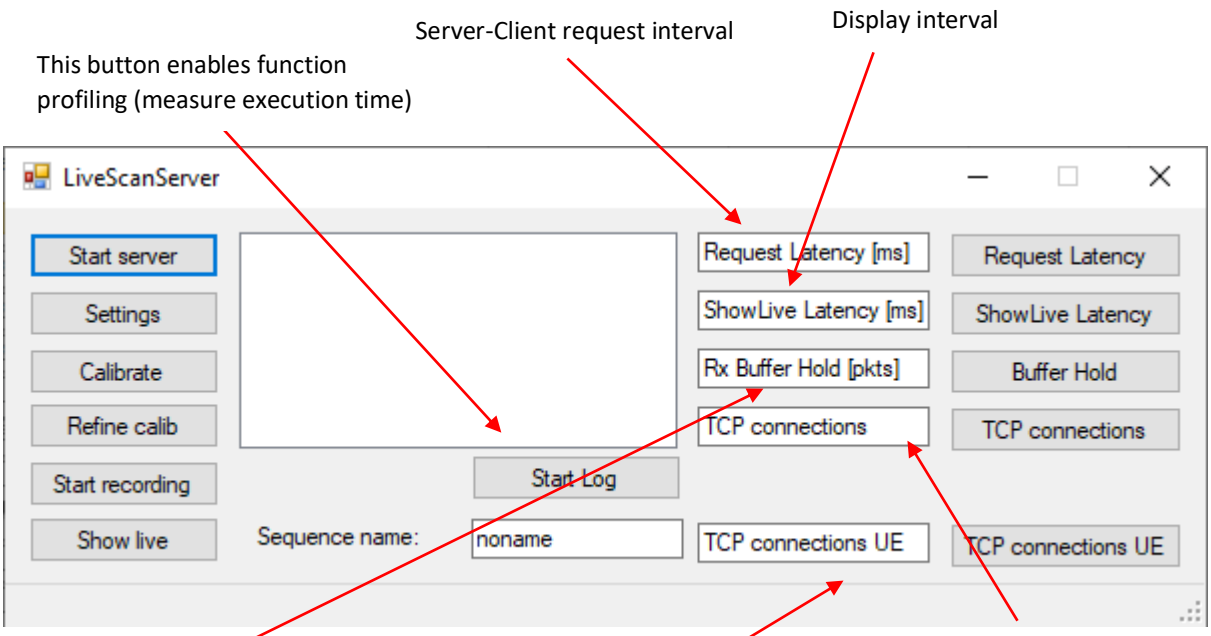


Figure 7: Server GUI

Buffer classes

clientBuffer.cpp: This class contains the buffer deployed at the *Client.*

- Enqueue: Function for enqueuing a frame in the buffer. The frame is now ready for transmissions. Along with the frame, I record some statistics.
- Dequeue: Function for dequeuing all the available frames in the buffer upon request.

BufferRxAlgorithm.cs: This class contains the 1st level of the buffering scheme at the *Server.* One buffer per *Client,* irrespectively of the number of parallel connections. This Buffer is responsible for the synchronisation.

- Enqueue: Function for enqueuing the frames received from a *Client.*
- Dequeue: Function for dequeuing a frame. Rendering is also happening with the frames dequeued (given that multiple *Clients* are used), and the final frame is produced on the live show.
- CheckStoredFrames: Function for monitoring the status of the buffer. If the number of frames stored in this buffer is lower than the threshold, the *Server* will request for a frame from the *Client.*
- IsFrameEligibleForDequeueing: Function for checking whether a frame is eligible for dequeuing based on the timestamp.

BufferLiveShowAlgorithm.cs: This class contains the 2nd level of the buffering scheme at the *Server.* There is only one buffer for the displaying object. Once the rendering of the frames finishes, the final frame is stored in this buffer and is ready for display.

- Enqueue: Function for enqueuing the frames ready for display.
- Dequeue: Function for dequeuing a frame for display.

BufferTxAlgorithm.cs: This class contains the 3rd level of the buffering scheme at the *Server.* There is only one buffer for all *UEs*.

- CreateFrameForTransmission: Function that constructs the final frame to be ready for a transmission (i.e. timestamp, headers etc.)
- BufferedFrames: Function for enqueuing the final frames.
- GetBufferedFrames: Function for dequeuing the frames for a *UE*. Since one buffer serves all the *UEs*, this function carefully selects the correct/appropriate frames for a *UE.*
- CleanBuffer: Function that removes from the buffer the frames that have been sent to the *UEs.*

LiveScanClient:

- SocketThreadFunction: Function that handles the sockets; receive/transmit. This function calls the HandleSocketReceive() and HandleSocketTransmit().
- StoreFrame: Function that constructs the frame ready for a transmission and stores it in the queue (if the *Client* is connected to the *Server)*. This function calls the CreateFramesReadyForTransmission() before Enqueue().
- NtpThreadFunction: Function for the NTP.
- ShowFps: Function for the statistics.

SocketCS:

This class is the one developed for the sockets at the *Client.* It also holds some TCP statistics per connection that are redirected to an output text file.

KinectServer:

This class is the backbone for the *Server.* It is responsible for the requests, calibration, NTP, and the statistics. Also, the requirements set by the user are handled by this class.

KinectSocket:

This class is responsible for handling the sockets, receive and transmit frames. Note that for the time being, only the RequestLastFrame() uses a different frame structure, i.e. one additional field has been added to carry the information about the % of dropped frames [values carried 0-1, with 0 → 0% and 1 → 100% of frames to be dropped] that the *Client* will do. As a future work, a unified header should be used!

TransferServer:

This class is responsible for handling everything for the *UEs*.

MobileUE:

Use Unity software to build and then Android Studio to install the app on the mobile device. You will find most of the classes/files in the Asset folder. Note that UE uses the ports 48006-48010. If we want to use 1 port we type e.g. 192.168.1.3:1:10 (it uses the 1st port → 48006 and prefetching threshold of 10 (buffer), or 192.168.1.3:2:10 for ports 48006 & 48007 and prefetching threshold of 10). Note that the number of ports should match with the one used at the *Server* for the *UE* (before we start the application). **The format is IP:NumberOfPorts:PrefetchingThreshold.** You can "debug" the application on the mobile device through Android Studio (& Unity) with the logs printed on the console (Android Studio).

As a side note, I had a look at the application for the Hololens and should relatively easy to develop or migrate the latest Android version on Hololens. The buffer mechanism and the headers (requests/responses) should follow the same pattern (hdr fields) as on the mobile device and as described at the TransferServer/TransferSocket (*Server*).

Note: Always remember to open the ports if a demo takes place outside ICS premises (also internally I had requested for this, as we use the cloud)!!

****
****

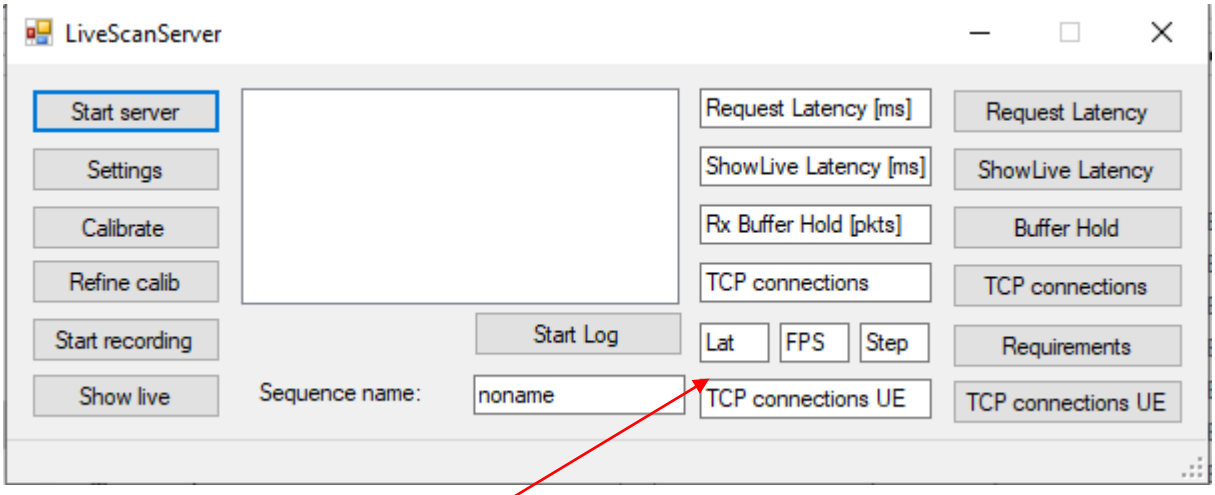
********** **Future Work** **********

****
****

Integration of the next generation of sensors into the platform.

Known issues:

- Only (S)NTP clients are supported for the time being and sometimes I had observed issues with the timestamps, i.e. t0 > t1 which is not correct. The synchronisation relies on public NTP servers, either there is a bug in the code (debugging) or an issue with the public servers. In any case, debugging is required and the development of an NTP server co-located with Kinect Server.
- Header structure needs to be updated across all links and for all frame types, to use a unique header like a proper system.
- Calibration needs to be checked whether it works with the latest developments, as I only focused on a single *Client.*
- The functionality of the *Server* that based on the requirements set by the user, calculates the % of the dropped frames is almost done (as a prototype) with the only issues to be the header (unified hdr at some point, for now it is not an issue) and the issue with the NTP.
- Application on the mobile device is too sensitive.
- Further debugging might be required to ensure that a release is stable.



Latency [ms] and FPS: These are the requirements set by the user.

The user can also set the step for the algorithm (0-1), where a value of 0 means that the *Server* will calculate the step as: (FPS_Current – FPS_Requirement) / FPS_Current

This step is the information transmitted to the *Client.*

Useful links:

Body tracking:

https://www.codeproject.com/Articles/743862/Kinect-for-Windows-Version-Body-Tracking

Kinect guide:

https://pterneas.com/kinect/

Coordinate system:

https://medium.com/@lisajamhoury/understanding-kinect-v2-joints-and-coordinate-system-4f4b90b9df16

Object recognition:

https://github.com/r4ghu/kinect_recognition

Original Code:

https://github.com/MarekKowalski/LiveScan3D

Original Code – Hololens:

https://github.com/MarekKowalski/LiveScan3D-Hololens