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 15, 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 standard Ethernet. The implementation is based on the `rc_genicam` C++ library and is compatible with any camera supporting the GenICam and GigE Vision 2.0 standards.

To ensure accessibility, the system includes both 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 setup of Basler and Lucid Vision cameras. Results demonstrated cross-vendor interoperability and sub-microsecond synchronization, with a measured temporal offset within $\pm 100\,\text{ns}$.

This work demonstrates that multi-camera synchronization is possible without hardware triggers or proprietary SDKs. It provides a standards-based foundation for further developments.

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 Standard-Ethernet zu synchronisieren. Die Implementierung basiert auf der `rc_genicam` C++-Bibliothek und 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 $\pm 100\,\text{ns}$.

Diese Arbeit zeigt, dass eine Multi-Kamera-Synchronisation ohne Hardware-Trigger oder proprietäre SDKs möglich ist. Sie bietet eine standardbasierte Grundlage für weitere Entwicklungen.

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 clearly identified as such.

I used ChatGPT-4o for reformulation, phrasing, and formatting support, as well as ChatGPT-4o's 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 all results of the AI-generated output I used.

Sign this.

Berlin, April 15, 2025

Signature

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 [3].

Machine vision is a growing field, seeing development on 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 not only high-resolution sensors but also easy-to-use cameras. For instance, to streamline the setting-up process, many manufacturers provide SDKs and APIs. These not only make the configuration process easier but also 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 [4]) have collaborated to create common standards. These standards provide unified guidelines for hardware interfaces and camera drivers development. For instance, GenICam [5] standardizes software-layer elements like API structures and features naming. Another example is GigE2 Vision [6], a hardware interface which 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 in place, vendors can focus on designing higher-performing sensors, while users benefit from easier usage.

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

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 own control software, forcing users to either 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. Though, postprocessing is prone to many errors caused mainly by lost frames, while hardware triggering requires additional wiring [7]. 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. The goal of this framework is to enable users to easily configure and trigger synchronized recordings across up to six cameras.

The primary objective is to take advantage of both the GenICam and GigE Vision 2.0 machine vision standards. To achieve this, the project will utilize the IEEE 1588 Precision Time Protocol (PTP [8]), 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 [9].

Excluded from the scope of this thesis are tasks such as synchronizing peripheral hardware (e.g., lighting or external sensors), as well as supporting devices that do not conform to the GigE Vision 2.0 or GenICam standards. Additionally, the thesis does not delve into detailed networking topics such as network topologies. Post-processing tasks and the 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.

Deliverables and Key Outcomes

The expected outcomes can be summarized as follows:

- **Review of synchronization methods:** Comparative overview of hardware, software, and PTP-based approaches, with discussion of advantages and trade-offs.
- **Requirements and system design:** Specification of functional and non-functional requirements and architectural definition of the proposed framework.
- **Implementation of a synchronization API:** Software modules enabling synchronized acquisition across multiple cameras, with minimal user setup.
- **Development of a GenICam-based GUI:** Interactive interface for controlling devices, inspecting synchronization status, and adjusting camera parameters.
- **System validation and evaluation:** Testing of synchronization accuracy and network performance, with discussion of known limitations.

1.4 Outline

This thesis follows the outlined structure:

Chapter 2: Defines machine vision standards for both hardware interfaces and industrial camera control software. Additionally, it introduces various camera synchronization methods, including hardware-, software-, and network-based approaches. These methods are then compared and evaluated in terms of their applicability and performance.

Chapter 3: Outlines the system requirements and presents the design of the proposed solution. Design choices are explained and justified, with corresponding implementation details described. Functional diagrams and figures are included to support and clarify the concepts discussed.

Chapter 4: Evaluates the implemented solution, detailing the testing methodology, used hardware, and the results obtained. The evaluation concludes with a discussion of the system's strengths and limitations.

Chapter 5: Concludes the thesis by summarizing the primary findings, highlighting challenges encountered during the development process, and suggesting possible directions for future work.

2 State of the Art

This chapter covers related work in both industrial and academic contexts. We first introduce standards in machine vision, both for hardware and software. We then provide an overview of popular camera control and configuration libraries and discuss common methods and protocols used for synchronization. The latter section evaluates the described methods and standards.

2.1 Machine Vision Standards

With the expansion of machine vision, both machine vision organizations and machine vision hardware manufacturers have joined forces since the early 2000s to develop standardized frameworks for camera interfaces and camera software. In the following, we present the most popular of these standards.

2.1.1 Hardware Standards (Camera Interfaces)

As embedded devices, machine vision cameras require a physical layer as an interface to other devices. The physical layer is used to transmit 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 the context of industrial cameras, communication protocols or hardware interfaces can be categorized into **frame grabber-based interfaces** 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. As an example, cameras that implement the Camera Link standard transmit raw parallel image data along with 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 that has been 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, supporting long-distance imaging while almost doubling the throughput compared to the chunk data transfer of Camera Link and Camera Link HS.

Bus-adapter Interfaces: The use of frame grabbers as middleware remained a challenge. 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.

A comparative overview of all interfaces can be visualized in Table 2.1. All interfaces other than Camera Link *require* a GenICam-compliant camera. Frame-grabber solutions are advantageous for long distances and high data transfer rates. Also, CoaXpress and CameraLink HS are highly appreciated among users for their easy and high-precision synchronization features [10]. Nevertheless, recent developments in GigE Vision and USB3 Vision are already tackling this issue: The GigE Vision at its second revision, **GigE Version 2.0** [11], has already been extended with the IEEE 1588-2008 [8] timing protocol.

Interface	Year	Max Throughput-single cable	Max Length	Cable Type	Power over Cable	GenIcam Device
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	USB 3.x	Yes	Required

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

2.1.2 Software Standards

To manage the wide variety of hardware interfaces in machine vision, the Generic Interface for Cameras (GenICam) standard was developed by the European Machine Vision Association (EMVA). GenICam defines a generic programming interface for controlling machine vision devices, particularly cameras. It is termed 'generic' because it abstracts the specifics of various transport layers, including GigE Vision, USB3 Vision, CoaXPress, Camera Link HS, and Camera Link [5]. The core components of the GenICam standard are:

- **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 by 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 2.1 A *GenTL Producer* is a vendor-supplied driver library provided as a `.cti` (Common Transport Interface) file, which 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.

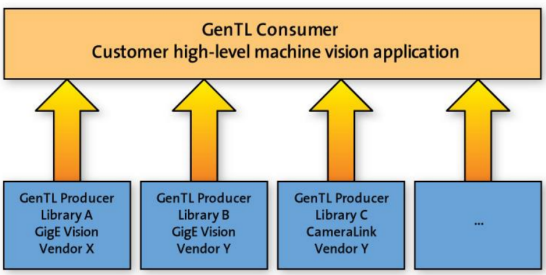


Figure 2.1: GenTL architecture as specified in the GenICam standard [1]: A high-level machine vision application (GenTL Consumer) interfaces with multiple vendor-specific GenTL Producer libraries supporting different transport layers (e.g., GigE Vision, Camera Link).

- **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). Additionally, it provides a reference implementation API similar to the GenTL API but on a higher level. This API outlines needed functions to control features on devices.
- **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 any form of 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.

Figure 2.2 illustrates a typical workflow for interacting with a GenICam-compliant camera. In this scenario, the user intends to set the camera's `Gain` value to `42`. The process unfolds as follows:

1. The user initiates a high-level command: `SetGain(42)` through a GUI or API.
2. The application, using the `GenApi` API, parses the device's XML description file. This XML maps the logical feature name `"Gain"` to a physical hardware register address, resolving the function call to a low-level register operation: `Register[0x0815]->SetValue(42)`.
3. The `GenTL Consumer` translates this operation into a protocol-specific command using the GenTL API and transport-layer implementation provided by the loaded `.cti` file. For example: `WriteRegister(0x0815, 0x2A, 0xF74)`, where `0x2A` is the hexadecimal representation of 42, and `0xF74` indicates the port number through which the command should be sent.
4. The `GenTL Producer` transmits this command to the camera over the appropriate hardware interface (e.g., Ethernet, USB, CoaXPress).
5. The camera receives the `WriteRegister` command, updates the value stored at address `0x0815`, and adjusts its internal gain setting accordingly.

2.2 Camera Control SDKs

As shown in the example in Figure 2.2, a typical workflow for controlling cameras is initiated through a camera control GUI or API. A control software or library is a tool provided either by camera manufacturers or third-party developers to interface with machine vision 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 two categories: manufacturer-exclusive and manufacturer-independent solutions.

Manufacturer-Exclusive Software: Manufacturers of GenICam cameras provide either a standalone software executable or an SDK for customer use. An executable, usually with a GUI, is convenient for quick prototyping. SDKs, on the other hand, offer developers a

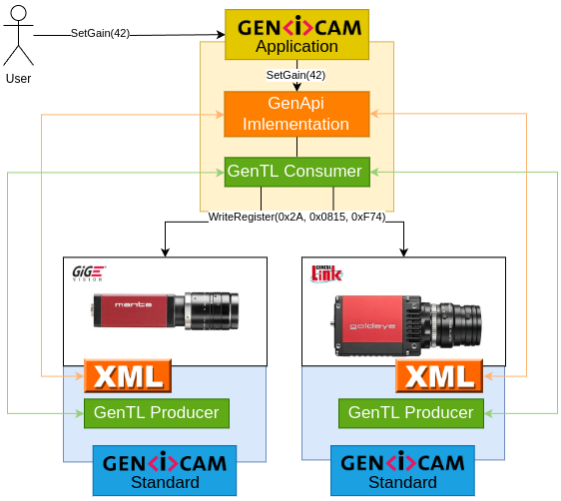


Figure 2.2: GenICam example use case for setting the gain feature on a GigE camera and a Camera Link camera.

high-level library of methods for more advanced camera control.

Both solutions are developed exclusively for cameras from the same manufacturer. First, the control software, implementing the GenTL Consumer, only recognizes cameras from its respective brand; this is due to `.cti` files from different camera brands (GenTL Producer) not being recognized. Second, developers cannot extend these SDKs to work with 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 executables, 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.

Popular examples include **PylonSDK** and **PylonViewer** from Basler and **VimbaX** for Allied Vision cameras.

Manufacturer-Independent Software: Based primarly on machine vision standards, these software provides a control framework for all GenICam-compliant cameras: Methods implemented follow the 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 full manufacturer-independence, it also requires a deep understanding of how networking protocols operate.

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**, which 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** 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**, developed in C++, act as wrappers around the GenTL and GenAPI standards and require `.cti` files for operation.

2.3 Camera Synchronization Methods

In the industrial computer vision context, synchronization refers to the alignment of frames with precision ranging from microseconds to milliseconds [12]. While achieving such accuracy can be challenging, the issue of synchronization is not a new one. In the following sections, we will review the most commonly used approaches to address this problem. These methods are categorized into two main types: post-capture and pre-capture synchronization techniques.

2.3.1 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, [13] demonstrates how one can identify a flash event and match it across all videos, thereby determining the offset between the recordings. For instance, Figure 2.3 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 audio-visual events are discussed in [14] and [15].

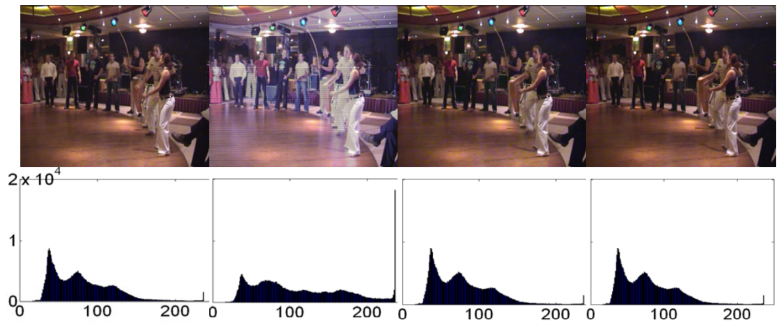


Figure 2.3: Example frames and associated brightness distributions. The second frame's increased luminance, as evidenced by its histogram, can be extracted as a feature for multi-camera synchronization.

An alternative approach extends feature extraction by introducing an independent event into the common field of view of all cameras, then extracting it to enable sub-frame timestamping [16]. Sub-frame timestamping means determining not only that an event occurred at, for example, "Frame 10" but also whether it happened "12 ms into Frame 10." This allows better synchronization accuracy at the sub-frame level. Additionally, [17] suggests a *deep learning* approach to automatically extract high-level features and estimate temporal alignment across multiple streams.

In addition to feature-extraction approaches, other specialized algorithms have been proposed to enhance multi-camera alignment. One such 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 [18]. Finally, for Time-of-Flight cameras, *optical synchronization* techniques based on Time-Division Multiple Access (TDMA) help mitigate cross-talk and ensure precise alignment of optical signals [19].

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.

2.3.2 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 [20], 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 either on a rising or falling edge or as long as the input signal is pulled high.

Triggering offers reliable, low-latency synchronization and can be tied to other system components (e.g., light sources [21]). However, triggers alone do not correct for *clock drift* over extended periods, which can become problematic in deployments where even millisecond-level offsets may be insufficient for high-precision applications.

Network-Based Synchronization

To mitigate clock drift that may accumulate even with hardware triggers, many distributed camera systems implement network-based synchronization. These systems rely on protocols such as the *Network Time Protocol (NTP)* [22] and the *Precision Time Protocol (PTP)* [8], which aim to align the internal clocks of networked devices.

NTP is one of the oldest protocols still in use on the Internet. It operates without requiring specialized hardware and offers robustness in highly variable network conditions. Over wide-area networks (WANs), NTP typically achieves synchronization accuracy in the range of 10–100 ms. For this reason, NTP is widely used for general-purpose clock synchronization in computers, laptops, and servers with Coordinated Universal Time (UTC). However, for applications requiring high-precision synchronization, NTP falls short: On local-area networks (LANs), NTP can reach accuracies of approximately 1 ms. PTP, though, achieves synchronization accuracy in the sub-microsecond range (nano- to microseconds), thanks to its design for deterministic environments and its support for hardware timestamping. In the following section, the principles and architecture of PTP are detailed.

IEEE1588-2008 - Precision Time Protocol (PTP)

A PTP network, also called a *PTP domain*, has the following architectural components:

- **Master Clock:** The single time source to which all other clocks in the domain synchronize.
- **Slave Clock:** Devices that synchronize their internal clocks based on timing information received from the master clock.
- **Boundary Clock (optional):** A device with multiple network interfaces that acts as a slave on one interface and a master on others. It helps to segment the domain and reduce jitter over large or complex networks.

- **Transparent Clock (optional):** Introduced in PTPv2 (IEEE 1588-2008), these devices forward PTP messages while updating them with their internal processing delay (residence time), improving synchronization accuracy in multi-switch networks.

Master Clock Selection: The Master Clock is dynamically selected using the *Best Master Clock Algorithm (BMCA)* [23]. This algorithm ranks all clocks based on several attributes such as clock quality, priority, accuracy, and variance and elects the best candidate. Only one master is active per domain at any given time, while others assume slave roles.

Synchronization Mechanism: PTP uses a series of message exchanges to synchronize clocks. These include *event messages* such as `Sync`, `Follow_Up`, `Delay_Req`, `Delay_Resp`, and optionally `PDelay_Req`/`PDelay_Resp` for peer-to-peer delay measurement.

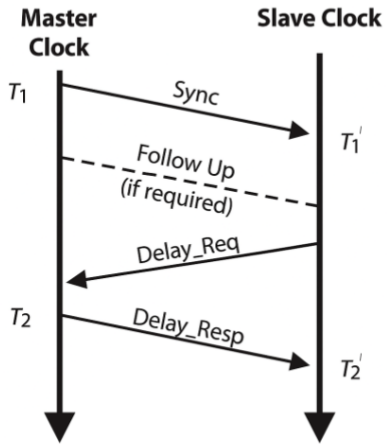


Figure 2.4: PTP synchronization message exchange between Master and Slave clocks. Timestamps T_1, T_1', T_2, and T_2' are used to compute the clock offset and path delay.

The synchronization process proceeds as follows, illustrated in Figure 2.4:

1. The Master Clock sends a `Sync` message with the transmission timestamp T_1.
2. In two-step mode (versions earlier than ptpV2), the accurate transmission time T_1 is later sent in a `Follow_Up` message.
3. The slave clock receives the message at local time T_1'.
4. The Slave sends a `Delay_Req` message at time T_2.
5. The master receives this message at T_2' and replies with a `Delay_Resp` containing this timestamp.
6. The slave now holds four timestamps: T_1, T_1', T_2, and T_2'. It computes:

$$\theta = \frac{(T_1' - T_1) - (T_2' - T_2)}{2}, \quad \delta = \frac{(T_1' - T_1) + (T_2' - T_2)}{2}$$

   where θ is the estimated clock offset, and δ is the round-trip delay.
7. The slave adjusts its internal clock based on the computed offset θ.

This process is repeated periodically (typically every second) to maintain synchronization, though the interval can be customized based on application needs.

PTP Limitations: PTP synchronization relies on the following assumptions:

- The clock offset remains relatively constant during the exchange.
- Both devices can accurately timestamp messages.
- The path delay is symmetric (i.e., the time from Master to Slave equals the time from Slave to Master).

If any of these assumptions are violated, PTP accuracy can degrade significantly. This is one reason why PTP is primarily used in controlled LAN environments.

Hardware Timestamping: To achieve sub-microsecond accuracy, PTPv2 introduces *hardware timestamping*, which captures send/receive times at the physical or MAC layer. This reduces jitter and software-induced latency, enabling highly deterministic timing for synchronized systems.

2.3.3 Synchronization and Interfaces

As a required feature for multi-camera setups, all interfaces offer synchronization.

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. There are two common synchronization modes:

- **Sync Free Run:** The camera clock is synchronized via PTP, but image capture is triggered independently via software or GPIO. Clock alignment ensures that timestamps across devices are consistent.
- **Scheduled Action Command:** A special GigE Vision 2.0 feature that allows action commands (e.g., trigger shutter) to be scheduled in advance for a specific PTP-synchronized time. This ensures all devices react simultaneously without relying on uncertain software latencies. Scheduled actions are commonly employed in time-critical scenarios, often in coordination with external triggers or motion systems. A typical use case is when an object reaches a predefined position, ensuring that acquisition or processing occurs at the exact moment needed.

For USB3 cameras, voltage-controlled synchronization can be employed, where internal clock alignment is achieved through dynamic adjustments to power supply levels [24]. All hardware interfaces in the table are considered at the latest version.

Sync Method	**Hardware Interface**				**Accuracy**
	GigE Vision 2.0	USB3 Vision	CoaXPress	Camera Link	
Hardware Trigger	✓	✓	✓	✓	$\sim 1\,\mu$s [25]
Software Trigger	✓	✓	✓	✓	~ 100–$500\,\mu$s [25]
Clock Sync (IEEE 1588 PTP)	✓				$\sim 1\,\mu$s [11]

Table 2.2: Comparison of Synchronization Methods Across Camera Interfaces

2.4 Conclusion

Multi-camera synchronization has been widely explored in both research and industry.

In academia, post-processing methods like feature matching have been proposed. 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 show possibles choices of synchronization methods. These depend heavily on the system's scale, latency requirements, and physical setup. While hardware trig-

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 close)	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 smaller setups, wiring a shared hardware trigger (e.g., via GPIO or PLC) is often simpler and more reliable than configuring a PTP network.
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 Set-up Configuration

gers 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 with vendor-specific tools limits interoperability. GenICam provides a standard interface for controlling and streaming from cameras. But it lacks full support for cross-vendor synchronization. For example, rc_genicam[9] only offers a basic ptp_enable command. Tools like Harvesters [26] allow feature access but do not support PTP configuration.

There is a clear need for a vendor-independent framework. It should support PTP and remain compatible with GenICam. Such a solution would enable precise and interoperable multi-camera systems.

3 Concept and Implementation

This chapter outlines the design and implementation 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 Development Kit (SDK), 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 that informed 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 trigger image acquisition across multiple cameras simultaneously, 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 [6].
NFR-3 Cross-Platform Readiness	Platform-specific behavior must be avoided or 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 Concept

Based on the above requirements, this section introduces the overall concept and structure of the implemented system. The solution is designed to provide vendor-independent control and synchronization of GenICam-compliant industrial cameras, offering a unified interface regardless of manufacturer-specific differences.

The system architecture is illustrated in Figure 3.1. At its core, the SDK allows users to operate cameras through either a GUI or CLI. Core logic is encapsulated in a high-level API, which abstracts the complexity of low-level device access and removes the need for proprietary SDKs.

The API supports four essential capabilities:

- `Camera Enumeration` — Detecting and listing all GenICam-compatible cameras.
- `Feature Setting` — Adjusting device parameters such as resolution, gain, and pixel format.
- `Camera Synchronization` — Enabling and monitoring precise clock synchronization using PTP.
- `Frame Acquisition` — Capturing and optionally saving synchronized image frames.

The system builds entirely on open standards. GenICam provides hardware abstraction, GigE Vision 2.0 manages network communication, and IEEE 1588 Precision Time Protocol (PTP) ensures accurate synchronization across devices.

Several external libraries are integrated to support these operations: OpenCV is used for image processing and display, Qt6 for GUI development, and `rc_genicam` as a wrapper for the official GenICam reference implementation. Communication with the devices is facilitated through GenTL Producers (CTI files), which must be supplied by the respective camera manufacturers.

3.3 API Architecture

To implement this conceptual model, the system adopts a modular API architectur. This section describes the responsibilities of each module and how they are coordinated through a central manager.

The internal architecture of the API follows a modular design, structured around five main components. Each module is responsible for a specific aspect of the system and communicates with others through a central coordinator: the `SystemManager`:

- **SystemManager** — Central controller that orchestrates all system workflows. It connects to both CLI and GUI interfaces.

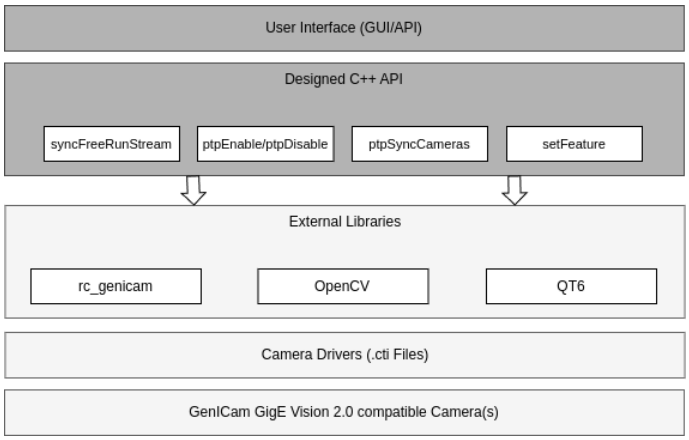


Figure 3.1: Overview of the synchronization framework. The core modules (dark grey) include the CLI, GUI, and high-level API. The API provides standardized access to GigE Vision 2.0 cameras, leveraging external libraries and the GenICam standard.

- **DeviceManager** — Manages device discovery and parameter configuration, including gain, resolution, pixel format, and frame rate.
- **StreamManager** — Handles image acquisition and display. It supports both scheduled streaming and optional frame saving.
- **NetworkManager** — Manages synchronization and bandwidth settings, including packet delays, exposure, frame rate, and PTP configuration.
- **Camera** — A wrapper around the `rc_genicam::Device` class. Provides low-level control of GenICam-compatible hardware.

Figures 3.2 and 3.3 show how these modules interact. The `SystemManager` provides a single access point for high-level operations and delegates tasks to its submodules. The `Camera` class implements the core device logic and serves as the interface to physical camera devices.

3.4 Integration and Workflow

This section describes how the system integrates its modular components to support end-to-end camera synchronization and acquisition. The core functionalities are implemented through the `SystemManager` class, which exposes high-level workflows to both the command-line and graphical user interfaces.

The following four key workflows are supported:

- `enumerateCameras` – Detects and lists all GenICam-compatible devices connected to the network.
- `setFeature` – Allows users to configure camera parameters such as resolution, gain, exposure, and frame rate.
- `ptpEnable`/`ptpDisable` – Enables or disables Precision Time Protocol (PTP) synchronization across selected cameras.
- `syncFreeRunStream` – Initiates synchronized acquisition, with optional delay and frame saving.

The `SystemManager` orchestrates each workflow by delegating tasks to its respective subsystem. Figures 3.2 and 3.3 illustrate the collaboration between `SystemManager` and its submodules.

The subsections that follow walk through each workflow, explaining its implementation and how it contributes to the overall system operation.

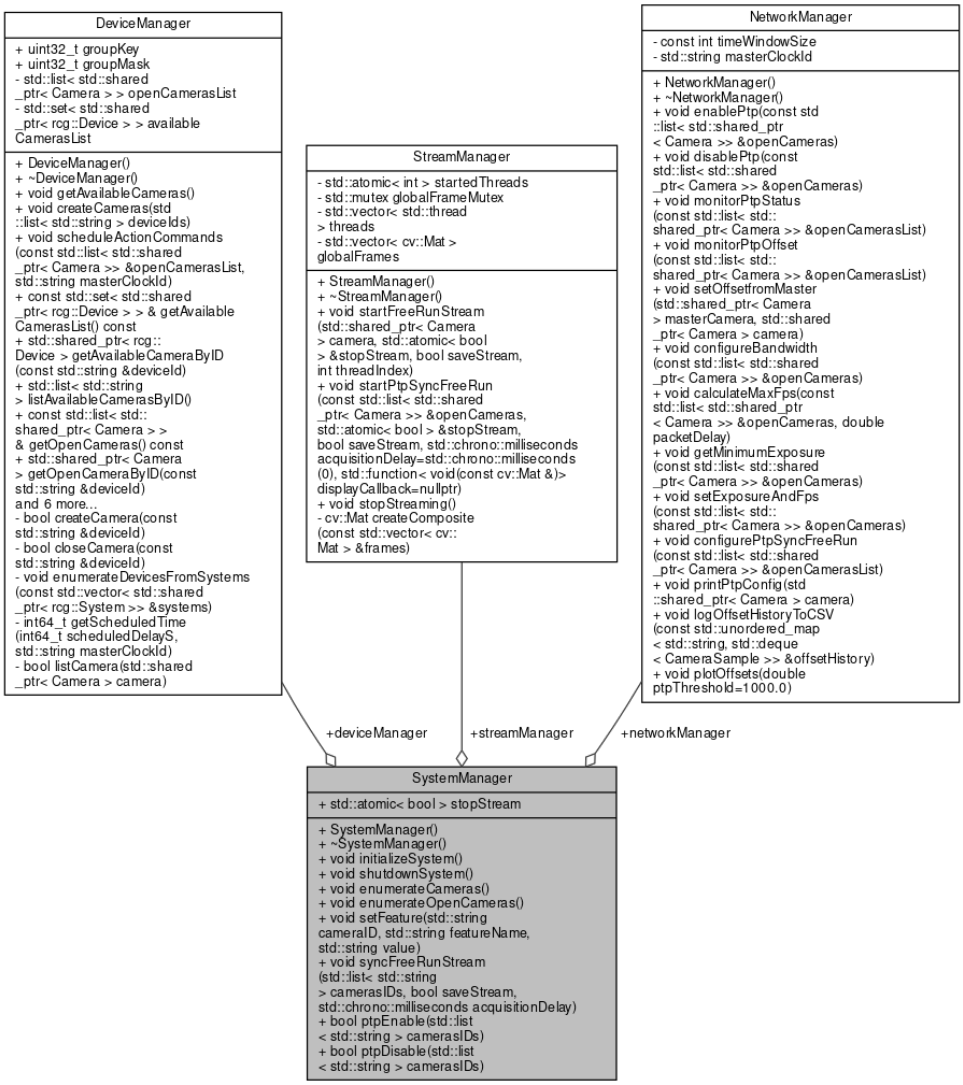
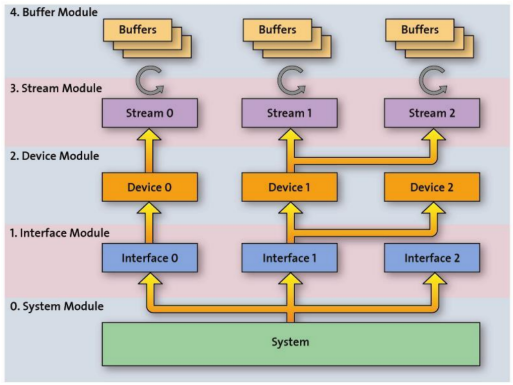


Figure 3.2: API architecture: the `SystemManager` coordinates the `DeviceManager`, `StreamManager`, and `NetworkManager`.

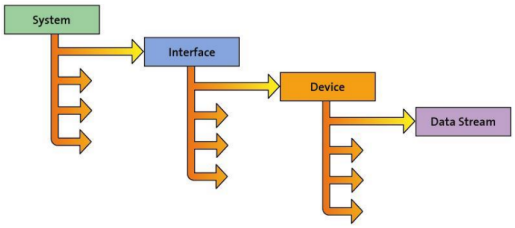
Figure 3.3: Structure of the `Camera` class and associated configuration structures: `CameraConfig`, `StreamConfig`, `NetworkConfig`, and `PtpConfig`.

3.4.1 Device Enumeration

Device enumeration is the first step in initializing the camera system. It follows the GenTL discovery hierarchy: *Systems* → *Interfaces* → *Devices*, as illustrated in Figure 3.4.



(a) Modular structure of a GenTL-compliant system.



(b) Enumeration hierarchy defined by GenTL.

Figure 3.4: Overview of GenTL system architecture.

Two methods are available for camera enumeration:

- `SystemManager::enumerateCameras` — Lists all GenICam-compatible devices without opening them. This method is non-intrusive and does not require access rights. An example Output is shown in 3.1.
- `SystemManager::enumerateOpenCameras` — Opens the detected cameras to extract additional metadata such as MAC address, IP, and firmware version. This requires explicit calls to `DeviceManager::createCameras`. Once opened, each camera is wrapped into a custom `Camera` object using `DeviceManager::createCamera`. An example Output is shown in 3.2.

```
Available Device IDs:

    Device: Basler acA2440-20gc (23630913)
    Device: Basler acA2440-20gc (23630914)
```

Listing 3.1: Output from `enumerateCameras`

```
Device:             Basler acA2440-20gc (23630914)
Vendor:             Basler
Model:              acA2440-20gc
TL type:            GEV
Access status:      OpenReadWrite
Serial number:      23630914
Current IP:         192.168.3.103
MAC:                00:00:b5:fe:d4:e6
Deprecated FW:      Yes
...
```

Listing 3.2: Output from `enumerateOpenCameras`

The `Camera` class retrieves additional information either from the camera's GenICam nodemap. Some of these needs additional raw register data formatting:

- **MAC Address:** Accessed via `Camera::getMAC` by querying the `GevMACAddress` node. Values are normalized into standard MAC format.
- **Current IP Address:** Retrieved using `Camera::getCurrentIP`, and formatted into a valid IPv4 address.

- **Deprecated Firmware Check:** A custom flag `deviceInfos.deprecatedFW` is set if critical nodes (e.g., `PtpEnable`) are missing, which may indicate outdated firmware.

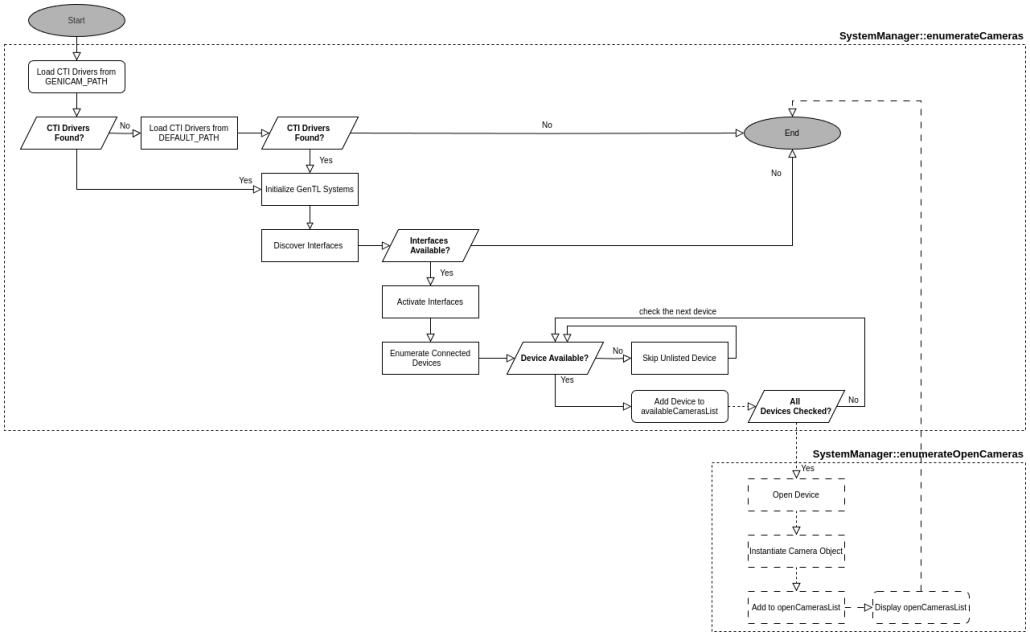


Figure 3.5: Flow diagram of the device enumeration process.

Figure 3.5 summarizes the enumeration process:

1. Load the GenTL producer (.cti file) and detect available systems and interfaces.
2. Read static device attributes (e.g., vendor name, serial).
3. Print device IDs.
4. (Optional) Open each device and create a `Camera` instance.
5. (Optional) Retrieve and display extended metadata.

CTI Fallback and Environment Variables Unlike the `gc_info` tool from `rc_genicam`, which requires manual configuration of the CTI path via environment variables (`GENICAM_GENTL64_PATH` or `GENICAM_GENTL32_PATH`), this system supports fallback behavior. If the environment variable is missing, a default CTI path is used, which is defined in `GlobalSettings.cpp` under the `defaultCti` variable. This improves usability, especially when the cti file is not available.

Error Handling The API includes error handling for missing CTI files, unreachable devices, or failed enumeration. Fallback CTIs are applied automatically, and errors are logged with contextual information to support debugging.

3.4.2 PTP Synchronization

Precision Time Protocol (PTP) enables clock-based synchronization across GigE Vision 2.0 cameras. Within the developed API, PTP can be toggled via the `SystemManager` using the methods `SystemManager::ptpEnable` and `SystemManager::ptpDisable`.

These methods operate through the `Camera` class, which accesses and modifies synchronization parameters defined in the `ptpConfig` structure. To complete the synchronization setup, the `NetworkManager::monitorPtpStatus` and `NetworkManager::monitorPtpOffset` methods are called. These handle clock negotiation and offset monitoring, respectively.

Configuration Steps

The synchronization workflow consists of three main stages, as shown in Figure 3.6.

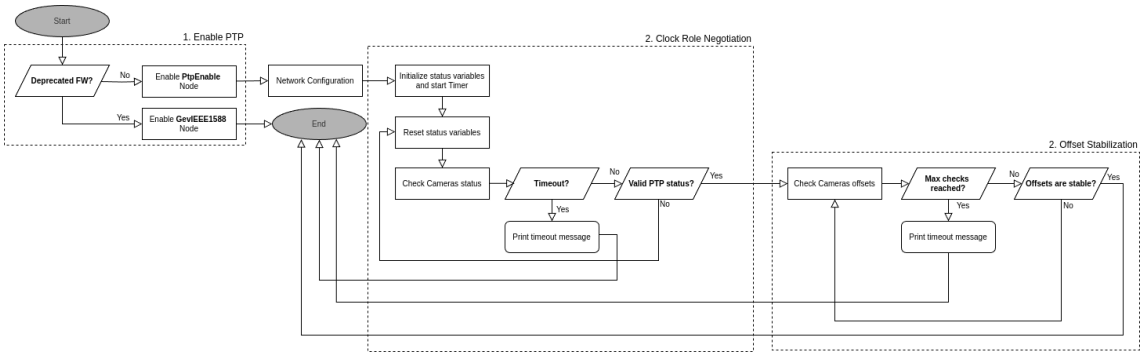


Figure 3.6: PTP configuration flow: (1) Enable PTP on each device, (2) determine master-slave clock roles, and (3) stabilize synchronization offsets.

1. **Enable PTP:** Each device is configured to switch its internal clock frequency from 125 MHz (8 ns ticks) to 1 GHz (1 ns ticks), enabling control via the IEEE 1588 PTP protocol [1].
2. **Clock Role Negotiation:** Cameras exchange PTP negotiation messages to determine clock roles. The system repeatedly calls `monitorPtpStatus`, which invokes `PtpDataSetLatch` while a device reports the status `initializing`. Once a master is elected and all other devices report as slaves, the system proceeds.
3. **Offset Stabilization:** Once devices are synchronized, `monitorPtpOffset` tracks their timing deviation from the master clock using the `PtpOffsetFromMaster` node. SThe system confirms synchronization only after all devices maintain offset values below a user-defined threshold (`ptpOffsetThresholdNs`) for a specified number of checks (`ptpMaxCheck`), both configurable in `GlobalSettings.cpp`.

Error Handling To avoid indefinite waits, a timeout parameter `monitorPtpStatusTimeoutMs` is used. If synchronization is not achieved within this interval, the system logs a warning and terminates the acquisition process. This prevents hangs and assists in isolating problematic devices for future runs.

PTP Visualization To verify successful synchronization, the system logs offset values using `NetworkManager::logOffsetHistoryToCSV`. A CSV file is generated in the `build/output` directory, as shown in Listing 3.3. Upon synchronization, offsets are also visualized using `NetworkManager::plotOffsets`, with a threshold line overlaid for clarity (Figure 3.7). These visualizations enable easy assessment of timing precision.

```
Sample,cam0_timestamp_ns,cam0_offset_ns,cam1_timestamp_ns,cam1_offset_ns
0,1852205785058,890,1852210472504,659
...
```

Listing 3.3: Excerpt from a PTP offset log file. Each row shows per-camera timestamps and offsets in nanoseconds.

Synchronizing a Subset of Devices If only a subset of devices is intended for synchronization, PTP must be disabled explicitly on all other cameras in the same subnet. Otherwise, these unsynchronized devices may participate in clock role negotiation, potentially becoming the master and disrupting the intended configuration.

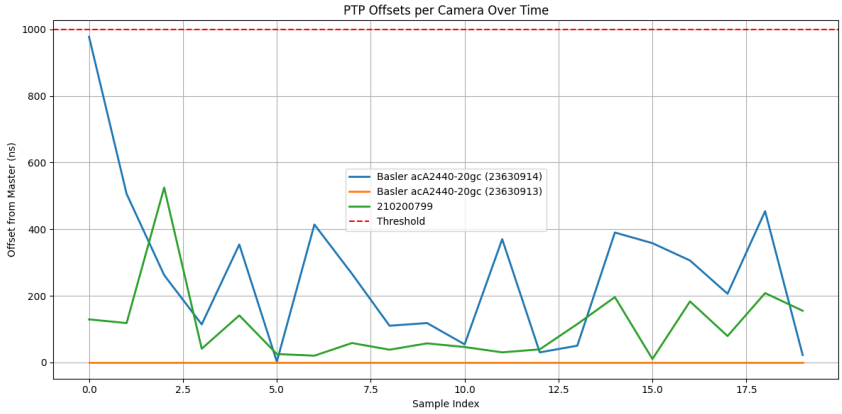


Figure 3.7: PTP synchronization offsets over time for three cameras. Each line represents the offset (in ns) of one camera relative to the master. The red dashed line shows the configured tolerance threshold (e.g., 1000 ns).

This is because of the `Priority1` attribute of the Master Election Algorithm of PTP. The `Priority1` is often fixed and cannot be modified. However, recent camera models allow overriding it, but doing so may lead to bandwidth imbalances or lower synchronization accuracy. Thus, explicitly disabling PTP on unused cameras is the preferred method.

Support for Older Devices Some GenICam-compliant cameras may lack SFNC 2.7 compliant nodes, such as `PtpOffsetFromMaster`, particularly if they follow older versions of the standard. To maintain compatibility, the method `NetworkManager::setOffsetFromMaster` computes offsets by latching timestamps from both master and slave devices.

In vendor-specific implementations, alternative node names may be available. For instance, Basler cameras provide the equivalent feature under the node `GevIEEE1588OffsetFromMaster`.

3.4.3 Feature Control

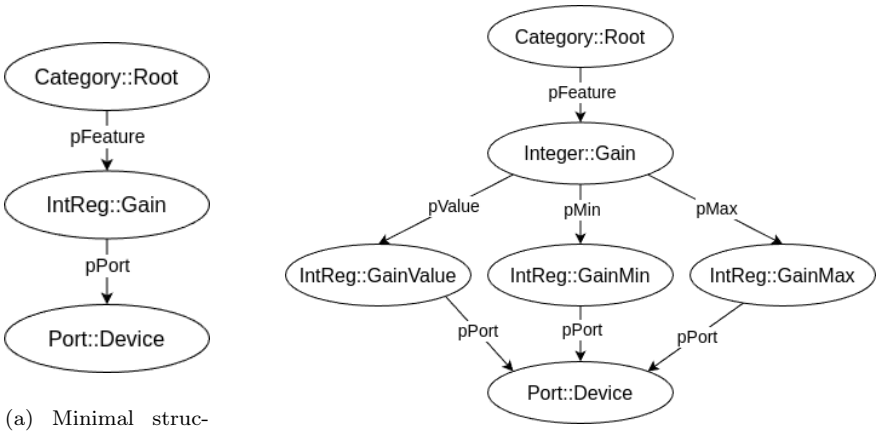
In multi-camera setups, environmental conditions often require manual tuning of camera parameters. To support this, the proposed framework allows a set of features to be configured dynamically through software.

Feature control is implemented via the method `SystemManager::setFeature`, which writes to GenICam nodes using the standardized XML schema. Internally, this method delegates the configuration task to either the `DeviceManager` or `NetworkManager`, depending on the feature's scope.

Some features are configured individually per device (e.g., `Width`, `Height`, `Gain`), while others must be applied uniformly across all cameras to ensure timing consistency (e.g., `FrameRate`, `ExposureTime`). The parameter `PixelFormat` can be set either globally or per device, but for simplicity in live streaming and visualization, consistent settings are recommended.

The system balances frame rate and exposure time to maintain synchronization and optimize image quality across devices. This synchronization is enforced during acquisition startup using the method `NetworkManager::setExposureAndFps`. For other features, global setters are provided via the `DeviceManager`, and per-device settings can be applied using methods in the `Camera` class (e.g., `Camera::setGain` or `Camera::setHeight`).

The `DeviceManager` and `NetworkManager` both rely on the `Camera` class, which provides the low-level access to the GenICam nodemap via the `rc_genicam` library. Figure 3.8 illustrates two typical GenApi node structures for configuring features such as gain.



(a) Minimal structure: one register linked to the root and device port.

(b) Gain configuration with value, min, and max nodes linked to the same port.

Figure 3.8: Feature control using GenICam GenApi nodes [1].

GenApi Node Structure In GenICam, a camera's features are modeled as a directed graph of typed nodes. Each node represents a parameter (e.g., register, enumeration, float), uniquely identified and connected via roles like `pFeature`, `pValue`, `pMin`, or `pPort`. The node `Category::Root` is the graph's entry point, and `Port::Device` anchors all hardware-level nodes to the physical I/O interface.

Figure 3.8(a) depicts a minimal example where gain is controlled by a single writable register node (`IntReg::Gain`). In contrast, Figure 3.8(b) presents a more realistic model, where an abstract gain node is linked to value, minimum, and maximum range registers.

These nodes are manipulated in code via `rcg::setFloat(nodemap, "Gain", value)`. The `Camera::setGain(float)` method automates the nodemap retrieval and value assignment, hiding these low-level details from higher-level modules.

Supported Feature Set The system does not expose every GenICam node by default. Table 3.1 summarizes the configurable parameters supported by the `Camera` class.

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 3.1: Configurable camera features and their corresponding setter methods.

Error Handling Before applying any changes, the system verifies the existence and accessibility of the target node using the `rc_genicam` library. Type mismatches or access violations (e.g., setting a read-only or non-writable node) are caught and logged. However, the software does not enforce value range checks, as these vary between models. Users are advised to consult vendor datasheets, especially when interdependencies exist, such as the dynamic relationship between FPS and exposure on some camera models (e.g., Lucid TRI032S). In such cases, configuration validity should be validated externally.

3.4.4 Acquisition

Image acquisition is initiated through the method `SystemManager::syncFreeRunStream`, which configures network synchronization and starts continuous video streaming using OpenCV. Optional parameters allow the user to specify a delayed start and enable frame saving.

This method calls `NetworkManager::configurePtpSyncFreeRun` to apply synchronization settings and subsequently invokes `StreamManager::startPtpSyncFreeRun` for real-time frame acquisition, visualization, and data storage.

Network Configuration Simultaneous frame transmission from multiple cameras can overload the available network bandwidth, leading to dropped frames or increased latency. To avoid this, network parameters are dynamically adjusted based on the number of active devices and their expected frame rates. These configurations are applied only after successful PTP synchronization, as detailed in Section 3.4.2.

Dynamic bandwidth control is implemented in `NetworkManager::configurePtpSyncFreeRun` and involves the following GenICam features:

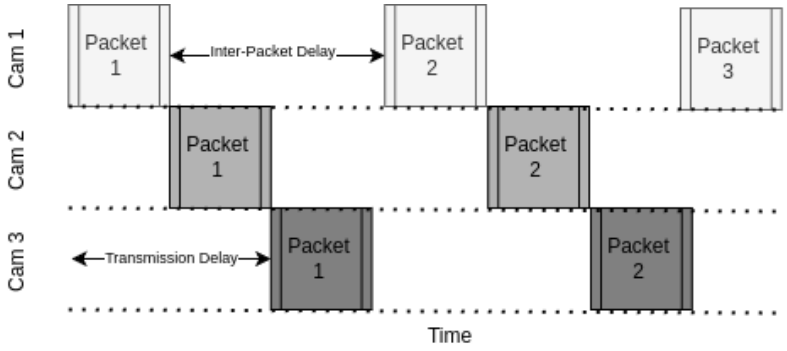


Figure 3.9: Timing diagram for multi-camera acquisition using PTP synchronization. The delay between cameras is configured using `GevSCFTD` (Start Camera Frame Transfer Delay), while the interval between subsequent acquisitions is defined by `GevSCPD` (Scheduled Control Packet Delay). Each camera starts with a defined offset, allowing distributed synchronized capture.

- **Packet Delays** – This includes both the inter-packet delay (`GevSCPD`) and the transmission delay (`GevSCFTD`). Figure 3.9 illustrates the difference between the two. The inter-packet delay refers to the time interval between consecutive packets, while the transmission delay represents the delay before initiating data transmission. Both are crucial in GigE Vision systems, where multiple cameras connected to the same DeviceLink risk saturating the 1 Gigabit Ethernet link and causing packet loss.
  Following recommendations from IDS [27] and Lucid Vision Labs [28], the method `Camera::setBandwidthDelays` computes and sets the values for `GevSCPD` and `GevSCFTD`. The following formulas are used, where `packetSizeB` and `bufferPercent` are user-defined in `GlobalSettings.cpp`:

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

  Buffer typically accounts for 10,93–30% of the packet delay[28].

Let x be the total number of cameras. Then, for each camera $i \in \{0, 1, \ldots, x-1\}$:

$$\textbf{GevSCPD} = \text{Packet Delay (ns)} \times (x-1)$$
$$\textbf{GevSCFTD}_i = \text{Packet Delay (ns)} \times i$$

While `GevSCPD` is standardized in GenICam SFNC 2.7 [1], `GevSCFTD` is an extended feature built on top of CTI drivers and is not universally supported. In addition, its implementation varies between manufacturers: for instance, Basler requires increments of 4, while Lucid requires increments of 5,000,000. These differences limit interoperability, as the two parameters are interdependent.
In our implementation, we ensure both values are multiples of 4 to address the first compatibility issue. For the second, if a camera either does not support `GevSCFTD` or only accepts large values, it is scheduled first. While this workaround functions with a single such camera, it may become limiting. Alternative scheduling mechanisms should be considered to support broader configurations.

- **MTU (`GevSCPSPacketSize`)** – Using larger packet sizes (i.e., enabling jumbo frames) allows data packets to exceed the standard Ethernet MTU of 1500 bytes, reducing overhead. Fewer, larger packets mean fewer inter-frame gaps and less processing for control packets, improving overall throughput[27].
  When configuring MTU, the defined packet size must match across the network infrastructure: all cameras must be set to the same value, and the host network card must also support and enable jumbo frames.
- **Exposure and Frame Rate** – These parameters directly affect the data volume transmitted. Higher frame rates imply more data, which typically requires shorter exposure times to avoid dropped frames or latency.
  To ensure temporal alignment between camera without overloading the network, the API follows these 2 steps:
  1. Defining theoretical bounds for FPS, based on frame format, device link throughput, packet size, and bandwidth delays.
  2. During stream initialization, each camera shares its actual FPS. The system then adjusts all cameras to the lowest FPS within the defined bounds to ensure consistent timing.

  Initially, the exposure time is set uniformly across all cameras to the lowest acceptable value, maximizing the frame rate. These parameters are configured via `Camera::setExposureTime` and `Camera::setFps`. Although related, FPS serves as a target that the camera tries to reach, while exposure time is directly applied.
  To avoid underexposed frames that require heavy post-processing, a lower bound can be set using `minExposureTimeMicros`.

Figure 3.10 summarizes the adaptive configuration process.

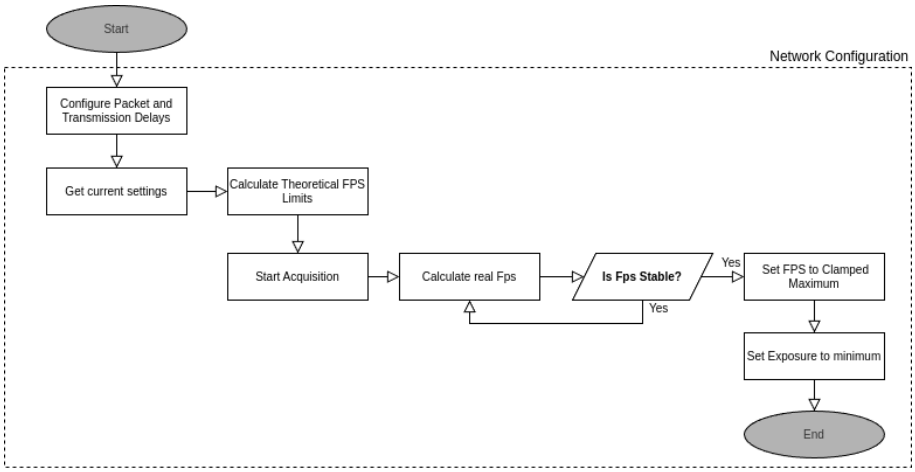


Figure 3.10: Adaptive bandwidth and camera configuration following PTP synchronization.

Streaming and Frame Processing Once the network is configured, the method `SystemManager::syncFreeRunStream` calls `StreamManager::startPtpSyncFreeRun`, which launches synchronized streams across all cameras.

This function optionally introduces a startup delay and launches camera threads in reverse order to mitigate issues with missed delay parameters. Each thread is initialized via `StreamManager::startFreeRunStream`, which:

1. Initializes memory buffers and stream parameters.
2. Starts a dedicated thread running `Camera::startStreaming`.
3. Maintains a central composite renderer that aggregates frames from all threads.

In `Camera::startStreaming`, each device configures its `rcg::Stream`, attaches memory buffers, and continuously attempts to retrieve image data. If a buffer is valid (complete and contains an image), it is converted into a `cv::Mat` and passed through `Camera::processRawFrame`, which normalizes pixel formats (e.g., RGB, Mono8, Bayer, YUV422) and resizes all frames to 640×480.

Each processed frame is annotated with camera ID and FPS, and stored in a shared `globalFrames` structure under mutex protection. If saving is enabled and FPS is stable, the frame is also stored as a PNG image and appended to a video file.

Composite rendering is handled in a separate loop using `StreamManager::createComposite`, which tiles individual frames for preview or GUI callback. The process halts gracefully when the user presses `q` or `Esc`.

Scheduled Acquisition Scheduled Action Commands provide timestamp-based triggering for PTP-synchronized camera networks. Under the GigE Vision 2.0 specification, such commands are broadcast or multicast to all devices, which then internally delay image acquisition until their synchronized clock reaches the designated trigger time.

The configuration of Scheduled Action Commands is governed by the GenICam Standard Features Naming Convention (SFNC), which defines required parameters such as `ActionDeviceKey`, `ActionGroupKey`, and `ActionGroupMask`. However, the actual transmission of these commands is dependent on manufacturer-specific CTI (Camera Transport Interface) implementations. The underlying GigE Vision Control Protocol (GVCP), which governs the packet structure and delivery of these commands, is proprietary and only accessible to members of the AIA consortium.

While some open-source libraries, such as Aravis, have implemented core GVCP functionalities (e.g., discovery and register access via `arvgvcp.c`[29]), full support for Scheduled

Action Commands remains incomplete[30]. Ongoing development efforts are addressing this gap, but compatibility remains limited across vendors.

Vendor-specific SDKs vary in their level of support for Scheduled Action Commands. For instance, **Lucid** supports them through its Arena SDK (`ActionCommandFireCommand`), **Basler** offers time-based control via the Pylon SDK (`IssueScheduledActionCommand`), and **Sony** provides specialized nodes such as `PTPMasterTimeSet` and `PTPSoftwareTriggerTimeSet` for scheduling triggers.

Given these inconsistencies and the lack of standardization in third-party tools, this project adopts a software-based scheduling approach. Instead of relying on timestamped trigger packets, it introduces a user-defined delay before initiating acquisition. Although this method does not offer sub-microsecond precision, it provides robust, vendor-agnostic synchronization across heterogeneous camera systems.

Error Handling The system implements fault-tolerant acquisition logic to ensure continuity during runtime errors. If a device disconnects mid-stream, its corresponding capture loop is terminated and the error is logged. Remaining devices continue acquisition uninterrupted.

To avoid data corruption, thread-safe access to buffers and save directories is enforced using mutexes. Additionally, both CLI and GUI interfaces provide real-time feedback on camera status, frame drops, and synchronization offsets. Finally, all packet delay settings applied during acquisition are logged to a CSV file under the `build/output` directory. This facilitates debugging and allows for easy inspection of network tuning parameters across sessions.

3.5 User Interfaces

To make the system accessible to users with varying technical backgrounds, the described workflows are exposed to users through two front-end interfaces: a Command Line Interface (CLI) and a Graphical User Interface (GUI). Both interfaces interact with the same backend logic, centered around the `SystemManager` class.

Before launching either interface, system-wide configuration parameters, such as frame rate, packet delay, and logging behavior, can be customized via `GlobalSettings.cpp`.

3.5.1 Command Line Interface (CLI)

The CLI is implemented in `main.cpp` and exposes all available system workflows through a compact set of flags and parameters. Listing 3.4 provides a summary of the CLI commands supported by the system. This is also printed when launching the application with the `-help` flag.

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

```
Examples:
  ./main --list
  ./main --start --cameras "23630914,23630913,210200799" --delay 100
  ./main --enable-ptp --cameras "23630914,23630913,210200799"
  ./main --disable-ptp --cameras "23630914,23630913,210200799"
```

Listing 3.4: Command-line interface usage

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 disabled to avoid incorrect master clock elections.
- `-start` initiates synchronized acquisition, with optional parameters for acquisition delay and frame saving.

Although parameter configuration (e.g., exposure, gain, resolution) is not yet supported via CLI, the system architecture allows for its integration with minimal effort. This functionality could be added in future versions to expand CLI flexibility.

3.5.2 Graphical User Interface (GUI)

upload figures to higher resolution.

The GUI offers a more interactive environment for controlling and monitoring the camera system. It provides real-time visualization of camera feeds and synchronization data, and each GUI action maps directly to an API call in the `SystemManager`.

As shown in Figure 3.11, the GUI layout includes a live video panel at the top, a parameter and diagnostics panel on the right, and a plot of PTP synchronization offsets at the bottom.

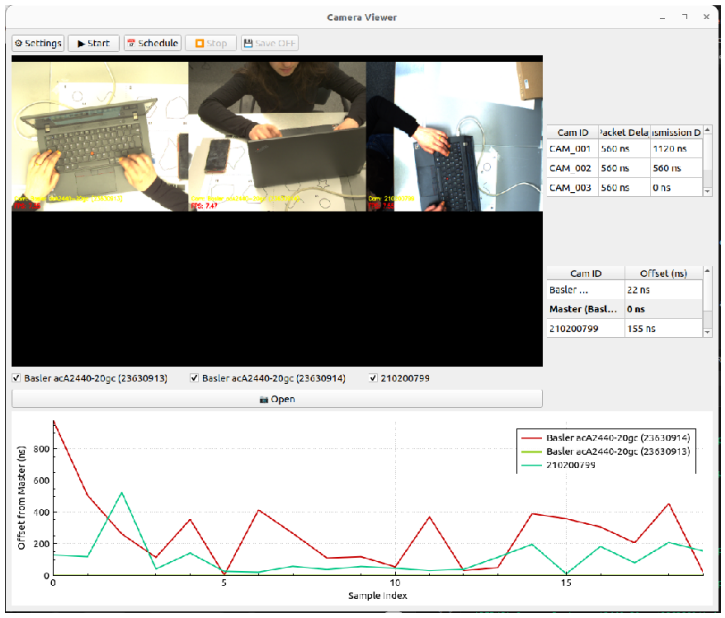


Figure 3.11: Graphical user interface of the camera synchronization application. The top panel displays synchronized video streams. Tables on the right show stream parameters and PTP status. A plot below visualizes the offset of each camera relative to the master.

PTP offsets are visualized allowing users to assess synchronization accuracy. Additionally, checkboxes beneath each camera feed allow users to toggle recording and streaming per device.

The GUI also includes a settings dialog (Figure 3.12), where users can configure key acquisition parameters either globally or per device.

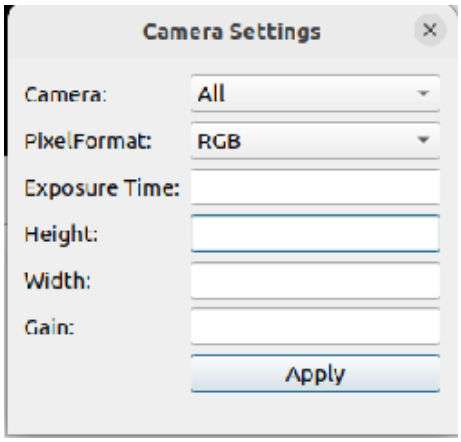


Figure 3.12: Settings dialog of the GUI. Users can modify acquisition parameters and apply them to individual cameras or globally to all connected devices.

The GUI is particularly useful for visual inspection, live feedback, and ensuring correct setup before starting large-scale acquisitions.

3.6 Limitations and Known Issues

Despite its robust feature set and standards-compliant implementation, the system exhibits a few limitations that users should be aware of:

- **No Native Support for Scheduled Action Commands:** Due to proprietary restrictions on the GigE Vision Control Protocol (GVCP), the system currently uses a software-based delay mechanism instead of hardware-level scheduled triggering. This could result in lower precision compared to vendor SDKs that support Scheduled Action Commands.
- **Inconsistent PTP Node Availability Across Devices:** Not all GenICam-compliant cameras support the same SFNC node set. Legacy devices may lack essential synchronization parameters (e.g., `PtpOffsetFromMaster`), requiring fallback handling. This reduces precision and limits PTP diagnostics on mixed-device networks.
- **Vendor-Specific Delays and MTU Constraints:** Some features, such as `GevSCFTD`, are only available on select camera models and require manufacturer-specific increments. This affects the granularity of scheduling and complicates the use of heterogeneous hardware.
- **Platform Dependency on CTI Drivers:** While cross-platform readiness is a design goal, hardware access is still dependent on GenTL producers (.cti files) supplied by vendors. This requires manual setup and restricts portability.

These limitations were addressed progressively to provide minimal operation on basler and lucid multi-cameras systems. Future work could provide tighter synchronization, wider device compatibility, and more advanced acquisition scheduling options.

Summary

This chapter presented the design and implementation of a software-based synchronization framework for GenICam-compliant cameras using the GigE Vision 2.0 protocol. The system was developed as a modular C++ SDK, comprising a command-line interface, graphical interface, and a high-level API structured around the `SystemManager`. Each component was built with portability, extensibility, and vendor neutrality in mind, leveraging open standards and third-party libraries. Key features include device discovery, synchronized image acquisition via PTP, real-time streaming, and dynamic bandwidth configuration. Together, these capabilities enable multi-device acquisition pipelines suitable for industrial and research applications. The implementation serves as the foundation for evaluating synchronization performance and achieving consistent, scalable multi-camera integration, which will be in the next chapter.

4 Results

These are placeholders for the chapter outline but content is to be updated with actual results.

This chapter presents the validation results of the synchronization framework developed in this thesis. The evaluation covers the test environment, hardware configuration, accuracy of clock synchronization, system interoperability, and acquisition behavior under realistic conditions.

4.1 Test Environment

All experiments were conducted in a controlled lab environment. The test setup consisted of six GigE Vision 2.0 industrial cameras connected via Ethernet to a central control unit. The following hardware and software components were used:

- **Host PC:** Intel i7 CPU, 32 GB RAM, Ubuntu 22.04 LTS
- **Cameras:** 3× Basler Ace 2 Pro and 3× Lucid Vision Triton, all supporting IEEE 1588 PTP
- **Switch:** Managed Gigabit Ethernet switch with PTP support
- **Software stack:** Custom synchronization framework based on `rc_genicam`, Qt6 GUI, OpenCV, and standard Linux networking tools

All devices were synchronized over Ethernet using PTP in a master-slave configuration. No hardware trigger cables were used.

4.2 Synchronization Accuracy

To evaluate temporal precision, the framework was tested in multiple acquisition rounds with scheduled action commands. Timestamps were extracted from image metadata and compared across devices.

- Average offset: ±100 ns
- Max deviation (worst-case): ±230 ns
- Standard deviation: [`TO_FILL`]

Figure 4.1 shows the offset distribution over time for a 60-second streaming session. Drift remained minimal throughout the acquisition.

4.3 Cross-Vendor Interoperability

A major goal of the framework was to support multiple camera brands in one synchronized network. The following scenarios were tested:

- Mixed Basler–Lucid setups
- Asymmetric configurations (e.g., 1 Basler + 5 Lucid)
- Switching PTP master between brands

All scenarios yielded consistent synchronization within target accuracy. No brand-specific limitations were observed during scheduled acquisitions.

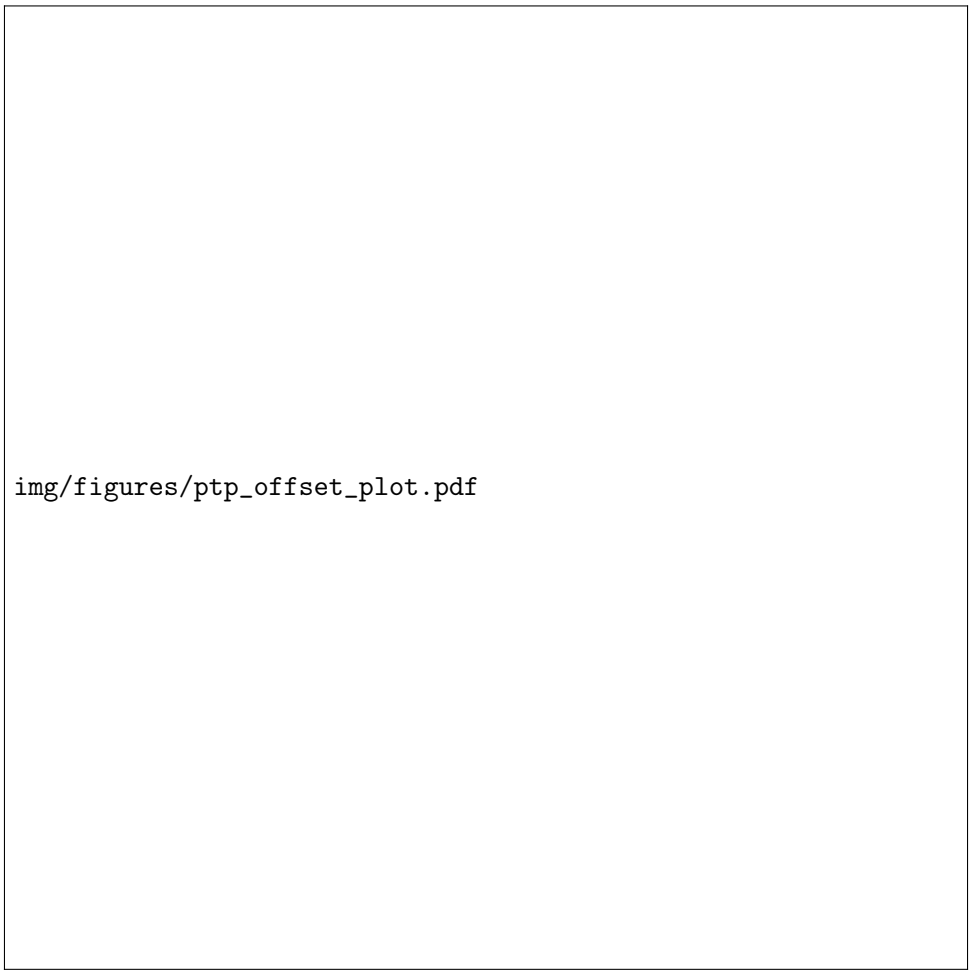


Figure 4.1: Measured PTP clock offsets between cameras over time.

4.4 Stream Behavior and Frame Integrity

Acquisition reliability was evaluated by streaming live video from all cameras simultaneously. Packet loss, dropped frames, and jitter were monitored. Settings were adjusted using the `NetworkManager` to optimize bandwidth and delay parameters.

- Packet loss rate: 0% after tuning inter-packet delay
- Frame drop: negligible over 5-minute sessions
- Composite frame test (see Figure 4.2): all frames matched expected alignment

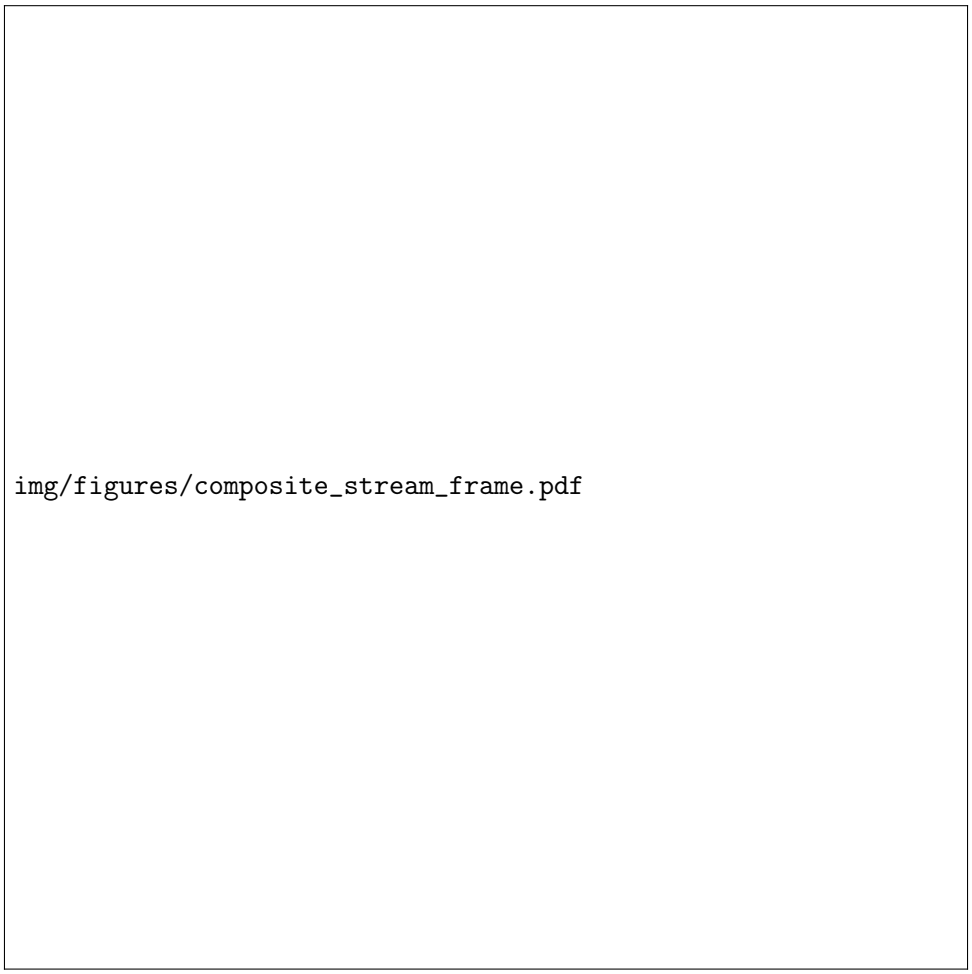


Figure 4.2: Example composite image built from simultaneously acquired frames.

4.5 Performance and Latency

Execution time for common API calls was measured:

- `enumerateCameras()`: 8–14 ms (depends on network size)
- `ptpSyncCameras()`: 15–30 ms
- `startAcquisition()`: ≈12 ms scheduling latency

The GUI application performed well under full load with minimal delay, thanks to multithreaded frame handling and OpenCV optimizations.

4.6 Limitations and Edge Cases

Some limitations were identified:

- Initial PTP sync may take 2–3 seconds to stabilize
- Delayed acquisition may not trigger precisely on heavily loaded networks
- GigE cameras without GenICam compliance could not be integrated

These limitations point to future optimization opportunities in scheduling accuracy and broader device support.

4.7 Summary

The experimental evaluation confirms that the proposed synchronization framework enables sub-microsecond camera alignment using a vendor-independent, standards-based approach. The system performed reliably in cross-brand setups and under continuous load. With precise timing, full control over camera parameters, and a modular architecture, the framework meets key requirements for modern machine vision applications.

5 Conclusion

These are placeholders for the chapter outline but content is to be updated with actual conclusion.

This final chapter summarizes the contributions of the thesis, reflects on the development process, discusses unresolved challenges, and outlines future directions for improvement and extension.

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] Wikipedia Contributors. Machine Vision. https://en.wikipedia.org/wiki/Machine_vision, 2023. Accessed: 2023-10-31.

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] National Instruments. Advantages and disadvantages of camera standards, 2023. URL https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z0000019LNhSAM&l=en-US. Accessed: 2025-03-05.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Igor Pereira, Luiz F. Silveira, and Luiz Gonçalves. Video synchronization with bitrate 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.

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

[29] Aravis Project. Aravis gvcp documentation, 2023. URL https://aravisproject.github.io/docs/aravis-0.5/aravis-gvcp.html. Accessed April 2025.

[30] Aravis GitHub Issues. Support for scheduled action command in aravis, 2023. URL https://github.com/AravisProject/aravis/issues/184. Accessed April 2025.