Technische Universität Berlin

TU Berlin Industrial Automation Technology Department
Fraunhofer Institute for Production Systems and Design Technology
Pascalstraße 8-9, 10587 Berlin, Germany

Bachelor Thesis

Development and Evaluation of a Manufacturer-Independent Synchronization Framework for GenICam Industrial Cameras

Yessmine Chabchoub

Matriculation Number: 465977

Berlin, April 29, 2025

Supervised by
Prof. Dr.-Ing. Jörg Krüger
Prof. Dr. Sabine Glesner

Assistant Supervisor
M.Sc. Oliver Krumpek

Abstract

Synchronizing multiple industrial cameras from different vendors is a core challenge in machine vision, especially when sub-microsecond precision is required. Traditional methods rely on hardware triggers, which increase wiring complexity.

This thesis presents a vendor-independent synchronization framework for GenICam-compliant GigE Vision 2 cameras. It uses IEEE 1588 Precision Time Protocol (PTP) to align camera clocks over Ethernet networks. The implementation is compatible with any camera supporting the GenICam and GigE Vision 2.0 standards.

To ensure accessibility, the system includes a command-line interface (CLI) and a graphical user interface (GUI). These tools allow users to discover connected cameras, configure synchronization parameters, and perform scheduled image acquisition.

The framework was tested on a mixed Basler and Lucid Vision camera setup. Results demonstrated cross-vendor interoperability and sub-microsecond synchronization, with a measured temporal offset within ± 100 ns.

This work demonstrates that multi-camera synchronization is possible without hardware triggers or proprietary SDKs. It provides a standards-based foundation for further development and integration into existing systems.

Kurzfassung

Die Synchronisierung mehrerer Industriekameras verschiedener Hersteller ist eine zentrale Herausforderung in der industriellen Bildverarbeitung, insbesondere wenn eine Präzision im Sub-Mikrosekundenbereich erforderlich ist. Traditionelle Methoden basieren auf Hardware-Triggern, die die Komplexität der Verkabelung erhöhen.

Diese Arbeit stellt ein herstellerunabhängiges Synchronisations-Framework für GenICam-kompatible GigE Vision 2-Kameras vor. Es verwendet das IEEE 1588 Precision Time Protocol (PTP), um die Uhren der Kameras über Ethernet-Netzwerke zu synchronisieren. Die Implementierung ist mit jeder Kamera kompatibel, die die Standards GenICam und GigE Vision 2.0 unterstützt.

Um die Zugänglichkeit zu gewährleisten, umfasst das System sowohl eine Befehlszeilenschnittstelle (CLI) als auch eine grafische Benutzeroberfläche (GUI). Mit diesen Werkzeugen können Benutzer angeschlossene Kameras erkennen, Synchronisationsparameter konfigurieren und geplante Bilderfassungen durchführen.

Das Framework wurde mit einer gemischten Konfiguration aus Basler und Lucid Vision Kameras getestet. Die Ergebnisse zeigten die herstellerübergreifende Interoperabilität und die Synchronisation im Submikrosekundenbereich mit einem gemessenen zeitlichen Versatz von nur ± 100 ns.

Diese Arbeit zeigt, dass eine Multi-Kamera-Synchronisation ohne Hardware-Trigger oder proprietäre SDKs möglich ist. Sie bietet eine standardbasierte Grundlage für weitere Entwicklung und Integration in bestehende Systeme.

Declaration of Authorship

I hereby declare that I have completed this bachelor thesis independently, without the assistance of third parties, and solely using the sources and resources listed. All parts taken from these sources and resources, whether quoted directly or paraphrased, have been identified as such.

I used ChatGPT-4o for reformulation, phrasing, and formatting support and its DeepResearch mode to explore additional literature and relevant academic papers in the "State of the Art" chapter. I take full responsibility for the selection, adoption, and results of the AI-generated output I used.

[illegible]

Berlin, April 29, 2025

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Industrial automation relies heavily on sensor data, driving the need for increasingly accurate sensing technologies. Among these are image sensors embedded in industrial cameras, which capture visual information for downstream analysis. This visual data is processed to support automated decision-making and control, a process referred to as machine vision.

Machine vision is a growing field, seeing development in both software and hardware technologies. On the software side, machine vision image processing techniques are the focus of much ongoing research. With the integration of machine learning, for example, machine vision systems can now handle more complex scenes and unstructured environments. On the hardware side, cameras are the key component. With the growing market of machine vision, manufacturers are competing to offer high-resolution sensors and easy-to-use cameras. For instance, many manufacturers provide SDKs and APIs to streamline the setup process. These make the configuration process easier and facilitate the integration with existing software components, such as the processing software.

While growing research and competition between manufacturers have accelerated advancements in camera technologies, they have also led to significant diversification. To address this, manufacturers and associations such as the European Machine Vision Association (EMVA [3]) have collaborated to create common standards. These standards provide unified guidelines for hardware interfaces and camera driver development. For instance, GenICam [4] standardizes software-layer elements like API structures and feature naming. Another example is GigE2 Vision [5], a hardware interface that supports extended capabilities: GigE2 supports IEEE 1588 PTP, enabling precise camera synchronization over Ethernet.

These standards have elevated camera innovation for both users and vendors. With a defined framework, vendors can focus on designing higher-performing sensors while users benefit from easier usage.

However, despite the significant benefits of standards, implementing machine vision applications in real-time with multi-camera systems remains challenging.

1.2 Problem Statement

Although most cameras comply with industry standards, the full potential of these standards is often unrealized due to two main constraints. First, camera control is limited by proprietary software. Even with standardized APIs, manufacturers enforce their control software, forcing users to commit to a single brand or manage multiple software. Second, camera synchronization is both complex and critical for real-time processing. This is particularly relevant for moving objects and dynamic environments. Currently, hardware triggering and postprocessing methods, such as timestamp alignment, are the primary methods used for synchronization. Nevertheless, postprocessing is prone to many errors caused mainly by lost frames, while hardware triggering requires additional wiring [6]. These constraints hinder camera integration and prevent developers from fully leveraging open standards such as GenICam and GigE Vision.

1.3 Goal and Scope

This thesis presents a manufacturer-independent software solution for configuring, controlling, and synchronizing multiple GenICam-compliant GigE Vision 2.0 cameras. The approach involves reviewing existing synchronization methods, designing a software-based synchronization framework, and evaluating the resulting outcomes. This framework aims to enable users to easily configure and trigger synchronized recordings across up to six cameras.

The primary objective is to take advantage of the GenICam and GigE Vision 2.0 machine vision standards. To achieve this, the project will utilize the IEEE 1588 Precision Time Protocol (PTP [7]), available in GigE Vision 2.0 devices, for synchronization and will develop a control interface based on the GenICam standard and its wrapper, rc_genicam_api [8].

This thesis excludes tasks such as synchronizing peripheral hardware (e.g., lighting or external sensors) and supporting devices that do not conform to the GigE Vision 2.0 or GenICam standards. It does not delve into detailed networking topics such as network topologies. Postprocessing tasks and integration into broader machine vision pipelines are also not addressed. While these areas are critical for specific applications, they lie outside the immediate focus of this project, which centers on camera synchronization and user interface development.

These expected outcomes of this thesis can be summarized as follows:

1. **Evaluation of existing camera synchronization methods:** Includes an in-depth review of both hardware-based and software-based synchronization mechanisms. The evaluation will also focus on the key advantages and drawbacks of the different approaches, considering factors like integration, scalability, and costs.
2. **Requirements specification and concept definition:** Details the hardware and software specifications, including network dependencies. It also identifies performance metrics for timing accuracy, sets latency thresholds, and defines constraints related to network bandwidth. From these requirements, implementation concepts are derived and illustrated, for instance, through diagrams.
3. **Implementation of a multi-camera synchronization solution:** Builds a software layer for synchronous acquisition-triggering and frame-timestamping of a multi-camera setup.
4. **Development of a GenICam-based graphical user interface (GUI):** Streamline a GUI for the configuration and management of diverse cameras. This GUI features options for camera recognition, features control, and synchronous acquisition triggering.
5. **Testing and evaluation:** Validates the proposed solution's functionality, reliability, and performance. The tests ensure the system meets specified requirements and can be integrated effectively into industrial environments.

1.4 Outline

This thesis follows the outlined structure:

Chapter 2 This section introduces various methods for camera synchronization, including hardware-based, software-based, and network-based approaches. It proceeds with a review of machine vision standards that govern both hardware interfaces and industrial camera control software. These synchronization methods are then compared and evaluated in terms of their applicability and performance in different industrial scenarios. Furthermore, the section explores key libraries and frameworks commonly used in the development of camera control software.

Chapter 3 outlines the system requirements and presents the design of the proposed solution. Design choices are explained and justified, with functional diagrams and figures included to support and clarify the concepts discussed.

Chapter 4 describes how the introduced concepts were implemented within an object-oriented framework, presenting the main classes' methods used to realize each workflow.

Chapter 5 prepares the groundwork for the following chapter by defining test cases, success criteria, the test environment, the testing methodology, and the hardware used.

Chapter 6 evaluates the implemented solution, providing a detailed analysis of the results obtained. The evaluation also discusses the system's strengths and areas for potential improvement.

Chapter 7 concludes the thesis by summarizing the main findings, highlighting the challenges encountered during development, and suggesting potential directions for future research and development.

2 State of the Art

This chapter reviews related work from both industrial and academic perspectives. It begins with an overview of common synchronization methods and protocols, followed by a discussion of relevant machine vision standards, both hardware and software. Then, widely used libraries for camera control and configuration are reviewed. The chapter concludes with an evaluation of the synchronization techniques and standards presented.

2.1 Camera Synchronization Methods

In the industrial computer vision context, synchronization refers to the alignment of frames with precision ranging from microseconds to milliseconds [9]. While achieving such accuracy can be challenging, the issue of synchronization is not a new one. The following sections review the most commonly used approaches to address this problem. Synchronization methods are categorized into two main types: post-capture and pre-capture synchronization techniques.

2.1.1 Post-Capture Synchronization

Post-capture synchronization methods estimate the exact timing or timestamps of frames from multiple cameras after recording. These approaches typically rely on matching temporal markers or events to compute offsets and align frames.

A widely used post-capture strategy is *feature extraction*, in which the system identifies and correlates common fingerprints or events. These can be visual (such as a sudden flash of light), audio (such as a sharp noise), or audiovisual (a combined cue). By applying an adaptive threshold to detect luminance variations across frames, [10] demonstrates how one can identify a flash event and match it across all videos, thereby determining the offset between the recordings. For instance, Figure 2.1 presents four sequential frames alongside their corresponding luminance histograms. The second frame exhibits notably higher brightness, which can serve as a detectable event. Similar approaches that rely on audio signals or audiovisual events are discussed in [11] and [12].

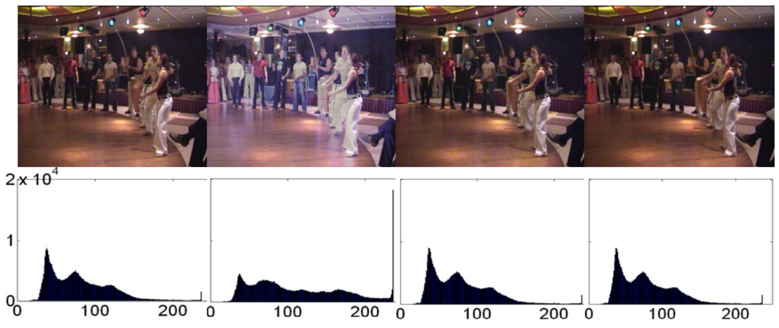


Figure 2.1: Example frames and associated brightness distributions. The second frame's increased luminance, as evidenced by its histogram, can be extracted as a feature for multi-camera synchronization.

An alternative approach extends feature extraction by introducing an independent event into all cameras' common field of view, then extracting it to enable sub-frame timestamping [13]. Sub-frame timestamping means determining whether an event occurred at, for example, "Frame 10" and whether it happened "12 ms into Frame 10". This allows better synchronization accuracy at the sub-frame level. Additionally, [14] suggests a *deep learning* approach to automatically extract high-level features and estimate temporal alignment across multiple streams.

Besides feature-extraction approaches, other specialized algorithms have been proposed to enhance multi-camera alignment. One method, known as *bit-rate-based synchronization*, leverages fluctuations in the compressed video bit rate as an alignment signal using a statistical measure called correntropy [15]. For Time-of-Flight cameras, *optical synchronization* techniques based on Time-Division Multiple Access (TDMA) were proposed to mitigate cross-talk and ensure precise alignment of optical signals [16].

Post-capture methods do not require specialized hardware. They are thus relatively easy to deploy. Nevertheless, they depend on clearly identifiable events (e.g., flashes or audio cues) that must be visible in all camera feeds. In some environments, such events may be infeasible or difficult to reproduce, and poor signal quality or environmental noise can further impair feature detection. Moreover, post-processing methods assume that frames are not being dropped and that most frames contain a detected feature. This assumption leaves other frames dependent on interpolation, which introduces additional uncertainty.

2.1.2 Pre-Capture Synchronization

Pre-capture synchronization methods ensure that cameras are synchronized during acquisition. This is particularly useful when real-time synchronization is required or to minimize the complexity of post-processing algorithms.

Trigger-Based Synchronization

In trigger-based synchronization, a trigger signal initiates coordinated image acquisition. Triggers can originate from either hardware or software. A software trigger is, for example, a command sent from the camera control software [17], usually based on other events or conditions. Hardware triggering occurs when an electrical signal from an external device (e.g., a PLC or another sensor) directly initiates image acquisition on a rising or falling edge or as long as the input signal is pulled high.

Triggering enables accurate synchronization and can be coordinated with components like light sources [18]. While software triggers are simple to implement, they often introduce millisecond-level delays due to system latency and variability. Hardware triggers offer greater precision by providing deterministic timing. However, they increase system complexity because each device requires dedicated trigger lines, careful signal routing, and often a central timing controller.

Network-Based Synchronization

To address these limitations, many distributed camera systems implement network-based synchronization. Network synchronization relies on protocols such as the Network Time Protocol [19] (NTP) and the Precision Time Protocol [7] (PTP), which aim to align the internal clocks of networked devices.

Network Time Protocol (NTP) is one of the oldest Internet protocols still in active use today. It operates without specialized hardware and remains robust even under highly variable network conditions. Over wide-area networks (WANs), NTP typically achieves synchronization accuracy within the range of 10–100 ms. Consequently, NTP is widely adopted for general-purpose clock synchronization across computers, laptops, and servers, aligned with Coordinated Universal Time (UTC). However, NTP proves insufficient for applications that demand high-precision synchronization: in local-area networks (LANs), it can achieve accuracies on the order of approximately 1 ms. In contrast, the Precision Time Protocol (PTP) achieves synchronization in the nano- to microsecond range thanks to its hardware timestamping. This mechanism ensures precise capture of send and receive times at the lowest layers of the network stack, reducing jitter and mitigating software-induced latency.

PTP operates on a hierarchical master-slave architecture, where devices are categorized based on clock roles in time distribution. The protocol defines four clock types: the Master Clock, which acts as the primary time reference; the Slave Clock, which synchronizes with the master; the Boundary Clock, which bridges network segments to reduce jitter; and the Transparent Clock, which improves timing accuracy by compensating for internal delays during message forwarding.

During PTP configuration, each clock must be assigned a specific role. This process begins with the election of a Master Clock, which is determined using the Best Master Clock Algorithm (BMCA) [20]. The BMCA evaluates all candidate clocks based on clock quality, priority, accuracy, and variance and selects the most suitable one. Only one master is active per domain at any time; all other clocks function as slaves.

Synchronization between clocks is established through a sequence of timestamped message exchanges. These include the event messages `Sync`, `Follow_Up`, `Delay_Req`, and `Delay_Resp`. Optional peer-to-peer delay measurement can be performed using `PDelay_Req` and `PDelay_Resp` messages.

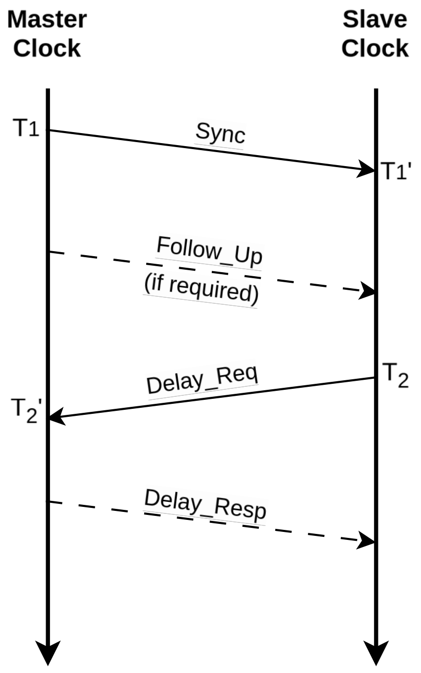


Figure 2.2: PTP synchronization message exchange between Master and Slave clocks. Timestamps T_1, T_1', T_2, and T_2' are used to compute the clock offset and path delay.

These messages are used for the synchronization process, illustrated in Figure 2.2. Synchronization, then, includes these steps:

1. The master clock sends a `Sync` message containing the transmission timestamp T_1.
2. In two-step mode (as in earlier versions before PTPv2), the precise transmission time

T_1 is sent separately in a `Follow_Up` message.
3. The slave clock receives the `Sync` message and records the local reception time as T_1'.
4. The slave then requests a `Delay_Req` message at time T_2.
5. The master receives this message at time T_2' and responds with a `Delay_Resp` message containing this timestamp.
6. At this point, the slave has all four timestamps: T_1, T_1', T_2, and T_2'. It computes the clock offset θ and the round-trip delay δ as follows:

$$\theta = \frac{(T_1' - T_1) - (T_2' - T_2)}{2} \tag{2.1}$$

$$\delta = \frac{(T_1' - T_1) + (T_2' - T_2)}{2} \tag{2.2}$$

   where θ represents the estimated offset between the master and slave clocks, and δ denotes the measured round-trip propagation delay.
7. Finally, the slave adjusts its internal clock using the computed offset θ.

This process is repeated periodically (typically every second) to maintain synchronization.

During the synchronization process, the Precision Time Protocol operates based on several underlying assumptions: that the clock offset remains constant during the message exchange, that both the master and slave devices can accurately timestamp transmitted and received messages, and that the network path delay is symmetric in both directions. The protocol's precision can be significantly degraded when these assumptions are violated. Consequently, PTP performs best in controlled local area network (LAN) environments, where such conditions are typically satisfied.

2.2 Machine Vision Standards

With the expansion of machine vision technologies, machine vision organizations and hardware manufacturers have collaborated since the early 2000s to develop standardized frameworks for camera interfaces and camera software. Key organizations such as the European Machine Vision Association[3] (EMVA), the Automated Imaging Association[21] (AIA, now part of A3 – the Association for Advancing Automation), and the Japan Industrial Imaging Association[22] (JIIA) are central to the development of these global standards. Joined efforts have led to the creation of widely adopted interface and protocol standards. In the following, the most popular of these standards are presented.

2.2.1 Hardware Standards (Camera Interfaces)

As embedded devices, machine vision cameras require a physical layer to interface with other devices. The physical layer transmits data to and from the sensors. Examples of such data include control commands for modifying camera parameters or triggering frame acquisition. Exchanging information over a network, whether between a camera and a computer or among multiple cameras, requires a well-defined protocol that dictates how the data should be transmitted. This is referred to as the communication protocol. In industrial cameras, communication protocols or hardware interfaces can be categorized into frame-grabber-based and bus-adapter interfaces.

Frame grabber interfaces

A frame grabber is a hardware intermediary between cameras and computers. While the cameras handle image acquisition, the frame grabber manages temporary storage and prepares the data for later processing. For example, cameras that implement the Camera Link standard transmit raw parallel image data, timing, and control signals over multiple physical connections. Without built-in protocol handling, the host cannot process the data directly. This is where the frame grabber comes in: it deserializes the incoming signals, reconstructs them into coherent image frames, buffers the data temporarily, and then forwards it to the host computer, typically via a PCIe interface.

Camera Link and CoaXPress are the main interfaces that work with frame grabbers. Camera Link is one of the earliest interfaces developed; however, its short cable length led to the development of Camera Link HS, which incorporates fiber optics for extended reach while maintaining high data throughput. CoaXPress introduces a packet-based transmission model that supports long-distance imaging while almost doubling the throughput compared to the chunk data transfer of Camera Link and Camera Link HS.

Bus-adapter Interfaces

Using frame grabbers as middleware remained challenging, as it required additional hardware and increased system complexity. To this end, GigE Vision and USB3 Vision were introduced as direct-connect interfaces utilizing standard bus-adapter communication protocols. GigE Vision is widely adopted due to its long-distance capability compared to other bus-adapter interfaces. USB3 Vision provides an easy, cost-effective alternative but is limited by cable length and bandwidth.

Table 2.1 provides a comparative overview of all interfaces. All interfaces other than Camera Link require a GenICam-compliant camera. Frame-grabber solutions offer advantages for long-distance transmission and high data transfer rates, whereas bus-adapter interfaces are preferred for their reduced system complexity.

Interface	**Year**	**Max. Throughput**	**Max. Length**	**Cable Type**	**Power over Cable**	**GenIcam Compliance**
Camera Link	2000	255 MB/s	15m	MDR/SDR	PoCL	Optional
GigE Vision	2006	1100 MB/s	5km (Fiber)	Ethernet	Yes (PoE)	Required
CoaXPress	2010	4800 MB/s	10km	Coaxial/Fiber	Yes	Required
Camera Link HS	2012	8400 MB/s	5 km+	Fiber/CX4	No	Optional
USB3 Vision	2013	400 MB/s	100m (fiber)	USB 3.x	Yes	Required

Table 2.1: Comparison of Camera Hardware Interfaces [2]. Dark grey lines represent frame-grabber interfaces; Light grey lines indicate bus interfaces.

All interfaces implement synchronization methods as a required feature for most multi-camera setups. A detailed comparison of the typical synchronization methods supported by various interfaces is provided in Table 2.2. In the case of frame-grabber interfaces, triggered synchronization is relatively straightforward, as a single frame-grabber can handle precise timing for multiple cameras.

Bus-adapter interfaces also support triggered acquisition but also offer more precise methods. GigE Vision 2.0-compliant cameras, for example, support Precision Time Protocol (PTP). GigE Vision 2.0 includes PTP as a complementary feature to software triggering, enabling clock synchronization between cameras for greater timing precision. With PTP, synchronized acquisition can be triggered using two methods. In the first, known as Sync Free Run, each camera's internal clock is synchronized over PTP, while image acquisition is triggered independently, either through software commands or GPIO signals. The second mode, Scheduled Action Command, allows for time-critical coordination by enabling action commands to be scheduled for execution at a precise PTP-aligned time. This approach is advantageous when deterministic timing is required, such as capturing an image exactly when an object reaches a designated position within a scene. For USB3 cameras, voltage-controlled synchronization can be employed, wherein internal clock alignment is achieved by dynamically adjusting the power supply levels [23]. Although this approach has been proposed in recent research, it has not yet seen widespread adoption in practical industrial systems.

Hardware Interface	**Synchronization Method**		
	Hardware Trigger	**Software Trigger**	**PTP (IEEE-1588)**
GigE Vision 2.0	✓	✓	✓
USB3 Vision	✓	✓	
CoaXPress	✓	✓	
Camera Link	✓	✓	
Accuracy	~1 µs [24]	~100–500 µs [24]	~1 µs [25]

Table 2.2: Supported Synchronization Methods by Camera Interface. All interfaces are considered to be the latest version.

2.2.2 Software Standards

To address the diversity of hardware interfaces in machine vision, two principal software interface standards have emerged: the Generic Interface for Cameras (GenICam), developed by the European Machine Vision Association (EMVA) [3], and IIDC2, a specification maintained by the Japan Industrial Imaging Association (JIIA) [22].

IIDC2

IIDC2 defines a standard for camera control register layout. It outlines a fixed set of register addresses and their corresponding functionality as memory-mapped registers located at fixed, predefined addresses. In IIDC2, the register address space is organized into logical categories covering general device information (vendor name, device ID), imaging controls (exposure, gain, region of interest), acquisition controls (trigger mode, frame start), and streaming and buffer management. As the standard defines both the presence and function of these registers, a software client familiar with IIDC2 can interact with the device without dynamic feature discovery, knowing in advance which registers exist, what each controls, the expected data types (typically 32-bit unsigned integers), and the correct access operations. Each camera that claims IIDC2 compliance must arrange its internal registers following this layout.

IIDC2 is an evolution of the original IIDC (Instrumentation and Industrial Digital Camera) protocol, initially designed for FireWire (IEEE 1394) communication. The revised version expands support to modern high-speed transport layers such as USB3 Vision and GigE Vision, addressing the limitations of its predecessor while retaining a streamlined register-based control model.

GenICam

GenICam introduces a more abstract and transport-layer-independent approach to camera feature mapping. While it provides a standardized naming convention for device features, the actual register mapping remains device-specific. Each camera supplies its feature mapping in a standardized XML file, which client SDKs interpret to enable control and configuration. Furthermore, GenICam defines a modular, layered architecture that allows software developers to build applications independently of the underlying communication protocol. It supports many transport standards, including GigE Vision, USB3 Vision, CoaXPress, Camera Link HS, and Camera Link [4]. It has been widely adopted across the imaging industry as a required interface standard. In contrast, IIDC2 remains an optional standard and is used more selectively in specific application domains where lightweight, register-based control models are preferred. Besides defining a standardized approach to feature mapping and supporting a wide range of hardware transport layers, GenICam is organized as a set of standards structured into several core components:

- SFNC (Standard Features Naming Convention): This component standardizes the name, type, meaning, and usage of device features. For example, "shutter speed" and "exposure time" refer to the same concept. To avoid confusion, SFNC 2.7[1] specifies this feature as "ExposureTime." This unification allows software developers and device vendors to reduce implementation variability using consistent terminology for most functionalities.
- GenTL (Generic Transport Layer): This component standardizes the interface between software applications and machine vision devices. It introduces the concepts of GenTL Producers and GenTL Consumers. As shown in 2.3, A GenTL Producer is a vendor-supplied driver library provided as a CTI (Common Transport Interface) file implements the GenTL API. This API defines a standardized set of C++ functions and structures intended for use by a GenTL Consumer. It provides access to the device's physical ports, such as for reading from or writing to its registers.

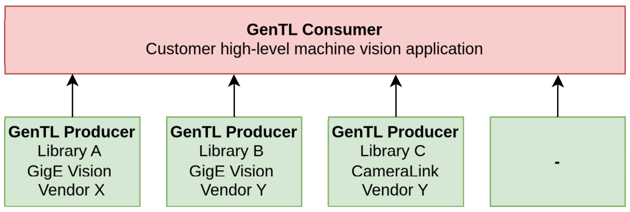


Figure 2.3: A high-level machine vision application (GenTL Consumer) interfaces with multiple vendor-specific GenTL Producer libraries supporting different transport layers, as specified in the GenTL standard [1].

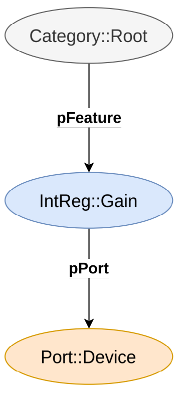
- GenApi (Generic Application Programming Interface): This component defines a standardized way to describe, access, and control device features. It introduces an XML-based description file that should be provided by the device (typically retrieved via the GenTL Producer) to include data such as feature names, types, allowed value ranges, enumeration options, and device metadata (e.g., model name, vendor).
- GenCP (Control Protocol): This component specifies guidelines for a common control protocol instead of developing a new packet layout and format for each control command and hardware interface.
- GenDC (Data Container): This component standardizes the data packet format to allow devices to send data to a host system. Different data types (e.g., 1D, 2D, 3D, multi-spectral, metadata) can be transported independently from the transport layer specifications.
- Other Modules:
  - PFNC (Pixel Format Naming Convention): Defines pixel formats across different devices.
  - CLProtocol: Extends GenICam standards to Camera Link.
  - FWUpdate: Provides a standardized method for device firmware updates.

The GenICam standard defines a set of specifications intended to guide developers in creating software for controlling GenICam-compliant devices. In addition to the specifications themselves, EMVA provides a reference implementation, offering a practical framework to support the development of compatible software solutions. The GenICam reference implementation is structured into two main components: the GenApi and the GenTL API.

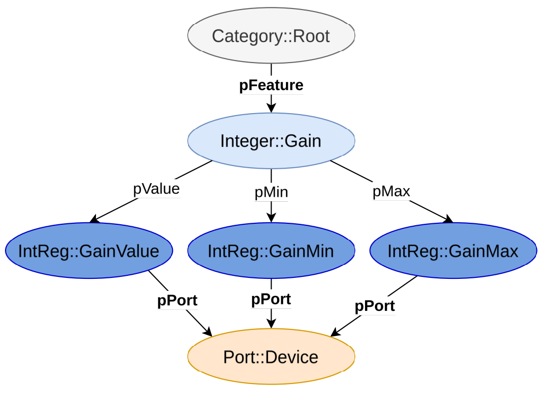
The GenApi module provides the C++ interface for accessing and controlling device features through a standardized node-based model. In GenICam, a camera's features are represented as a directed graph of typed nodes. Each node models a configurable parameter, such as a hardware register, enumeration, or floating-point value. Nodes are uniquely identified and interconnected through standardized roles such as `pFeature`, `pValue`, `pMin`, or `pPort`, which define logical relationships within the feature graph. The node `Category::Root` serves as the entry point to the graph, while `Port::Device` anchors all hardware-level nodes to the physical communication interface.

Figure 2.4(a) illustrates a minimal configuration in which a gain parameter is controlled directly through a single writable register node (`IntReg::Gain`). In contrast, Figure 2.4(b) presents a more realistic model, where an abstract gain node is associated with additional nodes representing the minimum, maximum, and current value registers.

GenApi formalizes the feature graph model in the reference implementation through a collection of C++ abstract interfaces. The core interface, `INode`, defines the basic properties and behaviors shared by all nodes in the graph, such as access control, visibility, and caching. Specific node types extend this base interface to represent different categories of features: `INodeMap` provides access to the complete graph of features exposed by a device; `ICategory` groups related features together into logical collections; `IInteger` and `IFloat` represent features with numerical values that can be read and written; and `IEnumeration` models features whose value can be selected from a predefined set of options. Within this model, a `NodeMap` acts as a centralized access point through which developers can locate individual features by name, inspect their metadata, and read or modify their values.



(a) Minimal feature model with a register connected to the root and device port.

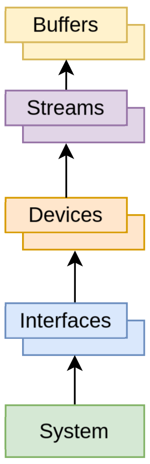


(b) Extended feature model for a gain control.

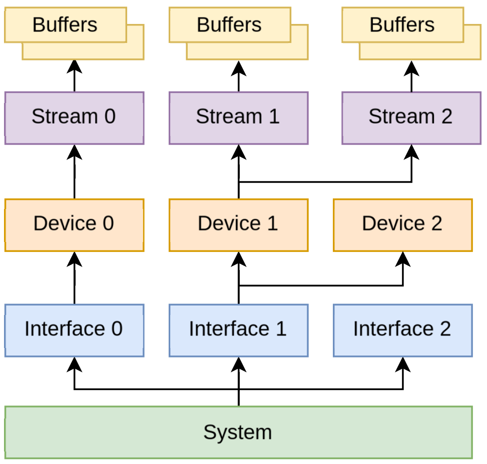
Figure 2.4: Examples of feature control structures using GenApi nodes.

Parallel to GenApi, the GenTL Consumer API defines the standard mechanisms for device discovery, communication channel management, image stream handling, and buffer processing. The key header file, `GenTL.h`, provides a set of enumerations, structures, and function prototypes required to interact with transport layer providers (CTI producers).

The typical enumeration process in GenTL follows a hierarchical model, beginning with identifying systems and interfaces and discovering individual devices and their associated data streams. This hierarchical structure is illustrated in Figure 2.5. In particular, Figure 2.5(a) presents the modules defined by the GenTL standard, while Figure 2.5(b) depicts the enumeration process, showing how devices are discovered through the system and interface hierarchy. Following this hierarchy, a developer can enumerate available systems (representing collections of transport adapters), interfaces (such as physical network cards or USB controllers), devices (cameras), data streams, and individual data buffers. Functions such as `GCInitLib`, `GCGetSystemList`, `IFOpenInterface`, `DevOpen`, `DSStartAcquisition`, and `DSGetBuffer` standardize the interactions needed to navigate through this hierarchy and control devices. Nevertheless, GenTL standardizes the structure and naming of these functions but does not provide a working implementation. Therefore, developers must combine the GenTL headers with an appropriate CTI (GenTL Producer) runtime library, typically offered by hardware vendors, to perform real device operations.



(a) System and interface enumeration in GenTL.



(b) Device and stream discovery in GenTL.

Figure 2.5: Device enumeration hierarchy following the GenTL model.

In summary, the GenICam reference implementation, through its GenApi and GenTL definitions, provides a complete and rigorous foundation for interacting with GenICam devices at the protocol and feature model level. However, this reference implementation provides only structural definitions and interface specifications, not including complete operational tools for direct camera control or image acquisition.

2.3 GenICam Cameras Control Software

Camera manufacturers or third-party developers provide control software or libraries to interface with machine vision GenICam cameras. These tools typically offer functionalities such as camera enumeration, configuration, synchronization, and streaming. Control software available on the market can generally be classified into manufacturer-exclusive and manufacturer-independent solutions.

2.3.1 Manufacturer-Exclusive Software

GenICam camera manufacturers provide users with camera control options such as SDKs, graphical viewer tools, or programmable APIs. A graphical viewer is convenient for quick prototyping. On the other hand, APIs offer developers a high-level library of methods for more advanced camera control. For example, issuing action commands is only possible using the provided API.

Both solutions are developed exclusively for cameras from the same manufacturer. First, most control software, both SDKs and GUIs, implementing the GenTL Consumer, only recognizes cameras from its respective brand due to CTI files from different camera brands (GenTL Producer) not being recognized. Second, developers cannot extend these SDKs to other GenICam-compliant cameras. That's because these libraries only expose high-level methods for camera interaction, but the underlying GenAPI and GenTL implementations are only available as compiled libraries, making them inaccessible to developers. These restrictions are primarily intended to control market competition. However, they challenge developers who want to integrate cameras from different manufacturers.

As leading camera manufacturers, both Basler [26] and Allied Vision [27] offer proprietary software solutions to control and integrate their devices. Basler provides the Pylon SDK and Pylon Viewer, while Allied Vision offers the Vimba X SDK for its range of cameras.

2.3.2 Manufacturer-Independent Software

Based primarily on machine vision standards, this software provides a control framework for all GenICam-compliant cameras: Methods implemented follow the GenICam reference implementations and only the CTI file from the camera is required. This makes third-party software partly dependent on GenTL producers or camera manufacturers. This is because hardware interfaces are proprietary, and only licensed parties, usually the camera manufacturers, can supply the transport layer interface. Another option is to implement the protocol functionalities from scratch. While this allows complete manufacturer independence, it also requires a deep understanding of network protocols.

Third-party libraries used in machine vision applications are either proprietary or open-source. Examples of proprietary solutions include Stemmer CVB, HALCON, and eBUS Player. These solutions provide customer support and advanced features but are limited by restrictive licensing, making them less suitable for open-source or custom development.

In contrast, open-source libraries such as Aravis, Harvesters, and rc_genicam_api offer greater accessibility, though often at the cost of limited functionality and community support. Aravis stands out by avoiding dependency on CTI files, instead reimplementing the GigE Vision and USB3 Vision protocols to remain manufacturer-independent. Harvesters, written in Python and rc_genicam_api, developed in C++, act as wrappers around the GenTL and GenAPI standards and require CTI files for operation.

2.4 Core Libraries

The integration of open-source libraries can benefit the development of GenICam-compliant camera control software. This section provides a detailed discussion of three key libraries: rc_genicam_api, OpenCV, and Qt6. Each library contributes distinct capabilities: rc_genicam_api for camera communication, OpenCV for image visualization, and Qt6 for graphical user interface (GUI) development.

2.4.1 rc_genicam_api

GenICam reference implementation delivers detailed and standardized interface definitions but remains relatively incomplete from an application developer's perspective. It defines only the structure of device interactions without providing full operational capabilities: actual communication with devices requires integration with a GenTL Producer supplied by the hardware vendor. Furthermore, the reference does not include higher-level abstractions such as unified device classes, automatic buffer management, event handling, or simplified feature access wrappers. Higher-level client libraries are left to implement these more practical functionalities, which bridge the gap between the raw GenICam interfaces and real-world camera control applications.

The rc_genicam_api library, developed by Roboception GmbH [28], is an open-source C++ library designed to provide a client-side implementation of the GenICam standard for controlling industrial cameras that expose a GenICam interface. It builds upon the GenICam reference implementations of GenApi and GenTL by offering a higher-level, type-safe, and more convenient C++ interface for device control, feature configuration, and image acquisition. The rc_genicam_api library provides a structured set of C++ classes that correspond to the central concepts of the GenICam and GenTL models. At the top level, the System class represents an instance of a GenTL system. The Interface class corresponds to physical or logical interfaces such as Ethernet network cards or USB controllers. Each Interface contains one or more Device objects, each representing an attached GenICam-compatible camera. A Device exposes its configuration features via a GenAPI INodeMap and provides access to one or more Stream objects used for image acquisition. Image data is represented using the Buffer class, which wraps the raw payload and metadata of acquired frames.

Device enumeration, which in raw GenTL would involve manual system, interface, and device discovery through TLSystem, TLInterface, and TLDevice, is simplified via the rcg::System, rcg::Interface, and rcg::Device classes. The process of discovering devices becomes straightforward.

```
std::vector<std::shared_ptr<rcg::System>> systems = rcg::System::getSystemList();
for (auto& sys : systems) {
    sys->open();
    std::vector<std::shared_ptr<rcg::Interface>> ifaces = sys->getInterfaces();
    for (auto& iface : ifaces) {
        iface->open();
        std::vector<std::shared_ptr<rcg::Device>> devices = iface->getDevices();
        for (auto& dev : devices) {
            std::cout << "Found device: " << dev->getDisplayName() << std::endl;
        }
    }
}
```

Similarly, feature configuration is simplified. Instead of manually retrieving and casting nodes from the INodeMap, developers use convenience functions such as rcg::getFloat(), rcg::setFloat(), rcg::getInteger(), and rcg::setInteger(). For example, setting the camera exposure time:

```
std::shared_ptr<rcg::Device> dev = iface->getDevices().front();
dev->open();
rcg::setFloat(dev->getRemoteNodeMap(), "ExposureTime", 5000.0);
```

For streaming and image acquisition, working directly with the GenTL API requires developers to handle the entire acquisition lifecycle manually: opening data streams, allocating memory buffers, announcing and queuing them, managing timeouts during buffer retrieval, and recycling buffers after use. Additionally, explicit API calls such as DSAnnounceBuffer, DSQueueBuffer, DSGetBuffer, and DSStartAcquisition must be used to control streaming operations.

In contrast, rc_genicam_api abstracts much of this complexity through its rcg::Stream and rcg::Buffer classes. The Stream class provides a unified interface for managing the lifecycle of image streams, encapsulating connection management, acquisition control, and buffer synchronization. The Buffer object provides direct access to the image payload and metadata without requiring manual buffer handling. Developers can open streams, start acquisition, and retrieve frames using simple methods such as open(), startStreaming(), and grab(), with buffer management, timeout handling, and resource cleanup performed internally:

```
std::shared_ptr<rcg::Stream> stream = dev->getStreams().front();
stream->open();
```

```
stream->startStreaming();
std::shared_ptr<const rcg::Buffer> buf = stream->grab(1000);
```

In addition to the C++ API, rc_genicam_api includes several useful command-line tools to assist development and debugging:

- `gc_config`: Configures network and device parameters of GenICam-compatible GigE Vision cameras.
- `gc_file`: Uploads or downloads files to and from the persistent storage of an industrial camera.
- `gc_info`: Displays information about available transport layers, interfaces, and connected devices.
- `gc_pointcloud`: Captures synchronized images from a Roboception rc_visard and generates a 3D point cloud.
- `gc_stream`: Streams and saves images from a GenICam device, with optional format selection and parameter configuration.

rc_genicam_api offers a convenient interface for interacting with GenICam-compliant devices. One of its main strengths is its object-oriented design, which simplifies device interaction and feature configuration. In addition, the library is portable across both Linux and Windows platforms. The only external requirement is the availability of a suitable CTI file (GenTL Producer) provided by the camera manufacturer. Although this introduces a dependency on vendor-supplied GenTL producers, it significantly reduces the need for proprietary SDKs.

2.4.2 OpenCV

The Open Source Computer Vision Library (OpenCV) is an open-source framework for efficient, scalable computer vision and image processing tasks. Architecturally, OpenCV is implemented in C++ with bindings for Python, Java, and other languages. It is organized into core modules, including `core` for basic data structures (e.g., Matrices, Scalars), `imgproc` for image processing algorithms, and `highgui` for simple graphical output.

At the foundation of OpenCV is the `Mat` object, a matrix structure representing images as n-dimensional pixel arrays. Each pixel can be a scalar (e.g., grayscale) or a vector (e.g., RGB, RGBA), and the library provides extensive methods for manipulating these matrices efficiently.

In the context of multi-camera synchronization, OpenCV provides multiple essential functionalities. First, it enables real-time frame visualization through simple GUI functions such as `imshow`, allowing individual camera streams or processed outputs to be displayed directly in OpenCV windows. Second, frame annotation is supported through functions like `putText` and `rectangle`, allowing metadata (e.g., device identifiers, frame indices, and timestamps) to be superimposed onto frames. Third, composite visualization is made possible using functions such as `hconcat` and `vconcat`, enabling the efficient construction of side-by-side or stacked views of multiple camera streams. Finally, OpenCV supports frame storage in lossless formats such as PNG and TIFF through functions like `imwrite`.

2.4.3 Qt6

Qt6 is a cross-platform application development framework built in C++. It's based on core modules such as `QtCore` for non-GUI functionality (e.g., data structures, file IO, event handling), `QtGui` for rendering and image management, and `QtWidgets` for classic desktop GUI elements. Dialogs can be built using `QDialog`, combined with layout managers such as `QFormLayout` to organize input fields like `QComboBox` for dropdown selections and `QLineEdit` for manual user input. Action buttons such as `QPushButton` can be connected to specific functions or slots.

To maintain responsiveness, especially when handling real-time data acquisition and processing, Qt6 defines background workers. A typical implementation involves creating a worker class derived from `QObject`. This worker can operate in a separate thread without blocking the main GUI thread. The signal-slot mechanism of Qt ensures efficient communication between the worker and the graphical components.

Data visualization and plotting in Qt6 applications can be efficiently achieved using the Qt C widget `QCustomPlot`. It supports generating various types of 2D plots, graphs, and charts. These include time-series graphs, scatter plots, and bar charts. Plots can be either integrated into the application or exported.

The application can present real-time plots by embedding `QCustomPlot` widgets directly into Qt6 dialogs or main windows. `QCustomPlot` supports features such as zooming, panning, and selectable items, which can be enabled to make the visualization more interactive.

`QCustomPlot` provides a set of core classes and methods that facilitate the creation of plots within Qt applications. To add new data curves to a plot, the `addGraph()` method is used, which creates a new `QCPGraph` object. Each graph can be assigned a label using `setName()`, which automatically appears in the plot's legend when enabled.

For drawing reference elements, `QCustomPlot` offers item classes like `QCPItemStraightLine`, which represents an infinite straight line across the plotting area, often used to indicate thresholds or critical values. Additional textual annotations can be added using `QCPItemText`, which allows placing labels at specific plot coordinates.

To automatically adjust the visible range of axes based on the plotted data, the `rescaleAxes()` function can be called. This ensures that all data points are appropriately framed within the current view, providing a clean and uncluttered visualization without requiring manual axis adjustments.

In multi-camera synchronization and PTP monitoring context, `QCustomPlot` can dynamically visualize key parameters such as clock offsets, synchronization drift, and network delay trends.

Qt6 allows the application to simultaneously display multiple OpenCV-generated image windows, display plots, and import log data for analysis within an integrated, user-friendly interface built on Qt6.

Finally, it's cross-platform design ensures that applications can be compiled and executed on Windows, Linux, and macOS with minimal codebase changes.

2.5 Conclusion

Multi-camera synchronization has been widely explored in both research and industry: Post-processing methods like feature matching have been proposed in academia. In industry, hardware triggering is the most common solution. As cameras evolved with network capabilities like GigE Vision, software triggers became more popular. While simplifying wiring, software triggers may introduce undesired latency and jitter.

Table 2.3 shows possible choices of synchronization methods. These depend heavily on the system's scale, latency requirements, and physical setup. While hardware triggers remain dominant in small or latency-sensitive setups, software-based approaches are recommended in large, distributed, or software-driven environments.

Software-based solutions could be paired with PTP to improve accuracy. PTP keeps clocks aligned across devices and reduces drift in distributed setups. However, it is currently only supported by GigE Vision 2.0 cameras.

Setting up PTP synchronization using vendor-specific tools often limits interoperability in multi-camera systems. While GenICam provides a standardized interface for controlling and streaming from cameras, it lacks comprehensive support for cross-vendor time synchronization. For example, libraries implementing GenICam such as Harvesters [29]

Use Case	**Preferred Method**	**Rationale**
Large-Scale Multi-Camera Setups (e.g., 8+ cameras)	Software Trigger / PTP	Avoids cabling complexity. With PTP, only Ethernet connections are required.
Distributed Systems (Cameras not physically closed)	Software Trigger / PTP	Hardware trigger lines may not be feasible over long distances due to signal degradation and noise.
Software-Controlled Events or Conditions	Software Trigger / PTP	Offers high flexibility for applications where triggering depends on software-defined conditions or logic events.
Very Low Latency Requirements	Hardware Trigger / PTP	Both PTP and external hardware triggers can offer sub-microsecond jitter and minimal latency.
Simple and Small Camera Systems (2–4 cameras)	Hardware Trigger	For setups with only a few cameras, hardware triggering avoids the complexity of setting up PTP. It eliminates the risk of timing degradation due to network congestion caused by frequent sync packets and image data.
Instantaneous Reaction to External Events	Hardware Trigger	When real-time reaction is critical (e.g., part crossing a light curtain), hardware triggers are ideal. PTP cannot respond instantly—it requires a few milliseconds of planning.
Interfaces Without PTP Support	Hardware Trigger	USB3 Vision and Camera Link lack PTP support, making hardware triggering the only deterministic synchronization option.
Legacy or Frame Grabber-Based Systems	Hardware Trigger	CoaXPress and Camera Link systems often use centralized hardware triggering via frame grabbers—accurate and deterministic. Adding PTP would increase costs without clear benefits.

Table 2.3: Recommended Synchronization Methods Based on Setup Configuration

enable access to device features but do not offer mechanisms for configuring or managing PTP synchronization.

The rc_genicam_api library provides a basic `ptpEnable` method; however, it assumes full compliance of the cameras with the latest SFNC (Standard Feature Naming Convention) and does not support older or deprecated firmware versions. Moreover, no library offers integrated network management functionalities or dedicated tools for monitoring PTP synchronization status and network performance metrics.

These limitations highlight the clear need to develop and evaluate a manufacturer-independent synchronization framework for GenICam industrial cameras. Such a solution would leverage standards with existing open-source libraries, including rc_genicam_api, OpenCV, and Qt6, to facilitate the development of control software.

3 Concept

This chapter outlines the design of a software-based synchronization framework for multi-camera systems based on the GigE Vision 2.0 protocol and the GenICam standard. The proposed solution is delivered as a software tool, which consists of three main components: a graphical user interface (GUI), a command line interface (CLI), and a modular application programming interface (API).

This chapter first defines the functional and non-functional requirements for the system design. It then presents the system architecture and details the design of each major component, including the integration of user interfaces.

3.1 Requirements

This section defines the system-level requirements that guided the design and implementation of the synchronization framework. These requirements are split into two categories: functional requirements and non-functional requirements, outlined below:

3.1.1 Functional Requirements

FR-1 Device Discovery	The application must detect all compatible cameras on the local network (up to six devices) and provide detailed metadata for each.
FR-2 Multi-Camera Synchronization	The system must support synchronization between a configurable subset of the connected cameras.
FR-3 Synchronization Monitoring	The application should provide real-time feedback on synchronization status, including clock offsets.
FR-4 Coordinated Image Acquisition	Users must be able to simultaneously trigger image acquisition across multiple cameras, with an option to save the resulting frames.
FR-5 User Interface Integration	A GUI must be provided for convenient access to the system's core features.

3.1.2 Non-Functional Requirements

NFR-1 GenICam Compliance	The system must comply with the GenICam standard, including GenApi 2.1.1, SFNC 2.7, and GenTL 1.6 [1].
NFR-2 GigE Vision Compatibility	Communication with cameras must conform to the GigE Vision 2.0 specification [5].
NFR-3 Cross-Platform Readiness	Platform-specific behavior must be avoided, clearly isolated, and documented to support portability.
NFR-4 Development Language	The system must be implemented entirely in C++ to ensure compatibility with existing industrial libraries and drivers.

3.2 System Architecture

This section introduces the overall concept and structure of the solution based on the above requirements. The goal is to provide a vendor-independent control and synchronization interface for GenICam-compliant industrial cameras.

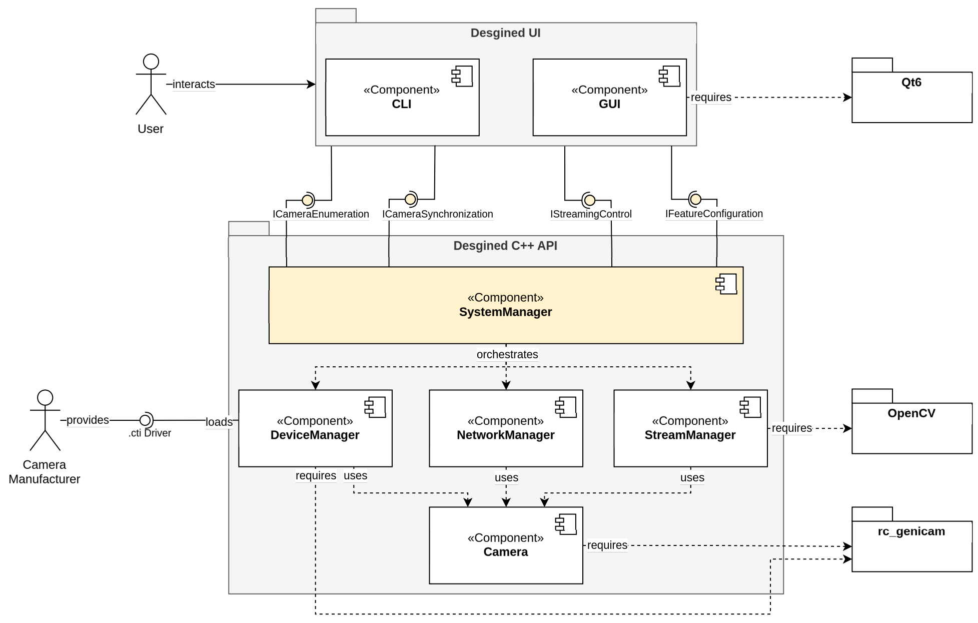


Figure 3.1: Component diagram of the synchronization system, illustrating the modular architecture of the designed library and its dependencies.

Figure 3.1 shows an overview of the system architecture. The software package is composed of three components: a graphical user interface (GUI), a command-line interface (CLI), and an application programming interface (API). The API encapsulates the core functionalities and is the central layer, interfaced by the GUI and CLI. The API has four core functionalities. First, it supports camera enumeration, allowing the detection and listing of all GenICam-compatible devices. Second, it enables feature configuration, giving access to key parameters such as resolution, gain, pixel format, and frame rate. Third, it implements camera synchronization using the IEEE 1588 Precision Time Protocol (PTP), ensuring accurate clock alignment across devices. Finally, it initiates frame acquisition for synchronized image capture and optional image storage. Internally, the designed package organizes its logic around five classes: The `SystemManager` class serves as the primary API entry point, orchestrating the interactions between three specialized managers: the `NetworkManager`, the `StreamManager`, and the `DeviceManager`. The `Camera` class provides an abstraction over individual GenICam-compliant cameras. The following sections describe these classes and their interactions in more detail. The system is built entirely on open standards. GenICam ensures hardware abstraction, GigE Vision 2.0 governs network-based Communication, and PTP provides temporal synchronization. In addition, external libraries are integrated to support the components of the system: OpenCV is used for visualization, Qt6 is employed for GUI development, and the rc_genicam_api library wraps the official GenICam reference implementation. Communication with physical devices is handled through GenTL Producers (CTI files), which the respective camera vendors must supply. This package extends the rc_genicam_api library by adding a fallback mechanism for enumerating cameras without a CTI driver, enabling PTP synchronization across multiple devices with visualization through plots, supporting dynamic configuration of bandwidth settings, and providing a viewer tool to display streams in a combined OpenCV window.

3.3 API Architecture

The API's internal architecture follows a modular and object-oriented design, divided into five classes. Each class encapsulates a well-defined set of responsibilities. Figure 3.2 presents the internal class architecture of the synchronization core. The `SystemManager` serves as the main API entry point, orchestrating the operations of three specialized manager classes: `NetworkManager`, `StreamManager`, and `DeviceManager`. Each manager is responsible for a distinct functionality domain, such as network synchronization, camera streaming, and device management. The `Camera` class encapsulates the access to GenICam-compliant devices, exposing feature control and acquisition capabilities. Relationships between classes are formalized through associations, highlighting the separation of concerns within the system design.

The figure shows only the main attributes and methods essential for understanding the system architecture; auxiliary methods and implementation-specific details are omitted to maintain clarity and avoid excessive complexity.

3.3.1 Camera Class

The Camera class acts as a structured wrapper around the `rcg::Device` interface, offering a unified representation of each GenICam-compatible device within the system. It extends the device class of rc_genicam_api by adding camera-specific attributes to each device, including network configuration parameters such as transmission delays, device metadata like the serial number, imaging characteristics including pixel format, and synchronization information relevant to IEEE 1588 PTP, such as the device's clock role.

In addition to providing internal state information, the class lays out a range of methods that abstract operations from the rc_genicam_api library. These are setter and getter functions for interacting with camera features and methods for initiating and controlling image streaming. To ensure compatibility with a broad range of hardware, the design of feature setting and getting methods support legacy devices that may rely on deprecated or non-standard feature names. This deprecated feature support is significant for cameras lacking firmware updates or not fully complying with the latest GenICam SFNC specification.

GenIcam values encoding can vary within manufacturers. To unify the format across getter methods, the class also performs additional processing to interpret raw register

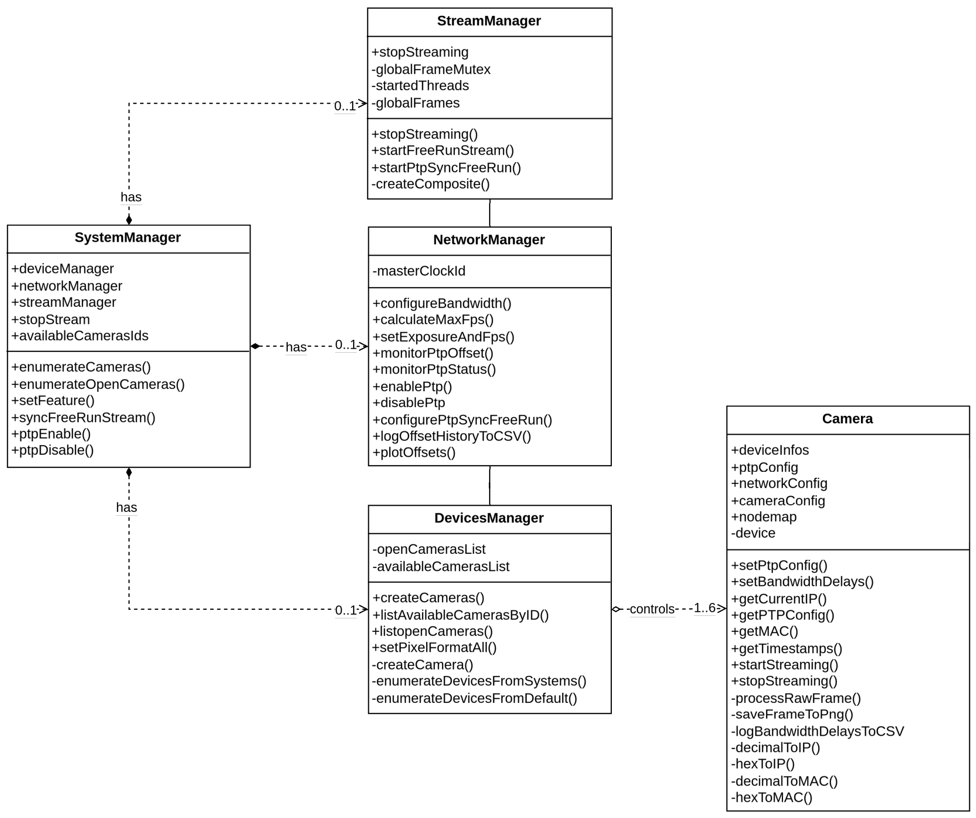


Figure 3.2: Class diagram of the developed API.

values correctly. For instance, the MAC address should have a standard MAC format. Similarly, the current IP address has to have a valid IPv4 format.

The `Camera` class allows direct access to the underlying GenICam interface and isolates transport layer-specific logic from the rest of the system. Such a design enables other modules—such as the `DeviceManager`, `NetworkManager`, and `StreamManager`—to interact with cameras in a consistent and simplified manner. Instead of managing raw nodemaps and device handles, these managers can rely on the `Camera` class.

3.3.2 System Manager

The `SystemManager` class is the central entry point to the API and manages all core system workflows. It is a unified interface to the CLI and GUI layers by exposing the four primary methods—device enumeration, feature configuration, PTP synchronization, and synchronized acquisition.

Since each high-level operation typically involves multiple steps, the `SystemManager` coordinates managers and ensures that each module performs its role in the correct order. This coordination is essential for maintaining consistency and avoiding conflicts between operations.

For instance, when setting network-related parameters on a specific camera, the `DeviceManager` must first be invoked to access and initialize the target device. Once the device is accessible, the `NetworkManager` sets the correct values to the desired feature. In this way, the `SystemManager` ensures that each operation is executed logically without adding complexity to the user interface layers.

3.3.3 Device Manager

The `DeviceManager` takes control of the initialization, discovery, and configuration of GigE Vision-compatible cameras. It provides functionality for detecting connected devices, opening them for configuration, and preparing them for streaming and synchronization.

Interfacing directly with the GenTL layer, the `DeviceManager` supports dynamic runtime discovery of available devices. It maintains an internal registry of detected cameras, including key metadata such as serial numbers, model identifiers, and vendor names. This registry is the foundation for managing the system's device state across workflows.

In addition to detection, the `DeviceManager` handles the controlled opening of device handles, a prerequisite for configuration, parameter retrieval, and streaming. Devices successfully opened are tracked in a separate internal list, allowing the system to distinguish between initialized and uninitialized cameras.

Once a device has been opened, the `DeviceManager` provides access to a broad range of configuration operations, covering image acquisition parameters such as exposure time, gain, pixel format, and spatial resolution (width and height). All configuration tasks are executed through the `Camera` class.

Unlike external tools such as `gc_info`, which require users to get driver files from manufacturers and manually configure environment variables (e.g., `GENICAM_GENTL64_PATH` or `GENICAM_GENTL32_PATH`) to locate these files, the `DeviceManager` implements an internal fallback mechanism. The system automatically defaults to a CTI path if no transport layer is detected at runtime. This feature is handy when the manufacturer's drivers are not installed. As the rc_genicam_api library provides an example CTI file, this file can be used as a default transport layer, enabling the system to at least discover GenICam-compatible devices on the network without requiring vendor-specific software.

3.3.4 Stream Manager

The `StreamManager` manages all aspects of image acquisition, real-time streaming, and data storage. It is the central module for coordinating synchronized capture across multiple devices and ensures that acquired frames are handled efficiently and reliably throughout the streaming pipeline.

This component supports real-time frame preview, buffered acquisition, and storage of captured images. Frames can be saved individually in PNG format. The `StreamManager` also provides functionality for generating composite streams that combine the outputs of multiple cameras into a single display.

3.3.5 Network Manager

The `NetworkManager` is responsible for all Ethernet-specific configurations required for camera synchronization and streaming. Its functionality is essential to maintaining stable performance in multi-camera setups, particularly when maximizing high frame acquisition rates.

Key responsibilities of this module include bandwidth negotiation, packet size configuration, and the dynamic adjustment of packet transmission timing. Specifically, the module manages both the inter-packet delay (`GevSCPD`) and the initial transmission delay per device (`GevSCFD`).

Figure 3.3 illustrates the timing relationship between packet transmission parameters. Cameras are configured to start transmission at staggered intervals to avoid network collisions and ensure synchronized, conflict-free streaming.

This is achieved by adding delay values computed based on the number of active cameras, frame dimensions, and the available network bandwidth. Formulas to compute these delays

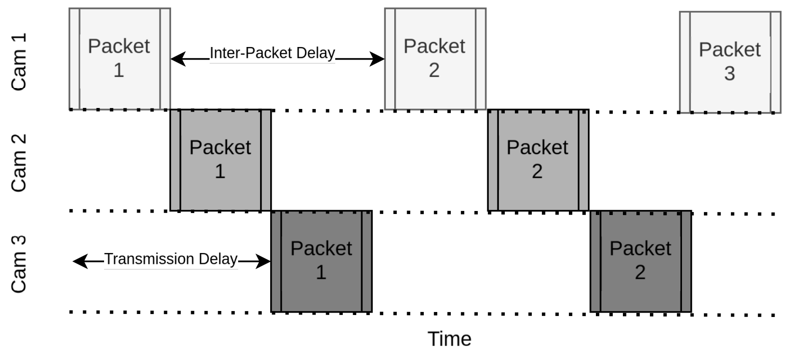


Figure 3.3: Timing diagram for multi-camera acquisition using PTP synchronization.

are provided in [30], [31], and [32]. These equations are detailed in 3.1–3.3.

$$\textbf{Packet Delay (ns)} = \text{Packet Size (Bytes)} \times \left(\frac{10^9}{\text{DeviceLinkSpeed (Bytes/sec)}} \right) + \text{Buffer} \tag{3.1}$$

where **Buffer** typically accounts for 10.93%–30% of the packet delay [30].

For each camera $i \in \{0, 1, \dots, x-1\}$, the following parameters are computed:

$$\textbf{GevSCPD} = \text{Packet Delay (ns)} \times (x-1) \tag{3.2}$$

$$\textbf{GevSCFTD}_i = \text{Packet Delay (ns)} \times i \tag{3.3}$$

Through dynamic adjustment of these values, the `NetworkManager` optimizes network throughput and ensures that packet collisions are avoided when multiple cameras are streaming simultaneously.

A second core function of this module is the implementation of IEEE 1588 Precision Time Protocol (PTP) for camera clock synchronization. The synchronization process involves periodic clock alignment and dynamic offset compensation routines. To assess and document synchronization quality, the `NetworkManager` logs clock status and offset values over time and automatically generates visual plots that display the drift and convergence of each camera relative to the elected master clock. These visualizations serve as a diagnostic tool, enabling users to validate synchronization accuracy and identify inconsistencies in networks containing heterogeneous devices.

Finally, the `NetworkManager` implements a mechanism to maintain compatibility with older or non-standard devices. In cases where standardized nodes such as `PtpOffsetFromMaster` are unavailable, alternative techniques such as timestamp latching or vendor-specific register access (e.g., `GevIEEE1588OffsetFromMaster`) are used to approximate synchronization status.

3.4 User Interfaces

The system offers two front-end interfaces: a Command Line Interface (CLI) and a Graphical User Interface (GUI). Both interfaces are tightly coupled to the same backend logic, which is exposed through the `SystemManager` class. This unified architecture ensures consistent behavior across interaction modes and minimizes redundancy in code implementation.

3.4.1 Command Line Interface (CLI)

The CLI exposes all core workflows—such as device enumeration, PTP control, and synchronized acquisition—through a compact set of command-line arguments.

Each CLI command maps directly to a function within the `SystemManager`, allowing users to configure devices, initiate streaming, or perform synchronization checks without relying on graphical components.

3.4.2 Graphical User Interface (GUI)

The GUI is designed to provide visual feedback when operating multi-camera systems. It presents live video feeds from all active cameras, displays synchronization status, and visualizes key acquisition metrics such as offset deviation and packet delays.

Each interactive component in the GUI maps directly to an API call, ensuring that all actions performed through the graphical interface are functionally equivalent to their command-line counterparts.

By offering graphical and terminal-based interfaces with equivalent backend access, the system provides a flexible and robust user experience suitable for various deployment contexts.

4 Implementation

The system has been implemented based on a modular API architecture to realize the concept introduced. This chapter presents how the individual components of the system have been integrated to support camera control.

4.1 API Methods

The `SystemManager` implements four principal workflows: camera enumeration, feature configuration, PTP synchronization management, and synchronized frame acquisition. Acting as the central coordinator, the `SystemManager` orchestrates these workflows and exposes them through both the graphical user interface (GUI) and the command-line interface (CLI). The following sections describe each workflow, including implementation strategies, supporting subsystems, relevant challenges, and how they've been solved.

4.1.1 Device Enumeration

The discovery of connected GenICam-compatible devices constitutes the first step in any acquisition sequence. Within the `SystemManager`, this functionality is implemented through two methods: `enumerateCameras` and `enumerateOpenCameras`.

The `enumerateCameras` method performs a lightweight scan to identify all available devices without requiring write access, whereas `enumerateOpenCameras` opens each device to retrieve extended metadata and initializes it into a corresponding `Camera` object. Listings 4.1 and 4.2 illustrate the formatted output of each method. The first listing shows only the device identifiers, whereas the second provides detailed information for each device. Device initialization and management are handled by the `DeviceManager`. The overall enumeration process, illustrated in Figure 4.1, begins by loading the CTI (GenTL Producer) file, detecting available interfaces and listing connected devices with optional detailed metadata.

The system also accesses the GenICam nodemap during this process and reformats specific raw register values into standardized representations. MAC addresses are converted into the conventional `XX:XX:XX:XX:XX:XX` format, while IP addresses are formatted according to standard IPv4 notation. Devices with outdated firmware are flagged during initialization by attempting to write to the `PtpEnable` node. Since `PtpEnable` is a critical feature introduced in the latest SFNC specification for synchronization, its absence is used as an indicator of deprecated firmware. If the write operation fails due to a missing node, the internal `deprecatedFW` flag of the `Camera` object is set to true.

Error handling mechanisms are integrated throughout the enumeration process. In cases where CTI files are absent, the function `enumerateDevicesFromDefault` attempts to access devices using the default CTI path specified in `GlobalSettings.cpp`. This default path points to a copy of the Baumer CTI file distributed with `rc_genicam_api`. If devices remain inaccessible, these enumeration failures are systematically managed through exception handling and detailed error logging to facilitate efficient debugging.

These two methods help mitigate specific limitations of `rc_genicam_api`: the requirement to configure the GenTL producer without a fallback option manually and the inconsistent formatting and incomplete display of device metadata such as IP and MAC addresses.

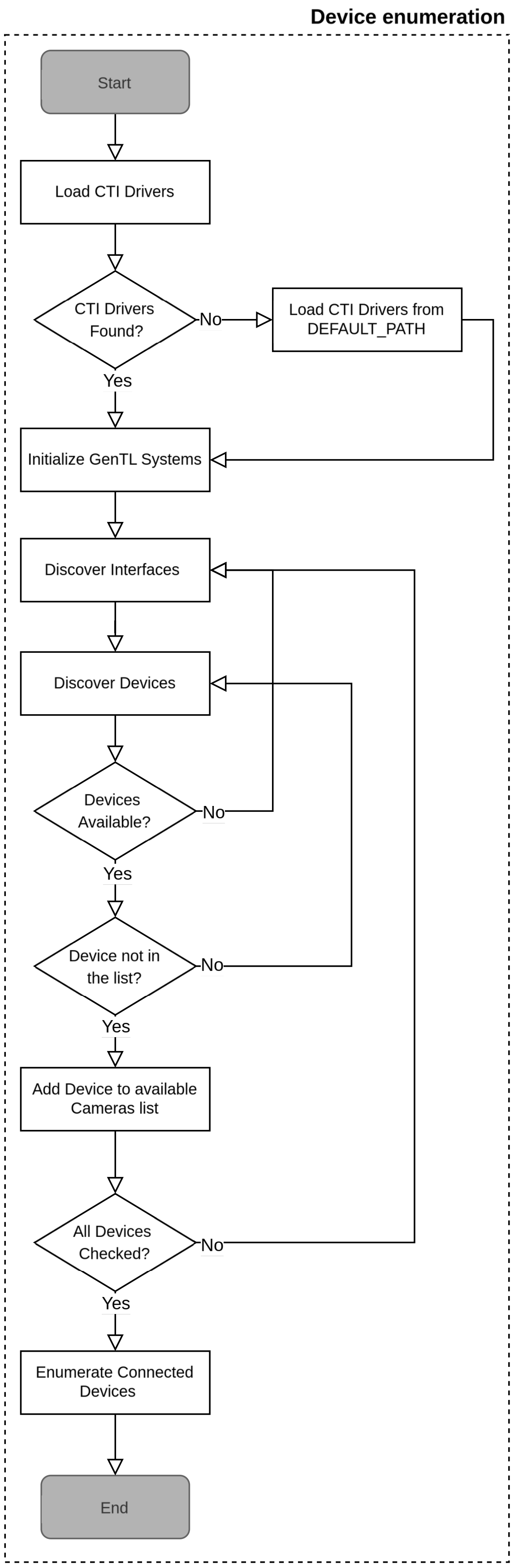


Figure 4.1: Device enumeration process flow.

```
Available Device IDs:

    Device: <Vendor Model> (<SerialNumber>)
    Device: <Vendor Model> (<SerialNumber>)
    ...
```

Listing 4.1: Template structure of output from `enumerateCameras`.

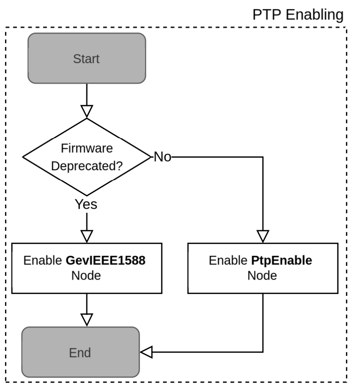
```
Device:             <Vendor Model>
Vendor:             <VendorName>
Model:              <ModelName>
TL type:            <TransportLayerType>
Access status:      <AccessMode>
Serial number:      <SerialNumber>
Current IP:         <IPv4Address>
MAC:                <MACAddress>
Deprecated FW:      <Yes/No>
...
```

Listing 4.2: Template structure of output from `enumerateOpenCameras`.

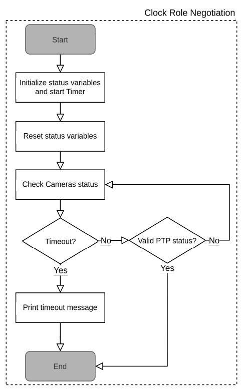
4.1.2 PTP Synchronization

Clock synchronization between cameras is achieved using the IEEE 1588 Precision Time Protocol (PTP). PTP synchronization is controlled within the `SystemManager` through the methods `ptpEnable` and `ptpDisable`.

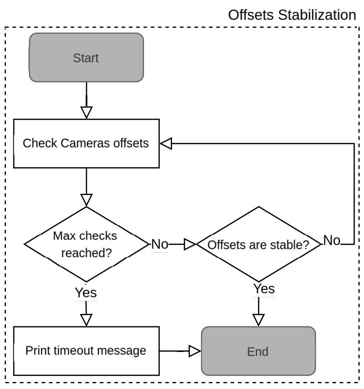
The `NetworkManager` class manages the overall synchronization process. The procedure follows a clearly defined sequence: it begins by enabling PTP on each camera, applying configurations defined in the `ptpConfig` structure through each camera's nodemap using the `enablePtp` method (Figure 4.2a). The process then proceeds through master-slave role negotiation and concludes with offset stabilization. Clock roles are monitored via the `monitorPtpStatus` method (Figure 4.2b), while clock offsets are continuously tracked through `monitorPtpOffset` (Figure 4.2c). Offset values are recorded in CSV format (Listing 4.3) and visualized as line plots showing the evolution of offsets over time relative to configured thresholds.



(a) PTP Enabling (`enablePtp`)



(b) Clock Role Monitoring (`monitorPtpStatus`)



(c) Offset Tracking (`monitorPtpOffset`)

Figure 4.2: Synchronization steps during multi-camera synchronization: (a) enabling PTP, (b) monitoring clock roles, and (c) tracking clock offsets.

If synchronization fails to stabilize within a predefined timeout period, the acquisition sequence is aborted, and a warning is issued. This mechanism prevents indefinite hanging and assists in identifying problematic devices. Successful synchronization is confirmed only if the clock offset remains within a configurable threshold for a specified duration. Threshold parameters, including `ptpOffsetThresholdNs` and `ptpMaxCheck`, are initialized based on empirical observations and are configurable through `GlobalSettings.cpp`.

```
Sample,Cam0_Timestamp_ns,Cam0_Offset_ns,Cam1_Timestamp_ns,Cam1_Offset_ns
0,<timestamp0>,<offset0>,<timestamp1>,<offset1>
...
```

Listing 4.3: Template structure of a PTP offset log file.

4.1.3 Feature Configuration

Configurable imaging parameters are essential for adapting frame acquisition to changing lighting conditions. Within the `SystemManager`, the method `setFeature` enables users to configure key parameters, including gain, resolution, exposure time, frame rate, and pixel format.

Changes to device parameters are delegated to the `DeviceManager` or the `NetworkManager`, depending on the nature of the feature. Features influencing acquisition timing, such as frame rate and exposure time, must be applied uniformly across all devices to maintain synchronization. In contrast, pixel format and gain can be configured individually for each device. Internally, all parameter modifications are routed through the `Camera` class, where each configurable feature is associated with a feature-specific wrapper method. Table 4.1 summarizes the supported feature set and the corresponding setter methods implemented in the `Camera` class. These parameters cover the essential configuration options required for most acquisition scenarios. The system does not enforce value range validation at runtime, relying instead on user responsibility and vendor datasheets to ensure the correctness of the provided values.

Feature	**Type**	**Setter Method**
Width	Integer	`Camera::setWidth`
Height	Integer	`Camera::setHeight`
PixelFormat	Enumeration	`Camera::setPixelFormat`
Gain	Float	`Camera::setGain`
ExposureTime	Float	`Camera::setExposureTime`
AcquisitionFrameRate	Float	`Camera::setFps`

Table 4.1: Configurable camera features and their corresponding setter methods.

4.1.4 Synchronized Acquisition

Streaming is initiated using the `syncFreeRunStream` method provided by the `SystemManager`. This method sequentially orchestrates camera synchronization, network configuration, and acquisition and display operations launch.

After successful synchronization and configuration, the `NetworkManager` begins the network optimization phase by calling to `configurePtpSyncFreeRun`. During this step, inter-packet delays (`GevSCPD`) and camera-specific start offsets (`GevSCFTD`) are computed and set according to the Packet Delay defined in Equation 3.1, 3.2 and 3.3. Exposure time and frame rate, being tightly interdependent, are managed in a two-step process to ensure temporal alignment without overloading the network: first, theoretical FPS bounds are derived from device throughput, frame format, packet size, and delay parameters; second, during stream initialization, each camera reports its current FPS and the system synchronizes all devices to the lowest common frame rate within the bounds. Initially, all cameras uniformly minimize exposure time to maximize frame rate while ensuring image quality through a minimum exposure threshold. These parameters are configured through the API calls `setExposureAndFps`. The adaptive configuration of frame rate and exposure time, including how these interdependencies are managed across the system, is illustrated in Figure 4.3.

Once streaming parameters are configured, the `StreamManager` takes control. Each camera stream is initialized in a separate thread via the `startPtpSyncFreeRun` method. The camera opens its internal stream within each thread, attaches memory buffers, and enters a continuous acquisition loop. Captured frames are extracted, validated, and processed through `processRawFrame`, where they are converted into OpenCV matrices. This method also handles pixel format normalization and annotates frames with metadata such as device identifiers and current frames-per-second (FPS) values.

The `StreamManager` coordinates all acquisition threads and maintains a shared memory structure (`globalFrames`), protected by mutexes, where frames from all devices are aggregated and made accessible to the graphical user interface (GUI) or other consumer modules. Optional frame storage is supported; validated frames are saved as PNG images if enabled. Additionally, introducing a delay enables scheduled acquisition. The choice to use delays instead of Scheduled Action Commands is due to the lack of standardization for firing scheduled actions in GenICam. Thus, the system supports adding a fixed delay before acquisition to trigger a simultaneous start.

For real-time visualization, a composite view is generated using `createComposite`, which tiles frames from all devices into a single rendered output. Frame rendering, format conversion, and layout tiling are handled using OpenCV. The acquisition process continues until user input explicitly terminates it (e.g., a keyboard signal when pressing the "q" Key). At this point, all resources are properly released, and all threads are cleanly shut down.

Figure 4.4 summarizes the entire streaming process.

During the implementation, constraints were encountered when configuring the network. Camera features are limited by their available nodes and value-increment constraints. To ensure broader device compatibility, both delays are configured as multiples of four. For cameras that either do not support `GevSCFTD` or require unusually large scheduling values, the system prioritizes their scheduling during initialization. To optimize network throughput, the system configures the maximum transmission unit (MTU) through the `GevSCPSPacketSize` parameter on the cameras. The host network interface MTU must be adjusted to 9000 bytes, enabling jumbo frames. This significantly reduces packet overhead and inter-frame gaps, improving data transfer efficiency. However, the MTU setting is configurable to accommodate cases where specific cameras do not support large packet sizes.

4.2 User Interfaces

All these functions can be called through a command-line interface (CLI) and a graphical user interface (GUI). Both interfaces interact with a shared backend based on the `SystemManager` class.

System-wide configuration parameters—such as default exposure time, frame rate, CTI path, and logging levels—are centralized in the `GlobalSettings.cpp` file. These values can be modified before the system launch to adapt behavior for specific hardware setups or experimental conditions.

4.2.1 Command-Line Interface (CLI)

The CLI is implemented in `cli.cpp`. The available commands, summarized in Listing 4.4 summarizes the CLI commands supported by the system. This is also printed with the

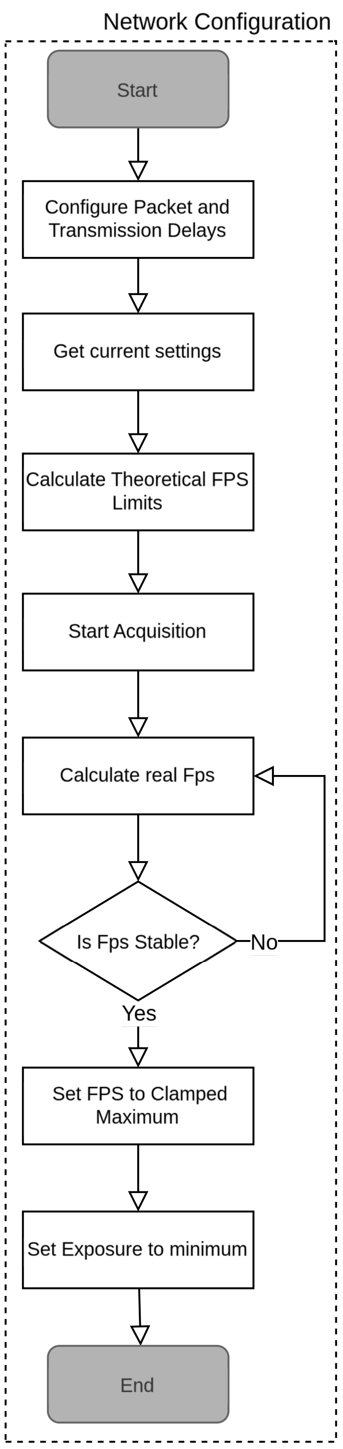


Figure 4.3: Flowchart for Network Configuration and Frame Rate Stabilization.

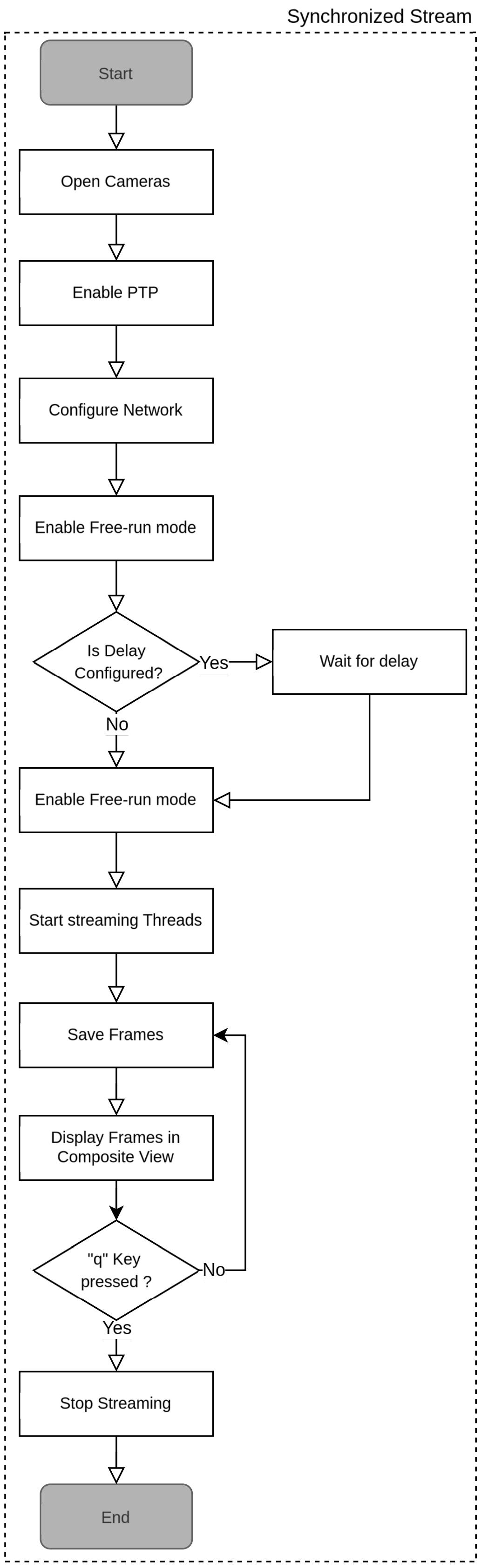


Figure 4.4: Streaming process flowchart illustrating camera opening, configuration, synchronization, and composite display.

`-help` flag when launching the application. Each CLI command maps directly to a core workflow:

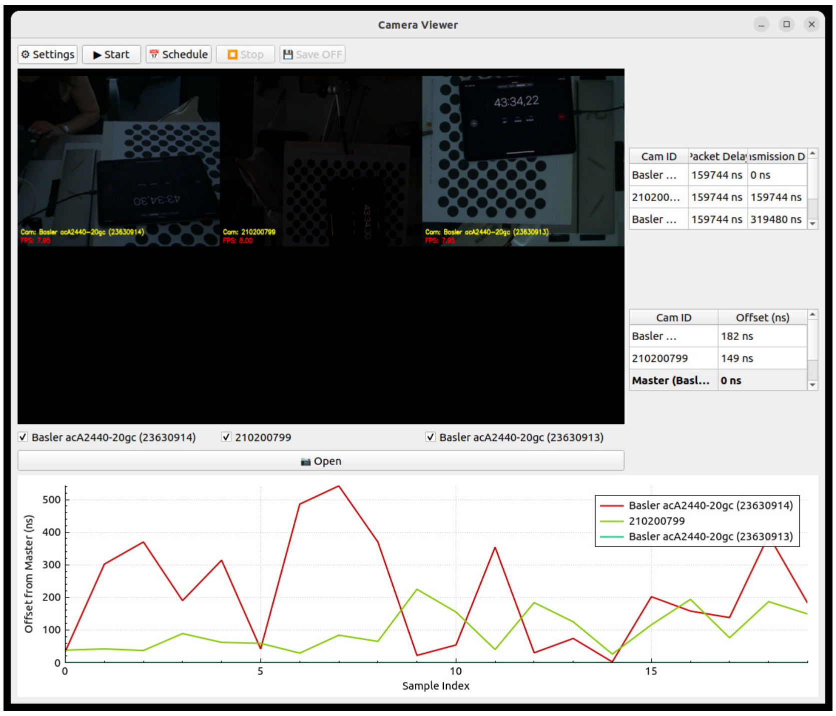
- `-list` performs device discovery, scanning the network for available cameras and printing their IDs.
- `-enable-ptp` and `-disable-ptp` control PTP synchronization. Cameras not intended for synchronization should have PTP explicitly turned off to avoid incorrect master clock elections.
- `-start` initiates synchronized acquisition, with optional parameters for acquisition delay and frame saving.

```
Usage:
  ./main --list
  ./main --start --cameras "cam1, cam2,..." [--delay <ms>] [--no-
     save]
  ./main --enable-ptp --cameras "cam1, cam2, ..."
  ./main --disable-ptp --cameras "cam1,cam2,..."
...
```

Listing 4.4: Command-line interface usage

4.2.2 Graphical User Interface (GUI)

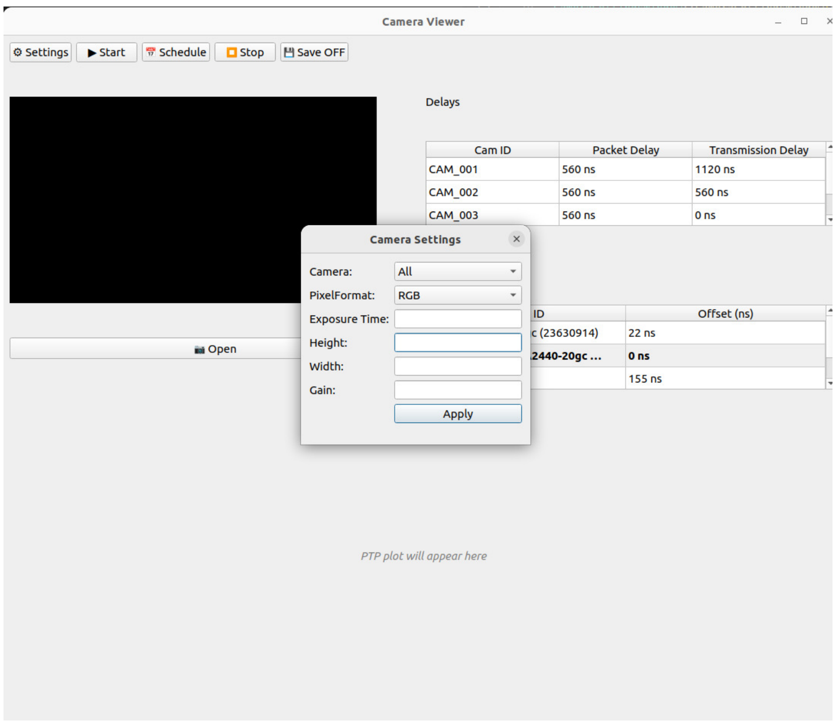
The GUI, developed with Qt6, is organized into three main regions: a live feed panel displaying real-time synchronized video streams from all connected cameras, a diagnostics panel presenting stream parameters, synchronization statuses, and network statistics, and a plot view visualizing the PTP clock offsets of each device relative to the master. Each GUI action directly triggers the same backend workflows as the CLI. Users can toggle recording and streaming individually for each device using checkboxes beneath each video feed. In addition, a dedicated settings dialog also allows users to modify acquisition parameters globally or per device. The graphical user interface (GUI) elements, such as checkboxes, tables, buttons, and plotting components, are implemented using the Qt6 library. The video streams are embedded in OpenCV windows integrated into the main application window. These elements are illustrated in Figure 4.5. The GUI layout is organized into three main sections: a live video panel at the top, a parameters and diagnostics panel on the right, and a plot displaying the PTP synchronization offsets at the bottom. The GUI also features a settings dialog (Figure 4.6), where users can configure key acquisition parameters globally for all devices or individually per device.


Camera Viewer
Settings
Start
Schedule
Stop
Save OFF
Cam: Basler acA2440-20gc (23630914)
FPS: 7.95
Cam: 210200799
FPS: 8.00
Cam: Basler acA2440-20gc (23630913)
FPS: 7.95
✓ Basler acA2440-20gc (23630914)
✓ 210200799
✓ Basler acA2440-20gc (23630913)
Open
Offset from Master (ns)
Sample Index
Basler acA2440-20gc (23630914)
210200799
Basler acA2440-20gc (23630913)

Cam ID	›acket Dela	ısmission D
Basler ...	159744 ns	0 ns
210200...	159744 ns	159744 ns
Basler ...	159744 ns	319480 ns

Cam ID	Offset (ns)
Basler ...	182 ns
210200799	149 ns
Master (Basl...	**0 ns**

Figure 4.5: Graphical user interface of the camera synchronization application. The top panel displays synchronized video streams. Tables on the right show stream parameters and PTP status. A plot below visualizes the offset of each camera relative to the master.



Delays

Cam ID	Packet Delay	Transmission Delay
CAM_001	560 ns	1120 ns
CAM_002	560 ns	560 ns
CAM_003	560 ns	0 ns

Camera Settings

Camera:	All
PixelFormat:	RGB
Exposure Time:	
Height:	
Width:	
Gain:	

Apply

ID	Offset (ns)
c (23630914)	22 ns
2440-20gc ...	**0 ns**
	155 ns

Figure 4.6: Settings dialog of the GUI. Users can modify acquisition parameters and apply them to individual cameras or globally to all connected devices.

5 Experimental Setup and Methodology

This chapter presents the validation results of the synchronization framework introduced in Chapter 3 and implemented in chapter 4. The evaluation aims to verify that the solution fulfills the defined functional requirements. The chapter begins by outlining the test environment and methodology.

5.1 Test Environment

All tests were conducted using a multi-camera setup connected via a common network switch and controlled by an Ubuntu host computer. The hardware configuration of the test environment is illustrated in Figure 5.1. Cameras from 3 different manufacturers were

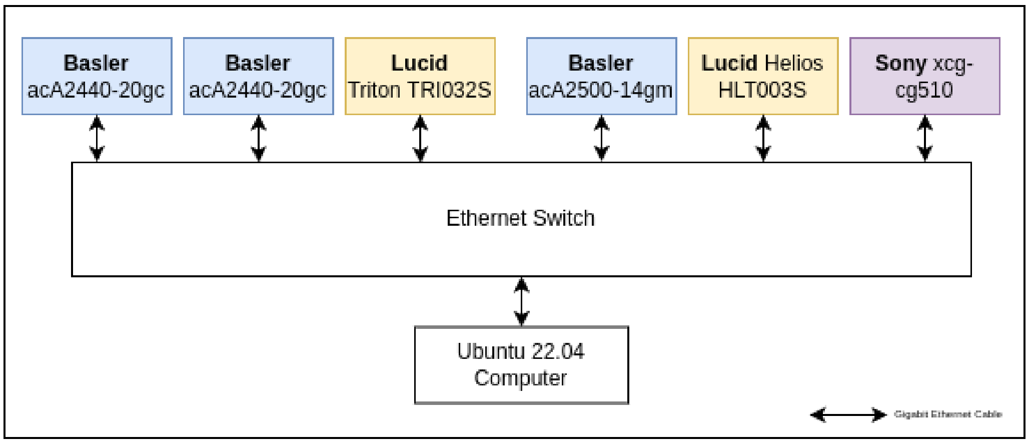


Figure 5.1: Schematic of the hardware test setup.

used across the various test cases.

The Lucid Helios HLT003S, Sony XCG-CG510, and Basler acA2500-14gm were added t for the device discovery Test (TC-1). The remaining tests use a simplified setup consisting of two Basler acA2440-20gc cameras and one Lucid Triton TRI032S.

This reduced configuration was chosen for compatibility reasons. The Basler acA2500-14gm camera implements the GigE Vision 1.x standard, which does not support the Precision Time Protocol (PTP). Although the Sony XCG-CG510 supports PTP, it lacks essential parameters for network configuration, such as `GevSCFTD`, which prevented its full integration into the framework. The Lucid Helios HLT003S, a time-of-flight (ToF) camera, introduces additional complexity due to its 3D depth data streams, making it unsuitable for initial synchronization testing within the current system design.

The cameras had to be assigned IP addresses within the same subnet for all tests. This configuration step was performed using the Aravis [33] toolkit and the vendor-specific APIs provided by the respective manufacturers.

5.2 Test Cases

Each test case maps to a set of functional requirements, defined in the requirements section 3.1. An additional test case evaluates the framework's overall acquisition and synchronization stability under extended usage. Table 5.1 summarizes all test cases.

Test ID	Description	Success Criteria	Requirements
TC-1	Device Discovery	All connected camera IDs are detected and listed in the console output.	FR-1, FR-5
TC-2	Multi-Camera Synchronization	Offset variance remains below 100 µs. Synchronization and network parameters are logged and saved to the output directory.	FR-2, FR-3, FR-5
TC-3	Coordinated Image Acquisition	No frames are dropped during acquisition. All images are saved to the output directory. Frame timestamps remain synchronized within the nano- to microsecond range.	FR-2, FR-4, FR-5
TC-4	Extended Acquisition	Zero frame drops and stable synchronization offsets are maintained during 10 minutes of continuous operation.	FR-2, FR-4, FR-5

Table 5.1: Test Case Specifications with Success Criteria and Associated Requirements

5.2.1 Device Discovery (TC-1)

The first test case focused on detecting and enumerating all connected camera devices in the system.

The required CTI files (GenTL producers) for each camera were first downloaded from the respective manufacturers to perform this test. After downloading, the GenTL search path was correctly configured by setting the environment variable `GENICAM_GENTL64_PATH`. This variable allows the application to locate the relevant producer libraries during runtime. The configuration was performed in the terminal as follows:

```
# Clear any previously set GenTL path
unset GENICAM_GENTL64_PATH

# Set path to multiple GenTL producers (e.g., Basler and Lucid)
export GENICAM_GENTL64_PATH=/opt/pylon/lib/gentlproducer/gtl:\
/home/test/Downloads/ArenaSDK_v0.1.92_Linux_x64_cti/ArenaSDK_\
Linux_x64/lib64

# Verify the environment variable
echo $GENICAM_GENTL64_PATH
```

Listing 5.1: Steps for setting GenTL path.

Once the environment was set, the client sent the enumeration command, which then accessed the defined GenTL producers to scan for connected devices.

5.2.2 Multi-Camera Synchronization (TC-2)

This test case evaluated the framework's ability to synchronize multiple cameras using the Precision Time Protocol (PTP) and visualize the synchronization behavior.

Any camera not intended to participate in synchronization was explicitly disabled via `PtpDisable`. The system was then configured with a time window of 20 consecutive checks, which was chosen empirically based on initial trial runs. The offset threshold was set to 1 microsecond to ensure synchronization precision within the sub-microsecond range.

Once this configuration was applied, the `PtpEnable` command is called with the IDs of devices to be synchronized.

5.2.3 Coordinated Image Acquisition (TC-3)

This test case assesses the framework's ability to perform synchronized image acquisition and saving across multiple PTP-synchronized cameras. Acquisition starts with a synchronized acquisition command `-start`, issued via the CLI. A precise real-time stopwatch on a tablet was used to verify frame alignment across camera previews visually. Both composites displayed Frames and saved single Frames from the output directory are evaluated to asses synchronization accuracy.

5.2.4 Extended Acquisition (TC-4)

This test case evaluated the overall system stability and performance under continuous usage conditions to determine whether the system can sustain its frame rate and synchronization accuracy over longer periods.

A prolonged streaming test was conducted over 10 minutes. The procedure involved maintaining uninterrupted image streaming from all synchronized cameras over 10 minutes while monitoring the system for frame loss, crashes, or memory leaks. After the test duration, offset values were read and analyzed from the camera logs and the captured frames to assess synchronization stability.

6 Results and Evaluation

In this chapter, the results observed from the conducted test cases are presented and evaluated. The objective is to assess the effectiveness of the proposed approaches and to identify potential areas for further improvement.

6.1 Device Discovery (TC-1)

During the initial stages of testing, devices were listed multiple times. Identifying devices based on their GenTL-assigned IDs led to inconsistencies. IDs are not guaranteed to be unique or persistent across sessions; they depend on the transport layer and may vary.

To address the listed Cameras, the system was modified to use the cameras' serial numbers for identification. Serial numbers provide a consistent and unique identifier that remains stable regardless of the driver.

After this adjustment, the application could reliably detect and enumerate all six connected devices without errors. Devices included the Sony XCG-CG510, which was successfully listed using the default Baumer CTI file.

The output of the `listdevices` command is shown in Listing 6.1.

```
Available Devices IDs:
    Device:            Basler acA2500-14gm (21639790)

    Device:            210201103

    Device:            210200799

    Device:            Basler acA2440-20gc (23630914)

    Device:            Basler acA2440-20gc (23630913)

    Device:            3201058
```

Listing 6.1: Available Device IDs

The device discovery mechanism successfully met the functional requirements defined under FR-1. The system detected all connected cameras using serial numbers, demonstrating capability in identifying a heterogeneous set of devices. Additionally, fallback support for Camera Transport Interface (CTI) through alternative GenTL producers was verified and functioned as intended.

6.2 Multi-Camera Synchronization (TC-2)

PTP mode was enabled without error; no timeouts or exceptions occurred throughout the procedure. Real-time debug messages confirmed the correct network setup, successful clock negotiation, and stable offset propagation. Both synchronization offsets and bandwidth usage metrics were logged and visualized.

Key parameters such as current PTP offset values, clock roles, and negotiation states were captured and stored in structured .csv files under the build/output directory. These log files were then used to generate plots that visualized synchronization performance over time. A representative snippet of the console output is shown in Listing 6.2. The corresponding synchronization offset plot is shown in Figure 6.1.

In this scenario, device Basler acA2440-20gc (23630913) served as the master clock with a fixed offset of 0, while devices 210200799 and Basler acA2440-20gc (23630914) operated as slave cameras. Both maintained an offset consistently below 1 microsecond throughout the observation window.

```
Available Devices IDs:
    Device:             Basler acA2440-20gc (23630913)
    Device:             Basler acA2440-20gc (23630914)
    Device:             210200799

[DEBUG] Camera Basler acA2440-20gc (23630914): PTP enabled
[DEBUG] Camera Basler acA2440-20gc (23630913): PTP enabled
[DEBUG] Camera 210200799: PTP enabled
...
[DEBUG] PTP synchronization successful!
[DEBUG] PTP synchronized with 2 slave cameras and 1 master
...
[DEBUG] Camera 210200799 stable within time window.
[DEBUG] Offset + timestamp history written to
./output/offset/ptp_offset_history_20250416_152624.csv

Camera ID:                  Basler acA2440-20gc (23630914)
PTP Status:                 Slave
PTP Offset From Master:     26 ns

Camera ID:                  Basler acA2440-20gc (23630913)
PTP Status:                 Master
PTP Offset From Master:     0 ns

Camera ID:                  210200799
PTP Status:                 Slave
PTP Offset From Master:     134 ns
```

Listing 6.2: Excerpt from the PTP synchronization log output

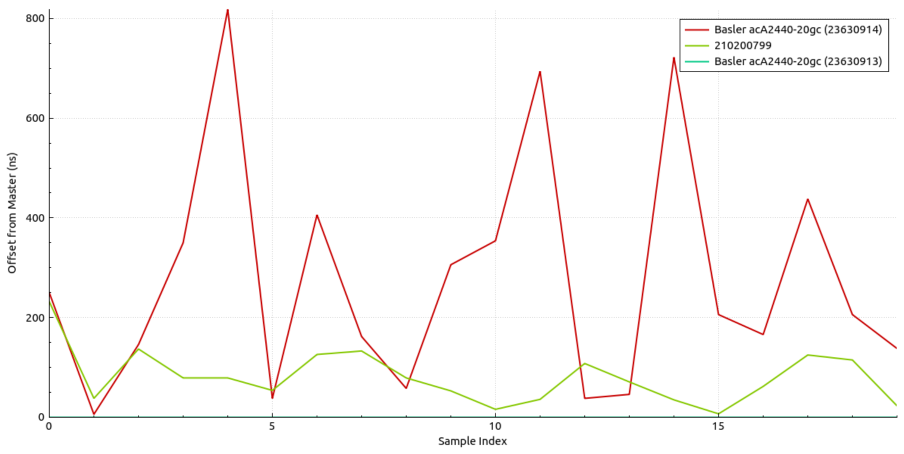


Figure 6.1: PTP synchronization offsets over 20 checks. Camera `Basler acA2440-20gc (23630913)` is the master clock; Cameras `Basler acA2440-20gc (23630914)` and `210200799` are slaves with sub-microsecond stability.

This test validated the framework's capability to synchronize multiple cameras using Precision Time Protocol (PTP), fulfilling the requirements of FR-2 and FR-3. The system consistently achieved clock role negotiation and stable offset propagation, confirming that synchronization was operating effectively under standard conditions. The absence of timeout warnings indicates that the empirically selected window and threshold parameters were well-tuned for the tested hardware configuration.

6.3 Coordinated Image Acquisition (TC-3)

Visual inspection of the live previews confirmed that images were captured simultaneously across all cameras. No frames were lost, and all images were correctly saved under `/build/output`. Additionally, frames from all devices were displayed in a common composite, shown in figure 6.2. The frame rate for all three cameras was capped at a maximum of 8 frames per second. Across three test trials, a temporal offset in a millisecond range was observed between frames: Analysis of the displayed frame in a composite view revealed an average temporal offset of 435 ms. In contrast, offset calculations based on retrieving the closest frame from the saved capture yielded a lower average offset of 141 ms.

Tables 6.1 summarize the per-camera offset observed during the trials.

Test ID	Composite View Offset [ms]			Saved Frame Offset [ms]		
	Cam 13	Cam 14	Cam 99	Cam 13	Cam 14	Cam 99
1	40	40	0	40	40	0
2	110	130	0	30	<1	0
3	100	90	0	10	70	0

Table 6.1: Comparison of PTP offset in milliseconds across three test cases and three cameras. **Cam 13:** 23630913, **Cam 14:** 23630914, **Cam 99:** 210200799 (Master Clock).

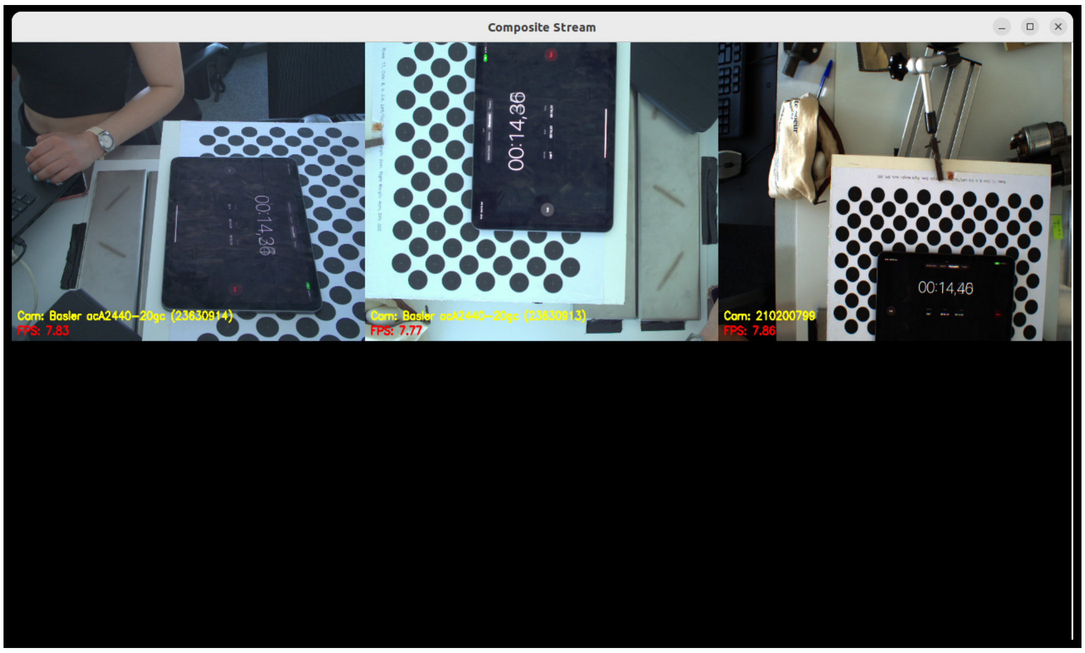


Figure 6.2: Composite stream display from three synchronized cameras showing coordinated capture with real-time previews. Each panel includes the camera identifier and current FPS.

This test demonstrated that the framework supports coordinated image acquisition across multiple cameras, fulfilling functional requirements FR-4 and FR-5. In most instances, frames were successfully displayed in real-time and saved to disk, with synchronization accuracy maintained at sub-100 microseconds.

However, this value does not fully align with the internal offset readings, which suggest synchronization in the range of approximately 100 nanoseconds. Notably, during all conducted tests, accurate microsecond-range synchronization between frames was observed only once—specifically in Test 2, between Camera 23630914 and Camera 210200799 (see Table 6.1). It is important to note that this alignment was not observed in the displayed composite view but was obtained manually by retrieving the frame with the closest timestamp to the master clock from the saved frames. This indicates that the composite display does not consistently select the temporally closest frame.

To address this, introducing a short stabilization process before starting the stream could help ensure that frames are synchronized more accurately at startup. The streaming process is also initiated via software commands, introducing network-induced delays. Utilizing hardware-based scheduled action commands could minimize this delay, as such commands are executed directly by the cameras and bypass the uncertainties of software-level triggering. While this solution improves synchronization, it diverges from the non-functional requirement NFR-1. The GenICam standard currently lacks a unified command-firing mechanism, and as such, implementing scheduled action commands requires the use of proprietary APIs provided by individual camera manufacturers. Introducing vendor-specific dependencies conflicts to maintain a fully standard-compliant and hardware-agnostic system.

The implemented frame rate stabilization mechanism also performed as intended. All cameras automatically adjusted their frame rates to match the maximum stable rate across devices, achieving consistent values of approximately 8 FPS. While this worked effectively under the configured dynamic network delay conditions, this frame rate may be insufficient for applications requiring higher temporal resolution.

Further optimization is recommended to support higher frame rates, particularly regarding image encoding and storage. For example, switching to a more compressed image format and avoiding saving full-color images could reduce bandwidth and disk usage, enabling higher throughput without compromising synchronization fidelity.

6.4 Extended Acquisition (TC-4)

The system exhibited stable operation throughout the 10-minute continuous streaming test, with no frame loss. Additionally, the framework showed no signs of crashes or memory leaks, and frame synchronization remained in the ms range.

During a sustained 10-minute acquisition session, the system demonstrated operational stability with no frame drops and consistent synchronization, satisfying FR-2, FR-4, and FR-5 requirements. Latency remained within acceptable limits, and the offset values recorded after the acquisition confirmed continued temporal alignment.

7 Summary and Conclusion

This thesis presented the design, implementation, and evaluation of a software-based synchronization framework for GenICam-compliant cameras using the GigE Vision 2.0 protocol. The framework was implemented as a modular C++ software tool with a command-line interface (CLI), a graphical interface (GUI), and a high-level API structured around a central `SystemManager`. It was developed with vendor neutrality in mind, relying on open standards and open-source libraries to support a wide range of cameras.

Key Achievements

The developed system enables precise temporal alignment of multiple image sources and provides flexible control over acquisition processes. The following milestones were successfully achieved:

1. **Device discovery:** All connected cameras were reliably identified using their serial numbers, independent of the manufacturer. A copy of a default CTI driver was included to enable device enumeration without the original manufacturer drivers. Extended metadata, such as IP and MAC addresses, was retrieved and displayed uniformly across all devices.
2. **PTP-based synchronization:** Precision Time Protocol (PTP) was used to establish time alignment across all devices, with synchronization status and clock offsets logged and visualized. The framework also supported devices running deprecated firmware versions. Internal clocks were successfully synchronized within a sub-microsecond range.
3. **Coordinated image acquisition:** Cameras were triggered to acquire images synchronized, with optional delay configurations available in free-run mode. Frames were successfully saved in `.png` format. Live streams could also be visualized in composite views. Bandwidth sharing was effectively managed through dynamic parameter adjustments (frame rate, exposure time, and acquisition delays) when the overall frame rate was maximized.
4. **Interface integration:** A functional GUI and CLI provided user-friendly access to system functionality, supported by visual feedback.

Experimental validation showed that sub-microsecond synchronization was attainable under controlled conditions. The system demonstrated the ability to perform stable multi-camera acquisitions while generating outputs for evaluating network conditions and synchronization performance.

Observed Limitations

Despite the overall success, several technical and usability limitations were encountered during testing. Each is outlined below with corresponding suggestions for mitigation:

1. **Millisecond-Scale Frame Offsets:** Saved and displayed frames occasionally exhibited offsets in the millisecond range, contrasting with the sub-microsecond synchronization values reported by internal camera clocks. This discrepancy is likely due to software-trigger delays, frame buffering, timestamp misalignment, and minor processing overheads during frame saving and display. To address this, future

improvements could include saving raw frames directly and implementing scheduled action command mechanisms to minimize additional delays.

2. **GUI Limitations:** Although the graphical user interface is operational, it lacks several advanced usability features, such as clear feedback messages, synchronization status indicators, and robust error handling. While real-time frame display is supported, the visualization of synchronization data requires users to specify the path to generate log files manually. Introducing an automatic log discovery mechanism—such as a configurable delay to locate and retrieve plots from the output directory—would significantly improve usability.
3. **Low Frame Rate:** The system achieves an average frame rate of approximately 7–8 FPS, which may be insufficient for high-throughput or time-critical applications. Potential optimizations include enabling frame compression to reduce network bandwidth usage, fine-tuning buffer allocation and data transmission parameters, and reducing frame resolution to lower the per-frame data size.

Conclusion and Future Work

The implemented framework satisfies the key functional requirements for multi-camera synchronization, configuration, and acquisition. It demonstrates the viability of a modular, vendor-independent system capable of operating over heterogeneous hardware in controlled network environments. While certain features remain areas for improvement, the results validate the core design principles and show that the system is suitable for practical deployment and further extension.

Throughout development and testing, both the command-line interface (CLI) and the graphical user interface (GUI) proved effective for controlling synchronized operations and device features.

Regarding protocol-level limitations, certain GenICam features such as `GevSCFTD` are not uniformly supported across all vendors, complicating packet delay configurations in mixed-device setups. Moreover, the reliance on vendor-supplied GenTL Producers (CTI files) introduces a degree of platform dependency, as driver-level behavior may vary between operating systems and hardware manufacturers.

Future development could prioritize the following directions:

- Enhancing synchronization stability through PTP scheduling mechanisms (e.g., Scheduled Action Commands) and deeper network diagnostics.
- Increasing acquisition rates by optimizing encoding strategies (e.g., introducing compression formats).
- Extending the GUI to support advanced control logic, synchronization status visualization, and more intuitive error reporting.
- Expanding the framework to handle depth and multi-modal data streams, enabling stereo and 3D camera systems support.
- Improving compatibility across diverse GenICam-compliant devices by implementing missing GenICam Nodes (e.g., Transmission Delay, IssueActionCommand).

The findings and implementation presented in this thesis establish a strong foundation for further exploration into high-precision, multi-camera synchronization frameworks. Addressing key challenges in synchronization accuracy, network configuration, and acquisition control offers a practical and extensible solution suitable for deployment in experimental and industrial environments.

List of Acronyms

API	Application Programming Interface
CLI	Command Line Interface
CTI	Camera Transport Interface
EMVA	European Machine Vision Association
FPS	Frames Per Second
GenApi	Generic Application Programming Interface
GenCP	Generic Control Protocol
GenDC	Generic Data Container
GenICam	Generic Interface for Cameras
GenTL	Generic Transport Layer
GigE	Gigabit Ethernet
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
LAN	Local Area Network
MTU	Maximum Transmission Unit
NTP	Network Time Protocol
OpenCV	Open Source Computer Vision Library
PFNC	Pixel Format Naming Convention
PTP	Precision Time Protocol
SDK	Software Development Kit
SFNC	Standard Features Naming Convention
UDP	User Datagram Protocol
XML	Extensible Markup Language

Bibliography

[1] EMVA – European Machine Vision Association. Genicam standard features naming convention (sfnc). https://www.emva.org/standards/genicam/, n.d. URL https://www.emva.org/standards/genicam/genicam-downloads. Available online at the official EMVA website.

[2] EMVA. Fsf vision standards brochure 2022, 2022. URL https://www.emva.org/wp-content/uploads/FSF-VS-Brochure-2022-A4-vcb-lowres.pdf. Accessed: 2025-03-05.

[3] European Machine Vision Association. Emva website. Online, N/A. URL https://www.emva.org/. Accessed: January 21, 2025.

[4] European Machine Vision Association. Genicam standard. Online, N/A. URL https://www.emva.org/standards-technology/genicam/. Accessed: January 21, 2025.

[5] Automated Imaging Association (AIA). GigE Vision Standard: A High-Performance Communication Protocol for Machine Vision. https://www.gigevision.org, 2023. Accessed: 2023-10-31.

[6] Steve Kinney. The future of machine vision imaging systems. *Quality Magazine*, 2019. URL https://www.qualitymag.com/articles/95531-the-future-of-machine-vision-imaging-systems.

[7] IEEE Instrumentation and Measurement Society. 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. https://standards.ieee.org/standard/1588-2019.html.

[8] Roboception GmbH. rc_genicam_api: A GenICam-compliant C++ Library for Camera Communication. https://github.com/roboception/rc_genicam_api, 2025. Accessed: 2025-01-31.

[9] Vasanth Subramanyam, Jayendra Kumar, and Shiva Nand Singh. Temporal synchronization framework of machine-vision cameras for high-speed steel surface inspection systems. *Journal of Real-Time Image Processing*, 19(2):445–461, 2022. ISSN 1861-8219. doi: 10.1007/s11554-022-01198-z. URL https://doi.org/10.1007/s11554-022-01198-z.

[10] 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*, MM ’06, page 137–140, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934472. doi: 10.1145/1180639.1180679. URL https://doi.org/10.1145/1180639.1180679.

[11] 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, 2012. doi: 10.1109/ISM.2012.79.

[12] Anna Casanovas and Andrea Cavallaro. Audio-visual events for multi-camera synchronization. *Multimedia Tools and Applications*, 74, 02 2014. doi: 10.1007/s11042-014-1872-y.

[13] Yunhyeok Han, Stefania Lo Feudo, Gwendal Cumunel, and Franck Renaud. Sub-frame timestamping of a camera network using a coded light signal. *Measurement*, 236:115046, 2024. ISSN 0263-2241. doi: https://doi.org/10.1016/j.measurement.2024.115046. URL `https://www.sciencedirect.com/science/article/pii/S026322412400931X`.

[14] Nicolas Boizard, Kevin El Haddad, Thierry Ravet, François Cresson, and Thierry Dutoit. Deep learning-based stereo camera multi-video synchronization, 2023. URL `https://arxiv.org/abs/2303.12916`.

[15] Igor Pereira, Luiz F. Silveira, and Luiz Gonçalves. Video synchronization with bit-rate signals and correntropy function. *Sensors*, 17(9), 2017. ISSN 1424-8220. doi: 10.3390/s17092021. URL `https://www.mdpi.com/1424-8220/17/9/2021`.

[16] Felix Wermke, Thorben Wübbenhorst, and Beate Meffert. Optical synchronization of multiple time-of-flight cameras implementing tdma. In *2020 IEEE SENSORS*, pages 1–4, 2020. doi: 10.1109/SENSORS47125.2020.9278667.

[17] Basler AG. Triggered image acquisition. `https://docs.baslerweb.com/triggered-image-acquisition`, N/A. Accessed on Month Day, Year.

[18] Hieu Nguyen, Dung Nguyen, Zhaoyang Wang, Hien Kieu, and Minh Le. Real-time, high-accuracy 3d imaging and shape measurement. *Appl. Opt.*, 54(1):A9–A17, Jan 2015. doi: 10.1364/AO.54.0000A9. URL `https://opg.optica.org/ao/abstract.cfm?URI=ao-54-1-A9`.

[19] D.L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991. doi: 10.1109/26.103043.

[20] Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–269, 2008. doi: 10.1109/IEEESTD.2008.4579760.

[21] AIA - Automated Imaging Association. Aia — part of the association for advancing automation (a3). `https://www.automate.org/aia`, 2024. Accessed April 2025.

[22] Japan Industrial Imaging Association. Jiia — japan industrial imaging association. `https://jiia.org/en/`, 2024. Accessed April 2025.

[23] Ricardo Sousa, Martin Wäny, and Pedro Santos. Multi-camera synchronization core implemented on usb3 based fpga platform. *Proceedings of SPIE - The International Society for Optical Engineering*, 9403, 02 2015. doi: 10.1117/12.2082928.

[24] DALSA Corporation. Gige vision for real-time machine vision. Technical report, Teledyne DALSA, 2010. URL `https://nstx.pppl.gov/nstxhome/DragNDrop/Operations/Diagnostics_%26_Support_Sys/D1CCD/GigE_Vision_for_Realtime_MV_11052010.pdf`. Accessed: April 2025.

[25] Basler AG. Synchronous and in real time: Operation of multiple cameras in a gige network, 2021. URL `https://assets-ctf.baslerweb.com/dg51pdwahxgw/2uJZ0zs5jf5fQjhN6oOs7U/be0edb3de81e40cf6b823ae45d18969a/BAS1601_White_Paper_Multi_Camera_applications__EN.pdf`. Accessed: 2025-03-05.

[26] Basler AG. pylon camera software suite. `https://www.baslerweb.com/en/products/software/basler-pylon-camera-software-suite/`, 2024. Accessed April 2025.

[27] Allied Vision Technologies GmbH. Vimba x sdk. `https://www.alliedvision.com/en/products/software/vimba-sdk/`, 2024. Accessed April 2025.

[28] Roboception GmbH. Roboception gmbh: 3d perception and robotics solutions, 2025. URL `https://roboception.com`. Accessed: 2025-04-26.

[29] GenICam Community. Harvesters: A Python Library for GenICam-based Machine Vision Cameras. `https://github.com/genicam/harvesters`, 2025. Accessed: 2025-01-31.

[30] Lucid Vision Labs. Bandwidth sharing in multi-camera systems. `https://support.thinklucid.com/app-note-bandwidth-sharing-in-multi-camera-systems/`, n.d. Accessed: April 11, 2025.

[31] Sanxo. Gige vision bandwidth control. `https://www.sanxo.eu/content/techtips/TechTip_GEV_BandwidthCtl_EN.pdf`, n.d. Accessed: April 11, 2025.

[32] Basler AG. Controlling packet timing with delays in gige vision cameras. `https://assets-ctf.baslerweb.com/dg51pdwahxgw/1Rtd9lVAHHw5lJQ7uVqxqx/bf7a8091f70cd803be9e179b07f4e630/AW00064902000_Controlling_Packet_Timing_With_Delays.pdf`, 2022. Accessed: 2025-04-27.

[33] Aravis. Aravis github repository, 2023. URL `https://github.com/AravisProject/aravis`.

Annex

Command-Line Interface (CLI) Demonstration

This appendix illustrates the functionality of the developed command-line interface (CLI), which enables full control of camera discovery, configuration, synchronization, and acquisition directly from the terminal.

Supported Commands

The CLI supports the following operations:

- `-list` List all connected cameras.
- `-start` Start synchronized image acquisition.
- `-enable-ptp / -disable-ptp` Enable/disable PTP time synchronization.
- `-set-feature` Adjust camera features such as exposure time, width, or pixel format.

CLI Help Output

```
Usage:
  ./cli --list [--cameras "cam1,cam2,..."]
  ./cli --start --cameras "cam1,cam2,..." [--delay <ms>] [--no-save]
  ./cli --enable-ptp --cameras "cam1,cam2,..."]
  ./cli --disable-ptp --cameras "cam1,cam2,..."]
  ./cli --set-feature --feature <name> --value <value>
  [--cameras "cam1,cam2,..."]
Options:
  --list                       List all connected cameras
  --start                      Start synchronized acquisition
  --cameras "cam1,cam2,..."    Comma-separated list of camera IDs
  --delay "ms"                 Acquisition delay in milliseconds (default:
     0)
  --no-save                    Do not save video or PNGs
  --enable-ptp                 Enable PTP on selected cameras
  --disable-ptp                Disable PTP on selected cameras
  --set-feature                Set a feature on cameras
  --feature <name>             Feature name (e.g., width,
  height, gain, exposure_time)
  --value <value>              Value to assign to the feature
  --help                       Show this help message
```

Listing 1: CLI Help Message

Example: Listing Available Cameras

```
$ ./cli --list
Available Devices IDs:
    Device: Basler acA2500-14gm (21639790)
    Device: Basler acA2440-20gc (23630913)
    Device: Basler acA2440-20gc (23630914)
    Device: 210200799
```

Listing 2: Listing Detected Cameras

Example: Enabling PTP Synchronization

```
$ ./cli --enable-ptp --cameras "23630913,23630914,210200799"
[DEBUG] Camera ...: PTP enabled
[DEBUG] PTP synchronization successful!
[INFO] Offset plot saved to ./output/plots/ptp_offsets_<timestamp>.png
```

Listing 3: Enabling PTP

Example: Starting Acquisition

```
$ ./cli --start --cameras "23630913,23630914,210200799" --no-save
[DEBUG] Cameras synchronized and streaming...
Press 'q' on the OpenCV window to stop.
```

Listing 4: Synchronized Free-Run Acquisition

Example: Setting Camera Feature

```
$ ./cli --set-feature --feature height --value 2000 --cameras "210200799"
[DEBUG] Camera 210200799 Height set to: 1536
```

Listing 5: Setting Feature Height

Directory Structure Overview

The following listing outlines the folder structure of the project directory. It highlights the separation between core source files, GUI components, and build artifacts.

```
camcontrol/
|-- build/                    # Compiled binaries and output files
|-- default_cti/              # Default Baumer CTI files (for camera
   initialization)
|-- gui/                      # Graphical User Interface (GUI) components
|   |-- build/                # GUI-specific build artifacts
|   |-- src/                  # GUI source files (MainWindow,
   CameraSettingsWindow, etc.)
|   `--  CMakeLists.txt       # Build configuration script for GUI
|-- src/                      # Core source files (camera, device,
   network, streaming)
|   |-- Camera.cpp/.hpp
|   |-- DeviceManager.cpp/.hpp
|   |-- NetworkManager.cpp/.hpp
|   |-- StreamManager.cpp/.hpp
|   |-- SystemManager.cpp/.hpp
|   |-- cli.cpp               # Command-line interface (CLI) logic
|   |-- GlobalSettings.cpp/.hpp
|   `-- qcustomplot.cpp/.h    # Plotting components for PTP offsets
|-- Acknowledgements.txt      # Acknowledgements and references
|-- CMakeLists.txt            # Build configuration script
`-- Doxyfile                  # Doxygen documentation configuration
```

Listing 6: Project Directory Layout

Note: The `build/` and `gui/build/` directories are generated during compilation and are not tracked in source control.

Note: Some auto-generated folders such as `output/offset` or `output/plots` may not exist until runtime.