FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Multi-Camera Synchronization for Object Detection in Multiple Scenes by Computer Vision

Pedro Emanuel Moura de Castro

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: João Manuel R. S. Tavares (FEUP/DEMec)

Orientador na empresa: Pedro Miguel Cruz (Bosch Security Systems S.A.)

July 29, 2021

Resumo

Nos últimos anos, o crescimento exponencial e desenvolvimento de tecnologias relacionadas com inteligência artificial e "deep learning" têm tornado as suas aplicações bastante claras em áreas como a visão computacional. O uso destas soluções para o desenvolvimento das chamadas "smart cities", onde um sistema distribuído de câmaras pode ser implementado em conjunto com algoritmos de análise de imagem, "crowd management", análise estrutural e Re-ID, com o objectivo de melhorar as condições de segurança e conforto dos seus cidadãos. Alguns destes processos requerem uma representação em tempo-real das cenas capturadas para poder exercer eficientemente o seu propósito. Assim, um método de sincronização que assegure que as transmissões de video obtidas das câmaras se mantêm dentro da precisão "sub-frame" para uma dada frequência de captura é vital.

O principal objetivo desta dissertação é o desenvolvimento de um método que permita a sincronização de múltiplas câmaras mesmo que estas não possuam nenhum dos requirimentos enunciados na literatura atual em obtenção de sincronização de vídeo tais como: 1) as câmaras estarem a observar a mesma cena, de modo a perimitir a extração de características visuais em comum que permitam a correção do "offset" dos vídeos; 2) o uso de um "trigger" em forma de hardware of software que permita que a captura de vídeo em todos os dispositivos se inicie exatamente no mesmo instante. Tendo isto em mente, na presente implementação, os relógios internos de cada dispositivo são sincronizados através do uso de um "network time protocol", e depois, extraindo os "timestamps" dentro dos pacotes RTCP recebidos em conjunto com a transmissão de cada câmara, somos capazes de calcular o "offset" entre os vídeos e corrigí-los adequadamente.

Os resultados obtidos são demonstrados através do uso de métodos que validam a precisão da sincronização, ao observar as diferenças temporais entre as transmissões das câmaras enquanto capturam um relógio exibido num visor externo e demonstrar que o "offset" em ambos os vídeos se mantém dentro do limite de precisão de "sub-frame" para a frequência de captura escolhida. E ao verificar a continuidade do movimento numa cena onde "image stitching" é aplicada. Outros testes como: aumentar a latência das transmissões, o "bit-rate" do vídeo ou deixar o sistemas correr durante um extenso período de tempo, foram realizados de maneira a demonstrar a robustez da sincronização. Finalmente, esta implementação é comparada com outras soluções da literatura atual.

Abstract

In recent years, the exponential growth and development of technologies related to artificial intelligence and deep learning as made its potential uses clear in areas such as computer vision. The use of these solutions for the development of smart cities, where a distributed system of cameras can be implemented along with image analysis algorithms such as event detection, crowd management, structural analysis and Re-ID, help improve the safety and comfort of its citizens. Some of these processes require an accurate real-time representation of the captured scenes in order to efficiently carry out their purpose. Therefore, a synchronization method that ensures that the video feeds obtained from the cameras stay within the sub-frame accuracy for a given capture frequency is vital.

The main objective of this dissertation is the development of a method that allows the synchronization of multiple cameras even if they do not possess any of the requirements described in recent literature to obtain video synchronization such as: 1) the cameras observing the same scene in order to extract common visual features between images that allow the correction of the video offsets; 2) the use of a synchronous hardware or software camera trigger to ensure that the video capture of all devices initiates at the exact same time. For this, in the presented implementation, the clocks of every device are synchronized using a network time protocol and then by extracting the timestamps inside the RTCP packets that come with every camera stream, we are able to calculate the offset between the videos and correct their frames accordingly.

The results obtained are demonstrated using tests that validate accuracy of the synchronization. By observing the time difference between the camera feeds when capturing a running clock on an external display and showing that the offset exhibited stays within the sub-frame accuracy threshold for the corresponding capture frequency, and by verifying the continuity of movement in a scene where image stitching is applied. Other tests such as increasing the latency of the streams, video bit-rate or even letting the system run for an extended period of time, are carried out in order to demonstrate the robustness of the synchronization. Finally, this implementation is compared with other solutions from previous literature.

Keywords: Multi-Camera Synchronization, Distributed Systems, Network Synchronization, Clock Synchronization, Stream Optimization

Agradecimentos

A realização desta dissertação culmina os últimos 5 anos da minha vida, ao longo dos quais evoluí muito quer pessoalmente, quer profissionalmente. Foi um percurso longo, mas não o percorri sozinho e, por isso, gostaria de agradecer:

Ao meu orientador Pedro Cruz pela disponibilidade e orientação ao longo deste semestre, especialmente durante o tempo que passei na empresa. Aos restantes colegas da equipa, os quais apesar de não saber ainda o nome da maioria de vocês, me acolheram na empresa com companheirismo e boa disposição no pouco tempo que estivemos juntos.

Ao meu orientador na faculdade o Prof. Doutor João Tavares, pelo suporte prestado nas incessantes burocracias que rodeiam o processo de dissertação e pela orientação na realização deste documento.

Ao meu pai, por ser o exemplo de pessoa calma que almejo ser mas nunca conseguirei alcançar. Por me ter guiado ao longo da vida no caminho da responsabilidade e bondade.

À minha mãe, que sempre me ajudou em todos os momentos da minha vida. Pela educação e paciência infinita para aturar a minha pessoa. E que me abençoou com a preocupação necessária para me sentir responsável em dar o máximo de mim mesmo em tudo o que faço.

À minha irmã, que mesmo estando sempre carregada de matéria para estudar, nunca se recusou a ouvir as minhas queixas acerca da minha carga de trabalho comparativamente diminuta. Por ser o pilar de responsabilidade e bom senso da minha vida.

À minha família, que apesar de nunca ter bem a certeza de que em que ano de faculdade já vou, sempre me apoiou direta ou indiretamente ao longo dos anos.

Aos amigos que fiz ao longo da minha vida académica, pela amizade e companheirismo durante estes anos. Que ajudaram a tornar os bons momentos ainda melhores e os piores momentos não tão maus. Pela verdadeira conexão com outra pessoa que nasce do medo de saberem que estão ambos prestes a reprovar a uma cadeira.

Aos amigos que fui fazendo ao longo da vida, por estarem sempre ao meu lado. Pelos convívios e aventuras. Pelas conversas pela noite a dentro. E por me ajudarem a manter a sanidade mesmo nas situações de maior stresse graças à vossa reconfortante nabice.

A todos os que se incluem acima e a aqueles, que mesmo não sendo citados, fizeram ou ainda fazem parte da minha vida. Um obrigado.

"Forget the risk. Take the fall.
If it is what you want, it's worth it all."

Kamina - Tengen Toppa Gurren Lagann

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
AVI	Audio Video Interleave
FPS	Frames per second
GIL	Global Interpreter Lock
GT	Ground Truth
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IoT	Internet of Things
LSW	Least Significant Word
MSW	Most Significant Word
MVG	Multiple View Geometry
NP	Network Protocol
NTP	Network Time Protocol
PC	Personal Computer
PoE	Power over Ethernet
PTP	Precision Time Protocol
RANSAC	Random Sample Consensus
Re-ID	Re-Identification
RTP	Real Time Protocol
RTCP	Real-time Transport Control Protocol
RTSP	Real Time Streaming Protocol
SOTA	State of the Art
TCP	Transmission Control Protocol
VLC	Video-LAN Client (multimedia framework)
VLCom	Visible Light Communication (computer vision method)
VR	Virtual Reality

Chapter 1

Introduction

1.1 Context

Recently, the development of technologies related to artificial intelligence and deep learning has been evolving exponentially. Computer vision is encompassed within this area. The use of this solution for smart cities, industrial applications or even daily tasks, optimizes processes that would otherwise be of longer duration or even impossible to perform. These processes sometimes require the use of live multimedia data from distributed systems of cameras, in order to obtain an accurate representation of the captured scenes.

1.2 Motivation

The Security Systems branch of Bosch in Ovar, Portugal, in collaboration with the University of Porto is responsible for the Safe Cities Project [1], which has the goal of applying IoT and AI technologies in urban areas in order to guarantee safety and comfort to its citizens. For this, it is necessary that the implemented artificial intelligence algorithms have access to data that more closely depict the real time environments in which they are used, so that processes like crowd management and Re-ID can be performed accurately and efficiently. Therefore, the camera systems which are positioned in multiple points of the city, must be able to keep themselves synchronized with each other in order for the whole system to accomplish its objective.

Although various approaches to video synchronization have been proposed for surveillance systems [2], due to the ambiguous nature of the camera positioning presented in this case, some approaches may be unfeasible as the long distances between cameras, which may possibly reach up to several hundred meters, and the disparity between observed scenes, make it hard or outright impossible to apply synchronization methods based on image analysis. Other methods based on the application of an hardware or software trigger to guarantee that all cameras initiate capture at the same time instant are also not suitable, as the system uses free-running cameras that stream their video feeds through RTSP to a server, meaning that the system cannot be stopped and

restarted to obtain a new capture, and also because several network connection layers may be in cause, which render the trigger methods useless.

1.3 Objectives

This dissertation proposes a method that allows the synchronization of multiple cameras even if they do not possess any of the requirements described in SOTA literature to obtain video synchronization such as: 1) the cameras observing the same scene in order to extract common visual features between images that allow the correction of the video offsets; 2) the use of a synchronous hardware or software camera trigger to ensure that the video capture of all devices initiates at the exact same time. The proposed method also aims to be reliable enough to maintain sub-frame accuracy even in systems subjected to high network latency or sudden spikes in bit-rate in the camera feeds which could compromise its synchronization.

1.4 Structure

This dissertation is organized as follows: In Chapter 2 we review the current State-of-the-Art approaches to multi-camera synchronization and compare them between each other. In Chapter 3 we analyse the problem characteristics of trying to synchronize multiple cameras given an ambiguous positioning of the devices and impossibility of using SOTA methods, whilst also proposing an implementation along with a detailed explanation of all the elements that take part in it, in order to better understand the developed work. In Chapter 4, we present the results for the proposed implementation in a series of tests that validate the accuracy of the synchronization, along with some other experiments that confirm the robustness of it. Finally in Chapter 5, we draw the final conclusions of the developed work and discuss some possible improvements and deployments of this technology in the future.

Chapter 2

Literature Review

By analysing the current techniques for multi-camera synchronization, we can see that approaches for this problem are traditionally done by extracting characteristics from the video footage captured by the cameras, either audio or video, and then determining the offset of each recording between one another. However, with the rise of accurate internet protocols to synchronize clocks in distributed devices, other approaches that don't require the analysis of the footage from the cameras have surfaced. Based on this information multi-camera synchronization approaches can be divided into five groups: video-based approaches, audio-based approaches, hybrid-based approaches, bit-rate-based approaches and network-protocol-based approaches.

2.1 Video-based Synchronization

Video-based approaches rely on extracting features or global changes that can be observed simultaneously in all the cameras, such as a flash of light, as demonstrated in [3]. This method detects the presence of flashes in the video content by using an adaptive threshold on the intensity of light variation across the frames and then matches those frames in the multiple recordings which allows the determination of the offset time between cameras. An example of the capture of a flash occurrence and synchronization process can be seen in Figures 2.1 and 2.2.

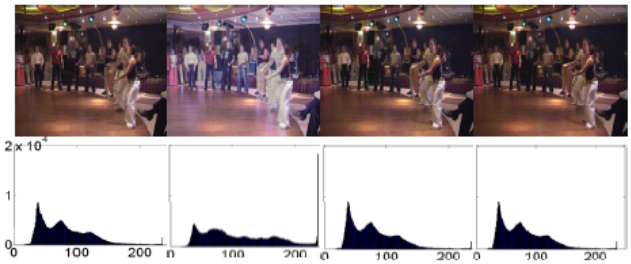


Figure 2.1: Example of video frames and corresponding histograms, during the occurrence of a camera flash [3].

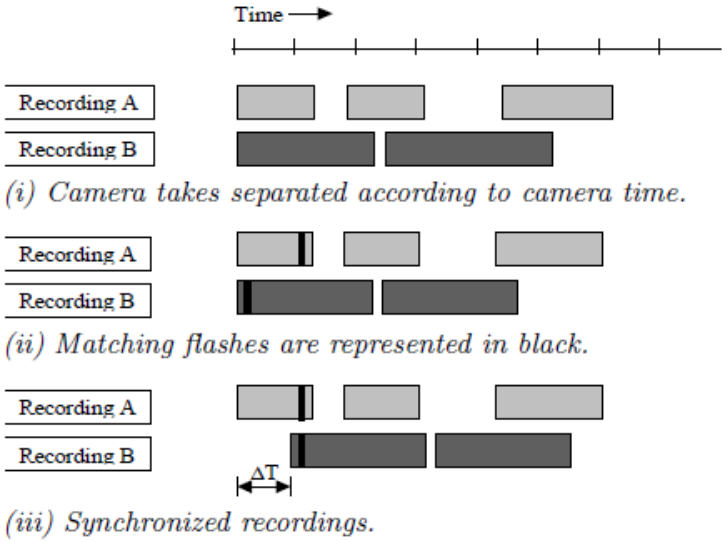


Figure 2.2: Simplified explanation of the synchronization process between two recordings [3].

Mehrabi et al. (2015), propose another similar implementation by harnessing the capabilities of Visible Light Communication (VLCom) [4]. As depicted in the system architecture in Figure 2.3, different digital phone cameras video streams are synchronized through LED lights by embedding the necessary information as light patterns that can be recorded and later extracted by a synchronization server.

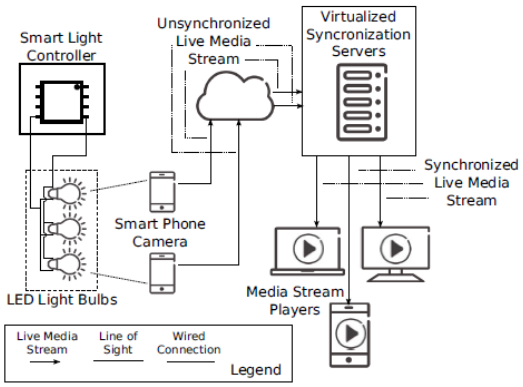


Figure 2.3: Diagram of the video synchronization system architecture [4].

Both of these approaches, however, are limited in the sense that they require the presence of flashes or lights and the detection of these by all the cameras that we wish to synchronize, which may not feasible in some environments. The majority of implementations that use video information are based on Multiple View Geometry (MVG) methods, which align sequences in space and time by estimating the fundamental matrix and the shift in time of the extracted features

among cameras [5, 6, 7, 8]. One of these MVG methods is the feature-based alignment which extract object trajectories and uses algorithms such as Random Sample Consensus (RANSAC) [8] in order to obtain a match between the key features among the recordings of different cameras [5]. This allows the synchronization to occur even without the insertion of an additional auxiliary element in the capture scenery. Methods like the feature-based approach are usually implemented in situations where the camera network geometry is considered to be approximately stationary [9].

2.2 Audio-based Synchronization

Audio-based approaches use the audio fingerprints information as a way to find synchronization points among the multiple video footage. Guggenberger et al. (2012), take these fingerprints from an arbitrary number of audio tracks and determine the matching points between tracks [10]. By finding *n-1* matching points for *n* overlapping tracks, a spanning tree is built using tracks as vertices and matches as edges. This implementation has the disadvantage of requiring user input to determine if the matches are correct, for example through an Graphical User Interface (GUI) like the one in Figure 2.4, which in the context given in Chapter 1 is not optimal, although in the same paper it is referenced that concepts for possible automated solutions have already been developed.

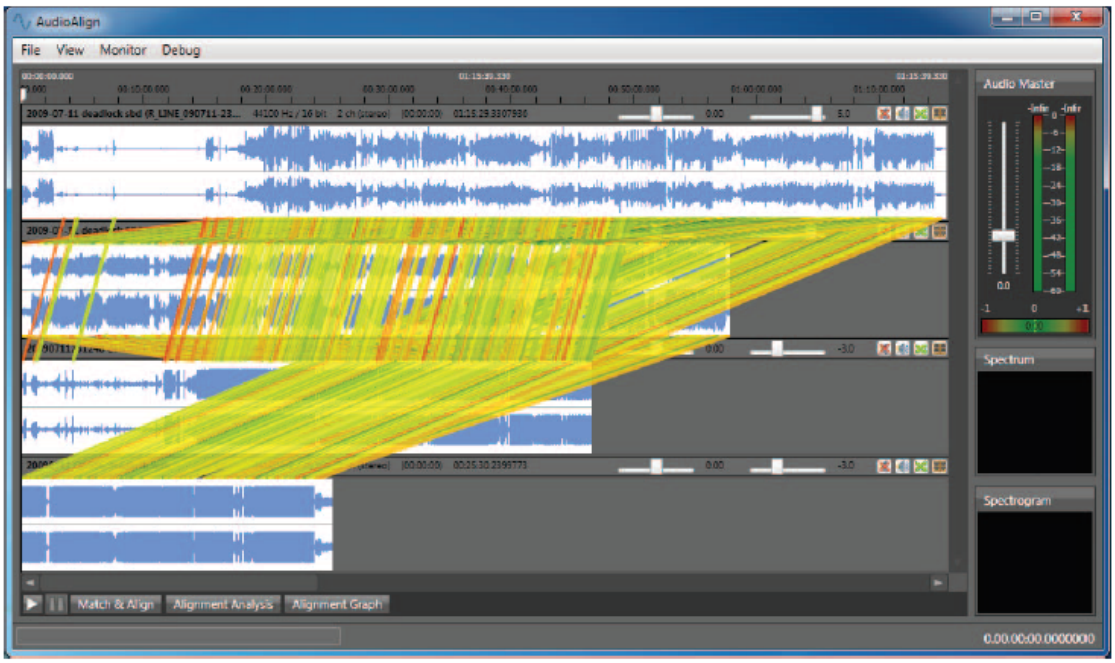


Figure 2.4: GUI showing the calculated match points of four tracks [10].

2.3 Hybrid-based Synchronization

Approaches using hybrid methods utilize both the audio and video processes simultaneously in order to obtain higher reliability for the alignment. Casanovas and Cavallaro (2015) propose an implementation of this approach by extracting temporally sharp audio-visual events that can be easily identifiable in the video, estimating the delay among cameras by matching the occurrence of these events in various recordings and cross-validating the results for all camera pairs, thus making possible the aligning of the recordings in a global timeline [9]. An architecture and brief explanation of this approach is presented in Figure 2.5.

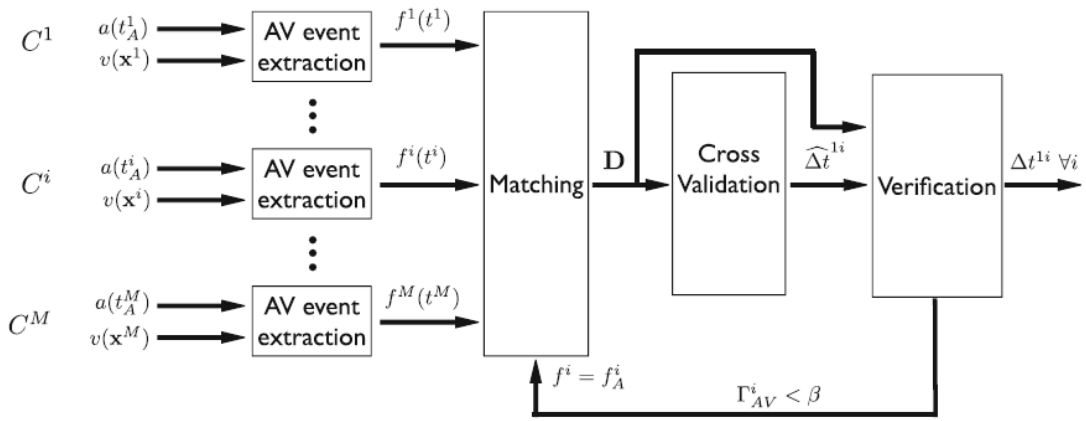


Figure 2.5: Block diagram of the hybrid-based approach for the synchronization of multiple camera recordings. First, audio-visual activation vectors f_{AV}^i indicating the presence of audio-visual events are extracted from each camera recording. Then, a delay matrix D containing the time shifts between each pair of recordings is computed from the audio-visual activation vectors $\left(f^i = f_{AV}^i\right)$ obtained in the previous step using cross-correlation. Next, this information is used in the cross-validation step to obtain time-shift estimates that are consistent among all cameras and sort the recordings on a global timeline. Finally a final verification is performed to detect unreliable audio-visual results and opt for an audio-based analysis $\left(f^i = f_A^i\right)$ if that is the case [9].

This implementation, however, cannot be used in cases where the confidence in the joint audio-visual analysis is low due to poor video quality or very different views, and will automatically adopt an audio-only strategy if that is the case.

2.4 Bit-rate-Based Synchronization

Another type of approach to achieve the synchronization of multiple videos is based on cross-correlating bit rate profiles [11, 12, 13]. This approach has the advantage of not requiring major restrictions on viewing angles, or camera properties, and its synchronization is done by retrieving the conditional entropy of video frames, which quantifies the remaining information given the information in previous frames, thus describing the amount of unexpected change in the scene, and uses this information as a fingerprint of the video. Knowing this, the temporal offset can be determined by a normalized cross-correlation of the conditional entropy sequences. In Figure 2.6 it is depicted how different events like camera motion and scene change contribute to the bit rate changes.

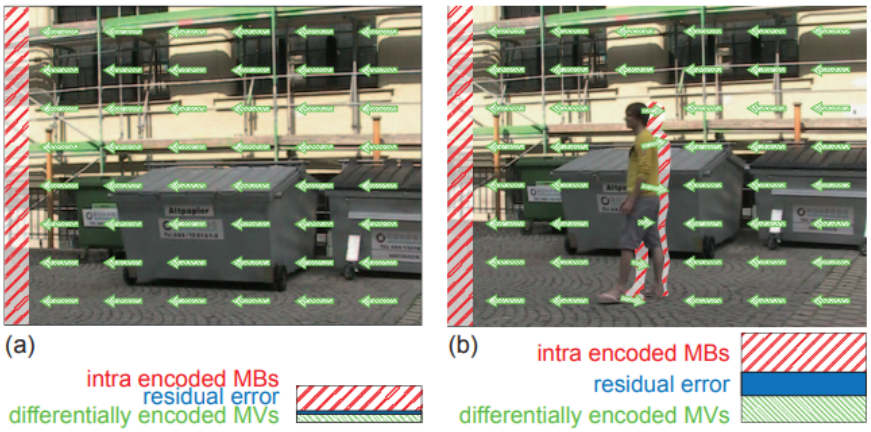


Figure 2.6: Simplified qualitative view on bit rate contributions in the cases of (a) where only sheer camera motion is present and in (b) where additional scene changes take place. Individual image Macro Blocks (MB) and their respective relative translation Motion Vectors (MV) are represented in both cases [11].

More recent implementations of this approach such as the one proposed in [13], state that previous approaches [12], where a window segmentation strategy with consensual zero-mean normalized cross-correlation is used, has an elevated computational complexity, which proves to be a difficulty when synchronizing online data streaming. Therefore, they propose a different window strategy that achieves the same results while requiring much less computational power. This is done by increasing the probability of picking a correct time reference classification by only selecting windows without any uncorrelated segments and thus reducing the number of samples necessary to analyse before building a synchronization set.

2.5 Network-Protocol-based Synchronization

Contrary to the methods presented before, where the sequences of image frames are aligned post-capture, this approach instead synchronizes the cameras in advance by using time synchronization protocols such as Network Time Protocol (NTP) [14] or Precision Time Protocol (PTP) [15] which removes inherent problem of the synchronization algorithm break down when implemented on noisy scenes (i.e. hard-visibility or low texture). Time protocols are used to synchronize networks of distributed systems to guarantee time cohesion between devices as explained further in section 3.2. Additionally, there is the advantage of not having to interpolate the captured frames to a global sub-frame time or requiring the capture of large amounts of data in order to ensure that the synchronization algorithm as enough information to determine a synchronization point between the various recordings.

The main problem with precision time synchronization in distributed systems, arises from the fact that its devices are physically independent, meaning that each device has its own perception of time provided from its hardware clock which may suffer drift even if aligned with an external source such as a distributed network. Litos et al. (2006) propose a software method to establish a time consensus between multiple cameras before capture through NTP synchronization of computer clocks [16]. This system is composed by a camera server that runs on each computer, which can have one or more cameras connected to it and a client that can run on any machine with a network connection. An example architecture of this system is illustrated in Figure 2.7.

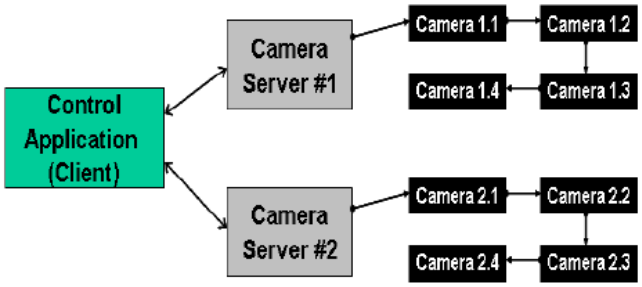


Figure 2.7: Multi-camera system architecture [16].

This solution, however, does not take into the account the delay introduced by the software of the cameras, which may create additional variable latency. Ansari et al. (2019) fixes this issue by phase shifting a frame stream between a leader and a client camera [17]. This is done by either *reset sampling*, a method that is slower to converge but can be applied in many devices, or *frame injection*, which is faster to converge but requires device-specific modeling. In *reset sampling* the client camera is restarted by sleeping for a random amount of time, until its streaming phase relative to leader camera reaches a value low enough to be considered as synced. In *frame injection*, because some cameras APIs allow high priority capture requests to be injected into a normal priority continuous image stream, it is possible to inject a frame with a longer exposure of $T + \delta$ (with T being the exposure time of a single frame, and δ being the phase shift required

to align the streams) which allows the alignment of the streams. This proposed system basically operates in two stages. Firstly, all the clocks of each device are synchronized with the leader's clock using a variant of the NTP protocol and then by phase shifting all the client streams to that of the leader. It is important to note that in this implementation an NTP protocol is used over PTP even though the latter is more accurate, due to the fact that, in this case, the devices used are smartphones which typically don't possess PTP hardware.

Another approach similar to this one was done in [18], where a sensor network composed of multiple high-speed cameras (capture speeds of up to 500 FPS), is setup to precisely capture synchronized frames across a system. By using a network time protocol such as PTP, to synchronize the shutters of the cameras and then timestamping each frame at the vision sensor node prior to sending it to the host terminal, this remote processing node can then correctly choose the correct frame combinations when receiving data from the network.

Easterbrook et al. (2010), described a distributed multi-camera capture system for real-time media applications [19]. The main objective was to develop distributed algorithms for free-viewpoint applications such as Virtual Reality (VR), but it also approaches the issue of requiring processing and transmission of real camera data, which is very much like the one seen in surveillance applications. The synchronization is also done by synchronizing the PCs system clocks through NTP or PTP. However, with the different processing delays of cameras, even if they are gen-locked the frames may still be captured at different times. To compensate this effect, a synchronization test can be setup using capturing a short clip of a 'clapperboard'. By manually examining the recordings of each camera, and finding the frame number where the 'clapperboard' action is completed, the PC's clock time can be adjusted to obtain a 'true capture time'.

2.6 Conclusions

Based on the approaches shown in this chapter we can draw some conclusions. Firstly although highly popular and presenting a wide variety of methods for its implementation (i.ex. feature extraction, intensity histogram), video-based approaches lack robustness to noise in the captured frames (although some solutions have been presented to diminish this effect like in the one in [7]), that may lead these methods to easily malfunction or critically fail in such conditions when compared with other types.

In audio-based approaches a slightly lower computational demanding solution than video-based is provided. Synchronization based on audio fingerprints does not require the analysis of large data files like videos, however it can still be heavily influenced by capture noise.

Hybrid-based methods have the advantages of both audio and video based approaches providing an even higher reliability to sync errors due to noise or camera angles that influence failure to extract image features, although much more computationally demanding because it has to associate visual features to audio events.

Bit-rate-based approaches are not restricted by camera viewing angles like vision-based approaches but still somewhat suffer from noise in the images. The computational demand of these methods is also lower when compared with others and even more with the development of new techniques to improve this characteristic such as the one provided in [13].

Network-Protocol-based seem to be the most efficient of all the approaches stated above, due to the increased reliability and robustness provided by the network synchronization protocols that exist nowadays. Also, the fact that the cameras can be synced without relying on feature or data extractions that are captured by the camera itself, highly decreases the computational power necessary to carry out this function and it is realistically only limited by the latency of the network it operates in.

Lastly in table 2.1 the different approaches to multi-camera synchronization are compared. Three characteristics were picked that represent the common elements between the studies, with them being: the devices used to carry out the implementation, the video frame accuracy and error when synchronizing frames.

Table 2.1: Evaluation of different approaches for multi-camera synchronization.

Approach type	Paper	Device	Video Frame Accuracy	Error Detection
Video-based	[3]	Not specified	$\pm 40ms$	210 of 238 frames correctly detected (88.2% (manual evaluation)
Video-based	[4]	Intel i5 NVIDIA NVS 5200M GPU (video: 720p 30fps)	Not specified[1]	Not specified[1]
Video-based	[7]	Not specified	$< \frac{1}{2}$ frame	Successful when used on synthetic data and real data with susbtantial noise different frame rates and varying levels of initial sync
Video-based	[8]	Not specified	In a subjective evaluation [0-5] by four testers, three gave a score above 4	48 of 50 key frames correctly detected (96% (manual evaluation)
Audio-based	[10]	Not Specified	Results in another paper[2]	Errors can be corrected by manual user input
Hybrid-based	[9]	32 processors Intel Xeon CPU E7-8837 @ 2.67 GHz 530 GBmemory	$< 100ms$	94 % precision and a 83 % recall
Bit-rate-based	[13]	Not specified	81% Max Synchronization Score (Same as other SOTA methods)	Not specified
NP-based	[17]	Google pixel smartphone camera	$< 250\mu s$	120 seconds to achieve 95% probability of alignment (reset sampling) 3 seconds to achieve 100% probability of alignment (frame injection)[3]
NP-based	[16]	2 x 3.2GHzwith Firewire-800 PCI boards cameras with 30fps capture speed	Mean Value or Standard Deviation of $9.988ms$ from GT	$92ms$ of maximum latency for 4 cameras (2 camera/PC)
NP-based	[18]	Two vision nodes composed by: 1 EoSens MC1362; 1 controller PC	Difference in camera timestamps of less than 375 ns	The signals that trigger the exposure of the cameras have errors of $< 10\mu s$

[1] Although the video frame accuracy and error detection are not specified, it is stated that the processing time for the implemented algorithm is 33*ms* and that the results were sufficient for real-time application.

[2] Results are in a Master thesis written in German [20]

[3] Tested both methods in a data set of 239 trials

Chapter 3

Methodology

The main objective when attempting to find a solution for synchronization of multiple cameras, is finding the amount of time offset between their streams and shift the frames accordingly. In some cases this can be solved at the start of a capture by synchronizing the shutters of the cameras. However, in the case of systems composed of free-running cameras, we cannot rely on a restart of a system as the cameras are continuously capturing. Therefore, other solutions must be explored in order to achieve the same result.

By analysing the characteristics of the video streaming protocols used by the cameras, in combination with other network clock synchronization protocols seen previously in the Network-Protocol-based solutions in Chapter 2, we are able to design a method that can effectively synchronize multiple cameras.

In this chapter a study of the cameras and communication protocols is conducted in sections 3.1 and 3.2 in order to better understand the components that constitute this implementation. This is followed by the development of possible solutions for the synchronization of multiple camera feeds based in the information obtained in these sections and the literature review of the previous chapter.

3.1 Cameras

The camera model used will be the DINION IP 6000 HD from Bosch [21] which features a 1.6MP or 1080p at 25,30,50 or 60 FPS capture resolutions in order to provide clear and detailed content that can be later analysed. This device is powered by Power-over-Ethernet (PoE) at 12 VDC, meaning its communications are done by Ethernet. It is also compatible with multiple network protocolos such as NTP, which has been used to synchronize distributed surveillance systems as described in [2]. These cameras were designed to be free-running surveillance cameras with their videos feeds being streamed through a Real Time Streaming Protocol (RTSP).

Because the cameras may be distanced up to hundreds of meters from each other whilst also observing the same scenery, they also feature long distance identification with telephoto lenses.

Figure 3.1: DINION IP 6000 HD camera [21].

The configuration of camera settings is done by connecting to them via Ethernet on a local network and accessing their web page by inserting its IP and access credentials in any web browser. Various configuration capabilities are then presented ranging from basic camera settings such as Lens Configuration and Capture quality to Alarm and Event Response and Network Settings as displayed on Figure 3.2.

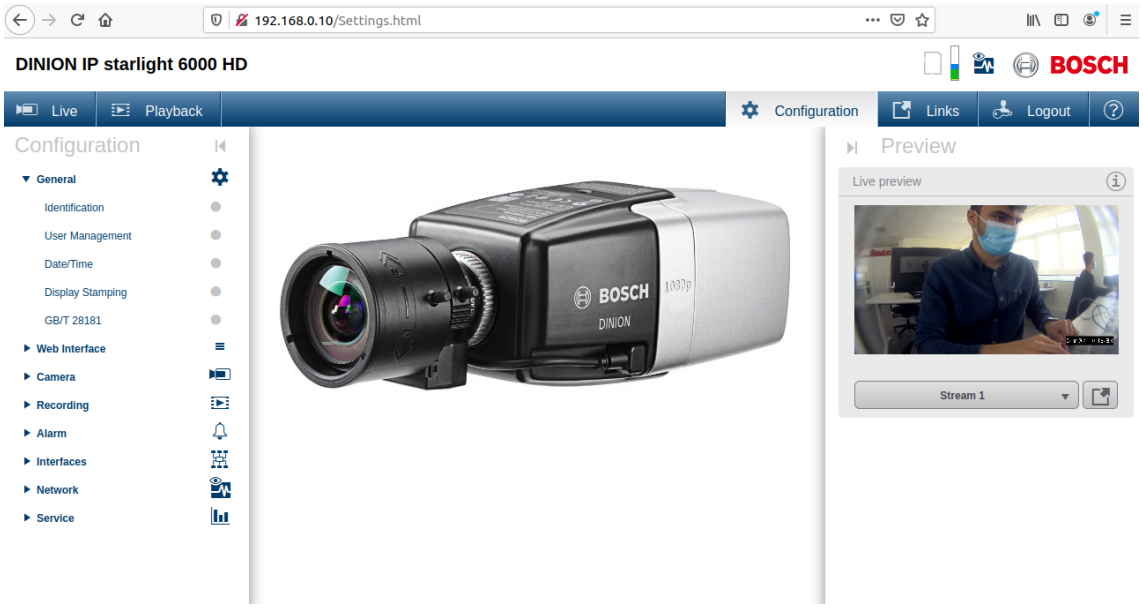


Figure 3.2: Camera web configuration page. [21].

3.2 Synchronization and Network Protocols

3.2.1 Network Time Protocol

The Network Time Protocol is an established Internet Standard protocol, that is used to synchronize and maintain transmission paths sub-nets of time servers [14] and is often deployed in large networks of distributed systems to maintain cohesion between the clocks of each device. It uses an hierarchical configuration where each layer is identified by a Stratum level (0-15), that represents the number of hops between nodes necessary to reach the root (level 0), meaning that as

the stratum level increases so do the clock drifts and time inaccuracies. Therefore, sub-networks are organized in a way where various devices connect to higher stratum level servers in order to maintain an equal time reference. If a connection to one of the nodes is lost, the network can re-configure itself automatically by using backup paths to the root node, but consequently dropping stratum levels on the nodes where the failure occurred. This is exemplified in Figure 3.6 where the link x is lost in (a) and the sub-network re-configures its topology to (b).

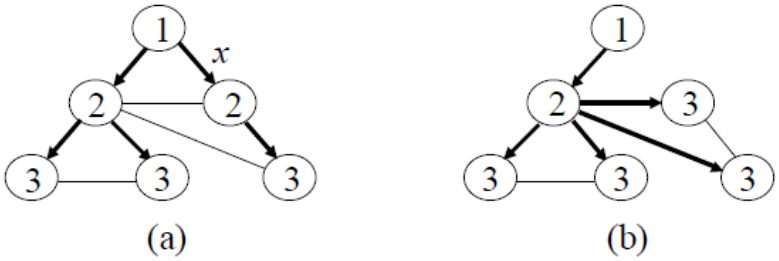


Figure 3.3: Sub-net reconfiguration example [14].

In NTP the individual round-trip delays and clock offsets between nodes are determined using the timestamps exchanged between communications. Figure 3.4 shows this exchange between a server A and a peer B.

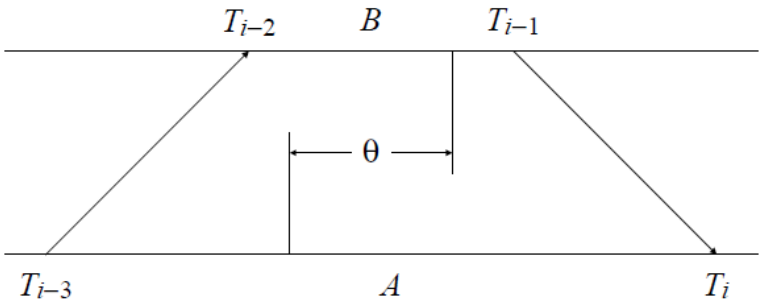


Figure 3.4: NTP message exchange [14].

With T, T_{i-1}, T_{i-2} and T_{i-3} representing the four most recent timestamps occurring in network, the NTP exchange works as follows: The master A sends a sync message to the slave and records the timestamp T_{i-3}; The slave B records the time at which this message is received T_{i-2} and sends back its own message with timestamp T_{i-1}, which is received by master who records its receipt time T.

Although calculating the network delays between the message exchanges from A to B and B to A is not possible due to T_{i-3} and T_i belonging to a different time scale from T_{i-2} and T_{i-1}, we are still able to calculate its round trip delay and offset through the following equations:

$$\delta_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}) \tag{3.1}$$

$$\theta_i = \frac{(T_{i-2} - T_{i-3} + T_{i-1} - T_i)}{2} \tag{3.2}$$

Having obtained these parameters, the filtration algorithm needs to recognize the reliability of this exchange which can be done by verifying if the values are within a margin of acceptable network errors. Calculating the bounds for this interval of the network delay can be achieved with the help of the following equation:

$$\theta_i - \frac{\delta_i}{2} \leq \theta \leq \theta_i + \frac{\delta_i}{2} \tag{3.3}$$

The validated offset values are then passed through a series of filters as illustrated in Figure 3.5 to create an estimate of the time offset through the use of various statistical techniques. This process continues to be repeated until the value converges [22].

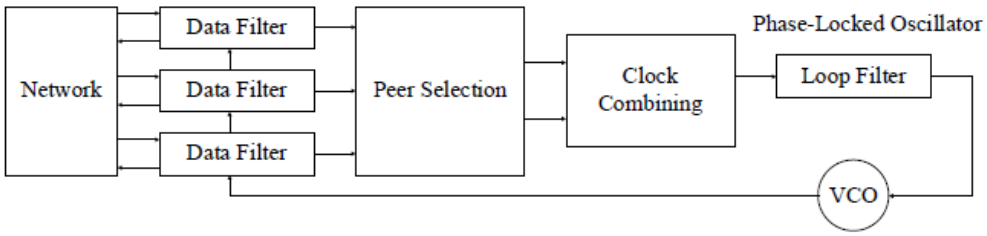


Figure 3.5: NTP Control Loop [14].

Clock synchronization between the cameras in the system can be achieved by assigning one camera as an NTP server that broadcasts its own internal clocks to any client cameras that choose to use it as their time reference, as seen in Figures 3.6 and 3.7. Synchronization through NTP possesses an accuracy up to the millisecond, which for the purpose of this implementation is enough to guarantee a solid base for the frame capture accuracy between cameras.

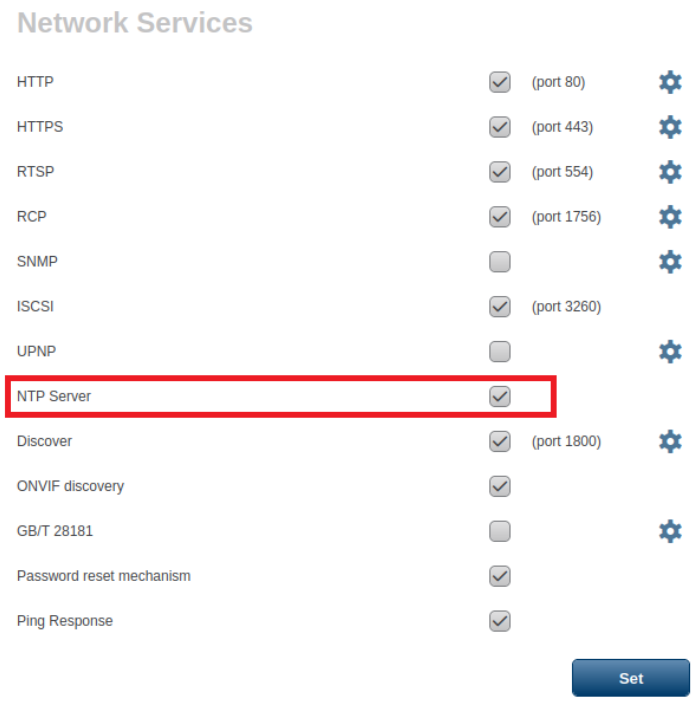


Figure 3.6: Enabling NTP Server on the cameras configuration page.

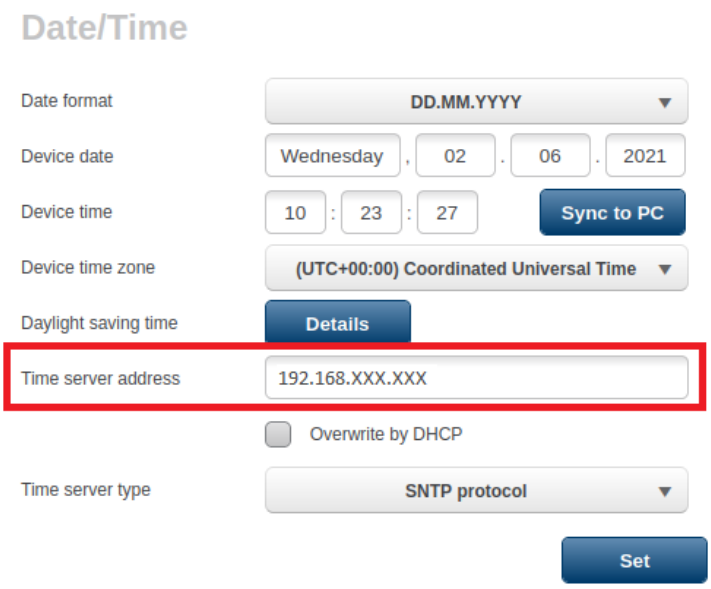


Figure 3.7: Choosing the NTP server address in the cameras configuration page.

3.2.2 Precision Time Protocol

Precision Time Protocol (PTP) is an improved version of the NTP protocol, aiming to provide accuracy up to nano-second or even pico-second level. This is done by using hardware timestamping instead of software and also by taking into account the time synchronization messages spend in each device. PTP is mostly used in scientific or financial environments where high precision timestamping is required [23]. Although PTP relies on specialized hardware in order to maintain a high level of precision in the timestamps, it can still be used without this hardware with relative success, and is referred as a software-only implementation [24], which applies the same principles as normal PTP except it is implemented in higher layers of the network causing jitter. The PTP protocol works similarly to NTP, therefore most implementations that already work with NTP can be easily reconfigured to use PTP, this is also true for the synchronization method proposed in this dissertation, as it only makes use of protocols synchronization capabilities without requiring any additional modifications to these.

3.2.3 Real Time Streaming Protocol

The cameras stream their feeds using RTSP which is an application-level protocol that enables data delivery with real-time properties such as video and audio [25]. This algorithm is also similar in context to HTTP to facilitate adding extension mechanisms that would work on HTTP.

To reduce the images into a more manageable size and be able to send more data over a network, the cameras use the H.264 compression format which works in 2 layers: The first layer divides the image in small grids and updates parts of the image as something changes; The second layer adapts the bit strings generated by the compression layer to various types of network and multiplex environments [26]. This ensures the performance and error resilience of videos which are very important characteristics for IP-based streaming applications. The response of the encoder to change in the image recorded is clearly visible in the encoder statistics tab of the cameras presented in Figure 3.8. The spikes in encoding are due to increased change of the grids in the recorded image.

Increasing the change in the image also increases the load on the network which may cause jitter on the streams if not taken into account. For that, the cameras allow us select between various encoder profiles and set a maximum bit-rate threshold like exemplified in Figures 3.9 and 3.10, which can be used to minimize this effect and, as it will be explained later, help with maintaining the synchronization of the streams.

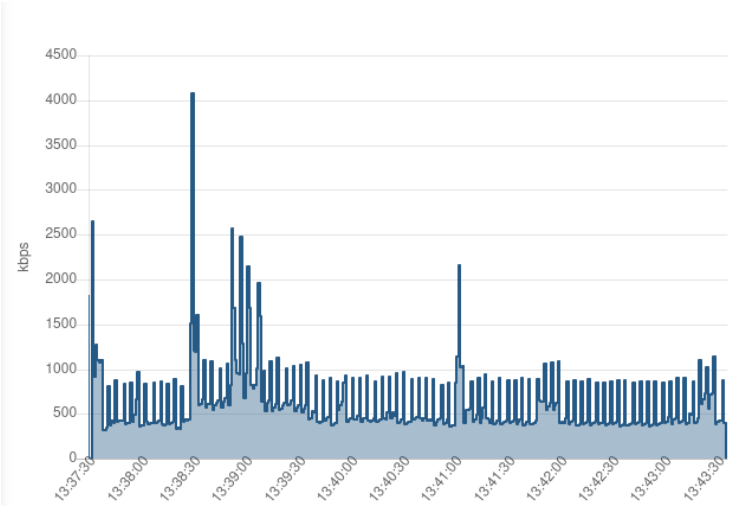


Figure 3.8: Encoder Statistics from the cameras. The rate of compression in the cameras is measured in kbps and a clear spike caused by change in the image can be seen at the timestamp 13:38:30.

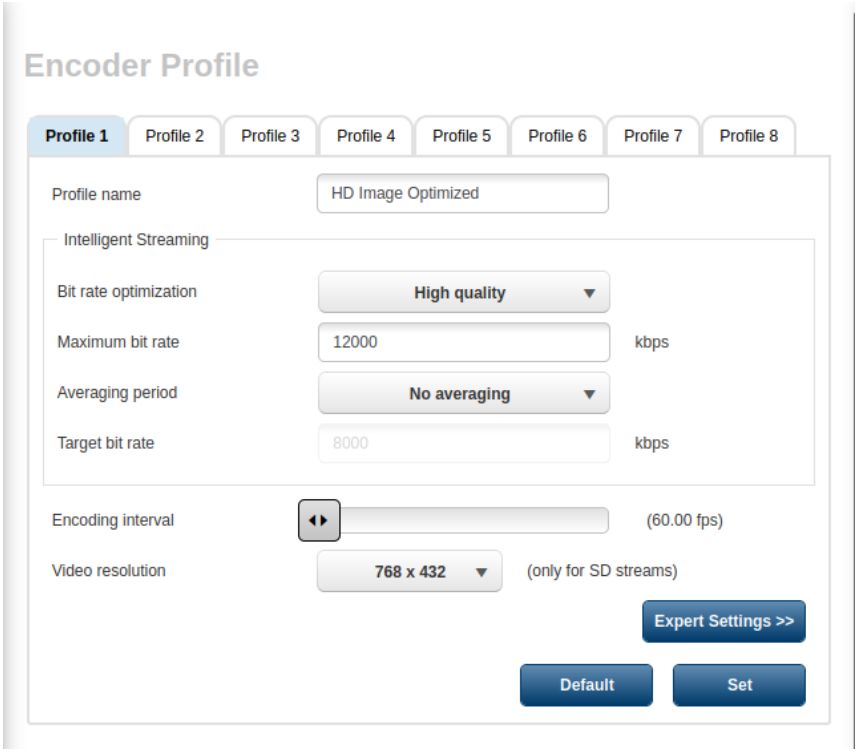


Figure 3.9: Encoder Profile 1 featuring a higher bit rate focusing on image quality.

The receiving of video feeds from the cameras is done by inserting the RTSP IP into the python script, in this case recurring to the open source computer vision library OpenCV [27], which provides all the necessary computer vision related functions required for this implementation. An RTP request is then done by the client in order to obtain the camera feeds from the RTSP server.

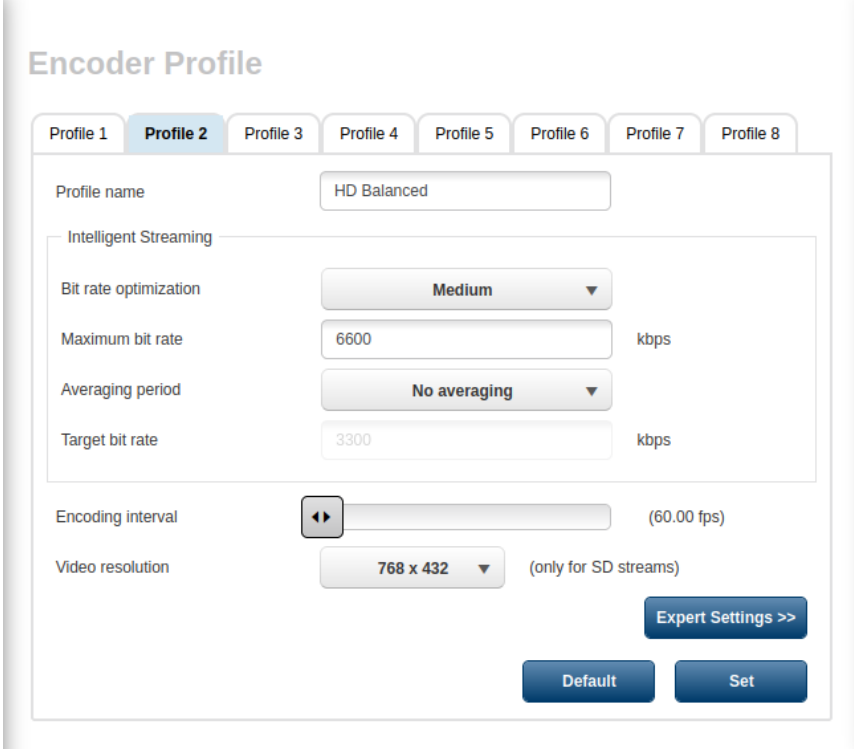


Figure 3.10: Encoder Profile 2 featuring a medium bit rate focusing on balanced image quality.

Between packets containing video footage, the protocol will also periodically send a Sender Report inside a Real-time Transport Control Protocol packet (RTCP) like the one described in Figure 3.11. This report contains information such as the "wall-clock" timestamp from the cameras. That is, the camera's internal clock timestamp at the time at which the packet was sent out from the camera to the Host PC. This timestamp value is described in the NTP format of Most Significant Word (MSW) and Least Significant Word (LSW), which represent respectively the number of seconds and microseconds since the NTP epoch time (January 1, 1990 00:00:00). It also contains information about the number of packets sent by the camera to the Host since the beginning of the stream [28].

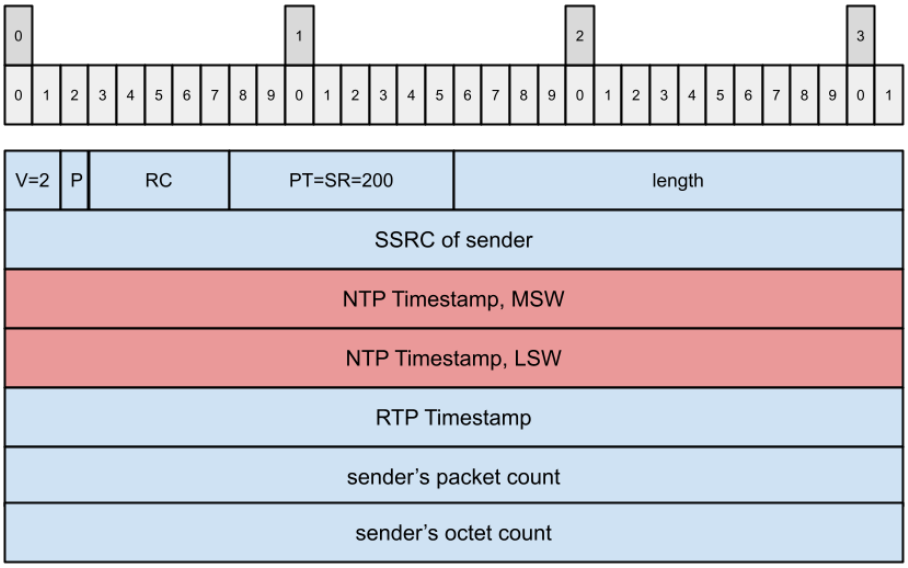


Figure 3.11: Sender Report RTCP Packet [28].

3.3 Timestamp Acquisition from RTCP packets

After having all camera clocks synchronized to a single NTP server, the cameras can begin transmitting video through RTSP. After an initial transmission of video packets the cameras send out a RTCP packet like the one shown in Figure 3.12. By then extracting the "wall-clock" timestamp that it contains, we can compare it with the ones from other cameras to obtain the time offset between the transmission of video streams from each device. This is only valid because it is assumed that the cameras are synchronized to the same time reference through NTP as stated before.

A TCP "packet sniffer" is used to filter the Sender Report RTCP packets from other exchanges happening in the network. In this case, a python script listens to the RTSP reception ports at the Host PC and grabs any packets containing the specifications of a Sender Report such as their length and packet type, and then extracts the timestamps in the NTP notation of MSW and LSW described previously which is then used to calculate it's offset relative to the other timestamps received.

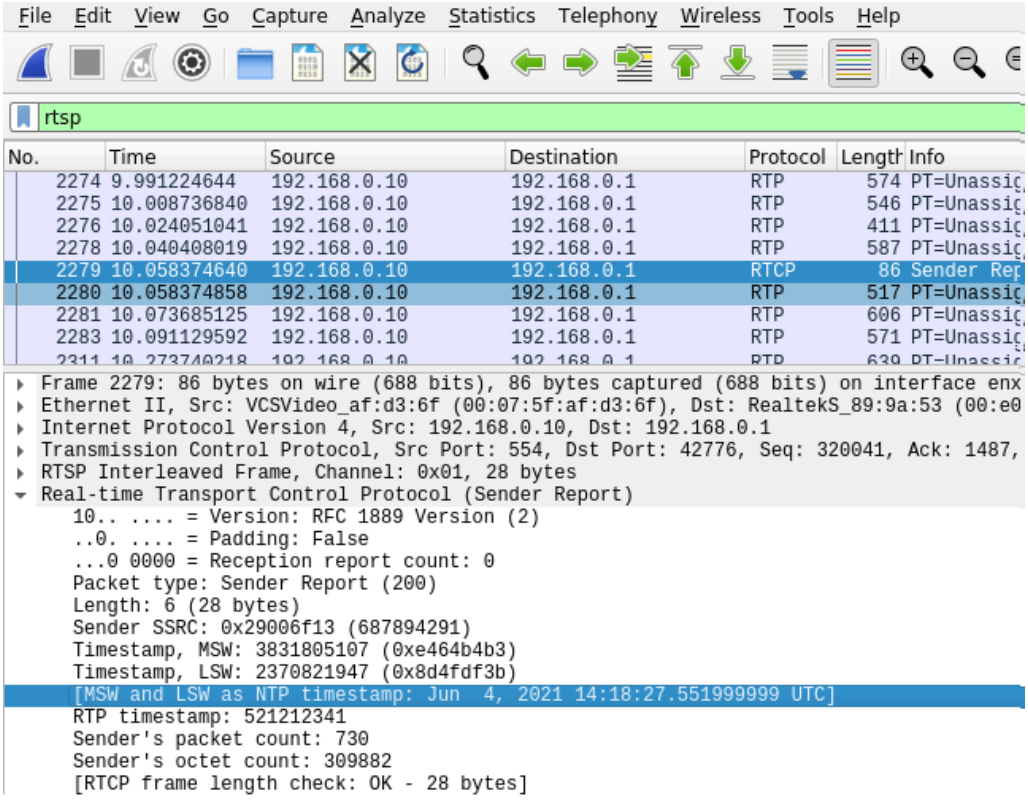


Figure 3.12: Observed RTCP Sender Report packet in Wireshark from one of the cameras during a stream.

This time offset is then converted to number of frames needed to shift the video streams based on the frequency of capture in which the cameras are setup, and thus we obtain the synchronized video feeds. A visual representation of this implementation is shown in Figure 3.13.

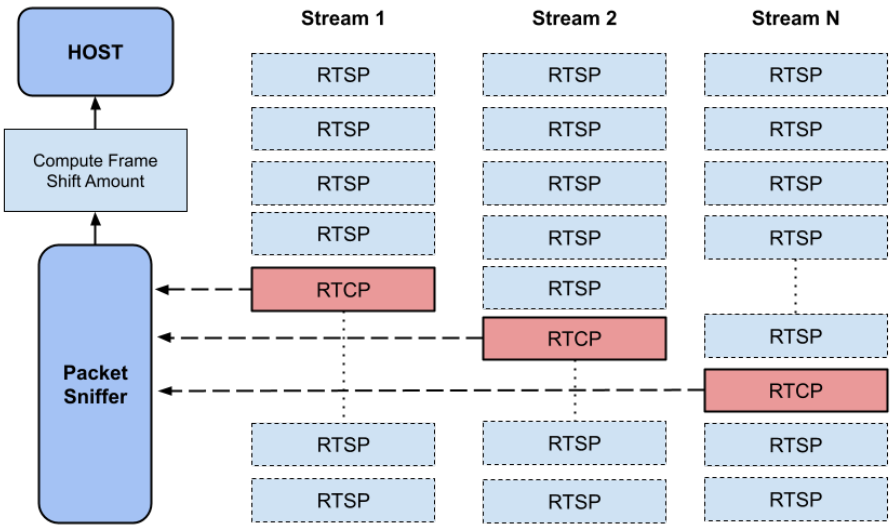


Figure 3.13: Obtaining frame shift from RTCP packets.

It is important to note that sometimes the RTCP packets come inside regular RTP packets instead of a single packet containing the RTCP with the timestamp like shown in Figure 3.14. These RTP packets include other payloads that are not related to the control part of the stream and therefore must be filtered to allow the extraction of the RTCP packet.

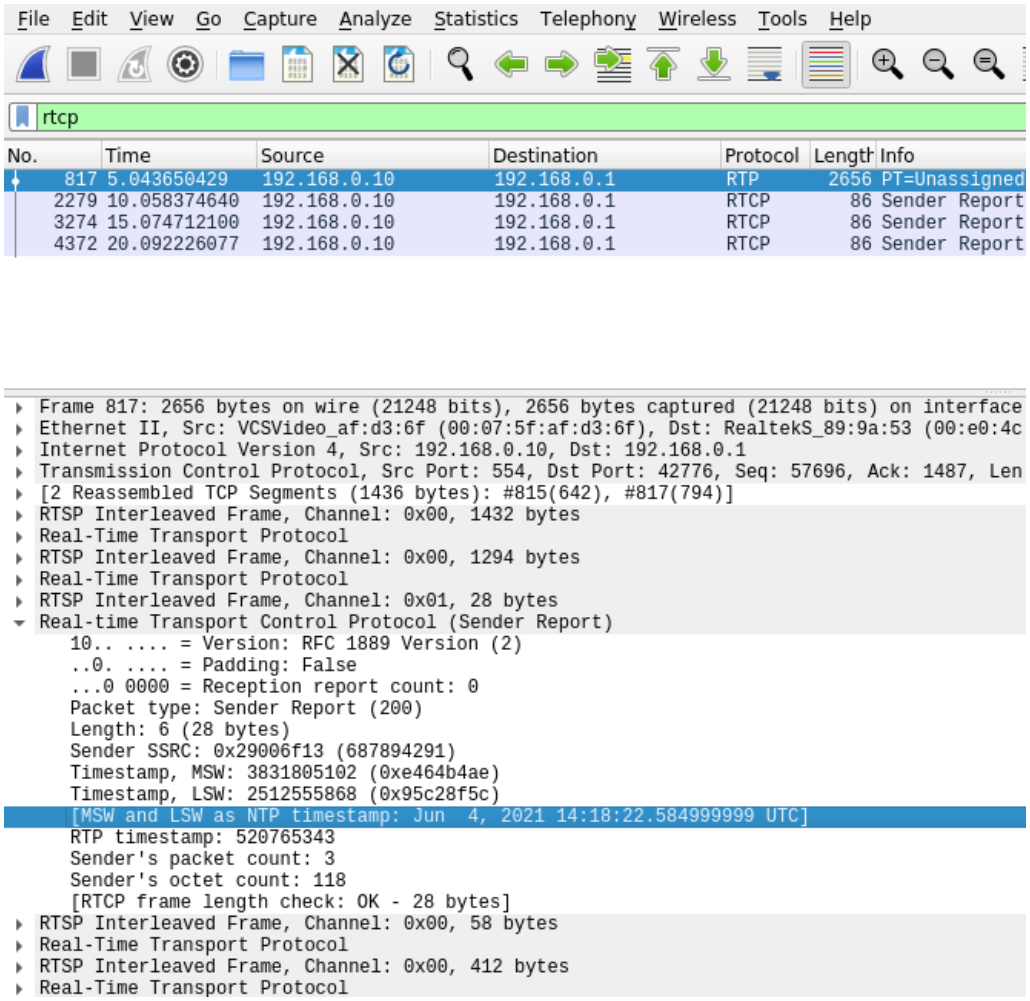


Figure 3.14: Observed RTCP Sender Report packet in Wireshark from one of the cameras during a stream which is interleaved inside a regular RTP packet.

In figure 3.15, we can observe an example of the terminal print obtained when two different RTCP Sender Report packets are detected by the "packet sniffer". Information from each packet is shown along with the decimal and hexadecimal values for the MSW and LSW NTP timestamps for debbuging purposes. These timestamp values are also converted to actual time and in the last output line we can see the calculated offset between the two reports.

```
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 6 Source Address: 192.168.0.10 Destination Address: 192.168.0.1
MSW int:
3833014800
LSW int:
926000.0001932494
MSW hex:
e4772a10
LSW hex:
ed0e5604
2021-06-18 14:20:00.926
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 6 Source Address: 192.168.0.11 Destination Address: 192.168.0.1
MSW int:
3833014801
LSW int:
987000.0001944136
MSW hex:
e4772a11
LSW hex:
fcac0831
2021-06-18 14:20:01.987
Calculated time offset: 0:00:01.061000
```

Figure 3.15: Terminal output of the received RTCP packets and offset value obtained.

3.4 Video Pipelines

A video pipeline is a set of components used to process an image from its source (in this case a video camera) until its desired image renderer. It can be composed by a series image encoders, decoders, format converters and communication protocol specifications, depending on the pipeline's configuration.

3.4.1 OpenCV default pipeline

OpenCV's I/O module is built as a two-layer interface which allows the read and write functions to use available video APIs such as Direct Show, Video for Windows, Video for Linux, GStreamer, FFMPEG, among others, as backend to perform these operations [29]. However OpenCV's default pipeline will use FFMPEG to decode RTSP streams if it is installed. FFMPEG is one of the most known media frameworks allowing operations such as decoding, encoding, streaming, filtering and other media related operations [30]. Using the FFMPEG's default OpenCV pipeline the following values of the transmission delay offset throughout a video capture for various capture frequencies were recorded and are displayed in the histograms of Figures 3.16, 3.17, 3.18 and 3.19.

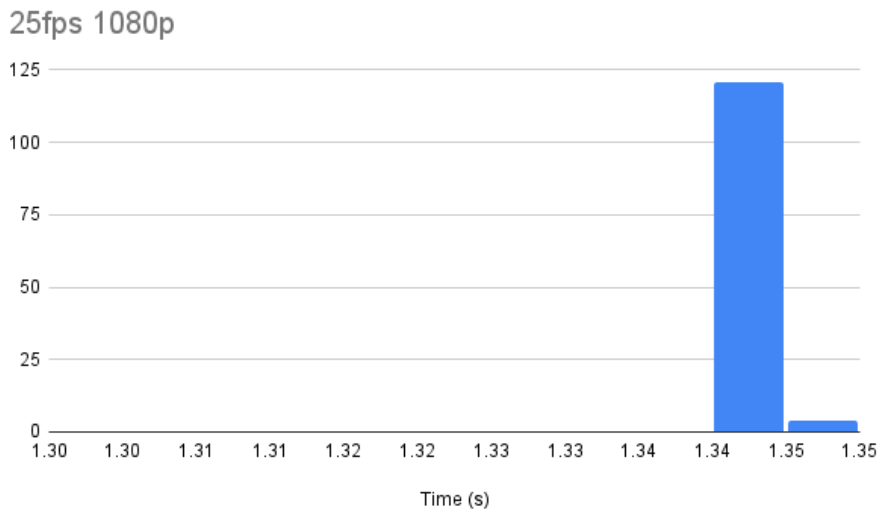


Figure 3.16: Histogram of the stream delay offset throughout a capture @ 1080p 25fps.

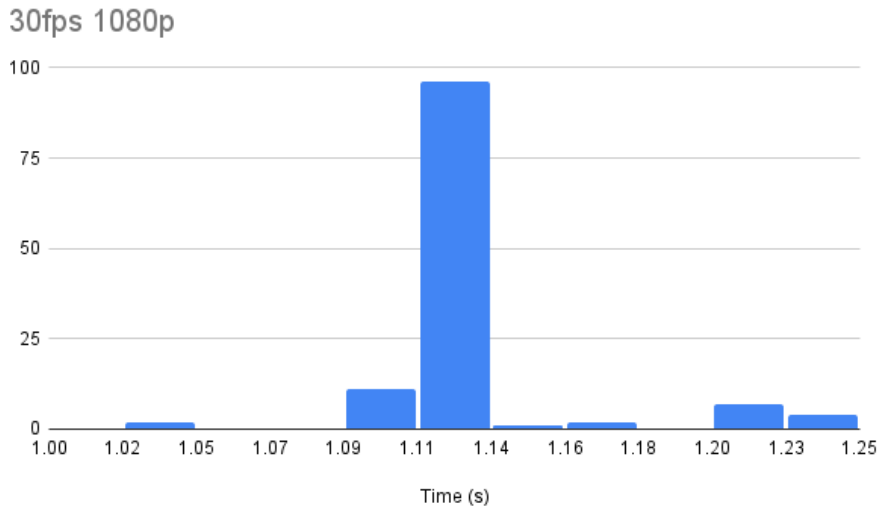


Figure 3.17: Histogram of the stream delay offset throughout a capture @ 1080p 30fps.

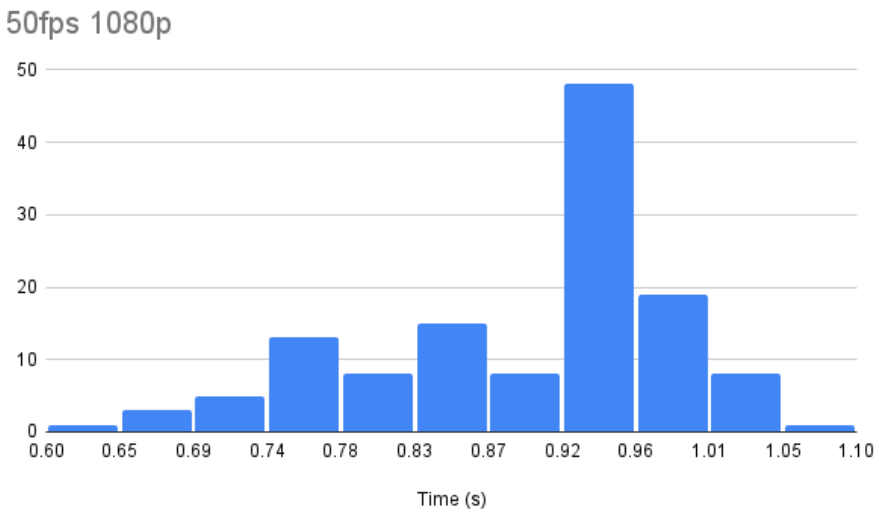


Figure 3.18: Histogram of the stream delay offset throughout a capture @ 1080p 50fps.

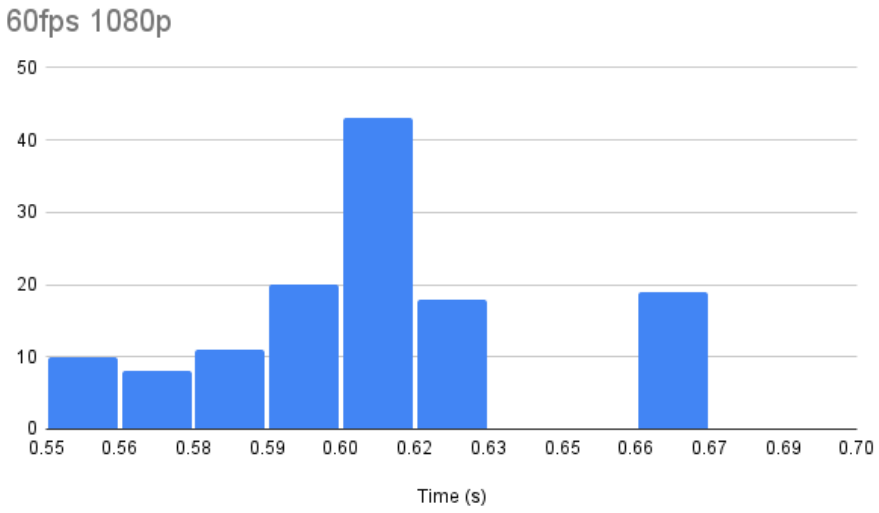


Figure 3.19: Histogram of the stream delay offset throughout a capture @ 1080p 60fps.

By analysing the histograms we can observe an inverse proportion between the capture frequency of a camera and its transmission offset delay, where a decrease in the capture frequency of the cameras equals an increase in the transmission offset delay between the streams, with the 60 FPS captures delivering approximately a 0.6 seconds offset while the 30 FPS capture ones almost double that value.

3.4.2 GStreamer pipeline

An alternative pipeline was setup using GStreamer with the aim of increasing the performance when receiving the video streams while also increasing its resistance to de-synchronization when high levels of bit-rate where imposed on videos. GStreamer is another multimedia framework that allows the linking of a variety of media processing operations, which can be designed to adapt to any development specific needs [31]. In this case, the streams are decoded and presented to the application using a new pipeline setup, which dramatically increased the speed of the streams reception while simultaneously decreasing the degradation of the sync level when exposed to high bit-rate feeds.

This new pipeline is setup as follows:

```
rtspsrc location=rtsp://service:Bosch2021.@192.168.0.10:554/?h26x=4
latency=0 protocols=GST_RTSP_LOWER_TRANS_TCP ! queue ! rtph264depay !
h264parse ! avdec_h264 ! videoconvert ! appsink
```

- **rtspsrc**: Plugin that instantiates an RTP session manager. It handles RTCP messages to and from server, jitter removal, packet reordering and provides a clock for the pipeline.
- **location**: URL of the RTSP stream
- **protocols**: Specifies the transport protocol used in this transmission (in this case TCP).
- **queue**: Allows the queuing of data until one of the limits specified has been reached. Gives the stream more stability as it won't have to wait for the next frame as long as it is queued.
- **rtph264depay**: Extracts H264 video from the RTP packets received.
- **h264parse**: Parses the H264 stream.
- **avdec_h264**: Decodes the H264 to audio-visual format
- **videoconvert**: Converts audio-visual format to video frames.
- **appsink**: Sink of the pipeline. Presents the final output ready to be used in an application (such as a python script).

The following histograms in Figures 3.20, 3.21, 3.22 and 3.23, show the transmission offset delay for every rate of capture obtained using this new pipeline.

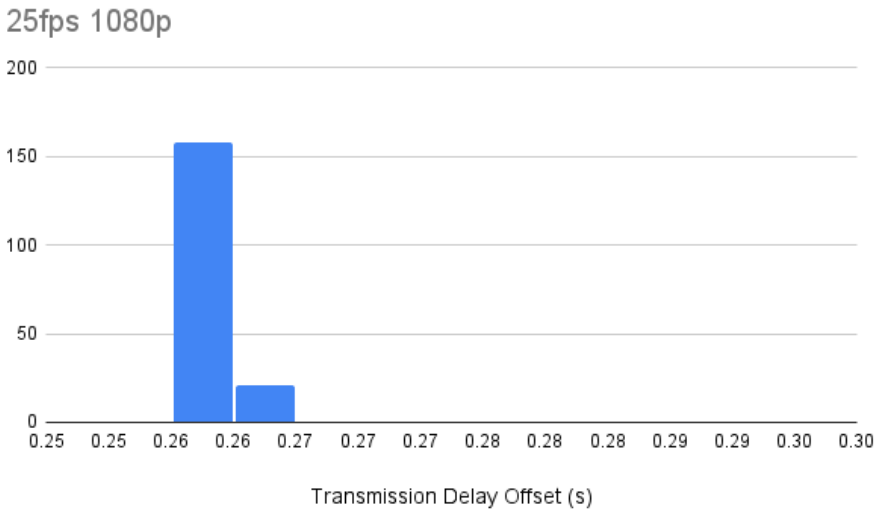


Figure 3.20: Histogram of the stream delay offset throughout a capture @ 1080p 25fps using the GStreamer pipeline.

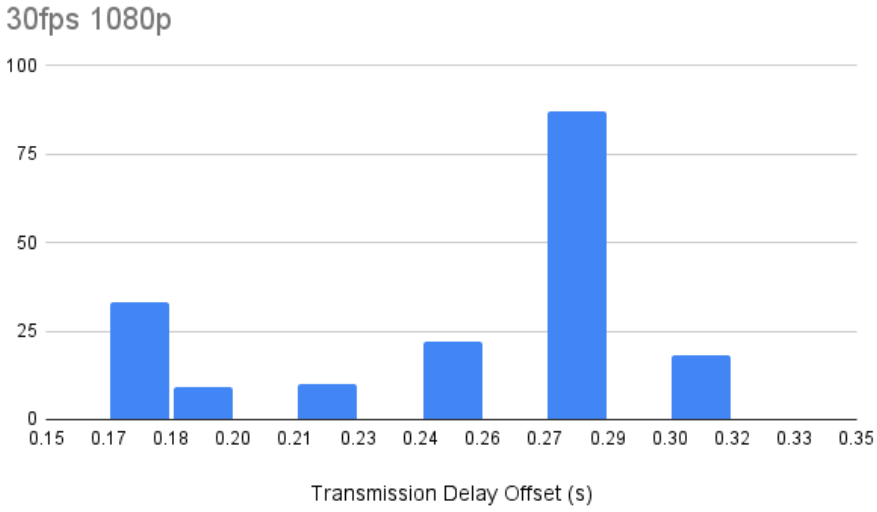


Figure 3.21: Histogram of the stream delay offset throughout a capture @ 1080p 30fps using the GStreamer pipeline.

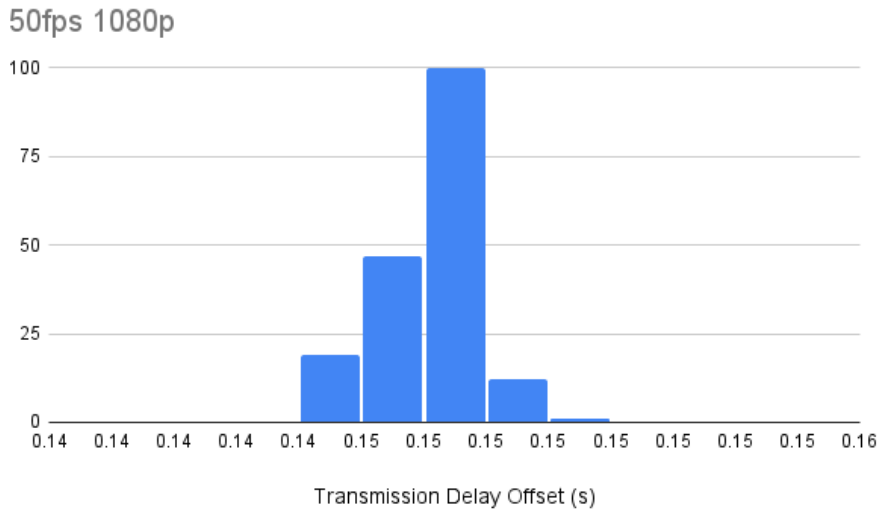


Figure 3.22: Histogram of the stream delay offset throughout a capture @ 1080p 50fps using the GStreamer pipeline.

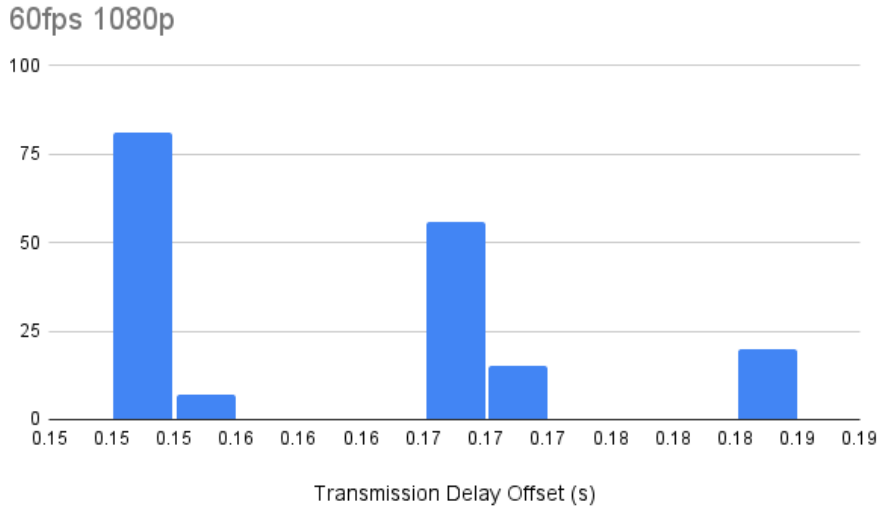


Figure 3.23: Histogram of the stream delay offset throughout a capture @ 1080p 60fps using the GStreamer pipeline.

As we can see the GStreamer based pipeline decreases the transmission offset by about a fifth of the original pipeline. Meaning that the performance of the video decoder and pipeline setup are the biggest contributors to the de-synchronization observed in video streams when they are deployed in closed local networks where the latency is minimal such as the one where these captures occurred. There is still the same tendency for lower delay values on the higher frequency captures like seen previously on the default OpenCV pipeline although not as prevalent. Some time offsets in the same capture are also big enough to shift the synchronization by order of a frame or two as seen in the 60 FPS capture in Figure 3.23 and the 30 FPS capture in Figure 3.21. It is important to state that none of these increases in latency affect the synchronization itself because the magnitude of the change is not big enough relative to the size of the queued frames (which was 100 in all captures).

3.4.3 VLC pipeline

A third multimedia framework was implemented using the VLC media player library for python. This time however OpenCV functions were not used as a frontend to read the video streams and instead were replaced by the equivalent operations from the library cited above. This allowed the streams to be decoded and inserted into the application at such speed that the synchronization algorithm wasn't even needed to correct any frame offsets between them. In other words, the NTP times reported by the Sender Reports in the RTCP packets were so close together in terms of time (sometimes less than 10 milliseconds apart), that no frame shift was needed to obtain synchronization between the observed streams. This was confirmed by looking at the real-time presentation of the streams and later, their recordings. An example of the offset calculated by the "packet sniffer" while using this pipeline at a capture frequency of 30 FPS is displayed in figure 3.24.

```
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 17 Source Address: 192.168.0.11 Destination Address: 192.168.0.1
MSW int:
3833013541
LSW int:
750000.000174623
MSW hex:
e4772525
LSW hex:
c0000000
2021-06-18 13:59:01.750
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 17 Source Address: 192.168.0.10 Destination Address: 192.168.0.1
MSW int:
3833013541
LSW int:
759000.0002549496
MSW hex:
e4772525
LSW hex:
c24dd2f2
2021-06-18 13:59:01.759
Calculated time offset: 0:00:00.009000
```

Figure 3.24: Terminal output of the received RTCP packets and offset value obtained using a VLC pipeline.

For the purpose of this dissertation most experiments for camera synchronization will still use the other pipelines described previously instead of this one, as they allow to more easily observe and study the effects of the proposed implementation on un-synchronous streams when the network characteristics such as latency are reduced, although it should be made clear that for any actual deployments were the elements that impact the de-synchronization of the streams are comprised of network constrictions such as latency and traffic, this pipeline should still be preferred.

3.5 Frame Synchronization

Frame synchronization between cameras in Network-Protocol-based solutions is usually done by a hardware or software trigger that syncs the starting moment of the cameras recording. For free running cameras such as the ones discussed previously, synchronization can be achieved by implementing a queue in each camera. This queue holds previously obtained frames from the video feeds, which allows us to retain information about the past captured frames or even the future frames that we have already captured but have not yet been shown in live feeds like shown in Figure 3.25. This way, within the size of the queue, we can shift the video frames independently to the "future" or past using the information about the frame offset value obtained from the RTCP Sender Reports, effectively obtaining frame synchronization between the cameras.

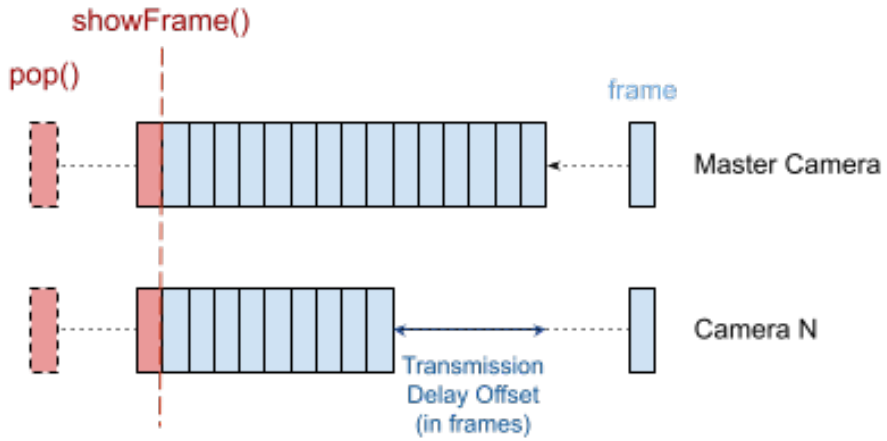


Figure 3.25: Transmission offset delay observed in the queues.

When shifting the video frames for multiple video sources, it is necessary to define a common instance to where all cameras will advance or go back to. If all streams are being read from the position zero in the queue, this can easily be achieved by finding the footage which is most advanced in time relative to the camera which is hosting the NTP server, and then shifting the queues of all the other cameras until that time frame. However, if we do not wish to lose such a big amount of frames by the other cameras, another method to determine the common point to which all cameras should sync to, is by determining the median time frame between all cameras and shifting the queues either forward or backwards to that point accordingly. This is done by

setting the pointer which extracts the current video frame from the camera queues to the middle of the queue, allowing the streams to be shifted to either the past or future. Diagrams explaining both procedures are exemplified in Figures 3.26 and 3.27.

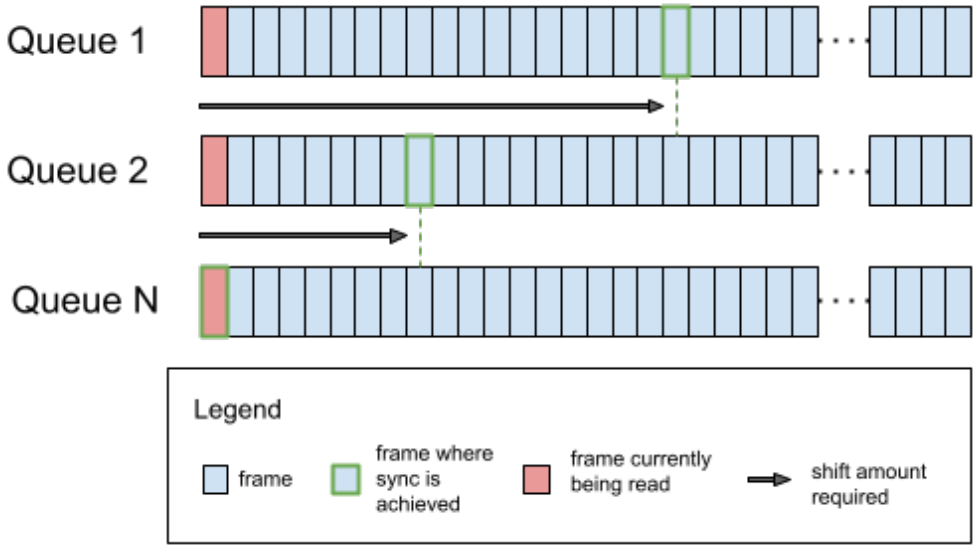


Figure 3.26: Diagram of the simplified frame shifting. All queues are initially being read at position zero and shift forward to synchronize with the other feeds.

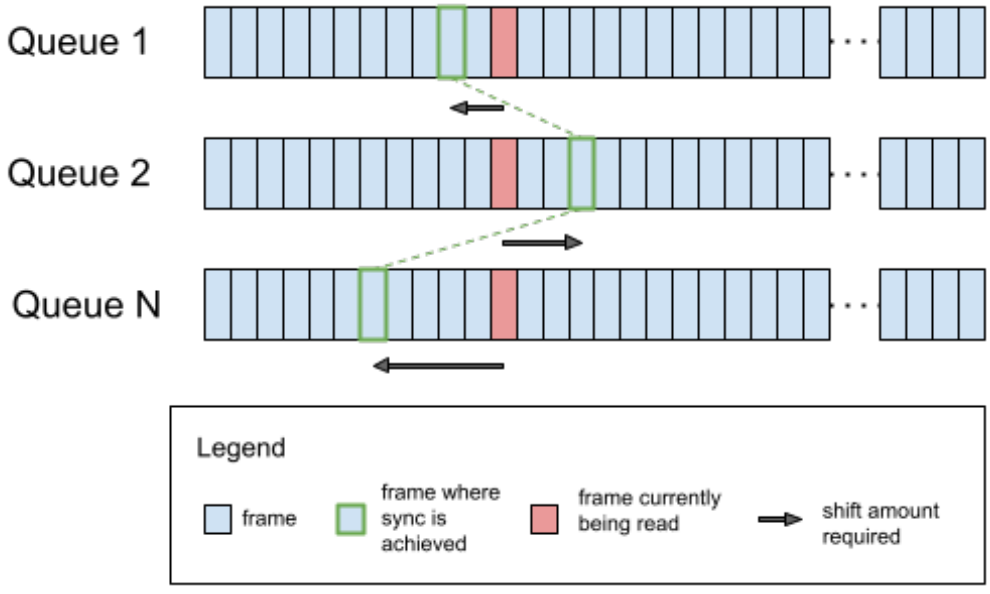


Figure 3.27: Diagram of the improved frame shifting. Queues are being read at their mid position as can shift forward or backward to synchronize with the other feeds.

3.5.1 Real-time Synchronization

The algorithm used to achieve synchronization between the video feeds in real time is described in the diagram of Figure 3.28. Each camera has a thread associated to it, whose responsibility is receiving and decoding the video frames that reach from the cameras at any time, and save those frames in the last position of the cameras associated queue. The Host PC, containing the main thread, accesses these queues to retrieve the frames in the position indicated by its pointer, and displays them to the user on-screen through the use of the *imshow* function from OpenCV. Another thread called TCP Sniffer is responsible of listening to any RTCP packets that come through the RTSP ports, retrieving their NTP timestamps and computing the respective stream offsets as described in section 3.3. These offset values are then passed to the Host's main thread who changes the index where the queue is being read, based on this value, to obtain synchronization between the video streams.

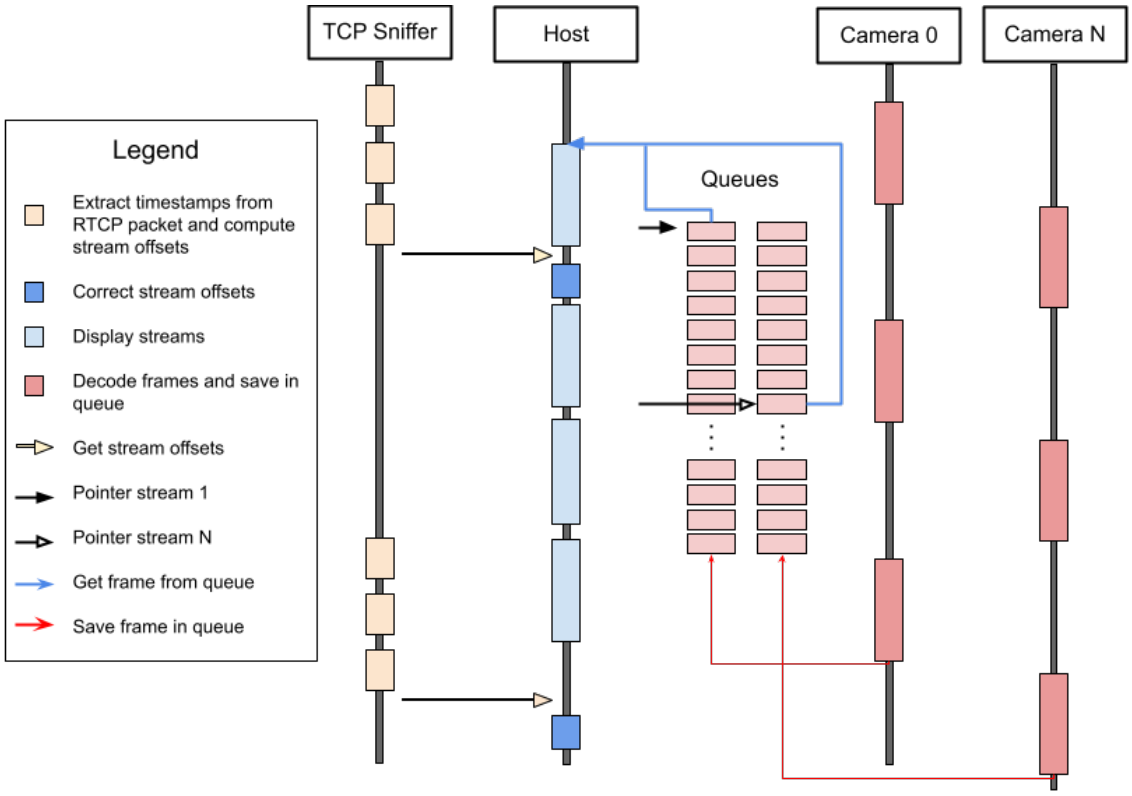


Figure 3.28: Diagram of the real-time synchronization. The threads for each object are visible along with the way they interact between each other.

The camera threads, unlike the Host and TCP sniffer ones, do not run freely and have a mutex associated to them to guarantee that no thread is starved because of another, and to stop them from running ahead of each other, which could create an additional offset of frames between the streams. This mutex makes the camera threads execute sequentially, looping back when the last thread is done with its task.

Locking in the capture threads is necessary because python does not feature real parallelism between threads due to the Global Interpreter Lock (GIL), which only allows one thread per process to be running at any time [32]. In consequence of this, without the mutex, sometimes the capture threads could run ahead of one another causing small frames offsets that could be observed on the video feeds.

3.5.2 Offline Synchronization

Similarly to the Real-Time synchronization the offline synchronization includes a Host, a TCP Sniffer thread and a thread for each camera with these last ones sharing a mutex to guarantee synchronization. The only difference in this case comes from writing the streams to an Audio Video Interleave (AVI) file after each frame is decoded and then, at the end of the capture, concatenating the already synced videos. A diagram demonstrating the described operation is shown in Figure 3.29.

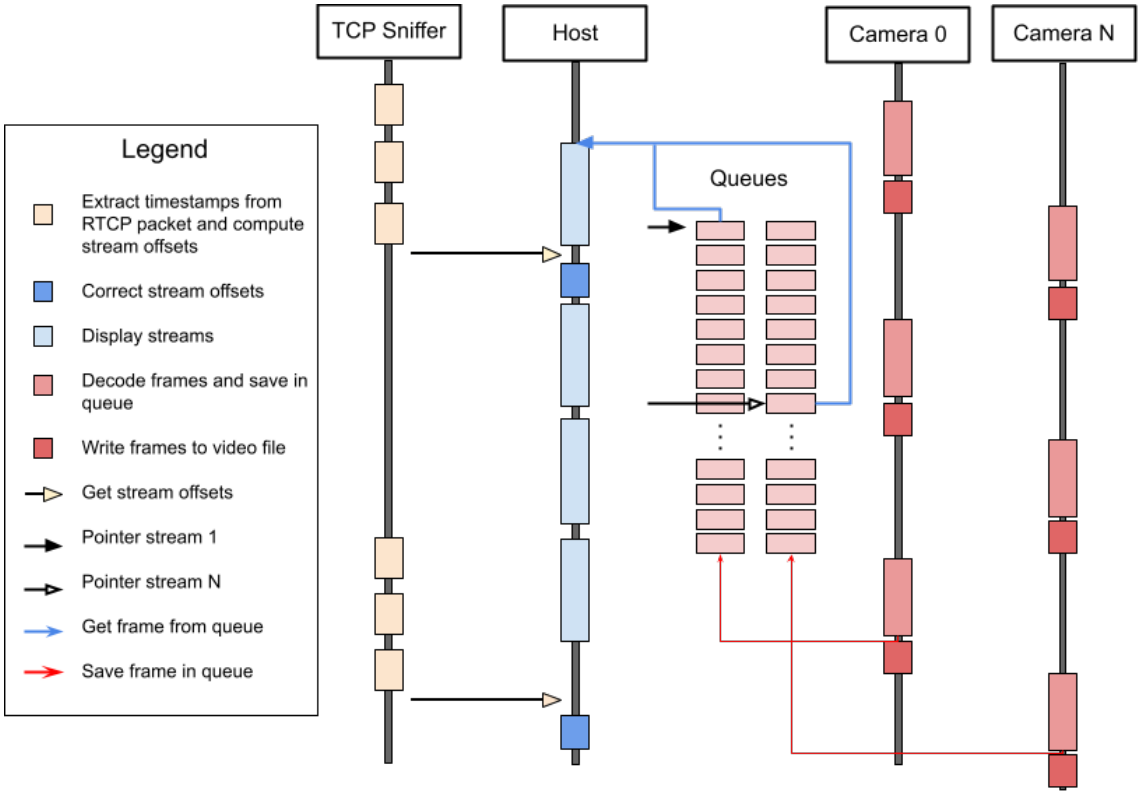


Figure 3.29: Diagram of the offline synchronization.

3.5.3 Alternative Frame Synchronization Method

Another way of obtaining synchronization can be done by removing restrictions from the capture process and limiting the size of the queue in situations where network characteristics such as latency are negligible. This works because when no limit is imposed on the speed that we are reading and showing the cameras feed, the frame that is presented will always be the most recently captured by the camera. So the offset between the feeds created at the beginning when starting to receive the feeds, disappears as the sizes of the video queues start to converge.

As shown in the graph in Figures 3.30 and 3.31, initially the queues are bombarded with information coming from the RTSP streams but at a certain point the amount of information processed by the program cycle responsible for presenting the footage on-screen, will begin to match the rate of the new incoming frames before overwhelming it and achieving real-time presentation of the streams. When done with multiple cameras, and the delay imposed by the network is much smaller than the other delays imposed in the system, the difference in size between the queues of each camera is equal to the offset amount of their respective streams, and so, when they converge to a value such as 1, their offset to one another will be non-existent and the streams are for all effects, synchronized. This characteristic is very evident in the graph of Figure 3.31, where we can observe that at a certain point all streams are initially apart by about 34 frames and then converge to a real-time synchronization.

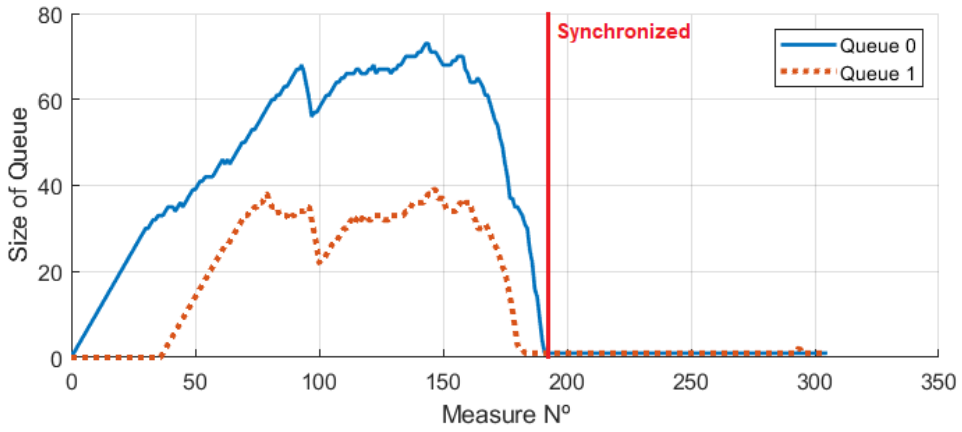


Figure 3.30: Time graph of the evolution of the queues size during a capture. Queue 0 is relative to the master camera which starts receiving packets first, and queues 1 is representative of a second camera which starts receiving after the transmission offset delay.

However, as mentioned before if the processing power of the Host's PC isn't fast enough, the queues will continue to grow without ever converging until the process runs out of memory. This effect can be seen in Figure 3.32 where the 3 queues continued to increase in size until the program ultimately ran out of memory. This occurred due to an increase in the size of display from the camera feeds on the terminal's screen, which increased the computing power necessary to carry out the program.

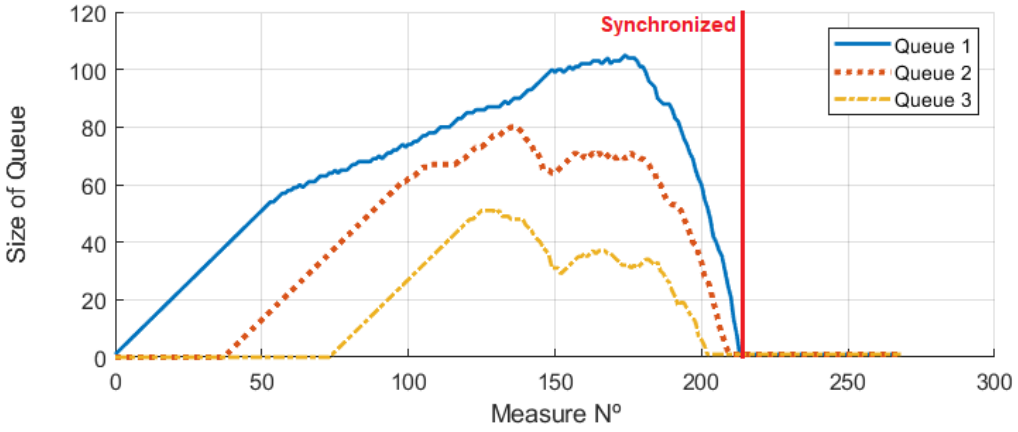


Figure 3.31: Time graph of the evolution of the queues size during a capture. Queue 1 is relative to the master camera which starts receiving packets first, and queues 2 and 3 are representative of a second and third camera which start receiving after the transmission offset delay.

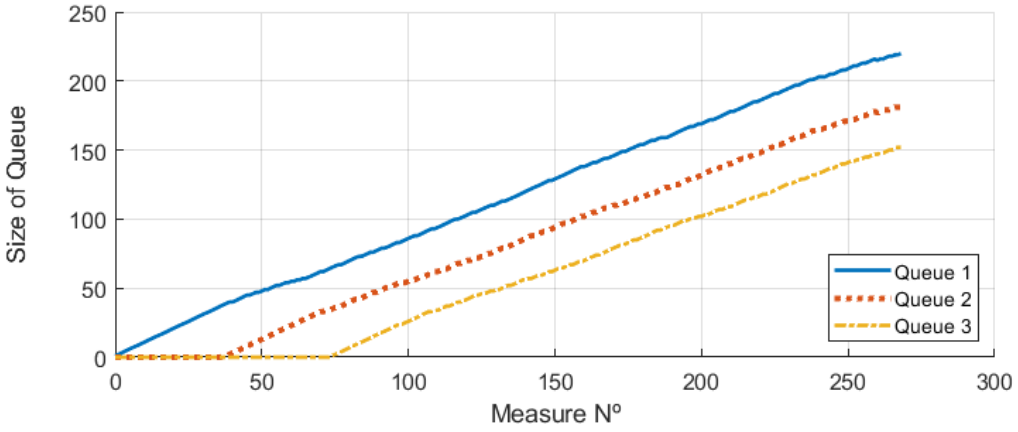


Figure 3.32: Time graph of the queue sizes during a capture where their values do not converge. Note that the delay imposed by the initial transmission offset is constant throughout the whole plot.

Therefore this method is flawed in the sense that it is highly dependent on processing power of the device where the program is implemented. If the size of the queues does not converge to one and instead caps at the max size of the queue, then to maintain the real-time display of the streams the algorithm will pop frames from the queue that have not been processed yet, which will cause a loss of frames in the video.

Also, because we are not controlling the rate at which the streams are captured, sometimes a process might lag behind the others by at least one frame for a bit before re-catching up. The result is a stream of videos that are sometimes behind one another by about 1 or 2 frames multiple times throughout a single capture. Which might be acceptable in some applications where how fast the streams are presented to an user relative to the time when they were captured, is more important than the accuracy of the synchronization itself, specially at high frame rates where the

time difference between one frame and another decreases greatly.

For the purpose of this dissertation, where the deployment of the sync is aimed at systems that require high precision in its synchronization of streams rather than presenting it faster to an user, we will focus on the first synchronization method described in this section.

3.6 Summary

Throughout the duration of this chapter we have conceived a method of synchronizing the video feeds that reach a Host server through the use of the camera characteristics along with the time information provided through the streaming protocols. This, along with the clock synchronization protocol that was already supported by the cameras, allowed us to calculate the amount of frame offset required between the video footage to obtain sub-frame synchronization, and shift the video streams in the queue of a camera object by the same amount.

Other elements such as video stream pipelines were also studied, in order to obtain faster and more reliable decoding methods of the stream that reach the Host from the cameras, which increases the systems reliability against sudden spikes in bit-rate.

A method that depends solely on the processing capability of the device in which this implementation is deployed, was also described and although we conclude it is not as reliable as the alternative proposed method, it revealed important characteristics in the way the video streams are processed by the algorithm which were used to increase the robustness of the final implementation.

Chapter 4

Results and Discussion

Having finally developed a solution to the synchronization problem, in this Chapter a series of experiments to determine the accuracy of this implementation and validate its deployment in the target use case are carried out. Firstly, an experimental case using two cameras recording a display with a running clock is carried out, and the differences in both the timestamps of the camera clocks and the timestamps of the display are analysed. This is followed by a second experiment were the feeds of two cameras are stitched to form a single extended view of the same scene and the continuity of the image is evaluated by having a person walk along this scene. Lastly various stress tests are conducted to confirm the robustness of the code when non-ideal conditions are imposed on the system.

4.1 Experimental test case with 2 cameras

4.1.1 System Architecture

To verify the accuracy of the proposed synchronization method, a system composed of a Host terminal running a python script and two cameras was setup like the one in Figure 4.1. Additionally a type of "clapperboard" which is composed by a single display with a higher frame rate than that of the cameras (120Hz), containing a running clock that shows the current time up to the millisecond is setup in the scene of recording.

The terminal runs on an Linux Ubuntu 20.04 x84 and uses an Intel Core i5-4300 @ 1.90GHz x 4 and 8 GB of RAM to process the operations required for this implementation. As for the running clock, a web page that displays a clock with millisecond accuracy [33] was used as it is enough to demonstrate the intended results of this work.

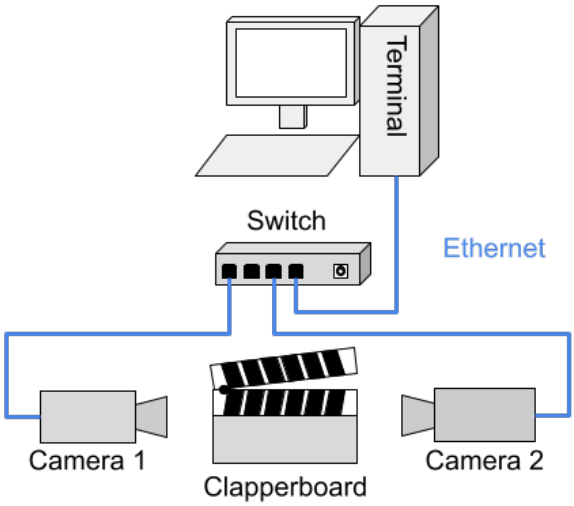


Figure 4.1: System architecture of the experimental case.

4.1.2 Results

In accordance to the architecture described previously, experiments were conducted for every one of the 4 available capture frequencies in the cameras (25, 30, 50 and 60 FPS). The captures consisted of approximately 10 minute sessions where the cameras are displaying their feeds at an 0.35 scale of the original image size to the user on-screen, while simultaneously recording the feeds into an AVI file. Because the displaying and writing of the videos are computationally demanding tasks, when the number of frames is increased these are performed more slowly, resulting in an increased latency relative to the real life events, meaning that even though all experiments were recorded during 10 minutes in real time, for the higher frequencies the actual amount of footage captured was of about 4 minutes.

In Figures 4.2, 4.3, 4.4 and 4.5 are displayed prints of both the "clapperboard" (mobile phone display) and cameras (black background) timestamp throughout various moments during a captured video. On top of every one of these prints there's also the approximate time offset between each one of them. Every Figure is accompanied by a table that indicates the values that we could make out from the timestamps observed in that respective print.

(a) Print at the beginning of the video.

(b) Print at the middle of the video.

(c) Print near the end of the video

Figure 4.2: Prints of the AVI file obtained at the end of a capture of a displayed clock @ 25 FPS.

Table 4.1: Values observed in the prints @ 25 FPS

Print nº	TS camera 1	TS camera 2	Δ Camera TS	TS diplay 1	TS display 2	Δ Display TS
1	17:44:48.142	17:44:48.149	7 ms	17:44:45.929	17:44:45.927	2 ms
2	17:48:47.622	17:48:47.631	9 ms	17:48:45.429	17:48:45.439	10 ms
3	17:52:45.101	17:52:45.111	10 ms	17:52:42.929	17:52:42:939	10 ms

(a) Print at the beginning of the video

(b) Print at the middle of the video

(c) Print near the end of the video

Figure 4.3: Prints of the AVI file obtained at the end of a capture of a displayed clock @ 30 FPS.

Table 4.2: Values observed in the prints @ 30 FPS

Print nº	TS camera 1	TS camera 2	Δ Camera TS	TS diplay 1	TS display 2	Δ Display TS
1	18:01:48.644	18:01:48.656	12 ms	18:01:46.509	18:01:46.536	27 ms
2	18:04:50.845	18:04:50.857	12 ms	18:04:48.723	18:04:48.735	12 ms
3	18:06:51.044	18:06:51.057	13 ms	18:06:48.939	18:06:48.956	17 ms

(a) Print at the beginning of the video

(b) Print at the middle of the video

(c) Print near the end of the video

Figure 4.4: Prints of the AVI file obtained at the end of a capture of a displayed clock @ 50 FPS.

Table 4.3: Values observed in the prints @ 50 FPS

Print nº	TS camera 1	TS camera 2	Δ Camera TS	TS diplay 1	TS display 2	Δ Display TS
1	18:22:16.375	18:22:16.371	4 ms	18:22:14.346	18:22:14.326	20 ms
2	18:24:17.975	18:24:17.971	4 ms	18:24:15.947	18:24:15:947	0 ms
3	18:26:19:715	18:26:19.732	17 ms	18:26:17:700	18:26:17:735	35 ms

(a) Print at the beginning of the video

(b) Print at the middle of the video

(c) Print near the end of the video

Figure 4.5: Prints of the AVI file obtained at the end of a capture of a displayed clock @ 60 FPS.

Table 4.4: Values observed in the prints @ 60 FPS

Print nº	TS camera 1	TS camera 2	Δ Camera TS	TS display 1	TS display 2	Δ Display TS
1	18:41:11.841	18:41:11.846	5 ms	18:41:09.897	18:41:09.897	0 ms
2	18:42:10.825	18:42:10.830	5 ms	18:42:08.883	18:42:08.883	0 ms
3	18:43:10.774	18:43:10.780	6 ms	18:43:08:833	18:43:08:833	0 ms

4.1.3 Analysis

By analysing the timestamps from both the cameras and displays, we can see that for all cases the accuracy is kept within the sub-frame range for all capture frequencies, namely: a time difference between frames bellow 40, 33.3, 20 and 16.6 milliseconds for a capture frequency of 25, 30, 50, 60 FPS respectively. The only discrepancy seen at one of the prints during the 50 FPS capture frequency is most likely due to the bad visibility of the numbers displayed in that particular print which causes us to have to more or less guess the actual value that is being shown on the display.

To better visualize the time difference between the two camera feeds, sixty samples containing the four timestamps, two from the cameras and two from the displays, were taken for every one of the four AVI files created previously. While studying the actual AVI files we can note that the remaining time difference between both feeds also never surpasses the value that enables the loss

of sub-frame accuracy. This is depicted in the charts of Figures 4.6, 4.7, 4.8 and 4.9, where the observed time difference (Δ) between the cameras and display timestamps were plotted.

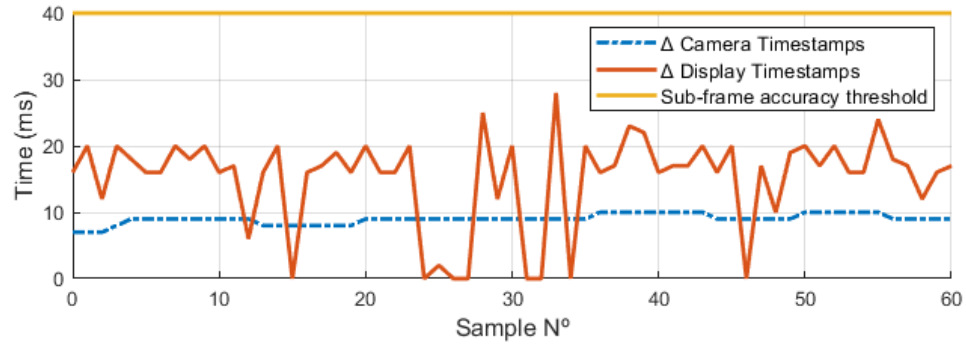


Figure 4.6: Chart of the Time Difference between feeds @ 25 FPS.

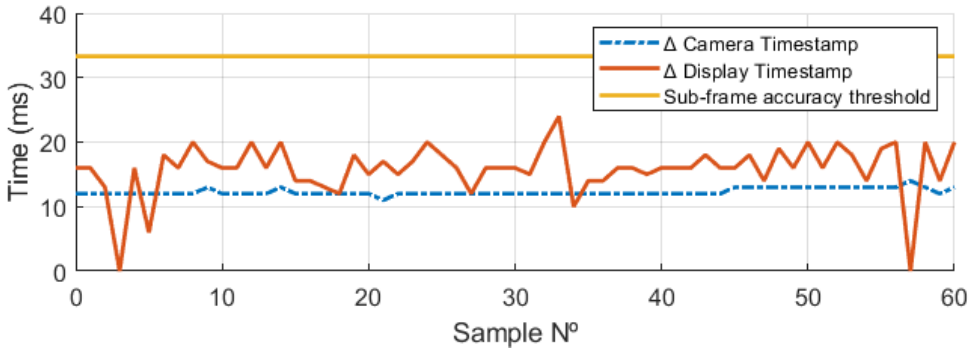


Figure 4.7: Chart of the Time Difference between feeds @ 30 FPS.

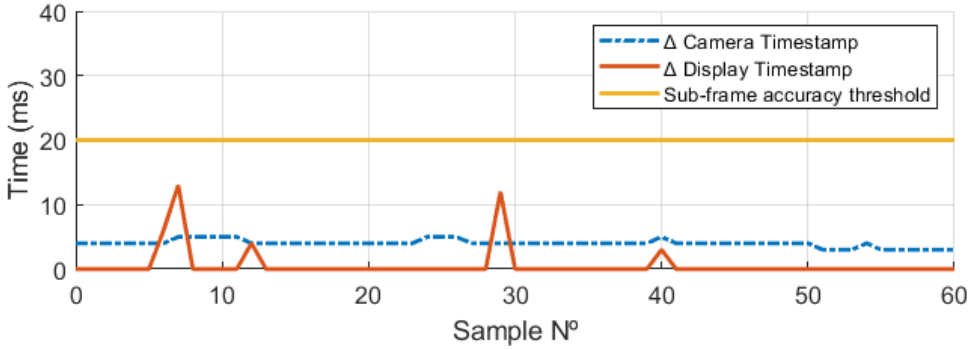


Figure 4.8: Chart of the Time Difference between feeds @ 50 FPS.

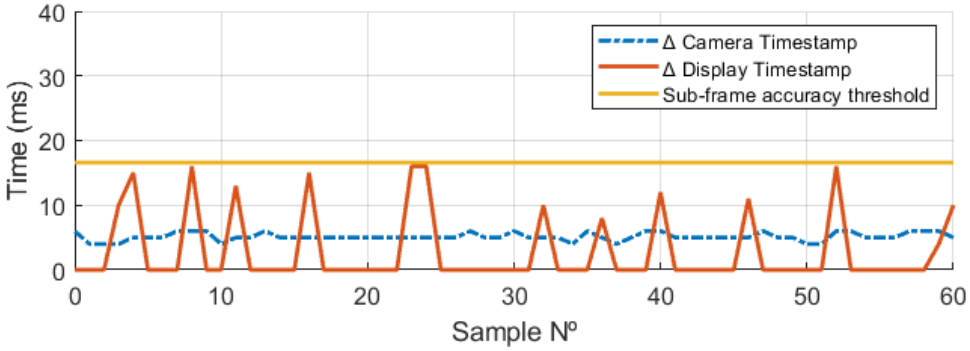


Figure 4.9: Chart of the Time Difference between feeds @ 60 FPS.

The variations in the difference between the display timestamps is caused by two things. First, because the capture rate of the cameras and the display frequency of the screen do not match, this sometimes causes a frame change in the display that is captured by one camera and is not yet seen in the other. Second, due to the difficulty in properly reading the values in the milliseconds part of the display, as sometimes two or even more different timestamps can be faintly seen in the same frame, causing ambiguity in determining the actual time that is being shown. We can see these effect are diminished as the difference between camera timestamps gets closer to zero, which is noticeable in the 50 FPS capture shown in Figure 4.8 where this difference was almost always 4 milliseconds, which translated into an non-existent visual difference between the timestamps values observed on the display by both cameras.

4.2 Experimental test case with video stitching

4.2.1 System Architecture

Another experiment used to verify the validity of synchronization was done by stitching the video feed of two cameras recording the same scene and observing the continuity of the movement of the model transitioning between the two feeds. An example of the system architecture for this demonstration using the same equipment specifications mentioned previously is shown in figure 4.10.

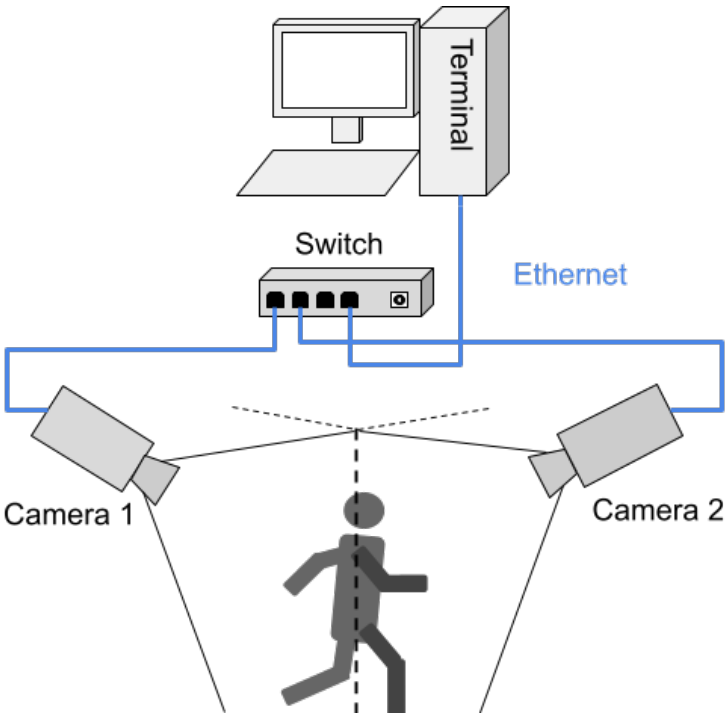


Figure 4.10: System architecture of the experimental case using video stitching.

4.2.2 Results

The continuity of movement was proved by having a person walk through a scene and then jumping through that same scene. The obtained results can be seen in a single frame of Figure 4.11 and in appendix A.0.1 for a complete transition between feeds of two different movements. For visibility purposes, the model's silhouette was highlighted and the cameras transition line is also indicated.

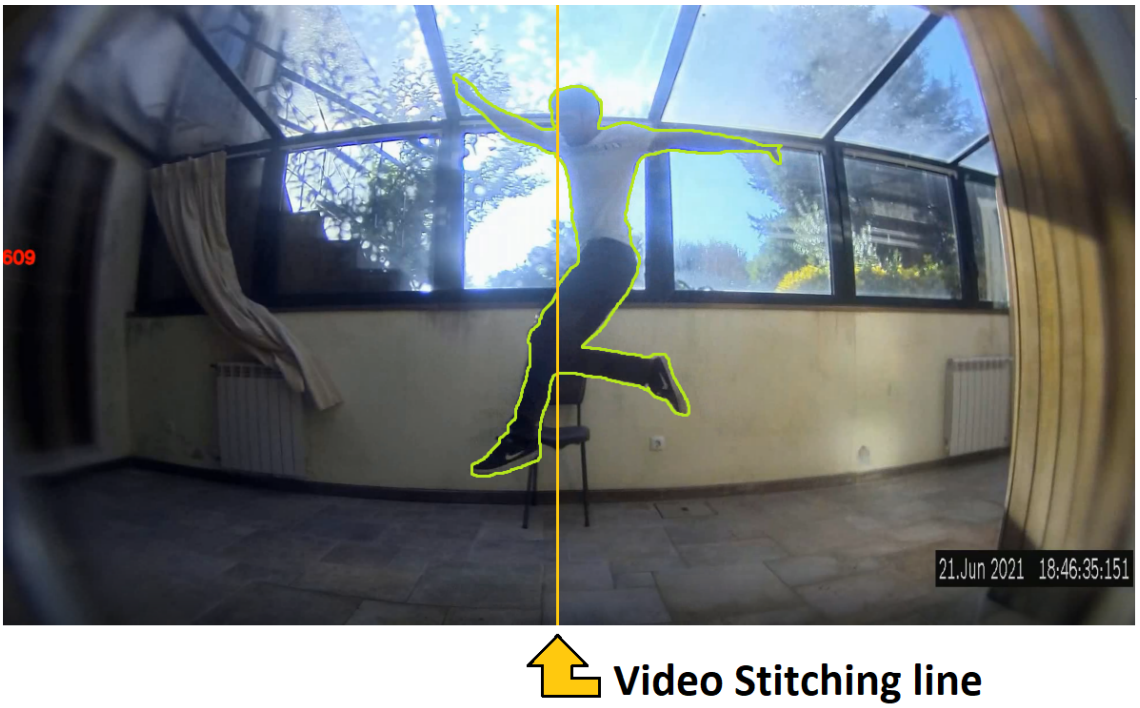


Figure 4.11: Image stitching example of two camera feeds.

As we can observe, the continuity of the movement in both actions is upheld as the deformation of the person when transitioning frames is imperceptible, proving that this implementation is accurate enough to be used for this kind of applications.

4.3 Stress Test

To guarantee the robustness of this implementation, some experiments were carried out with the intention of straining the network with increased latency, either by introducing this latency through software, by increasing the bit-rate captured by the cameras or leaving the system running for an extended period of time.

4.3.1 Latency Stress Test

When using GStreamer to decode the streams, the property *latency* can be inserted into the pipeline to simulate that same network characteristic. Latency in the network can only start to affect the synchronization of the streams once the latency amount increases above the inverse of the capture frequency of the camera. That is why for small network jitters (i.e. bellow 16,6 ms for a capture frequency of 60 FPS), the synchronization mechanism will not be affected because the latency increase or decrease is not enough to create a frame offset between the feeds, and much less a frame offset equal to the queue max size that it requires for it to be affected. In the following Figures we can see the obtained results when increasing the network latency for one of the cameras by 100, 500 and 1000 milliseconds:

```
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 6 Source Address: 192.168.0.10 Destination Address: 192.168.0.1
MSW int:
3833014888
LSW int:
959000.0002549496
MSW hex:
e4772a68
LSW hex:
f5810625
2021-06-18 14:21:28.959
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 6 Source Address: 192.168.0.11 Destination Address: 192.168.0.1
MSW int:
3833014889
LSW int:
220999.99995459802
MSW hex:
e4772a69
LSW hex:
389374bc
2021-06-18 14:21:29.221
Calculated time offset: 0:00:00.262000
```

Figure 4.12: Terminal output of the received RTCP packets and offset value obtained using a GStreamer pipeline with latency = 0 ms

```
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 6 Source Address: 192.168.0.10 Destination Address: 192.168.0.1
MSW int:
3833015156
LSW int:
327000.0001245644
MSW hex:
e4772b74
LSW hex:
53b645a2
2021-06-18 14:25:56.327
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 6 Source Address: 192.168.0.11 Destination Address: 192.168.0.1
MSW int:
3833015156
LSW int:
655000.0001804437
MSW hex:
e4772b74
LSW hex:
a7ae147b
2021-06-18 14:25:56.655
Calculated time offset: 0:00:00.328000
```

Figure 4.13: Terminal output of the received RTCP packets and offset value obtained using a GStreamer pipeline with latency = 100 ms.

```
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 6 Source Address: 192.168.0.10 Destination Address: 192.168.0.1
MSW int:
3833015220
LSW int:
593000.0002479646
MSW hex:
e4772bb4
LSW hex:
97ced917
2021-06-18 14:27:00.593
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 6 Source Address: 192.168.0.11 Destination Address: 192.168.0.1
MSW int:
3833015221
LSW int:
322000.0000023283
MSW hex:
e4772bb5
LSW hex:
526e978d
2021-06-18 14:27:01.322
Calculated time offset: 0:00:00.729000
```

Figure 4.14: Terminal output of the received RTCP packets and offset value obtained using a GStreamer pipeline with latency = 500 ms.

```
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 6 Source Address: 192.168.0.10 Destination Address: 192.168.0.1
MSW int:
3833014967
LSW int:
460000.0000698492
MSW hex:
e4772ab7
LSW hex:
75c28f5c
2021-06-18 14:22:47.460
----------------------FROM A RTCP PACKET-----------------
Version: 4IP Header Length: 5TTL: 64 Protocol: 6 Source Address: 192.168.0.11 Destination Address: 192.168.0.1
MSW int:
3833014968
LSW int:
688000.0002421439
MSW hex:
e4772ab8
LSW hex:
b020c49c
2021-06-18 14:22:48.688
Calculated time offset: 0:00:01.228000
```

Figure 4.15: Terminal output of the received RTCP packets and offset value obtained using a GStreamer pipeline with latency = 1000 ms.

As we can deter, the increase in the latency parameter value resulted in a corresponding increase in the delay between the streams. Even with the insertion of these characteristics in the pipeline, when analysing the footage it was concluded that the algorithm was still able to synchronize the feeds with the same sub-frame accuracy as before. The only way breaking the synchronization through this method is by increasing the latency to a value big enough to where the algorithm is unable to shift the stream due to lack of queue size, which is easily solved by increasing the number of frames that a queue from a camera can hold.

4.3.2 Bit-rate Stress Test

In the second stress test, an increase in the bit-rate in camera feeds is done by closely and rapidly waving a hand in front of the cameras which results in a increase in the amount of data that the cameras have to encode in H264, which itself creates a spike of latency in the network due to the amount of data being transmitted. In Figures 4.16 we can see an attempt to create this effect and in Figure 4.17 the subsequent spikes observed in the encoder statistics of the cameras.

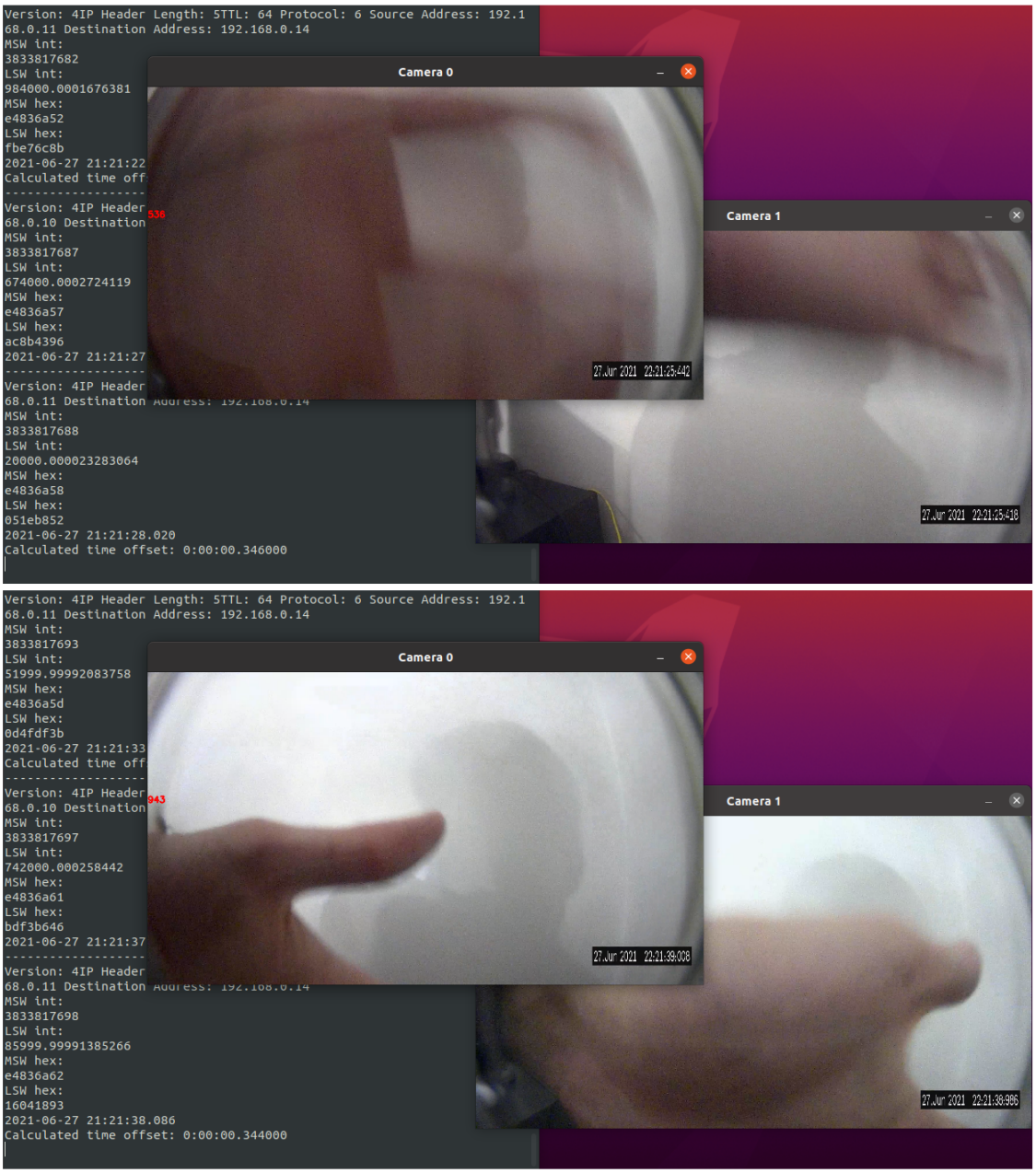


Figure 4.16: Bit-rate stress test. By waving a hand in front of the cameras it guaranteed that a pixel change will occur in all the grids of the H264 encoder, resulting in a increase of the transmitted data.

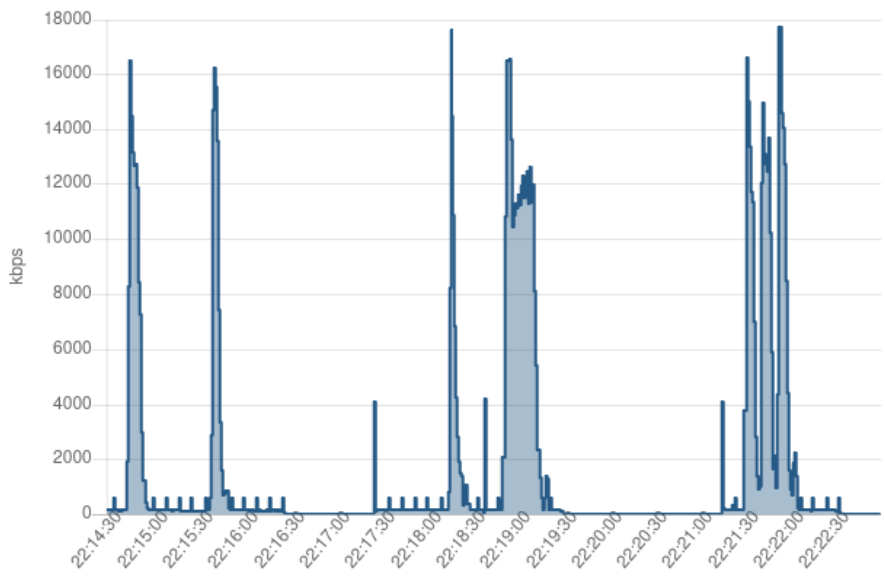


Figure 4.17: Encoder Statistics of one of the cameras during the stress test.

Having finished the experiment, it was concluded that even with the multiple raises of bit-rate the streams of the cameras were still able to maintain their synchronization. This is most likely due to the faster decoding pipeline provided by GStreamer, as in previous iterations of the implementation where the default OpenCV pipeline was used, a single spike in amount of encoded data would immediately result in a de-synchronization of the streams. However, as seen in section 3.2.3, the cameras maximum bit-rate can be configured in order to combat this issue at the cost of losing image quality.

4.3.3 Extended capture Stress Test

To be able to run through extended periods of time without losing its properties is one of the requirements of a algorithm implemented within a system composed of free-running cameras. Consequently, an experiment where the algorithm is left running for several hours on two cameras was carried out, having observed that even after a time period of 10 hours, the synchronization between the two feeds was still upheld, and therefore the algorithm is reliable enough to run for such extended periods.

4.4 Summary

With the achieved results throughout this chapter, we were able to determine that the accuracy of the proposed implementation stays within the sub-frame requirement for video synchronization and verify its viability for implementations such as video stitching, which consolidates its credibility as a reliable method for the video synchronization problem.

Also, when compared with other SOTA methods such as the ones in 2.1, we can see that it presents one of the highest accuracy rates in frame synchronization, only falling behind the methods like proposed in [17] and [18], which is to expect as these method do not operate on free-running cameras, allowing the initiation of the cameras capture times to be synchronized along with the reception of the streams, resulting in more accurate frame synchronization.

The stress tests also helped conclude that even in non-ideal system conditions such as high latency, high bit-rate or even extended capture periods of time, the algorithm should still be able to uphold the synchronization of the devices feeds.

Chapter 5

Conclusions & Future Work

5.1 Conclusions

The synchronization of video feeds in order to improve the performance of computer vision based algorithms such as structural analysis, Re-ID and event detection, is a highly desirable feature with the current developments in edge AI and IoT systems as we advance more and more into the era of smart cities.

Through the use of the characteristics from video stream packets sent by the cameras along with the properties of a network time synchronization protocol the implementation described in this work is able to synchronize the video feeds of free-running cameras even when the system's network in which they are deployed is faced with high latency constrictions.

Although most of the literature found on the camera synchronization subject did not have a practical use during the development of this solution, some of their inherent difficulties such as finding a common base frame of time, obtaining accuracy in adverse system environments and dealing with device hardware limitations, helped better understand the characteristics of the problem itself and consequently delineating a better solution for its resolution.

Initially a method was conceived to filter the control packets containing the timestamps from the cameras from other packets in the network, while simultaneously calculating the time offset between them and computing the frame amount for a given capture frequency.

Having obtained a way of temporally matching the video streams, a method was devised to shift the streams through the use of queues containing the capture frames. During the development of this solution, another way of obtaining a synchronized stream reception without almost no latency relative to the real time capture was achieved by removing any restrictions in the stream capture processes. However this method proved to be somewhat inaccurate in its frame synchronization relative to the final adopted solution, as sometimes the streams would run ahead of each other, causing some frames to be outright dropped before being presented to the user as the capture process outpaced the presentation of the obtained frames. With that being said, this method demonstrated that the use of a mutex to prevent the stream capture threads from running unrestricted, was a more optimal solution to obtain accurate sub-frame synchronization.

In order to validate the accuracy and robustness of the implementation, various experiments where carried out using a system composed by two of Bosch's free-running cameras, along with a Host terminal that ran the python algorithm. The synchronization tests allowed us to establish that the sub-frame accuracy was maintained throughout the various capture frequencies allowed by the camera, while the stress tests revealed how the method held its synchronization when non-ideal system conditions were imposed.

By then comparing the results with the current literature in multi-cam synchronization, it was determined that this solution is well encompassed within the target requirements for a synchronization method and can be used effectively as such in the target systems described throughout this work.

5.2 Future Work

As it currently is, the algorithm relies on code re-write to choose parameters such as number of cameras, stream URLs, capture frequency, etc. However in a future implementation, a graphical user interface could be designed to alter these parameters and add other functionalities that can be used mid capture such as start and stop recording video frames or creating a concatenated view of the presented streams.

Improvements can also be done to the code's processing speed by transitioning to a C++ implementation instead of the current python based one. This should allow it to surpass the limitation of one thread running concurrently per process due to GIL, which may be important when scaling this solution for a large number of devices.

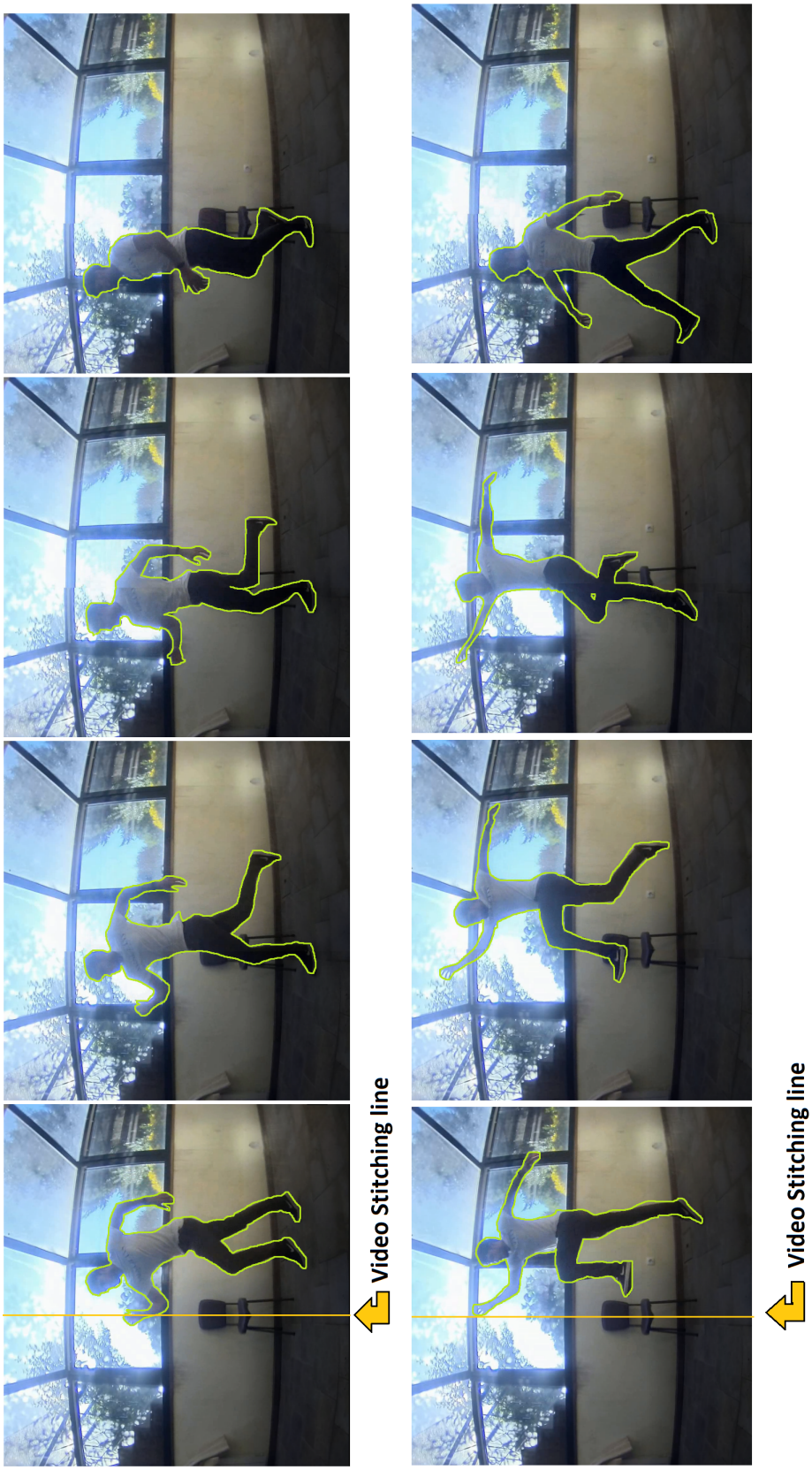
Also, although this implementation was tested for various types of non-ideal system characteristics, it is still yet to be deployed in the actual field, as the current ongoing coronavirus pandemic implied various restrictions and delays related to its experiment on a live-scenario.

Right now, this implementation is being prepared to be tested in the NVIDIA's Jetson which is an embedded system specially designed for processing computer vision and deep learning algorithms, which are great advantages for edge AI based systems.

Appendix A

Appendix

A.0.1 Image Stitching Examples



References

[1] Safe Cities Project. Safe cities. URL: https://www.bosch.pt/a-nossa-empresa/bosch-em-portugal/projetos-de-inovacao/safe-cities.html.

[2] Maria Valera and Sergio A Velastin. Intelligent distributed surveillance systems: a review. *IEE Proceedings-Vision, Image and Signal Processing*, 152(2):192–204, 2005.

[3] Prarthana Shrestha, Hans Weda, Mauro Barbieri, and Dragan Sekulovski. Synchronization of multiple video recordings based on still camera flashes. In *Proceedings of the 14th ACM international conference on Multimedia*, pages 137–140, 2006.

[4] M. Mehrabi, S. Lafond, and L. Wang. Frame synchronization of live video streams using visible light communication. In *2015 IEEE International Symposium on Multimedia (ISM)*, pages 128–131, 2015. doi:10.1109/ISM.2015.26.

[5] Flavio Padua, Rodrigo Carceroni, Geraldo Santos, and Kiriakos Kutulakos. Linear sequence-to-sequence alignment. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(2):304–320, 2008.

[6] Yaron Caspi, Denis Simakov, and Michal Irani. Feature-based sequence-to-sequence matching. *International Journal of Computer Vision*, 68(1):53–64, 2006.

[7] Anthony Whitehead, Robert Laganiere, and Prosenjit Bose. Temporal synchronization of video sequences in theory and in practice. In *2005 Seventh IEEE Workshops on Applications of Computer Vision (WACV/MOTION'05)-Volume 1*, volume 2, pages 132–137. IEEE, 2005.

[8] Bo-Song Huang, Day-Fann Shen, Guo-Shiang Lin, and Sin-Kuo Daniel Chai. Multi-camera video synchronization based on feature point matching and refinement. In *2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS)*, pages 136–139. IEEE, 2019.

[9] Anna Llagostera Casanovas and Andrea Cavallaro. Audio-visual events for multi-camera synchronization. *Multimedia Tools and Applications*, 74(4):1317–1340, 2015.

[10] Mario Guggenberger, Mathias Lux, and Laszlo Böszörmenyi. Audioalign-synchronization of a/v-streams based on audio data. In *2012 IEEE International Symposium on Multimedia*, pages 382–383. IEEE, 2012.

[11] G. Schroth, F. Schweiger, M. Eichhorn, E. Steinbach, M. Fahrmair, and W. Kellerer. Video synchronization using bit rate profiles. In *2010 IEEE International Conference on Image Processing*, pages 1549–1552, 2010. doi:10.1109/ICIP.2010.5653576.

[12] A. Al-Nuaimi, B. Cizmeci, F. Schweiger, R. Katz, S. Taifour, E. Steinbach, and M. Fahrmair. Concor+: Robust and confident video synchronization using consensus-based cross-correlation. In *2012 IEEE 14th International Workshop on Multimedia Signal Processing (MMSP)*, pages 83–88, 2012. doi:10.1109/MMSP.2012.6343420.

[13] Igor Pereira, Luiz F Silveira, and Luiz Gonçalves. Video synchronization with bit-rate signals and correntropy function. *Sensors*, 17(9):2021, 2017.

[14] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[15] Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008)*, pages 1–499, 2020. doi:10.1109/IEEESTD.2020.9120376.

[16] G. Litos, X. Zabulis, and G. Triantafyllidis. Synchronous image acquisition based on network synchronization. In *2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'06)*, pages 167–167, 2006. doi:10.1109/CVPRW.2006.200.

[17] Sameer Ansari, Neal Wadhwa, Rahul Garg, and Jiawen Chen. Wireless software synchronization of multiple distributed cameras. In *2019 IEEE International Conference on Computational Photography (ICCP)*, pages 1–9. IEEE, 2019.

[18] Akihito Noda, Satoshi Tabata, Masatoshi Ishikawa, and Yuji Yamakawa. Synchronized high-speed vision sensor network for expansion of field of view. *Sensors*, 18(4), 2018. URL: https://www.mdpi.com/1424-8220/18/4/1276, doi:10.3390/s18041276.

[19] Jim Easterbrook, Oliver Grau, and Peter Schübel. A system for distributed multi-camera capture and processing. In *2010 Conference on Visual Media Production*, pages 107–113. IEEE, 2010.

[20] Mario Guggenberger. *Synchronisation von Multimediadaten auf Basis von Audiospuren*. na, 2012.

[21] Bosch. Bosch global. URL: https://commerce.boschsecurity.com/xm/en/DINION-IP-ultra-8000-MP/p/12540006027/.

[22] How ntp works - official ntp documentation. URL: https://www.eecis.udel.edu/~mills/ntp/html/warp.html.

[23] Network timing technology: Ntp vs ptp. URL: https://www.masterclock.com/support/library/network-timing-ntp-vs-ptp.

[24] Kendall Correll and Nick Barendt. Design considerations for software only implementations of the ieee 1588 precision time protocol. 01 2006.

[25] rfc2326 - the real time streaming protocol (rtsp). URL: https://datatracker.ietf.org/doc/html/rfc2326.

[26] S. Wenger. H.264/avc over ip. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):645–656, 2003. doi:10.1109/TCSVT.2003.814966.

[27] Open source computer vision library - opencv, Nov 2020. URL: https://opencv.org/about/.

[28] rfc3550 - rtp: A transport protocol for real-time applications - section 6: Rtp control protocol – rtcp. URL: https://datatracker.ietf.org/doc/html/rfc3550#section-6.

[29] Video i/o with opencv overview. URL: https://docs.opencv.org/3.4/d0/da7/videoio_overview.html.

[30] Ffmpeg. URL: https://www.ffmpeg.org/about.html.

[31] Gstreamer. URL: https://gstreamer.freedesktop.org/.

[32] Global interpreter lock - python wiki. URL: https://wiki.python.org/moin/GlobalInterpreterLock.

[33] Javascript clock with milliseconds. URL: https://codepen.io/jasonleewilson/pen/gPrxwX.